MATHFUN Lecture FP5
List Patterns and Recursion

Matthew Poole

`moodle.port.ac.uk`

School of Computing
University of Portsmouth

2014/15

Introduction to Lecture

- The aim of this lecture is to cover further techniques for defining functions that operate over **lists**.

Introduction to Lecture

- The aim of this lecture is to cover further techniques for defining functions that operate over **lists**.
- The first technique we use is **pattern matching**:
  - we first review our use of pattern matching so far; then
  - we show how pattern matching is used in the case of lists.

Introduction to Lecture

- The aim of this lecture is to cover further techniques for defining functions that operate over **lists**.
- The first technique we use is **pattern matching**:
  - we first review our use of pattern matching so far; then
  - we show how pattern matching is used in the case of lists.
- We then see how lists can be processed using **recursion**:
  - we present several examples of primitive recursion over lists;
  - we discuss briefly a more general recursive function.

Introduction to Lecture

- The aim of this lecture is to cover further techniques for defining functions that operate over **lists**.
- The first technique we use is **pattern matching**:
  - we first review our use of pattern matching so far; then
  - we show how pattern matching is used in the case of lists.
- We then see how lists can be processed using **recursion**:
  - we present several examples of primitive recursion over lists;
  - we discuss briefly a more general recursive function.
- This lecture is based on, and further supported by, Chapter 7 of Thompson's book.

Summary of patterns

- Many of the functions seen so far have been defined using **patterns**.
- We'll briefly summarise the forms these patterns have taken so far, before extending these to allow **list patterns**.

Summary of patterns

- Many of the functions seen so far have been defined using **patterns**.
- We'll briefly summarise the forms these patterns have taken so far, before extending these to allow **list patterns**.
- Patterns can include **literal** values, **variables**, and **wildcards**, as in our (re-)definition of `or` from lecture FP3:

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

Summary of patterns

- Many of the functions seen so far have been defined using **patterns**.
- We'll briefly summarise the forms these patterns have taken so far, before extending these to allow **list patterns**.
- Patterns can include **literal** values, **variables**, and **wildcards**, as in our (re-)definition of `or` from lecture FP3:

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

- Patterns can also include **tuples** ...

Tuple patterns

- As an example, consider the Prelude functions `fst` & `snd`:

```
fst (x,_)      = x
snd (_,y)      = y
```

- The `fst` (first) function returns the first element of a tuple (a pair), and `snd` (second) returns the second element.

Tuple patterns

- As an example, consider the Prelude functions `fst` & `snd`:

```
fst (x,_)       = x
snd (_,y)       = y
```

- The `fst` (first) function returns the first element of a tuple (a pair), and `snd` (second) returns the second element.
- Note that these **projection** functions are **polymorphic**—they work for tuples of data of any kind.
- Question: What are the types of these functions?

```
ghci>
```

Tuple patterns

- As an example, consider the Prelude functions `fst` & `snd`:

```
fst (x,_)     = x
snd (_,y)     = y
```

- The `fst` (first) function returns the first element of a tuple (a pair), and `snd` (second) returns the second element.
- Note that these **projection** functions are **polymorphic**—they work for tuples of data of any kind.
- Question: What are the types of these functions?

```
ghci> :type fst
```

Tuple patterns

- As an example, consider the Prelude functions `fst` & `snd`:

```
fst (x,_)      = x
snd (_,y)      = y
```

- The `fst` (first) function returns the first element of a tuple (a pair), and `snd` (second) returns the second element.
- Note that these **projection** functions are **polymorphic**—they work for tuples of data of any kind.
- Question: What are the types of these functions?

```
ghci> :type fst
fst :: (a,b) -> a
ghci>
```

Tuple patterns

- As an example, consider the Prelude functions `fst` & `snd`:

```
fst (x,_)      = x
snd (_,y)      = y
```

- The `fst` (first) function returns the first element of a tuple (a pair), and `snd` (second) returns the second element.
- Note that these **projection** functions are **polymorphic**—they work for tuples of data of any kind.
- Question: What are the types of these functions?

