3 - Types and Classes

What is a Type?

A **type** is a name for a collection of related values.

`Bool`

contains the two logical values:

`False` `True`

Type Errors

Applying a function to one or more arguments of the wrong type is called a *type error*.

```
> 1 + False
error ...
```

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

Types in Haskell

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

```
e :: t
```

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

$$\frac{f :: A \to B \qquad e :: A}{f\ e :: B}$$

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

- The `:type` command calculates the type of an expression, without evaluating it:

```
> :type False
False :: Bool

> 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 / Integer` - fixed/arbitrary-precision integers

`Float / Double` - single/double-precision floating-point numbers

List Types

A **list** is sequence of values of the **same** type:

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

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

[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]]
```

Tuple Types

A **tuple** is a sequence of values of **different** types:

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

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

`(t1,t2,…,tn)` is the type of n-tuples whose i-th components have type `ti` for any i in 1…n.

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])
```

Function Types

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

```
not :: Bool → Bool

even :: Int → Bool
```

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

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]
```

Curried Functions

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

```
addc :: Int → (Int → Int)
addc x y = x+y
```

`addc` takes an integer `x` and returns a function `addc x`. In turn, this function takes an integer `y` and returns the result `x+y`.

Note:

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

```
add :: (Int,Int) → Int

addc :: Int → (Int → Int)
```

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

- 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.

Why is Currying Useful?

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

```
inc = addc 1
inc :: Int → Int

rest = drop 1
rest :: [Int] → [Int]
```

Currying Conventions

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

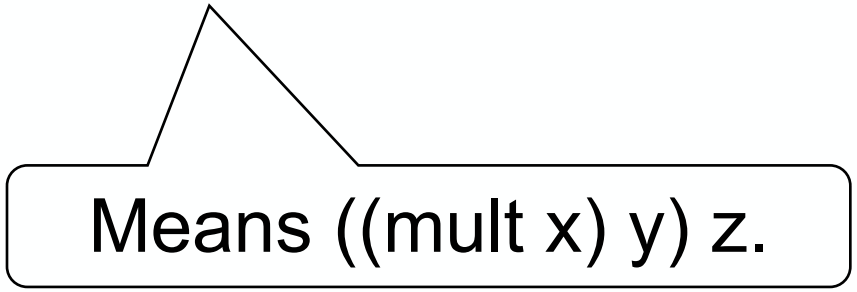
- The arrow → associates to the **right**.

```
Int → Int → Int → Int
```

Means Int → (Int → (Int → Int)).

- As a consequence, it is then natural for function application to associate to the **left**.

```
mult x y z
```



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

Polymorphic Functions

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

```
length :: [a] → Int
```

```
> length [1,2,3,4]
4
> length ['a','b','c','d']
4
```

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

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.

- Many of the functions defined in the standard prelude are polymorphic.

```
fst :: (a,b) → a

head :: [a] → a

take :: Int → [a] → [a]

zip :: [a] → [b] → [(a,b)]

id :: a → a
```

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

- Numbers are overloaded

```
> :t 3
3 :: Num p => p
```

- Haskell has a number of type classes, including:

`Num` - Numeric types

`Eq` - Equality types

`Ord` - Ordered types

`Show` - Showable types

`Read` - Readable types

`Integral` - Integral types

`Fractional` - Fractional types

`Num` - Numeric types

```
(+)    :: a → a → a    a → a

(-)    :: a → a → a    a → a

(*)    :: a → a → a    a → a

negate  :: a → a        → a

abs    :: a → a        a

signum  :: a → a        →
```

Instances

```
Int

Integer

Float

Double
```

`Eq` - Equality types

```
(==) :: a → a → Bool

(/=) :: a → a → Bool
```

`Ord` - Ordered types

```
(<)  :: a → a → Bool

(<=)  :: a → a → Bool

(>)  :: a → a → Bool

(>=)  :: a → a → Bool

min  :: a → a → a

max  :: a → a → a
```

Instances

```
Bool

Char

Int

Integer

Float

Double

lists

tuples
```

`Show` - Showable types
(whose values can be converted into Strings)

```
show :: a → String
```

```
> show [1..3]
“[1,2,3]"
```

`Read` - Readable types

```
read :: String → a
```

```
> read “[1,2,3]” :: [Int]
[1,2,3]
> read "[1,2,3]" :: [Float]
[1.0,2.0,3.0]
```

Instances

- Bool
- Char
- Int
- Integer
- Float
- Double
- lists
- tuples

`Integral` - Integral types

```
div :: a → a → a

mod :: a → a → a
```

Instances

```
Int

Integer
```

`Fractional` - Fractional types

```
(/) :: a → a → a

recip :: a → a → a
```

```
Float

Double
```

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

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]
```

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)
```