MATHFUN Lecture FP3
Pattern Matching and Recursion

Matthew Poole

moodle.port.ac.uk

School of Computing
University of Portsmouth

2014/15

Introduction to Lecture

- The main aim of this lecture is to introduce two ideas fundamental to functional programming: pattern matching and recursion.
- We start, however, by taking a quick look at:
  - modules in Haskell;
  - the difference between functions and operators, and
  - how they can be used interchangeably.

Introduction to Lecture

- The main aim of this lecture is to introduce two ideas fundamental to functional programming: pattern matching and recursion.
- We start, however, by taking a quick look at:
    - modules in Haskell;
    - the difference between functions and operators, and
    - how they can be used interchangeably.
- We then introduce the basics of Haskell's pattern matching mechanism for defining functions.
- Later, we introduce recursive function definitions.
- In later lectures we'll see how many functions are written using recursion on lists.

Modules

- As in Python, large programs are made up of modules, and there exist many module libraries (e.g. for networking, graphics, etc.).

Modules

- As in Python, large programs are made up of modules, and there exist many module libraries (e.g. for networking, graphics, etc.).
- To use definitions contained in a module, we need to **import** it.
- E.g., to be able to use the functions from the `Data.Char` module (for processing characters), we use the import declaration:

```
import Data.Char
```

Modules

- As in Python, large programs are made up of modules, and there exist many module libraries (e.g. for networking, graphics, etc.).
- To use definitions contained in a module, we need to **import** it.
- E.g., to be able to use the functions from the `Data.Char` module (for processing characters), we use the import declaration:

```
import Data.Char
```

- If we only wanted to import the functions `toUpper` and `toLower`, this would become:

```
import Data.Char (toUpper, toLower)
```

Modules

- As in Python, large programs are made up of modules, and there exist many module libraries (e.g. for networking, graphics, etc.).
- To use definitions contained in a module, we need to **import** it.
- E.g., to be able to use the functions from the `Data.Char` module (for processing characters), we use the import declaration:

```
import Data.Char
```

- If we only wanted to import the functions `toUpper` and `toLower`, this would become:

```
import Data.Char (toUpper, toLower)
```

- There is also a **standard prelude**; a special module which:
  - includes definitions of commonly used types and functions; and
  - is implicitly imported into every other module.

Functions and operators

- Haskell includes:
  - **functions** (e.g. `sqrt`, `mod`) which are used with **prefix** notation (e.g. `mod n 2`);
  - **operators** (`+`, `-`, `**`, etc.) which are used with **infix** notation (e.g. `1 + x`). There is also one prefix operator (unary minus).
- We can use any binary (two-argument) function as an operator by surrounding it with back-quotes; for example:

```
n `mod` 2    is equivalent to    mod n 2
```

Functions and operators

- Haskell includes:
  - **functions** (e.g. `sqrt`, `mod`) which are used with **prefix** notation (e.g. `mod n 2`);
  - **operators** (`+`, `-`, `**`, etc.) which are used with **infix** notation (e.g. `1 + x`). There is also one prefix operator (unary minus).
- We can use any binary (two-argument) function as an operator by surrounding it with back-quotes; for example:

```
n `mod` 2     is equivalent to     mod n 2
```

- Similarly, we can use an operator as a function by using `()`'s:

```
(+) 1 x       is equivalent to     1 + x
```

Functions and operators

- Haskell includes:
  - **functions** (e.g. `sqrt`, `mod`) which are used with **prefix** notation (e.g. `mod n 2`);
  - **operators** (`+`, `-`, `**`, etc.) which are used with **infix** notation (e.g. `1 + x`). There is also one prefix operator (unary minus).
- We can use any binary (two-argument) function as an operator by surrounding it with back-quotes; for example:

```
n `mod` 2     is equivalent to     mod n 2
```

- Similarly, we can use an operator as a function by using ()'s:

```
(+) 1 x       is equivalent to     1 + x
```

- Parentheses are needed to find the type of an operator; e.g.:

```
ghci> :t (+)
(+) :: Num a => a -> a -> a
```

Pattern matching

- We have seen two ways of defining functions so far:
  - using single expressions (e.g. `square n = n * n`); and
  - using guards.

Pattern matching

- We have seen two ways of defining functions so far:
  - using single expressions (e.g. `square n = n * n`); and
  - using guards.
- Many functions are instead given a simple and intuitive definition using **pattern matching**.
- E.g., the Boolean function `not` is defined (in the prelude), by:

```
not :: Bool -> Bool
not True     = False
not False    = True
```

Pattern matching

- We have seen two ways of defining functions so far:
  - using single expressions (e.g. `square n = n * n`); and
  - using guards.
- Many functions are instead given a simple and intuitive definition using **pattern matching**.
- E.g., the Boolean function `not` is defined (in the prelude), by:

```
not :: Bool -> Bool
not True     = False
not False    = True
```

- This definition consists of a sequence of **patterns**, with each one associated with a different result.