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

Tuple patterns

- As an example, consider the Prelude functions `fst` & `snd`:

```
fst (x,_)      = x
snd (_,y)      = y
```

- The `fst` (first) function returns the first element of a tuple (a pair), and `snd` (second) returns the second element.
- Note that these **projection** functions are **polymorphic**—they work for tuples of data of any kind.
- Question: What are the types of these functions?

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

Lists and list patterns

- Recall the `:` operator from the end of last lecture:

```
ghci>
```

Lists and list patterns

- Recall the `:` operator from the end of last lecture:

```
ghci> 4:[7,3]
```

Lists and list patterns

- Recall the `:` operator from the end of last lecture:

```
ghci> 4:[7,3]
[4,7,3]
```

Lists and list patterns

- Recall the `:` operator from the end of last lecture:

```
ghci> 4:[7,3]
[4,7,3]
```

- Every list can be built up using `[]` and `:`.

Lists and list patterns

- Recall the `:` operator from the end of last lecture:

```
ghci> 4:[7,3]
[4,7,3]
```

- Every list can be built up using `[]` and `:`.
- For example, the list `[7,3]` is build in the following way:

```
[]
```

Lists and list patterns

- Recall the `:` operator from the end of last lecture:

```
ghci> 4:[7,3]
[4,7,3]
```

- Every list can be built up using `[]` and `:`.
- For example, the list `[7,3]` is build in the following way:

```
[]
3:[] = [3]
```

Lists and list patterns

- Recall the `:` operator from the end of last lecture:

```
ghci> 4:[7,3]
[4,7,3]
```

- Every list can be built up using `[]` and `:`.
- For example, the list `[7,3]` is build in the following way:

```
[]
3:[] = [3]
7:[3] = [7,3]
```

Lists and list patterns

- Recall the `:` operator from the end of last lecture:

```
ghci> 4:[7,3]
[4,7,3]
```

- Every list can be built up using `[]` and `:`.
- For example, the list `[7,3]` is build in the following way:

```
[]
3:[] = [3]
7:[3] = [7,3]
```

- We can write (since `:` is right associative) the list `[7,3]` as:

```
7:3:[]
```

- In fact, Haskell lists are represented internally in this way, and `[7,3]` is just syntactic sugar for `7:3:[]`.

Lists and list patterns

- `[]` and `:` are known as the **constructors** for lists.
- We'll see (and define) other constructors in lecture FP7.
- What is important here is that constructors may also be used in patterns.

Lists and list patterns

- `[]` and `:` are known as the **constructors** for lists.
- We'll see (and define) other constructors in lecture FP7.
- What is important here is that constructors may also be used in patterns.
- We use `:` in list patterns when we want to deal with the first element and the rest of the list separately.
- For example, consider the Prelude `head` function below; this returns the first element (the head) of a list.

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

List patterns

- If we evaluate the expression:

  `head [7,3,9]`

  (or `head 7:[3,9]`) then the `x` will match `7` and the `xs` will match `[3,9]`.

List patterns

- If we evaluate the expression:

  `head [7,3,9]`

  (or `head 7:[3,9]`) then the `x` will match `7` and the `xs` will match `[3,9]`.

- We note here that:

  1. The naming scheme `x:xs` (and `y:ys`, etc.) is a standard convention; we say "x, exes and y, whys", etc;

List patterns

- If we evaluate the expression:

  `head [7,3,9]`

  (or `head 7:[3,9]`) then the `x` will match 7 and the `xs` will match `[3,9]`.
- We note here that:
  1. The naming scheme `x:xs` (and `y:ys`, etc.) is a standard convention; we say "x, exes and y, whys", etc;
  2. the pattern `x:xs` will not match the empty list; so evaluation of `head []` would fail (which is acceptable!);

List patterns

- If we evaluate the expression:

  `head [7,3,9]`

  (or `head 7:[3,9]`) then the `x` will match 7 and the `xs` will match `[3,9]`.

