Haskell – Definire funzioni

Conditional Expressions

As in most programming languages, functions can be defined using <u>conditional expressions</u>.

```
abs   :: Int → Int
abs n = if n ≥ 0 then n else -n
```

abs takes an integer *n* and returns *n* if it is non-negative and -*n* otherwise.

Conditional expressions can be nested:

```
signum  :: Int → Int
signum n = if n < 0 then -1 else
              if n == 0 then 0 else 1
```

Note:

In Haskell, conditional expressions must <u>always</u> have an else branch, which avoids any possible ambiguity problems with nested conditionals.

Pattern Matching

Many functions have a particularly clear definition using <u>pattern matching</u> on their arguments.

```
not       :: Bool → Bool
not False = True
not True  = False
```

`not` maps **`False`** to **`True`**, and **`True`** to **`False`**.

Functions can often be defined in many different ways using pattern matching. For example

```
(&&)            :: Bool → Bool → Bool
True  && True  = True
True  && False = False
False && True  = False
False && False = False
```

can be defined more compactly by

```
True && True = True
_    && _    = False
```

4

However, the following definition is more efficient, because it avoids evaluating the second argument if the first argument is False:

```
True  && b = b
False && _ = False
```

Note:

The underscore symbol _ is a wildcard pattern that matches any argument value.

5

Patterns are matched in order. For example, the following definition always returns False:

```
_    && _    = False
True && True = True
```

Patterns may not repeat variables. For example, the following definition gives an error:

```
b && b = b
_ && _ = False
```

Bi-directional pattern matching is called '*unification*'; it is used in Prolog, not in Haskell

Numeric patterns

- Pattern-matching can also be used with functions with numbers

```
fact 0 = 1
fact n = n*fact (n-1)
```

7

Non-exhaustive patterns

```
charName 'a' = "Albert"
charName 'b' = "Broseph"
charName 'c' = "Cecil"
```

- If no pattern matches, an error is raised

```
ghci> charName 'h'
"*** Exception: Non-exhaustive
patterns in function charName
```

8

List Patterns

Internally, every non-empty list is constructed by repeated use of an operator **(:)** called "cons" that adds an element to the start of a list.

```
[1,2,3,4]
```

Means 1:(2:(3:(4:[]))).

9

Functions on lists can be defined using x:xs patterns.

```
head        :: [a] → a
head (x:_)   = x

tail        :: [a] → [a]
tail (_:xs) = xs
```

head and tail map any non-empty list to its first and remaining elements.

10

Note:

x:xs patterns only match non-empty lists:

```
> head []
ERROR
```

x:xs patterns must be parenthesised, because application has priority over **(:)**. For example, the following definition gives an error:

```
head x:_ = x
```

11

- The pattern **`x:y:xs`** matches all lists containing at least 2 elements (≥ 2 elements)

- The pattern **`a:b:c:[]`** matches all lists of exactly three elements (= 3 elements). It can also be written as [a,b,c] (syntactic sugar)

Pattern matching on tuples

- Sum of two vectors:

```
addVectors (x1, y1) (x2, y2) =
  (x1 + x2, y1 + y2)
```

- extract the elements from a 3-tuple

```
first (x, _, _) = x
second (_, y, _) = y
third (_, _, z) = z
```


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]
```

Lambda Expressions

Functions can be constructed without naming the functions by using lambda expressions.

```
λx → x + x
```

the nameless function that takes a number x and returns the result x + x.

24

Note:

- The symbol λ is the Greek letter lambda, and is typed at the keyboard as a backslash \.
- In mathematics, nameless functions are usually denoted using the ↦ symbol, as in $x \mapsto x + x$.
- In Haskell, the use of the λ symbol for nameless functions comes from the lambda calculus, the theory of functions on which Haskell is based.

25

Why Are Lambda's Useful?

Lambda expressions can be used to give a formal meaning to functions defined using currying.

For example:

```
add x y = x + y
```

means

```
add = λx → (λy → x + y)
```

26

Lambda expressions are also useful when defining functions that return functions as results.

For example:

```
const     :: a → b → a
const x _ = x
```

is more naturally defined by

```
const   :: a → (b → a)
const x = λ_ → x
```

27

Lambda expressions can be used to avoid naming functions that are only referenced once.

For example:

```
odds n = map f [0..n-1]
         where
            f x = x*2 + 1
```

can be simplified to

```
odds n = map (λx → x*2 + 1) [0..n-1]
```


Sections

An operator written between its two arguments can be converted into a curried function written before its two arguments by using parentheses.

For example:

```
> 1+2
3

> (+) 1 2
3
```


This convention also allows one of the arguments of the operator to be included in the parentheses.

For example:

```
> (1+) 2
3

> (+2) 1
3
```

In general, if $\oplus$ is an operator then functions of the form $(\oplus)$, $(x\oplus)$ and $(\oplus y)$ are called sections.

Why Are Sections Useful?

Useful functions can sometimes be constructed in a simple way using sections. For example:

- `(1+)` - successor function
- `(1/)` - reciprocation function
- `(*2)` - doubling function
- `(/2)` - halving function

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

These slides were adapted from the material of the book

Graham Hutton, Programming in Haskell, Cambridge University Press, 2nd edition, 2016