Pattern matching

- We have seen two ways of defining functions so far:
  - using single expressions (e.g. `square n = n * n`); and
  - using guards.
- Many functions are instead given a simple and intuitive definition using **pattern matching**.
- E.g., the Boolean function `not` is defined (in the prelude), by:

```
not :: Bool -> Bool
not True     = False
not False    = True
```

- This definition consists of a sequence of **patterns**, with each one associated with a different result.
- For a given expression (e.g. `not p`), if the first pattern is matched (i.e. `p` is `True`), then the first result is chosen.
- Otherwise, the second pattern is tried, and so on.

Pattern matching

- Let's investigate pattern matching in a little more detail, by defining for ourselves the Boolean **or** operator `||`.

Pattern matching

- Let's investigate pattern matching in a little more detail, by defining for ourselves the Boolean **or** operator `||`.
- This operator is defined in the prelude; we first "hide" it by explicitly importing all prelude definitions **except** `||`:

```
import Prelude hiding ((||))
```

Pattern matching

- Let's investigate pattern matching in a little more detail, by defining for ourselves the Boolean **or** operator `||`.
- This operator is defined in the prelude; we first "hide" it by explicitly importing all prelude definitions **except** `||`:

```
import Prelude hiding ((||))
```

- Now, a simple but naïve way of defining `||` is:

```
(||) :: Bool -> Bool -> Bool
True  || True    = True
False || True    = True
True  || False   = True
False || False   = False
```

Pattern matching

- This can be simplified by combining the first three (True) cases into one, using the **wildcard** pattern _ which matches any value:

```
(||) :: Bool -> Bool -> Bool
False || False  = False
_ || _          = True
```

Pattern matching

- This can be simplified by combining the first three (True) cases into one, using the **wildcard** pattern _ which matches any value:

```
(||) :: Bool -> Bool -> Bool
False || False  = False
_  ||  _        = True
```

- Another alternative is the following which uses both a wildcard and a named parameter:

```
(||) :: Bool -> Bool -> Bool
True || _     =  True
False || a    = a
```

- **Question**: Can the final pattern be shortened?

Recursion

- A recursive definition is one that is defined in terms of itself.
- Most of you will have seen recursion used during earlier units as an alternative to iteration (while/for loops).

Recursion

- A recursive definition is one that is defined in terms of itself.
- Most of you will have seen recursion used during earlier units as an alternative to iteration (while/for loops).
- While and for loops are clearly **imperative** constructs: they are commands that operate on a program's state.
- Pure functional programming therefore cannot involve loops, and so recursion is fundamental to the functional paradigm.

Recursion

- A recursive definition is one that is defined in terms of itself.
- Most of you will have seen recursion used during earlier units as an alternative to iteration (while/for loops).
- While and for loops are clearly **imperative** constructs: they are commands that operate on a program's state.
- Pure functional programming therefore cannot involve loops, and so recursion is fundamental to the functional paradigm.
- We'll see recursion used heavily for most of the remainder of this section of the unit, particularly when using **lists**.
- We'll review the concept of recursion using a standard example, and then briefly see a couple of extra recursive definitions.

Recursive definition of factorial

- Consider the definition of the **factorial** of a positive integer n, written either as $n!$ or $fact(n)$:

$$fact(n) = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1$$

- Also, by convention, $fact(0) = 1$.

Recursive definition of factorial

- Consider the definition of the **factorial** of a positive integer n, written either as $n!$ or $fact(n)$:

$$fact(n) = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1$$

- Also, by convention, $fact(0) = 1$.
- So, for example:

$$\begin{aligned} fact(1) &= 1 \\ fact(2) &= 2 \times 1 = 2 \\ fact(3) &= 3 \times 2 \times 1 = 6 \end{aligned}$$

Recursive definition of factorial

- Consider the definition of the **factorial** of a positive integer n, written either as $n!$ or $fact(n)$:

$$fact(n) = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1$$

- Also, by convention, $fact(0) = 1$.
- So, for example:

$$\begin{aligned} fact(1) &= 1 \\ fact(2) &= 2 \times 1 = 2 \\ fact(3) &= 3 \times 2 \times 1 = 6 \end{aligned}$$

- We note that the factorial of a number $n > 0$ can be defined in terms of the factorial of $n-1$, e.g., $fact(4) = 4 \times fact(3)$.
- In general, for $n > 0$, $fact(n) = n \times fact(n-1)$.