- We note here that:

  1. The naming scheme `x:xs` (and `y:ys`, etc.) is a standard convention; we say "x, exes and y, whys", etc;
  2. the pattern `x:xs` will not match the empty list; so evaluation of `head []` would fail (which is acceptable!);
  3. since functions have higher precedence than operators, patterns with `:` always appear in parentheses (e.g. `(x:xs)`).

List patterns

- For this particular function, we don't use the remaining elements of the list, so we could match them with a wildcard; i.e

```
head (x:_) = x
```

List patterns

- For this particular function, we don't use the remaining elements of the list, so we could match them with a wildcard; i.e

```
head (x:_) = x
```

- We can similarly (re-)define the Prelude's `tail` function:

```
tail (_:xs) = xs
```

List patterns

- For this particular function, we don't use the remaining elements of the list, so we could match them with a wildcard; i.e

```
head (x:_) = x
```

- We can similarly (re-)define the Prelude's `tail` function:

```
tail (_:xs) = xs
```

- Some example evaluations are:

```
ghci>
```

List patterns

- For this particular function, we don't use the remaining elements of the list, so we could match them with a wildcard; i.e

```
head (x:_) = x
```

- We can similarly (re-)define the Prelude's `tail` function:

```
tail (_:xs) = xs
```

- Some example evaluations are:

```
ghci> tail [7,3,9]
```

List patterns

- For this particular function, we don't use the remaining elements of the list, so we could match them with a wildcard; i.e

```
head (x:_) = x
```

- We can similarly (re-)define the Prelude's `tail` function:

```
tail (_:xs) = xs
```

- Some example evaluations are:

```
ghci> tail [7,3,9]
[3,9]
ghci>
```

List patterns

- For this particular function, we don't use the remaining elements of the list, so we could match them with a wildcard; i.e

```
head (x:_) = x
```

- We can similarly (re-)define the Prelude's `tail` function:

```
tail (_:xs) = xs
```

- Some example evaluations are:

```
ghci> tail [7,3,9]
[3,9]
ghci> head [8]
```

List patterns

- For this particular function, we don't use the remaining elements of the list, so we could match them with a wildcard; i.e

```
head (x:_) = x
```

- We can similarly (re-)define the Prelude's `tail` function:

```
tail (_:xs) = xs
```

- Some example evaluations are:

```
ghci> tail [7,3,9]
[3,9]
ghci> head [8]
8
ghci>
```

List patterns

- For this particular function, we don't use the remaining elements of the list, so we could match them with a wildcard; i.e

```
head (x:_) = x
```

- We can similarly (re-)define the Prelude's `tail` function:

```
tail (_:xs) = xs
```

- Some example evaluations are:

```
ghci> tail [7,3,9]
[3,9]
ghci> head [8]
8
ghci> tail [8]
```

List patterns

- For this particular function, we don't use the remaining elements of the list, so we could match them with a wildcard; i.e

```
head (x:_) = x
```

- We can similarly (re-)define the Prelude's `tail` function:

```
tail (_:xs) = xs
```

- Some example evaluations are:

```
ghci> tail [7,3,9]
[3,9]
ghci> head [8]
8
ghci> tail [8]
[]
```

List patterns

- As another example, consider a function to return the **absolute** value of the first element of a list, or -1 for an empty list; i.e.:

```
ghci>
```

List patterns

- As another example, consider a function to return the **absolute** value of the first element of a list, or -1 for an empty list; i.e.:

```
ghci> absFirst [-4, 7, -3]
```

List patterns

- As another example, consider a function to return the **absolute** value of the first element of a list, or -1 for an empty list; i.e.:

```
ghci> absFirst [-4, 7, -3]
4
ghci>
```

List patterns

- As another example, consider a function to return the **absolute** value of the first element of a list, or -1 for an empty list; i.e.:

```
ghci> absFirst [-4, 7, -3]
4
ghci> absFirst []
```

