Haskell

Types and Classes

What is a Type?

A <u>type</u> is a name for a collection of related values. For example, in Haskell the basic type

```
Bool
```

contains the two logical values:

False **True**

```
> (True && False) || False
False
```


Type Errors

Applying a function to one or more arguments of the wrong type is called a <u>type error</u>.

```
> 1 + False
ERROR
```

```
> True && 1
ERROR
```

1 is a number and False is a logical value, but + requires two numbers.


```
> 30 < 31
True
```

```
> 30 == 31
False
```

```
> 30 /= 31
True
```

$\neq$

If then else

- if then else is also a function

```
fact n = if n==0 then 1 else n*fact (n-1)
```

```
> 3 * if 2>3 then 5 else 3
9
```

```
> 3 * if 2<3 then 5
ERROR!
```

Since it is a function it **must always** return a value!
The **else** is necessary!

Types in Haskell

- If evaluating an expression **e** would produce a value of type **t**, then **e** has type **t**, written

```
e :: t
```

- Every well formed expression has a type, which can be automatically calculated at compile time using a process called type inference.

- All type errors are found at compile time, which makes programs safer and faster by removing the need for type checks at run time.

- In GHCi, the **:type** command calculates the type of an expression, without evaluating it:

```
> not False
True

> :type not False
not False :: Bool
```


Basic Types

Haskell has a number of basic types, including:

`Bool`	- logical values
`Char`	- single characters
`String`	- strings of characters
`Int`	- fixed-precision integers
`Integer`	- arbitrary-precision integers
`Float`	- floating-point numbers

Everything has a Type

- Haskell secretly infers that True is a Bool.

```
Prelude> :type True
True :: Bool
```

You can also explicitly use a type.

```
Prelude> 3 :: Int
3

Prelude> 3 :: Double
3.0
```


Lists

- The most common data type in Haskell

[☺, ☹, ☺]

- elements are comma-separated
- surrounded by square brackets [...]
- an empty list is simply `[]`

```
> [3,1,5,3]
[3,1,5,3]

> ["list","of","strings"]
["list","of","strings"]
```


List Types

A list is sequence of values of the same type:

```
[False,True,False] :: [Bool]

['a','b','c','d']  :: [Char]
```

```
> [1,2,3,"a","bb","ccc"]
ERROR!
```

In general:

`[t]` is the type of lists with elements of type `t`.

Note:

- The type of a list says nothing about its length:

```
[False,True]        :: [Bool]

[False,True,False] :: [Bool]
```

- The type of the elements is unrestricted. For example, we can have lists of lists:

```
[['a'],['b','c']] :: [[Char]]
```

Enumeration

- Start at 1, end at 10

```
> [1..10]
[1,2,3,4,5,6,7,8,9,10]
```

- Start at 1, count up by 0.25, end at 2

```
> [1, 1.25 .. 3.0]
[1.0,1.25,1.5,1.75,2.0]
```

- Count down

```
>[10,9..0]
[10,9,8,7,6,5,4,3,2,1,0]
```


Enumeration

- Also in functions

```
zeroto n   = [0..n]
```

```
> zeroto 6
[0,1,2,3,4,5,6]
```


List operators

Concatenation (++)

```
> [1,2,3] ++ [4,5,6]
[1,2,3,4,5,6]
```

```
> (++) [1,2,3] [4,5,6]
[1,2,3,4,5,6]
```

Construct (:)

```
> 0 : [1,2,3]
[0,1,2,3]
```

```
> (:) 0 [1,2,3]
[0,1,2,3]
```

Most Efficient

In fact, a list is formally defined like this

```
> [1,2,3] == (:) 1 ((:) 2 ((:) 3 []))
True
```


String

A String is just a list of characters.

```
> "wahoo" == ['w', 'a', 'h', 'o', 'o']
True
```

- So (++) and (:) work on strings too.

Esercizi

- Scrivere una funzione `isPositive` che restituisce True se tutti gli elementi della lista sono positivi

```
> isPositive [3,2,4]
True
```

- Scrivere una funzione `elemento` che, dati una lista *xs* e un intero *n*, fornisce l'elemento *n* della lista (partendo da 1) (senza usare `!!`)

```
> elemento 2 [3,2,4]
2
```

- Scrivere una funzione `inverti` che, data una lista, fornisce la lista invertita

```
> inverti "abc"
"cba"
```

16

True or False?

```
> "" == []
```

```
> 'a':"bc" == ['a', 'b', 'c']
```

```
Prelude> 6:"789" == "6789"
```

17

Tuple Types

A <u>tuple</u> is a sequence of values of <u>different</u> types:

```
(False,True)     :: (Bool,Bool)

(False,'a',True) :: (Bool,Char,Bool)
```

In general:

(t_1,t_2,…,t_n) is the type of *n*-tuples whose *i*-th components have type t_i for any *i* in 1…*n*.

18

Note:

- The type of a tuple encodes its size:

```
(False,True)       :: (Bool,Bool)

(False,True,False) :: (Bool,Bool,Bool)
```

- The type of the components is unrestricted:

```
('a',(False,'b')) :: (Char,(Bool,Char))

(True,['a','b'])  :: (Bool,[Char])
```

19

Accessing Tuple Elements

`fst` retrieves the first element

```
Prelude> fst (1,2)
1
```

`snd` retrieves the second element

```
Prelude> snd (1,2)
2
```

Only for 2-tuples! Work only for tuples of exactly 2 elements

Function Types

A function is a mapping from values of one type to values of another type:

```
not  :: Bool → Bool

even :: Int → Bool
```

In general:

t1 → **t2** is the type of functions that map values of type **t1** to values to type **t2**.