Recursive definition of factorial

- This leads to the following **recursive** function definition:

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

Recursive definition of factorial

- This leads to the following **recursive** function definition:

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

- Notice that this definition, although perfectly correct, will fail for (illegal) negative integers (since no guard will be true).

Recursive definition of factorial

- This leads to the following **recursive** function definition:

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

- Notice that this definition, although perfectly correct, will fail for (illegal) negative integers (since no guard will be true).
- We can add an `otherwise` guard to trigger a nice error message:

```
fact :: Int -> Int
fact n
    | n > 0     = n * fact (n - 1)
    | n == 0    = 1
    | otherwise = error "undefined for neg ints"
```

Calculation of an expression involving recursion

```
fact 2
```

Calculation of an expression involving recursion

```
fact 2
??  2 > 0                                 first guard
```

Calculation of an expression involving recursion

	`fact 2`	
??	`2 > 0`	first guard
??	$\rightsquigarrow$ `True`	def of >

Calculation of an expression involving recursion

fact 2	
?? 2 > 0	first guard
?? $\rightsquigarrow$ True	def of >
$\rightsquigarrow$ 2 * fact (2 - 1)	def of fact

Calculation of an expression involving recursion

fact 2	
?? 2 > 0	first guard
?? ⇝ True	def of >
⇝ 2 * fact (2 - 1)	def of fact
?? 2 - 1 > 0	first guard

Calculation of an expression involving recursion

`fact 2`	
`?? 2 > 0`	`first guard`
`?? ⇝ True`	`def of >`
`⇝ 2 * fact (2 - 1)`	`def of fact`
`?? 2 - 1 > 0`	`first guard`
`?? ⇝ 1 > 0`	`arithmetic`

Calculation of an expression involving recursion

```
fact 2
??  2 > 0                          first guard
??  ⇝ True                         def of >
⇝ 2 * fact (2 - 1)                 def of fact
??  2 - 1 > 0                      first guard
??  ⇝ 1 > 0                        arithmetic
??  ⇝ True                         def of >
```

Calculation of an expression involving recursion

```
fact 2
??  2 > 0                        first guard
??   ⇝ True                      def of >
⇝ 2 * fact (2 - 1)               def of fact
??  2 - 1 > 0                    first guard
??   ⇝ 1 > 0                     arithmetic
??   ⇝ True                      def of >
⇝ 2 * (1 * fact (1 - 1))         def of fact
```

Calculation of an expression involving recursion

```
fact 2
??  2 > 0                          first guard
??  ⇝ True                         def of >
⇝ 2 * fact (2 - 1)                 def of fact
??  2 - 1 > 0                      first guard
??  ⇝ 1 > 0                        arithmetic
??  ⇝ True                         def of >
⇝ 2 * (1 * fact (1 - 1))           def of fact
??  1 - 1 > 0                      first guard
```

Calculation of an expression involving recursion

`fact 2`	
`?? 2 > 0`	`first guard`
`?? ⇝ True`	`def of >`
`⇝ 2 * fact (2 - 1)`	`def of fact`
`?? 2 - 1 > 0`	`first guard`
`?? ⇝ 1 > 0`	`arithmetic`
`?? ⇝ True`	`def of >`
`⇝ 2 * (1 * fact (1 - 1))`	`def of fact`
`?? 1 - 1 > 0`	`first guard`
`?? ⇝ 0 > 0`	`arithmetic`

Calculation of an expression involving recursion

```
fact 2
??  2 > 0                          first guard
??  ⇝ True                         def of >
⇝ 2 * fact (2 - 1)                 def of fact
??  2 - 1 > 0                      first guard
??  ⇝ 1 > 0                        arithmetic
??  ⇝ True                         def of >
⇝ 2 * (1 * fact (1 - 1))           def of fact
??  1 - 1 > 0                      first guard
??  ⇝ 0 > 0                        arithmetic
??  ⇝ False                        def of >
```

Calculation of an expression involving recursion

```
fact 2
??  2 > 0                          first guard
??   ⇝ True                        def of >
⇝ 2 * fact (2 - 1)                 def of fact
??  2 - 1 > 0                      first guard
??   ⇝ 1 > 0                       arithmetic
??   ⇝ True                        def of >
⇝ 2 * (1 * fact (1 - 1))           def of fact
??  1 - 1 > 0                      first guard
??   ⇝ 0 > 0                       arithmetic
??   ⇝ False                       def of >
??  0 == 0                         second guard
```

