# Modern Programing Languages

## Introduction to Haskell (1)

Wamberto Vasconcelos
w.w.vasconcelos@abdn.ac.uk

2016–2017

# Plan of Lecture

1. Functional Programming in Haskell – Basics
2. Defining Recursive Functions
3. Using the Haskell System
4. Specifying Types of Functions

# Functional Programming in Haskell

- In Haskell, $f(x)$ is "`f x`".
- Haskell's main programming concept is function application:

```
sin (square z)     maximum (abs -5) (abs 3)
```

- Function application: higher precedence than anything else

```
1 + f x   means  1 + (f x)   not  (1 + f) x
f x - 3   means  (f x) - 3   not  f (x - 3)
```

- N.B.: Precedence even over operators such as "+" and "$*$"!
- Function application is left associative:

```
  f g x    means     (f g) x      not     f (g x)
f g h x    means   ((f g) h) x    not   f (g (h x))
```

# Variables in Haskell

- Similar to variables in mathematical equations
- Not a pointer to a memory cell!!
- Place-holder for unknown values
- Local to their particular function definition
- A variable receives a value when a function is applied to its arguments
- No other way to assign a value to a variable
- No destructive assignment: not possible to alter value of variable

# Defining Functions in Haskell

- Function definitions in “equational” style, e.g.

```
a = 42.4
f x = x + a
g x y = x
poly a b c x = (a * x ^ 2) + (b * x) + c
```

- Haskell requires us to name each function definition explicitly
- Variable: any unquoted string not used as a function name

# Using Functions in Haskell

- After functions are defined as previously, we can run them.
- Edit functions in file (use extension `.hs`, as in `myprog.hs`)
- Load file with definitions in Haskell interpreter

# Function Definition by Cases (1)

- Haskell allows the definition of functions using guards:

```
relative a b
  | a > b    = "higher"
  | a == b   = "equal"
  | a < b    = "lower"
```

- N.B.: Make sure conditions for each case are mutually exclusive.
- Compiler does not check conditions!
- To help us, Haskell provides the “`otherwise`” keyword:

```
relative a b
  | a > b       = "higher"
  | a == b      = "equal"
  | otherwise   = "lower"
```

- Haskell selects the first alternative that holds (top-to-bottom).

# Function Definition with Patterns (1)

- Important syntactic sugar: arguments as patterns:

```
cave_man 0 = "none"
cave_man 1 = "one"
cave_man n = "lots"
```

- Equivalent function with guards:

```
cave_man n
  | n == 0      = "none"
  | n == 1      = "one"
  | otherwise   = "many"
```

- It is good practice to use the patterned form of function definition, where possible.

# Function Definition with Patterns (2)

- Ordering of cases is essential when patterns are not mutually exclusive!
- Haskell selects the first case whose pattern matches the value of the arguments:

right:
```
fn a 3 = True
fn a b = False
```
wrong:
```
fn a b = False
fn a 3 = True
```

- Rule of thumb: most specific cases first!
- A useful pattern for functions over natural numbers is "`(n+`$k$`)`"
  - `n` is a variable and $k$ is a positive number like 1 (not a variable):
  - Example: `predecessor (n+1) = n`
  - A given `x` matches `(n+`$k$`)` iff `x` is an integer greater than $k$
  - `n` takes on the value `x`$-k$: `predecessor 5` $\rightsquigarrow$ 4

# Defining Local Functions in Haskell

- Define a function local to another function's definition:

```
calc x y
  | x > y     = 2 * a/b
  | x == y    = 0
  | otherwise = 2 * b/a
    where a = x + y
          b = x - y
sum_cubes x y z = vol x + vol y + vol z
                  where vol c = c * c * c
sum_cylinders x y z = vol x + vol y + vol z
                      where vol c = pi * c * c
```

# The Offside Rule for Function Definitions

- Haskell allows the programmer a lot of freedom in laying out programs.
- Layout: for the benefit of humans, to improve visualisation.
- However, there are situations when layout is significant: offside rule

  > "*the right hand side of an equation is terminated by any text appearing to the left of its first character or at the same indentation (offside)*"

- Example:

```
expensive p = (p > 100)
vat p = 17.5 *
p/100
```

- Function `expensive` ends just before the "`v`" of `vat`
- The definition of `vat` is offside and will cause an error!

