

basic Datatypes. → types play an important role in the readability, safety and maintainability of Haskell code as they allow us to classify and delimit data, thus restricting the forms of data our programs can process

expresses → values  
evaluated → every value has a type  
↓

Note: Set theory is a precursor to type theory

↓  
Used in design of programming language.

type can be thought to be like sets in mathematics.

↓  
Set of Natural No.s, set of whole No.s, set of Rational No.s, set of Even no.s

→ Anatomy of a data declaration → used to define datatypes.

:) Type constructor : name of the type - Capitalized

↓

used when reading or writing type signatures

:) Data constructor : are the values that inhabit the type they are defined in. These are the values that our code evaluate to. This is said to appear at

term level, rather than type level.

For example `data Bool = False`

not true  
false

data constructor  
or value

→ Datatype Bool is repr.  
by the values True or False.

type constructor for  
Bool, i.e. name  
of the type and shows  
up in type signatures

→ Data constructor  
for False  
True  
↓  
Data constructor  
for True  
indicates a  
sum type or a  
logical disjunction i.e  
Or

This complete thing is called data declaration. They don't follow precisely the same pattern always i.e. there are datatypes that use 'and' instead of 'or', and some type and data constructors may have arguments.

↓

The thing that is common to all, is the keyword 'data' followed by the type constructor, the equals sign, and then data constructors.

→ We can define a function by matching on a data constructor, or value and describing the behaviour that the function should have based on which value it matches.

↓

pattern matching on the data constructor

→ Numeric types → ① Integral

int      ↓  
integer (higher no. of abs.  
value support)

② Fractional → i) float

↓  
single precision floating point numbers

ii) double → double-precision floating point number

(twice as many bits with which to describe numbers as the float type)

iii) Rational → represent ratio of two integer values.  
in the  $x/y$  format in simplest form.

iv) scientific → coefficient as integer and  
exponent as int

Arbitrary precision

(not limited precision  
like float, double)

↓  
not available by default,  
needed to be included

These numeric datatypes all have instances of a typeclass  
called 'num'



way of adding functionality

Most numeric types  
will have num typeclass  
And num typeclass provides  
the std '+' , '-' across different

← to types that is reusable

across all types that have  
instances of that type class.

types.

In case of Integer, and most of the numeric datatypes, data constructors are not written out, because they include an infinite series of whole numbers.



Done through GHC magic

→ we almost never want float unless we are doing graphics programming using OpenGL.

→ Some computations will lead to fractional results, for example the division operation.

$\Rightarrow :t (/)$

$(/): \text{Fractional } a \Rightarrow a \rightarrow a \rightarrow a$



denotes a typeclass constraint i.e.

the type variable a must implement the Fractional typeclass.



whatever type of a number a turns out to be, it must be a type that has an instance of a Fractional typeclass. There must be a declaration in how the operations from that typeclass will work for that particular type.



Fractional typeclass requires its types to already have an instance of Num typeclass.

↓  
So fractional typeclass  
can use the operations and  
functions of the Num typeclass

↓  
Here 'Num' is a **superclass**  
for 'Fractional'

↓  
using arbitrary-precision types  
as a matter of course

→ comparing numbers : not equal →  $\neq$

$\Rightarrow :t (\neq)$

$\Rightarrow \neq :: Eq\ a \Rightarrow a \rightarrow a \rightarrow Bool$

$\Rightarrow :t (c)$

$\Rightarrow c :: Ord\ a \Rightarrow a \rightarrow a \rightarrow Bool$

Eq: types that can be compared and determined to be equal  
in value

Ord: type that can be ordered.

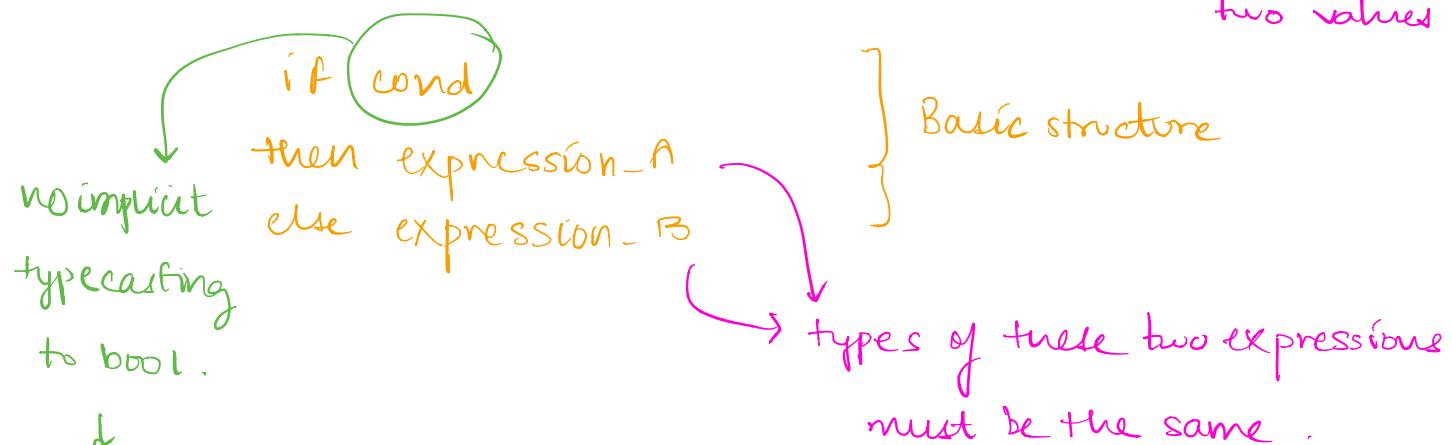
↓  
not limited to Numbers

.) list of items can be compared where the items are also instances of Ord typeclass.

:t expression → what it evaluates to

→ Conditionals with if - then - else .

Haskell doesn't have 'if' statements , but it does have if expressions . can be thought of as a way to choose between two values .



The expression should evaluate to a bool .

→ Tuples : type that allows us to store and pass around multiple values within a single value . we refer to tuples by the no. of values in each tuple , which is referred to its **arity** . Values within a tuple doesn't have to of the same type .

data-declaration of two-tuple → expressed at both type level and term level as (,)

: int (, ) → product-type (AND)  
data (, ) a b = (, ) a b . need to supply  
two parameters both arguments .  
↓  
applied to concrete types at term level .

convenience functions for two-tuple : fst, snd

(a, b) type constructor can be used to pattern match on functions.

not recommended to use tuples of size  $> 5$ .

cannot have 1 sized tuple but 0 sized tuple called 'unit' ✓

→ lists : a) all values of the same type

b) own distinct '[ ]' syntax → used at the type and term

c) no. of values in the list level to express list values.

isn't specified in the type.