Introduction to Programming: Lecture 5

K Narayan Kumar

Chennai Mathematical Institute

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

22 August 2013

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.

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.

```
transpose []:xs = []
transpose xs = (map head xs):transpose (map tail xs)
```

List notation: ranges

List notation: ranges

- `[m..n]` $\leadsto$ `[m,m+1,m+2,...,n]`
  - Empty list if `n < m`

List notation: ranges

- `[m..n]` $\leadsto$ `[m,m+1,m+2,...,n]`
  - Empty list if `n < m`

```
[1..7] = [1,2,3,4,5,6,7]
[3..3] = [3]
[4..3] = []
```

List notation: ranges

- `[m..n]` $\rightsquigarrow$ `[m,m+1,m+2,...,n]`
  - Empty list if `n < m`

```
[1..7] = [1,2,3,4,5,6,7]
[3..3] = [3]
[4..3] = []
```

- Arithmetic progressions

```
[1,3..8] = [1,3,5,7]
[2,5..19] = [2,5,8,11,14,17]
```

List notation: ranges

- `[m..n]` $\leadsto$ `[m,m+1,m+2,...,n]`
  - Empty list if `n < m`

```
[1..7] = [1,2,3,4,5,6,7]
[3..3] = [3]
[4..3] = []
```

- Arithmetic progressions

```
[1,3..8] = [1,3,5,7]
[2,5..19] = [2,5,8,11,14,17]
```

- Lists in descending order

```
[8,7..5] = [8,7,6,5]
[12,8..-9] = [12,8,4,0,-4,-8]
```

Tuples in Haskell

- It is often useful to keep multiple pieces of data together.

Tuples in Haskell

- It is often useful to keep multiple pieces of data together.
- Data about students: Name, Id Number, Date of Birth, ...

Tuples in Haskell

- It is often useful to keep multiple pieces of data together.
- Data about students: Name, Id Number, Date of Birth, ...

```
("Om Puri", 1007, "1 Jan 1942")
```

Tuples in Haskell

- It is often useful to keep multiple pieces of data together.
- Data about students: Name, Id Number, Date of Birth, ...

```
("Om Puri", 1007, "1 Jan 1942")
```

- Suppose the mark list for an assignment is a list of pairs where each pair consists of a name and a mark.

Tuples in Haskell

- It is often useful to keep multiple pieces of data together.
- Data about students: Name, Id Number, Date of Birth, ...

```
("Om Puri", 1007, "1 Jan 1942")
```

- Suppose the mark list for an assignment is a list of pairs where each pair consists of a name and a mark.

```
[("Amitabh", 89), ("Naseerudin", 92),
("Shahrukh", 29)]
```

Tuples

Haskell allows the grouping together of multiple types into tuples.

Tuples

Haskell allows the grouping together of multiple types into tuples.

- `(Int,Int)` is the type whose members are pairs of integers.

```
(3,-21) :: (Int,Int)
```

Tuples

Haskell allows the grouping together of multiple types into tuples.

- `(Int,Int)` is the type whose members are pairs of integers.

  ```
  (3,-21) :: (Int,Int)
  ```

- `(Int, Bool, Int)` is the type whose members are triples, the first and third component of which are integers and the second component is a boolean.

  ```
  (13,True,97) :: (Int, Bool,Int)
  ```

Tuples

Haskell allows the grouping together of multiple types into tuples.