List patterns

- As another example, consider a function to return the **absolute** value of the first element of a list, or -1 for an empty list; i.e.:

```
ghci> absFirst [-4, 7, -3]
4
ghci> absFirst []
-1
```

List patterns

- As another example, consider a function to return the **absolute** value of the first element of a list, or -1 for an empty list; i.e.:

```
ghci> absFirst [-4, 7, -3]
4
ghci> absFirst []
-1
```

- To define this function, we'll need two patterns: one for empty lists, and one for non-empty lists:

```
absFirst :: [Int] -> Int
absFirst []     = -1
absFirst (x:xs) = abs x
```

List patterns

- As another example, consider a function to return the **absolute** value of the first element of a list, or -1 for an empty list; i.e.:

```
ghci> absFirst [-4, 7, -3]
4
ghci> absFirst []
-1
```

- To define this function, we'll need two patterns: one for empty lists, and one for non-empty lists:

```
absFirst :: [Int] -> Int
absFirst []     = -1
absFirst (x:xs) = abs x
```

- Question: is the order in which the patterns occur important?

Recursion over lists

- Recall the recursive definition `fact` from lecture FP3:

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

Recursion over lists

- Recall the recursive definition `fact` from lecture FP3:

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

- Like all primitive recursive functions on ints, this definition has:
    - a base case for `n == 0`
    - a recursive case that gives the result for any value of `n > 0` from the result for `n - 1`.

Recursion over lists

- Recall the recursive definition `fact` from lecture FP3:

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

- Like all primitive recursive functions on ints, this definition has:
  - a base case for `n == 0`
  - a recursive case that gives the result for any value of `n > 0` from the result for `n - 1`.
- We define primitive recursive functions for lists in a similar way:
  - the base case considers the empty list `[]`;
  - the recursive case gives the result for any non-empty list (one matched by `x:xs`) from the result for the tail of the list, `xs`.

Primitive recursion over lists

- Consider implementing the Prelude function:

  ```
  sum :: [Int] -> Int
  ```

  to return the sum of a list of integers.

Primitive recursion over lists

- Consider implementing the Prelude function:

  ```
  sum :: [Int] -> Int
  ```

  to return the sum of a list of integers.
- The base case is easy: the sum of the elements in `[]` is clearly 0.

Primitive recursion over lists

- Consider implementing the Prelude function:

  ```
  sum :: [Int] -> Int
  ```

  to return the sum of a list of integers.
- The base case is easy: the sum of the elements in `[]` is clearly 0.
- Now consider the recursion case for a list `x:xs`:
  - We assume `sum xs` is the sum of the elements in the tail `xs` of the list;
  - We need to add the value of the head element `x` to this sum to give the sum of the whole list.

Primitive recursion over lists

- Consider implementing the Prelude function:

  ```
  sum :: [Int] -> Int
  ```

  to return the sum of a list of integers.
- The base case is easy: the sum of the elements in `[]` is clearly 0.
- Now consider the recursion case for a list `x:xs`:
  - We assume `sum xs` is the sum of the elements in the tail `xs` of the list;
  - We need to add the value of the head element `x` to this sum to give the sum of the whole list.
- Using pattern matching, the definition is written as:

  ```
  sum []       = 0
  sum (x:xs)   = x + sum xs
  ```

Primitive recursion over lists

- As another example, consider a function:

  ```
  doubleAll : [Int] -> [Int]
  ```

  that doubles all the elements of a list; e.g.

  ```
  ghci> doubleAll [4,7,2]
  [8,14,4]
  ```

Primitive recursion over lists

- As another example, consider a function:

```
doubleAll : [Int] -> [Int]
```

  that doubles all the elements of a list; e.g.

```
ghci> doubleAll [4,7,2]
[8,14,4]
```

- The base case is straightforward: the empty list `[]` with all its elements doubled is still `[]`.