Calculation of an expression involving recursion

```
fact 2
??  2 > 0                     first guard
??  ⇝ True                    def of >
⇝ 2 * fact (2 - 1)            def of fact
??  2 - 1 > 0                 first guard
??  ⇝ 1 > 0                   arithmetic
??  ⇝ True                    def of >
⇝ 2 * (1 * fact (1 - 1))      def of fact
??  1 - 1 > 0                 first guard
??  ⇝ 0 > 0                   arithmetic
??  ⇝ False                   def of >
??  0 == 0                    second guard
??  ⇝ True                    def of ==
```

Calculation of an expression involving recursion

fact 2	
?? 2 > 0	first guard
?? ⇝ True	def of >
⇝ 2 * fact (2 - 1)	def of fact
?? 2 - 1 > 0	first guard
?? ⇝ 1 > 0	arithmetic
?? ⇝ True	def of >
⇝ 2 * (1 * fact (1 - 1))	def of fact
?? 1 - 1 > 0	first guard
?? ⇝ 0 > 0	arithmetic
?? ⇝ False	def of >
?? 0 == 0	second guard
?? ⇝ True	def of ==
⇝ 2 * (1 * 1)	def of fact

Calculation of an expression involving recursion

```
fact 2
??  2 > 0                       first guard
??  ⇝ True                      def of >
⇝ 2 * fact (2 - 1)              def of fact
??  2 - 1 > 0                   first guard
??  ⇝ 1 > 0                     arithmetic
??  ⇝ True                      def of >
⇝ 2 * (1 * fact (1 - 1))        def of fact
??  1 - 1 > 0                   first guard
??  ⇝ 0 > 0                     arithmetic
??  ⇝ False                     def of >
??  0 == 0                      second guard
??  ⇝ True                      def of ==
⇝ 2 * (1 * 1)                   def of fact
⇝ 2 * 1                         arithmetic
```

Calculation of an expression involving recursion

fact 2	
?? 2 > 0	first guard
?? ⇝ True	def of >
⇝ 2 * fact (2 - 1)	def of fact
?? 2 - 1 > 0	first guard
?? ⇝ 1 > 0	arithmetic
?? ⇝ True	def of >
⇝ 2 * (1 * fact (1 - 1))	def of fact
?? 1 - 1 > 0	first guard
?? ⇝ 0 > 0	arithmetic
?? ⇝ False	def of >
?? 0 == 0	second guard
?? ⇝ True	def of ==
⇝ 2 * (1 * 1)	def of fact
⇝ 2 * 1	arithmetic
⇝ 2	arithmetic

Primitive versus general recursion

- The definition of *fact* is known as a **primitive** recursive definition; that is:
  - the **base case** considers the parameter value 0;
  - the **recursive case** considers how to get from value $n - 1$ to n.

Primitive versus general recursion

- The definition of *fact* is known as a **primitive** recursive definition; that is:
  - the **base case** considers the parameter value 0;
  - the **recursive case** considers how to get from value $n - 1$ to n.
- Here is another example of a primitive recursion definition (with two parameters) that defines multiplication in terms of addition:

```
mult :: Int -> Int -> Int
mult n m
    | n == 0        = 0
    | n > 0         = m + mult (n - 1) m
```

- (This definition only works for non-negative values of `n`.)

Primitive versus general recursion

- The definition of *fact* is known as a **primitive** recursive definition; that is:
  - the **base case** considers the parameter value 0;
  - the **recursive case** considers how to get from value $n - 1$ to n.
- Here is another example of a primitive recursion definition (with two parameters) that defines multiplication in terms of addition:

```
mult :: Int -> Int -> Int
mult n m
    | n == 0         = 0
    | n > 0          = m + mult (n - 1) m
```

- (This definition only works for non-negative values of `n`.)
- As an example of recursion that doesn't follow this pattern, consider an integer division function `divide` ...

General recursion

- Consider the following integer division:

```
divide 53 10 = 5
```

General recursion

- Consider the following integer division:

```
divide 53 10 = 5
```

- If we subtract the divisor (10) from the number being divided (53) we get a division where the result is one less:

```
divide 43 10 = 4
```

General recursion

- Consider the following integer division:

```
divide 53 10 = 5
```

- If we subtract the divisor (10) from the number being divided (53) we get a division where the result is one less:

```
divide 43 10 = 4
```

- If we continue, we'll get to a base case where the divided number is **smaller** than the divisor, and the result is 0:

```
divide 3 10 = 0
```

General recursion

- Consider the following integer division:

```
divide 53 10 = 5
```

- If we subtract the divisor (10) from the number being divided (53) we get a division where the result is one less:

```
divide 43 10 = 4
```

- If we continue, we'll get to a base case where the divided number is **smaller** than the divisor, and the result is 0:

```
divide 3 10 = 0
```

- This leads to the following recursive function definition:

```
divide :: Int -> Int -> Int
divide n m
    | n < m          = 0
    | otherwise      = 1 + divide (n - m) m
```