- `(Int,Int)` is the type whose members are pairs of integers.

  ```
  (3,-21) :: (Int,Int)
  ```

- `(Int, Bool, Int)` is the type whose members are triples, the first and third component of which are integers and the second component is a boolean.

  ```
  (13,True,97) :: (Int, Bool,Int)
  ```

- `([Int],Int)` is the type whose members are pairs, the first is a list of integers and the second is an integer.

  ```
  ([],67) :: ([Int], Int)
  ([1,2],73) :: ([Int],Int)
  ```

Tuples: `fst` and `snd`

- `fst (x,y) = x` and `snd (x,y) = y`

```
fst :: (a,b) -> a
snd :: (a,b) -> b
```

Tuples: `fst` and `snd`

- `fst (x,y) = x` and `snd (x,y) = y`

  ```
  fst :: (a,b) -> a
  snd :: (a,b) -> b
  ```

- `fst([1,2,3],'a') = [1,2,3]`
- `snd ([1,2,3],'a') = 'a'`

Pattern matching with tuples

- Haskell pattern matches tuples automatically.

Pattern matching with tuples

- Haskell pattern matches tuples automatically.
- Summing the integers in a pair.

```
sumpairs :: (Int,Int) -> Int
sumpairs (x,y) = x+y
```

Pattern matching with tuples

- Haskell pattern matches tuples automatically.
- Summing the integers in a pair.

```
sumpairs :: (Int,Int) -> Int
sumpairs (x,y) = x+y
```

- Summing up integers in a list of pairs.

```
sumpairlist :: [(Int,Int)] -> Int
sumpairlist [] = 0
sumpairlist ((x,y):ps) = x+y+(sumpairlist ps)
```

Pattern matching with tuples

- Haskell pattern matches tuples automatically.
- Summing the integers in a pair.

```
sumpairs :: (Int,Int) -> Int
sumpairs (x,y) = x+y
```

- Summing up integers in a list of pairs.

```
sumpairlist :: [(Int,Int)] -> Int
sumpairlist [] = 0
sumpairlist ((x,y):ps) = x+y+(sumpairlist ps)
```

- `sumpairlist l = sum ((map sumpairs) l)`

Tuples ...

- A tuple of type `(String,Int)` can store a student's mark and a marklist is of type `[(String,Int)]`.

```
[("Amitabh",89),("Naseerudin",91),
("Shahrukh",29)]::[(String,Int)]
```

Tuples ...

- A tuple of type `(String,Int)` can store a student's mark and a marklist is of type `[(String,Int)]`.

  ```
  [("Amitabh",89),("Naseerudin",91),
  ("Shahrukh",29)]::[(String,Int)]
  ```

- Given a marklist and a student's name return the mark obtained by that student.

Tuples ...

- A tuple of type `(String,Int)` can store a student's mark and a marklist is of type `[(String,Int)]`.

```
[("Amitabh",89),("Naseerudin",91),
("Shahrukh",29)]::[(String,Int)]
```

- Given a marklist and a student's name return the mark obtained by that student.

```
lookUp :: String -> [(String,Int)] -> Int
lookUp  s ((name,marks):l)
  | (s == name) = marks
  | otherwise   = lookUp  s l
```

Tuples ...

- A tuple of type `(String,Int)` can store a student's mark and a marklist is of type `[(String,Int)]`.

```
[("Amitabh",89),("Naseerudin",91),
("Shahrukh",29)]::[(String,Int)]
```

- Given a marklist and a student's name return the mark obtained by that student.

```
lookUp :: String -> [(String,Int)] -> Int
lookUp  s ((name,marks):l)
  | (s == name) = marks
  | otherwise   = lookUp  s l
```

- Given a list of marklists and a student's name, construct a list with the marks obtained by this student.

Tuples ...

- A tuple of type `(String,Int)` can store a student's mark and a marklist is of type `[(String,Int)]`.

```
[("Amitabh",89),("Naseerudin",91),
("Shahrukh",29)]::[(String,Int)]
```

- Given a marklist and a student's name return the mark obtained by that student.

```
lookUp :: String -> [(String,Int)] -> Int
lookUp  s ((name,marks):l)
  | (s == name) = marks
  | otherwise   = lookUp  s l
```

- Given a list of marklists and a student's name, construct a list with the marks obtained by this student.

```
getStudMarks :: String -> [[(String,Int)]] -> [Int]
getStudMarks s  = map (lookUp s)
```

Giving Names to complex types

- It is irritating to write `[(String,Int)]` repeatedly.