Primitive recursion over lists

- As another example, consider a function:

  ```
  doubleAll : [Int] -> [Int]
  ```

  that doubles all the elements of a list; e.g.

  ```
  ghci> doubleAll [4,7,2]
  [8,14,4]
  ```

- The base case is straightforward: the empty list `[]` with all its elements doubled is still `[]`.
- For a list `x:xs`, we assume `doubleAll xs` is the tail `xs` with all its elements doubled.

Primitive recursion over lists

- As another example, consider a function:

```
doubleAll : [Int] -> [Int]
```

  that doubles all the elements of a list; e.g.

```
ghci> doubleAll [4,7,2]
[8,14,4]
```

- The base case is straightforward: the empty list `[]` with all its elements doubled is still `[]`.
- For a list `x:xs`, we assume `doubleAll xs` is the tail `xs` with all its elements doubled.
- We need to put `2*x` at the head of this list; so

Primitive recursion over lists

- As another example, consider a function:

```
doubleAll : [Int] -> [Int]
```

  that doubles all the elements of a list; e.g.

```
ghci> doubleAll [4,7,2]
[8,14,4]
```

- The base case is straightforward: the empty list `[]` with all its elements doubled is still `[]`.
- For a list `x:xs`, we assume `doubleAll xs` is the tail `xs` with all its elements doubled.
- We need to put `2*x` at the head of this list; so

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

Primitive recursion over lists

- As another example, consider a function:

```
doubleAll : [Int] -> [Int]
```

  that doubles all the elements of a list; e.g.

```
ghci> doubleAll [4,7,2]
[8,14,4]
```

- The base case is straightforward: the empty list `[]` with all its elements doubled is still `[]`.
- For a list `x:xs`, we assume `doubleAll xs` is the tail `xs` with all its elements doubled.
- We need to put `2*x` at the head of this list; so

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

Primitive recursion over lists

- As another example, consider a function:

```
doubleAll : [Int] -> [Int]
```

  that doubles all the elements of a list; e.g.

```
ghci> doubleAll [4,7,2]
[8,14,4]
```

- The base case is straightforward: the empty list `[]` with all its elements doubled is still `[]`.
- For a list `x:xs`, we assume `doubleAll xs` is the tail `xs` with all its elements doubled.
- We need to put `2*x` at the head of this list; so

```
doubleAll []      = []
doubleAll (x:xs)  = 2*x : doubleAll xs
```

Primitive recursion over lists

- Another example (again a function from the Prelude):

  ```
  concat :: [[a]] -> [a]
  ```

  joins sub-lists together; e.g.

  ```
  ghci>
  ```

Primitive recursion over lists

- Another example (again a function from the Prelude):

  ```
  concat :: [[a]] -> [a]
  ```

  joins sub-lists together; e.g.

  ```
  ghci> concat [[1,2], [3], [4,5]]
  ```

Primitive recursion over lists

- Another example (again a function from the Prelude):

  ```
  concat :: [[a]] -> [a]
  ```

  joins sub-lists together; e.g.

  ```
  ghci> concat [[1,2], [3], [4,5]]
  [1,2,3,4,5]
  ```

Primitive recursion over lists

- Another example (again a function from the Prelude):

  ```
  concat :: [[a]] -> [a]
  ```

  joins sub-lists together; e.g.

  ```
  ghci> concat [[1,2], [3], [4,5]]
  [1,2,3,4,5]
  ```

- The base case is `[]` (there are no sub-lists), giving `[]`.
- For a non-empty list of lists `x:xs`, we assume `concat xs` is the concatenation of all the lists in the tail `xs`.

