Haskell – Definire funzioni

Conditional expressions can be nested:

```
\begin{array}{lll} \mbox{signum} & :: \mbox{Int} \rightarrow \mbox{Int} \\ \mbox{signum} \ n = \mbox{if} \ n < 0 \ \mbox{then} \ -1 \ \mbox{else} \\ \mbox{if} \ n == 0 \ \mbox{then} \ 0 \ \mbox{else} \ 1 \end{array}
```

Note:

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

Conditional Expressions

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

```
abs :: Int \rightarrow Int abs n = if n \ge 0 then n else -n
```

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

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 \rightarrow Bool \rightarrow Bool True && True = True
True && False = False
False && True = False
False && False = False
```

can be defined more compactly by

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

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

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

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

Bi-directional pattern matching is called 'unification'; it is used in Prolog, not in Haskell 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 <u>wildcard</u> pattern that matches any argument value.

5

Numeric patterns

 Pattern-matching can also be used with functions with numbers

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

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

Functions on lists can be defined using $\underline{x:xs}$ patterns.

```
head :: [a] \rightarrow a
head (x:_) = x

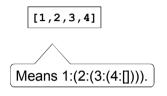
tail :: [a] \rightarrow [a]
tail (_:xs) = xs
```

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

10

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.



9

Note:

x:xs patterns only match non-empty lists:

> head [] ERROR

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

$$head x: = x$$

- 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)

12

Guarded Equations

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

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

14

Pattern matching on tuples

Sum of two vectors:

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

• extract the elements from a 3-tuple

13

Guards

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

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

16

Variables

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

Or with the where keyword.

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

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!

17

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</pre>
```

let

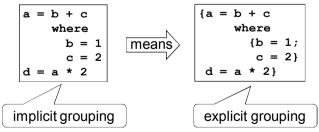
• let bindings are expressions themselves

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

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

20

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



Don't use tab. Use spaces ' '.

22

The Layout Rule

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

$$a = 10$$
 $a = 10$ $a = 10$
 $b = 20$ $b = 20$
 $c = 30$ $c = 30$ $c = 30$







Exercises

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

Lambda Expressions

Functions can be constructed without naming the functions by using <u>lambda expressions</u>.

$$\lambda x \rightarrow x + x$$

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

24

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

$$| add = \lambda x \rightarrow (\lambda y \rightarrow x + y)$$

26

Note:

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

25

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

For example:

const ::
$$a \rightarrow b \rightarrow a$$
 const $x = x$

is more naturally defined by

const ::
$$a \rightarrow (b \rightarrow a)$$

const $x = \lambda \rightarrow x$

Lambda expressions can be used to avoid naming functions that are only <u>referenced once</u>.

For example:

can be simplified to

odds n = map
$$(\lambda x \rightarrow x*2 + 1)$$
 [0..n-1]

28

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

For example:

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

30

Sections

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

For example:

> 1+2 3 > (+) 1 2 3

29

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 $null::[a] \rightarrow Bool$ can be used to test if a list is empty.

32

32

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

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

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

33

These slides were adapted from the material of the book

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