Giving Names to complex types

- It is irritating to write `[(String,Int)]` repeatedly.
- Haskell allows us to define short names for type expressions.

```
type Marklist = [(String,Int)]
```

Giving Names to complex types

- It is irritating to write `[(String,Int)]` repeatedly.
- Haskell allows us to define short names for type expressions.

```
type Marklist = [(String,Int)]
```

- We may then write

```
lookUp :: String -> Marklist -> Int
getStudMarks :: String -> [Marklist] -> [Int]
```

and so on.

Type definitions

- A `type` definition merely creates an alias.

Type definitions

- A `type` definition merely creates an alias.
- For instance `Marklist` and `[(String,Int)]` can be used interchangeably.

Type definitions

- A `type` definition merely creates an alias.
- For instance `Marklist` and `[(String,Int)]` can be used interchangeably.
- The following is a valid defintion.

```
type Marklist = [(String,Int)]

lookUp :: String -> Marklist -> Int
lookUp  s ((name,marks):l)
  | (s == name) = marks
  | otherwise   = lookUp  s l

getStudMarks :: String -> [[(String,Int)]] -> [Int]
getStudMarks s  = map (lookUp s)
```

The function `zip`

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

The function `zip`

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

Marklists again ...

- Given a list of `Marklist`s, construct a `Marklist` giving the total marks for each student.

Marklists again ...

- Given a list of `Marklist`s, construct a `Marklist` giving the total marks for each student.
- Assume that lists are sorted by student names.

```
totalMarks [[("Amitabh", 80),("Smita", 90)],
  [("Amitabh",46),("Smita",38)]]
 = [("Amitabh",126),("Smita",128)]
```

Marklists again ...

- Given a list of `Marklist`s, construct a `Marklist` giving the total marks for each student.
- Assume that lists are sorted by student names.

```
totalMarks [[("Amitabh", 80),("Smita", 90)],
  [("Amitabh",46),("Smita",38)]]
 = [("Amitabh",126),("Smita",128)]
```

- Reuse `addMarks`.

Marklists again ...

- Given a list of `Marklist`s, construct a `Marklist` giving the total marks for each student.
- Assume that lists are sorted by student names.

```
totalMarks [[("Amitabh", 80),("Smita", 90)],
  [("Amitabh",46),("Smita",38)]]
 = [("Amitabh",126),("Smita",128)]
```

- Reuse `addMarks`.
- Here is one way to do this.

Marklists again ...

- Given a list of `Marklist`s, construct a `Marklist` giving the total marks for each student.
- Assume that lists are sorted by student names.

```
totalMarks [[("Amitabh", 80),("Smita", 90)],
  [("Amitabh",46),("Smita",38)]]
 = [("Amitabh",126),("Smita",128)]
```

- Reuse `addMarks`.
- Here is one way to do this.
  - Construct a list of the names from one of the marklists.

Marklists again ...

- Given a list of `Marklist`s, construct a `Marklist` giving the total marks for each student.
- Assume that lists are sorted by student names.

```
totalMarks [[("Amitabh", 80),("Smita", 90)],
  [("Amitabh",46),("Smita",38)]]
 = [("Amitabh",126),("Smita",128)]
```

- Reuse `addMarks`.
- Here is one way to do this.
  - Construct a list of the names from one of the marklists.
  - Strip away the names from all the marklists.

Marklists again ...

- Given a list of `Marklist`s, construct a `Marklist` giving the total marks for each student.
- Assume that lists are sorted by student names.

```
totalMarks [[("Amitabh", 80),("Smita", 90)],
  [("Amitabh",46),("Smita",38)]]
 = [("Amitabh",126),("Smita",128)]
```

- Reuse `addMarks`.
- Here is one way to do this.
  - Construct a list of the names from one of the marklists.
  - Strip away the names from all the marklists.
  - Use `addMarks` to get the total marks

Marklists again ...