Primitive recursion over lists

- Another example (again a function from the Prelude):

  ```
  concat :: [[a]] -> [a]
  ```

  joins sub-lists together; e.g.

  ```
  ghci> concat [[1,2], [3], [4,5]]
  [1,2,3,4,5]
  ```

- The base case is `[]` (there are no sub-lists), giving `[]`.
- For a non-empty list of lists `x:xs`, we assume `concat xs` is the concatenation of all the lists in the tail `xs`.
- We need to prepend (the elements of) the list `x` onto this (and we use the list operator `++` to do this):

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

Primitive recursion over lists

- A final example of a primitive recursive definition, consider a function:

  ```
  reverse :: [a] -> [a]
  ```

  to reverse a list:

  ```
  ghci>
  ```

Primitive recursion over lists

- A final example of a primitive recursive definition, consider a function:

  ```
  reverse :: [a] -> [a]
  ```

  to reverse a list:

  ```
  ghci> reverse "hello"
  ```

Primitive recursion over lists

- A final example of a primitive recursive definition, consider a function:

  ```
  reverse :: [a] -> [a]
  ```

  to reverse a list:

  ```
  ghci> reverse "hello"
  "olleh"
  ```

Primitive recursion over lists

- A final example of a primitive recursive definition, consider a function:

  ```
  reverse :: [a] -> [a]
  ```

  to reverse a list:

  ```
  ghci> reverse "hello"
  "olleh"
  ```

- Questions:
  - What is the base case `reverse []`?
  - What is the recursive case `reverse (x:xs)`?

Primitive recursion over lists

- A final example of a primitive recursive definition, consider a function:

  ```
  reverse :: [a] -> [a]
  ```

  to reverse a list:

  ```
  ghci> reverse "hello"
  "olleh"
  ```

- Questions:
  - What is the base case `reverse []`?
  - What is the recursive case `reverse (x:xs)`?
- The solution uses ++ to append at list containing `x` to the end:

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

General recursion over lists

- Not all recursive functions over lists are primitive recursive.
- Consider, for example the Prelude function `zip`:

```
zip :: [a] -> [b] -> [(a,b)]
```

which joins two lists into a single list of tuples; e.g.

```
ghci> zip ['r', 'h', 'a'] [4, 7, 2]
[('r',4),('h',7),('a',2)]
ghci> zip [5, 7, 1, 5] ['a', 'b']
[(5,'a'),(7,'b')]
```

General recursion over lists

- Not all recursive functions over lists are primitive recursive.
- Consider, for example the Prelude function `zip`:

```
zip :: [a] -> [b] -> [(a,b)]
```

  which joins two lists into a single list of tuples; e.g.

```
ghci> zip ['r', 'h', 'a'] [4, 7, 2]
[('r',4),('h',7),('a',2)]
ghci> zip [5, 7, 1, 5] ['a', 'b']
[(5,'a'),(7,'b')]
```

- Notice that the lists don't have to be of the same type.
- Notice also that if the lists are of different lengths, the last few elements of the longer list are dropped.

General recursion over lists

- The easiest way to define `zip` is by beginning with the (recursive) case of two non-empty lists `x:xs` and `y:ys`.

General recursion over lists

- The easiest way to define `zip` is by beginning with the (recursive) case of two non-empty lists `x:xs` and `y:ys`.
- In this case we place `x` and `y` into a tuple, and zip up the tails `xs` and `ys`.

```
zip (x:xs) (y:ys)  = (x,y) : zip xs ys
```

- What happens when one of the arguments is `[]`? We simply drop the remaining elements from the other list (i.e. return `[]`).

General recursion over lists

- The easiest way to define `zip` is by beginning with the (recursive) case of two non-empty lists `x:xs` and `y:ys`.
- In this case we place `x` and `y` into a tuple, and zip up the tails `xs` and `ys`.

```
zip (x:xs) (y:ys)  = (x,y) : zip xs ys
```

- What happens when one of the arguments is `[]`? We simply drop the remaining elements from the other list (i.e. return `[]`).
- We could write all three possible cases separately:

```
zip (x:xs) []     = []
zip [] (y:ys)     = []
zip [] []         = []
```

but these can be merged using the wildcard _ to give:

```
zip _ _    = []
```