Function types

```
Prelude> head [1,2,3,4]
1

Prelude> :type head
head :: [a] -> a
```

head has the type List of **a**'s to just **a**

```
Prelude> fst ("left", "right")
"left"

Prelude> :type fst
fst :: (a, b) -> a
```

fst has the type tuple of **a** and **b** to just **a**

Note:

The arrow → is typed at the keyboard as **->**.

The argument and result types are unrestricted. For example, functions with multiple arguments or results are possible using lists or tuples:

```
add       :: (Int,Int) -> Int
add (x,y)  = x+y

zeroto    :: Int -> [Int]
zeroto n   = [0..n]
```

Esercizio

- Dire quali sono i tipi delle seguenti funzioni:

```
fac n = product [1..n]
```

```
(&&)
```

```
Prelude> 0:[1,2,3]
[0,1,2,3]

Prelude> :type (:)
```

24

Curried Functions

Functions with multiple arguments are also possible by returning functions as results:

```
add'     :: Int → (Int → Int)
add' x y = x+y
```

add' takes an integer **x** and returns a function **add' x**. In turn, this function takes an integer **y** and returns the result **x+y**.

25

Note:

add and **add'** produce the same final result, but **add** takes its two arguments at the same time, whereas **add'** takes them one at a time:

```
add  :: (Int,Int) → Int

add' :: Int → (Int → Int)
```

Functions that take their arguments one at a time are called curried functions, celebrating the work of H.B. Curry on such functions.

27

Functions with more than two arguments can be curried by returning nested functions:

```
mult       :: Int → (Int → (Int → Int))
mult x y z = x*y*z
```

mult takes an integer x and returns a function mult x, which in turn takes an integer y and returns a function mult x y, which finally takes an integer z and returns the result x*y*z.

28

Why is Curried Useful?

Curried functions are more flexible than functions on tuples, because useful functions can often be made by <u>partially applying</u> a curried function.

For example:

```
add' 1 :: Int → Int

take 5 :: [Int] → [Int]

drop 5 :: [Int] → [Int]
```


Currying Conventions

To avoid excess parentheses when using curried functions, two simple conventions are adopted:

- The arrow → associates to the <u>right</u>.

```
Int → Int → Int → Int
```

Means Int → (Int → (Int → Int)).

As a consequence, it is then natural for function application to associate to the <u>left</u>.

```
mult x y z
```

Means `((mult x) y) z`

Unless tupling is explicitly required, all functions in Haskell are normally defined in curried form.

Exercise

- Write the function **`gravity`** that, given a mass m_1, a distance *d,* and a mass m_2, computes the gravitational force

  $G=6.7\ 10^{-11}\ m^3/kg/s^2$

- Write the function **`earthGravity`** that, given a mass and a distance, computes the gravitational force of the Earth on the mass

  Earth mass = $5.96\ 10^{24}\ kg$

- Write a function **`earthGravitySurface`** that computes the weight of a mass on the surface of the Earth

  Earth radius = $6.37\ 10^{6}\ m$

Exercise

- Function **`logBase`** *b x* computes the logarithm in base *b* of *x*
- Write function **`log2`** that computes the logarithm in base 2 of a number

33

Polymorphic Functions

A function is called polymorphic ("of many forms") if its type contains one or more type variables.

```
length :: [a] → Int
```

For any type **a**, length takes a list of values of type **a** and returns an integer.

34

Note:

Type variables can be instantiated to different types in different circumstances:

```
> length [False,True]
2

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

a = Bool

a = Int

Type variables must begin with a lower-case letter, and are usually named **a, b, c**, etc.

35

Many of the functions defined in the standard prelude are polymorphic. For example:

```
fst  :: (a,b) → a

head :: [a] → a

take :: Int → [a] → [a]

zip  :: [a] → [b] → [(a,b)]

id   :: a → a
```

36

Overloaded Functions

A polymorphic function is called overloaded if its type contains one or more class constraints.

```
(+) :: Num a ⇒ a -> a -> a
```

For any numeric type **a**, **(+)** takes two values of type **a** and returns a value of type **a**.

Note:

Constrained type variables can be instantiated to any types that satisfy the constraints:

```
> 1 + 2
3

> 1.0 + 2.0
3.0

> 'a' + 'b'
ERROR
```

a = Int

a = Float

Char is not a numeric type

Haskell has a number of type classes, including:

- **Num** - Numeric types
- **Eq** - Equality types
- **Ord** - Ordered types

For example:

```
(+)  :: Num a ⇒ a → a → a
(==) :: Eq a  ⇒ a → a → Bool
(<)  :: Ord a ⇒ a → a → Bool
```


Hints and Tips

- When defining a new function in Haskell, it is useful to begin by writing down its type;
- Within a script, it is good practice to state the type of every new function defined;
- When stating the types of polymorphic functions that use numbers, equality or orderings, take care to include the necessary class constraints.

Exercises

(1) What are the types of the following values?

```
[’a’,’b’,’c’]

(’a’,’b’,’c’)

[(False,’0’),(True,’1’)]

([False,True],[’0’,’1’])

[tail,init,reverse]
```


(2) What are the types of the following functions?

```
second xs     = head (tail xs)

swap (x,y)    = (y,x)

pair x y      = (x,y)

double x      = x*2

palindrome xs = reverse xs == xs

twice f x     = f (f x)
```

(3) Check your answers using GHCi.

These slides were adapted from the material of the book

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