

) Haskell  $\rightarrow$  static type system i.e. the type of every expression is known at compile time which leads to safer code.  $\therefore$  everything in Haskell has a type.



Haskell also has type inference.



Type is a kind of a label that every expression has and it tells us the category of things that expression fits in.



:t cmd. can be used to identify the type of an expression in ghci.



:: is read out as 'has type of'



each tuple has its own type.

) Functions also have types. We can choose to give explicit type declaration to functions when writing them. This is considered good practice except when writing very short functions.

for example : list comprehension that filters a string so that only caps remain.

maps string to string

removeNonUpperCase :: [char]  $\rightarrow$  [char]

removeNonUpperCase st = [c | c  $\leftarrow$  st, c `elem` [A'..`Z']]



can also use String instead of [char]

for multiple parameters : function-name :: param1  $\rightarrow$  param2  $\rightarrow$  ...  $\rightarrow$  return-type

Note : If we want to give one function a type declaration but are unsure as to what it should be , we can just always write the function w/o first and then check it with :t , since functions are expressions too.

→ some common data types in Haskell : i) Int : bounded ,

(ii) Integer : unbounded  $\text{Int} \rightarrow$  meaning it can be used to represent very big no.s

↓

like really big

(iii) Float : real floating point with single precision

(iv) Double : real floating point with double precision.

(v) Bool

(vi) Char

(vii) Tuple : are types but are dependent on length as well as the types of their components , so there is theoretically an  $\infty$  no . of tuple types.

↓

even empty tuple is a type

→ functions that employ type variables are called polymorphic functions.

→ fst : fn. that returns the first component of a pair where pair is represented by a tuple .

•) Typeclass : sort of interface that defines some behaviour. If a type is a part of a typeclass , that means that it supports and implements the behaviour the typeclass describes.



Typeclasses are not like classes from OOP,  
rather can be understood better as JAVA interfaces .



type signature of `==` function

Note: In Haskell, if a fn. comprises only of special characters ,  
then it is considered infix by default .

class  
constraint,

$(==) :: (\text{Eq } a) \Rightarrow a \rightarrow a \rightarrow \text{Bool}$



can be read as : the equality function takes any two values  
that are of the same type and returns a `Bool` , given  
the type of those two values must be a member of the  
Eq. class. (the class constraint)

→ provides an interface for testing for equality .

Any type where it makes sense to test for equality  
between two values of that type should be a member  
of the Eq. class . All standard Haskell types except `IO`  
and functions are part of the Eq. typeclass .



`elem :: (\text{Eq } a) \Rightarrow a \rightarrow [a] \rightarrow [\text{Bool}]`



↓  
Used `==` over a list to check whether some value we're looking for is in it.

→ some basic typeclasses : (i) Eq : types that support equality testing.

The functions that its members support are  $\text{==}$  and  $\text{/=}$   
↓  
equal to  
↓  
not equal to

(ii) Ord : types that have an order → all types covered till now except fns. are part of Ord.

↓  
Supports  $<$ ,  $>$ ,  $\leq$ ,  $\geq$

↓  
Compare function takes two Ord members of same type and returns an ordering

To be a member of Ord, the type should be a member of Eq.

(iii) Show : members can be represented as strings. All types discussed till now except functions are part of Show.

↓  
most used function : show

(iv) **Read** \* : opposite typeclass of Show. The read function takes a string representation of a type and returns the data corresponding to the type

↓

this is only possible if there is a way for the program to infer the type based on some operation or context.

↓

for ex.  $\text{read } "4" - 2 \Rightarrow 2$

$\text{read } "4" \gg \text{error} \rightarrow \text{This can be}$

countered by using explicit type annotations. Type annotations

are a way of explicitly saying what the type of an expression should be.

↓

This can be done by adding '`::`' at the end of the expression and specifying a type.

```
ghci> read "5" :: Int  
5  
ghci> read "5" :: Float  
5.0  
ghci> (read "5" :: Float) * 4  
20.0  
ghci> read "[1,2,3,4]" :: [Int]  
[1,2,3,4]  
ghci> read "(3, 'a')" :: (Int, Char)  
(3, 'a')
```

