CS1JC3-Sept20-22

CS 1JC3

What Is A Type?

- A type is a name for a collection of related values. For example, in Haskell the basic type

  `Bool`

- contains the two logical values

  `False        True`

Type Errors

- Haskell is strictly typed (every variable has a type that cannot change), although it is not necessary to specify this type as ghci will infer it from context.
- Applying a function to one or more arguments of the wrong type is called a type error

```
Prelude> 1 + False
       Error
```

Type Errors

- Haskell is strictly typed (every variable has a type that cannot change), although it is not necessary to specify this type as ghci will infer it from context.
- Applying a function to one or more arguments of the wrong type is called a type error

```
Prelude> 1 + False
       Error
```

- The (+) function does not take an argument of type Bool, and so an error will be returned

Basic Types

Haskell has a number of basic types, including

- Bool → True, False
- Char → 'a' ,'b','1','2','*','!'...
- String → "Name","This is a Sentence" ...
- Int → 1,2,3,25,40050 ... (32 bit or 64 bit, varies by system)
- Integer → Same as Integer but Unlimited Range
- Float → 1.0, 2.555, −3.456 (32 bit)
- Double → Same as Float but 64 bit

Note: Try entering :type ['a','b','c'] into ghci

Types in Haskell

- For every expression e, it has a type t, written as

```
e :: t
```

Types in Haskell

- For every expression e, it has a type t, written as

```
e :: t
```

- Every well formed expression has a type, which can be displayed in ghci using the command :type or :t
- Example:

```
Prelude>:type ’a’
Prelude>:type False && True
```

Types in Haskell

- For every expression e, it has a type t, written as

```
e :: t
```

- Every well formed expression has a type, which can be displayed in ghci using the command :type or :t
- Example:

```
Prelude>:type ’a’
   ’a’ :: Char
Prelude>:type False && True
   False && True :: Bool
```

Tuples vs Lists

- A List is a sequence of values of the same type, Ex:

```
[1,'a',"String"]
    Error
```

Tuples vs Lists

- A List is a sequence of values of the same type, Ex:

```
[1,'a',"String"]
    Error
```

- A tuple, specified using brackets, is a sequence of values that can be of different types. Ex:

```
(1,'a',"String")
```

Tuple and List Types

- A Lists type is specified by brackets, and the type of the elements it contains. Ex:

```
[’a’,’b’,’c’,’d’]   ::  [Char]
[True,False,True]   ::  [Bool]
```

Tuple and List Types

- A Lists type is specified by brackets, and the type of the elements it contains. Ex:

```
['a','b','c','d']   ::  [Char]
[True,False,True]   ::  [Bool]
```

- Because tuples can contain different types, the tuples type specifies each element. Ex:

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

Tuple and List Types

- A Lists type is specified by brackets, and the type of the elements it contains. Ex:

```
['a','b','c','d']   :: [Char]
[True,False,True]   :: [Bool]
```

- Because tuples can contain different types, the tuples type specifies each element. Ex:

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

- Its possible to have a list of tuples, but the tuples must all be of the same type, Ex:

```
[(False,'a',True),(True,'b',False)]
[(True,False,'a'),(False,'b',True)]
```

Tuple and List Types

- A Lists type is specified by brackets, and the type of the elements it contains. Ex:

```
['a','b','c','d']  :: [Char]
[True,False,True]  :: [Bool]
```

- Because tuples can contain different types, the tuples type specifies each element. Ex:

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

- Its possible to have a list of tuples, but the tuples must all be of the same type, Ex:

```
[(False,'a',True),(True,'b',False)]   -- Valid
[(True,False,'a'),(False,'b',True)]   -- Error
```

Function Types

- A function takes arguments of certain types and returns a value of a certain type.
- In a functions type, this is specified using the $->$ operator, Ex:

```
not     :: Bool -> Bool
head    :: [a]  ->  a
```