# Recommended Layout

- A recommended layout is:

```
f p₁ p₂ ...  pₖ
  | cond₁      = res₁
  | cond₂      = res₂
  ...
  | otherwise = resₘ
    where
    v₁ a₁ a₂ ...  aₙ = sol₁
    v₂ = sol₂
    ...
```

# Defining Recursive Functions (1)

- Function to compute $x^n$ for any $x$ and any $n \geq 0$:
  $$x^0 = 1 \text{ and } x^{n+1} = x \times x^n$$
- Haskell version follows this definition closely:
```
power x 0     = 1                   -- base case
power x (n+1) = x * (power x n)     -- recursive case
```
- Recursive: defined in terms of itself.
- Function defines a result (i.e. has a normal form) because:
  - there is one equation which is not recursive and
  - recursive call on the RHS is a smaller instance of the problem
- Incorrect definition (will loop forever):
```
power x n = x * (power x (n+1))
```

# Defining Recursive Functions (2)

- It is useful to start by defining base case(s).
- For recursion over numbers, base case(s) usually 0 or 1.
- When writing recursive case(s):
  - Think of result in terms of arbitrary variable $n$
  - Imagine you have already defined the function you require
  - Use it to describe how the result for larger instance $n + 1$ relates to result of smaller instances $n$.

# Defining Recursive Functions (3)

- Factorial of $n \in \mathbb{N}, n > 0$: $n! = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1$
- Base case: $1! = 1$
- However, $0!$ is also part of the mathematical definition of factorial, the base case is actually:

```
factorial 0 = 1
```

- The case for $1!$ is subsumed by the more general recursive case.
- To define the recursive case, we must describe how $(n+1)!$ relates to $n!$:

```
factorial (n+1) = (n+1) * (factorial n)
```

# Defining Recursive Functions (4)

- Previous example highlights an advantage of recursive specifications.
- Typical imperative solution:

```
function fac(n) is
   begin
      f := 1;
      for i := n step -1 until 1 do f := f * i;
      return f
   end;
```

- What is the value of this function when `n` is `0`?
- We have to "run" program in our heads!
- In a recursive definition the base case(s) are explicit.

# Using the Haskell System

- Haskell interpreter runs in read, evaluate, print loop.
- Function definitions are introduced by creating a script file.
- Convention: script files with extension `.hs`
- Haskell is a strongly-typed language: type errors in definitions and expressions are detected by the interpreter.
- For instance, given the definition `f x = x ^ 2`, if we try to apply the function to a non-numeric argument, we get:

```
Main> f 'a'
ERROR - Type error in application
*** Expression :  f 'a'
*** Term :  'a'
*** Type :  Char
*** Does not match :  Double
```

# Specifying Types of Functions

- Type of functions declared with the "::" operator:

```
vat ::  Float
vat = 17.5

expensive ::  Float -> Bool
expensive p = (p > 100)

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

- Type declarations are optional: Haskell can infer types.
- It is a good practice to specify types for all but the simplest functions.
- Types provide an extra check for your definition!

# Base (Built-in) Types in Haskell

- Haskell provides four base types: `Int`, `Float`, `Bool` and `Char`.
- `Int`: integers, *e.g.* `0`, `-34`.
- `Float`: floating-point numbers, *e.g.* `0.0`, `-34.567`.
- `Bool`: the boolean values `True` and `False`
- `Char`: the individual characters, *e.g.* `'a'`, `'b'`, etc.
  - Single quotes are needed to distinguish characters from variables `a`, `b`.
  - N.B.: The string `'ab'` is not of type `Char`!! (more on this later...)
- Function types are declared with the "`->`" operator:

```
code ::  Char -> Int
```

- More complex types such as tuples, lists, algebraic and abstract types can also be defined.

# Tuples

- A tuple collects together a fixed number of values of different types.
- Examples of tuples and their types:

```
           (25,12,1996)  ::  (Int,Int,Int)
             ('a',True)  ::  (Char,Bool)
        ('x',(1,1,2001)) ::  (Char,(Int,Int,Int))
(current_year,next_year) ::  (Int,(Int -> Int))
```

- Fields of tuples accessed by pattern matching of definitions:

```
first (a, b) = a
second (a, b) = b
```