Types and Classes

What is a Type?

A type is a collection of related values.

`Bool`

The logical values False and True.

`Bool → Bool`

All functions that map a logical value to a logical value.

Types in Haskell

We use the notation e :: T to mean that evaluating the expression e will produce a value of type T.

```
False :: Bool

not :: Bool → Bool

not False :: Bool

False && True :: Bool
```

Note:

- Every expression must have a valid type, which is calculated prior to evaluating the expression by a process called type inference;

- Haskell programs are type safe, because type errors can never occur during evaluation;

- Type inference detects a very large class of programming errors, and is one of the most powerful and useful features of Haskell.

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

List Types

A list is sequence of values of the same type:

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

[’a’,’b’,’c’,’d’]  :: [Char]
```

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]]
```

Tuple Types

A tuple is a sequence of values of different types:

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

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

In general:

(T1,T2,…,Tn) is the type of n-tuples whose ith components have type Ti for any i in 1…n.

Note:

- The type of a tuple encodes its arity:

```
(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

isDigit :: Char → Bool
```

In general:

T1 → T2 is the type of functions that map arguments of type T1 to results of type T2.

Note:

- 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]
```

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

```
[’a’,’b’,’c’]

(’a’,’b’,’c’)

[(False,’0’),(True,’1’)]

[isDigit,isLower,isUpper]
```

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

- 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.

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

Curry 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
```

Means ((mult x) y) z.

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

Polymorphic Types

The function length calculates the length of any list, irrespective of the type of its elements.

```
> length [1,3,5,7]
  4

> length ["Yes","No"]
  2

> length [isDigit,isLower,isUpper]
  3
```

This idea is made precise in the type for length by the inclusion of a type variable:

```
length :: [a] → Int
```

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

A type with variables is called polymorphic

- 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)]
```

Overloaded Types

The arithmetic operator + calculates the sum of any two numbers of the same numeric type.

For example:

```
> 1+2
  3

> 1.1 + 2.2
  3.3
```

This idea is made precise in the type for + by the inclusion of a class constraint:

```
(+) :: Num a ⇒ a → a → a
```

For any type a in the class Num of numeric types, + takes two values of type a and returns another.

A type with constraints is called overloaded.

Classes in Haskell

A class is a collection of types that support certain operations, called the methods of the class.

Eq

Types whose values can be compared for equality and difference using

```
(==) :: a → a → Bool
(/=)  :: a → a → Bool
```

Haskell has a number of basic classes, including:

`Eq`	- Equality types
`Ord`	- Ordered types
`Show`	- Showable types
`Read`	- Readable types
`Num`	- Numeric types

Example methods:

```
(==) :: Eq a   ⇒ a → a → Bool

(<) :: Ord a  ⇒ a → a → Bool

show :: Show a ⇒ a → String

read :: Read a ⇒ String → a

(*)  :: Num a  ⇒ a → a → a
```

(1) 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)
```