- Given a list of `Marklist`s, construct a `Marklist` giving the total marks for each student.
- Assume that lists are sorted by student names.

```
totalMarks [[("Amitabh", 80),("Smita", 90)],
  [("Amitabh",46),("Smita",38)]]
 = [("Amitabh",126),("Smita",128)]
```

- Reuse `addMarks`.
- Here is one way to do this.
  - Construct a list of the names from one of the marklists.
  - Strip away the names from all the marklists.
  - Use `addMarks` to get the total marks
  - Combine the list of names and list of total marks using `zip`

Marklists again ...

- Given a list of `Marklist`s, construct a `Marklist` giving the total marks for each student.
- Assume that lists are sorted by student names.

```
totalMarks [[("Amitabh", 80),("Smita", 90)],
  [("Amitabh",46),("Smita",38)]]
 = [("Amitabh",126),("Smita",128)]
```

- Reuse `addMarks`.
- Here is one way to do this.
  - Construct a list of the names from one of the marklists.
  - Strip away the names from all the marklists.
  - Use `addMarks` to get the total marks
  - Combine the list of names and list of total marks using `zip`

```
totalMarks (x:xs) =
  zip (map fst x) (addMarks (map (map snd) (x:xs)))
```

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, you must practice speaking it.

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 [] = [[]]
```

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 [] = [[]]
```

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

Example: interleave

- `interleave x l` inserts `x` into all possible positions in the list `l`

```
interleave 3 [] = [[3]]
interleave 3 [2,3] = [[3,2,3],[2,3,3],[2,3,3]]
interleave 'a' "abcd" =
          ["aabcd","aabcd","abacd","abcad","abcda"]
```

Example: interleave

- `interleave x l` inserts `x` into all possible positions in the list `l`

```
interleave 3 [] = [[3]]
interleave 3 [2,3] = [[3,2,3],[2,3,3],[2,3,3]]
interleave 'a' "abcd" =
          ["aabcd","aabcd","abacd","abcad","abcda"]

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

Example: Permutations

- Write down function that computes all the permutations of a given list.

Example: Permutations

- Write down function that computes all the permutations of a given list.
- Notice that this problem does not have a unique answer. We are happy with a listing of all the permutations in any order.

Example: Permutations

- Write down function that computes all the permutations of a given list.
- Notice that this problem does not have a unique answer. We are happy with a listing of all the permutations 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.
- Notice that this problem does not have a unique answer. We are happy with a listing of all the permutations 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 [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.
- Notice that this problem does not have a unique answer. We are happy with a listing of all the permutations 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 [x] = [[x]]
perm (x:xs) = concat (map (interleave x)  (perm xs))
```

- The combination `concat (map f l)` appears so often that there is a function `concatMap` which does precisely this:

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

Example: Permutations

- Write down function that computes all the permutations of a given list.
- Notice that this problem does not have a unique answer. We are happy with a listing of all the permutations 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 [x] = [[x]]
perm (x:xs) = concat (map (interleave x)  (perm xs))
```

- The combination `concat (map f l)` appears so often that there is a function `concatMap` which does precisely this:

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

Exercise: What is the type of `concatMap`

Example: Partitions

- Given a list `l` is a collection of nonempty lists `l1, l2, ..., lk` such that `l = l1 ++ l2 ++ ... ++ lk`

Example: Partitions

- Given a list `l` is a 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

- Given a list `l` is a 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]]]
```

- The type of `part` is `part :: [a] -> [[[a]]]`

Example: Partitions

- Given a list `l` is a 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]]]
```

- The type of `part` is `part :: [a] -> [[a]]`

```
part [x]  = [[[x]]]
part (x:xs) =  map ([x]:) (part xs)  ++
                   map (f x) (part xs)
               where
                 f x (y:ys) = (x:y):ys
```