- Note: a is not a specific type, we'll talk about this in a few slides

Curried Functions

Currying is the process of transforming a function that takes multiple arguments into a function that takes a single argument, for example:

```
        add  :: Int -> Int -> Int
        add x y = x + y

-- is the same as
        add  :: Int -> (Int -> Int)
        add x y = x + y

-- So, when you call add 5 6, the function becomes
        add  :: Int - > Int
        add y = 5 + y

-- and is evaluated from there
```

Uncurried Functions

- It's possible to specify an uncurried function through use of a tuple
- For Example:

```haskell
add  :: (Int,Int) -> Int
```

Uncurried Functions

- It's possible to specify an uncurried function through use of a tuple
- For Example:

```
add  :: (Int,Int) -> Int
```

- Fact Of The Day: Haskell Curry was an American mathematician and logician best known for his work in combinatory logic

Polymorphic Functions

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

```
length :: [a] -> Int
```

The function length takes a list and returns its size. a can be any type. For example, in

```
length [False,True]     -- a is Bool
length [1,2,3,4]        -- a is Int
```

This allows the function length to work on any type of list

Type Classes

- In Haskell, a class is a collection of that support certain overloaded operations called methods.
- Overloaded basically means the method is defined in different ways based on the type of the argument it is passed

Type Classes

- In Haskell, a class is a collection of that support certain overloaded operations called methods.
- Overloaded basically means the method is defined in different ways based on the type of the argument it is passed
- For example, the Eq class contains the methods:

```
(==)    :: a -> a -> Bool
(/=)    :: a -> a -> Bool
```

- All the basic types, Bool,Char,String,Int,Integer, and Float, are said to be instances of the class (the methods are defined for them)

Predefined Classes

Basic Classes

- Eq → Bool, Char, String, Int, Integer, Float
- Show → Bool, Char, String, Int, Integer, Float
- Read → Bool, Char, String, Int, Integer, Float

Some classes can only have types that are already an instance of another class

- Eq => Ord → Bool, Char, String, Int, Integer, Float
- (Eq,Show) => Num → Int, Integer, Float
- Num => Integral → Int, Integer
- Num => Fractional → Float

Overloaded Functions

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

```
sum          :: Num a => [a] -> a
fromIntegral :: (Num b, Integral a) => a -> b
```

The binding operator (=>) is used to bind the class. So in the function fromIntegral, it takes an argument of any type a and returns a value of any type b so long as a is an instance of Integral and b is an instance of Num

Type Signatures

When writing your own functions, you should always write the functions type before hand. This is not necessary as ghci will infer the type, but it will make your code easier to read. You can tell alot about a function by it's type. For Example:

```
fst   :: (a,b) -> a
fst (x,y) = x

snd   :: (a,b) -> b
snd (x,y)  = y
```

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]
```

Exercises

What are the types of the following values?

```
['a','b','c']                  :: [Char]
('a','b','c')                  :: (Char,Char,Char)
[(False,'0'),(True,'1')]       :: [(Bool,Char)]
([False,True],['0','1'])       :: ([Bool],[Char])
[tail,init,reverse]            :: [[a]->[a]]
```

Exercises

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
palin xs    = reverse xs == xs
twice f x   = f (f x)
```

Exercises

What are the types of the following functions?

```
second xs  = head (tail xs)   :: [a] -> a
swap (x,y) = (y,x)            :: (a,b) -> (b,a)
pair x y   = (x,y)            :: a -> b -> (a,b)
double x   = x*2              :: (Num a) => a -> a
palin xs   = reverse xs == xs :: (Eq a) => [a] -> Bool
twice f x  = f (f x)          :: (a -> a) -> a -> a
```

Exercises

Is the following true (take a guess before trying in ghci)

```
1e-44 + 1e-45 == 1.1e-44
```

Why / why not?

What does the following evaluate to in ghci?

```
1e-45 * 0.1 :: Float
```

What if you don't specify the type? Why is this different?