UNIVERSITÀ DEGLI STUDI DI FERRARA
- EX LABORE FRUCTUS -

Capitolo 4.5
Haskell
Guarded equations
let e where

Corso di Laurea Magistrale in Ingegneria Informatica e dell'Automazione

Anno accademico 2019/2020

Prof. MARCO GAVANELLI

Part of these slides were adapted from the material of the book Graham Hutton, Programming in Haskell, Cambridge University Press, 2nd edition, 2016

Guarded Equations

As an alternative to conditionals, functions can also be defined using guarded equations

```
abs n | n >= 0     = n
      | otherwise = -n
```

$$\text{abs } n = \begin{cases} n \geq 0 & = n \\ \text{otherwise} & = -n \end{cases}$$

Guards

- Guards are clean if statements.
- Just like with pattern matching, order matters.
- A guard is introduced by the | symbol.
- And it's followed by a Bool expression.
- Then followed by the function body

```
guessMyNumber x
    | x > 27    = "Too high!"
    | x < 27    = "Too low!"
    | otherwise = "Correct!"
```


Guarded equations can be used to make definitions involving multiple conditions easier to read:

```
signum n | n < 0     = -1
         | n == 0    = 0
         | otherwise = 1
```

`otherwise` is just a fancy word for `True`

Variables

- These are not like your typical Java variables
- In Java or C++, you can redefine variables:

```
x = 1;
...
x = 2;
```

- Mathematically, this makes no sense.
- It implies 1=2 Preposterous!

Variables

- Haskell variables are immutable.
- Once defined, they can't change.
- They can be used with the `let` keyword.

```
slope (x1,y1) (x2,y2) = let dy = y2-y1
                            dx = x2-x1
                        in dy/dx
```

Or with the `where` keyword.

```
slope (x1,y1) (x2,y2) = dy/dx
                        where dy = y2-y1
                              dx = x2-x1
```

where

- **where** bindings can span to multiple guards

```
bmiTell weight height
| bmi <= 18.5 = "underweight"
| bmi <= 25.0 = "normal"
| bmi <= 30.0 = "fat"
| otherwise   = "whale"
where bmi = weight / height ^ 2
```


let

- **let** bindings are expressions themselves

```
> 4 * (let a = 9 in a + 1) + 2
42
```

- They can also be used to introduce functions in a local scope:

```
>[let square x = x * x in (square 5, square 3)]
[(25,9)]
```


The Layout Rule

In a sequence of definitions, each definition must begin in precisely the same column:

```
a = 10
b = 20
c = 30
```

```
a = 10
  b = 20
c = 30
```

```
  a = 10
b = 20
  c = 30
```

The layout rule avoids the need for explicit syntax to indicate the grouping of definitions.

```
a = b + c
    where
      b = 1
      c = 2
d = a * 2
```

means

```
{a = b + c
     where
       {b = 1;
        c = 2}
 d = a * 2}
```

implicit grouping

explicit grouping

Don't use tab. Use spaces' '.

Exercises

Fix the syntax errors in the program below, and test your solution using GHCi.

```
N = a 'div' length xs
    where
       a = 10
      xs = [1,2,3,4,5]
```


Exercises

(1) Consider a function **safetail** that behaves in the same way as tail, except that **safetail** maps the empty list to the empty list, whereas tail gives an error in this case. Define **safetail** using:

(a) a conditional expression;
(b) guarded equations;
(c) pattern matching.

Hint: the library function **null::[a] → Bool** can be used to test if a list is empty.

(2) Give three possible definitions for the logical or operator `(||)` using pattern matching.

(3) Redefine `(&&)` using conditionals rather than patterns:

Exercises

- Write a Caesar Cipher function called cipher

```
Prelude> cipher "hello" 13
"uryyb"
```

- Suggestion:
  - **`pred`** and **`succ`** can be used to get the previous and following character