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

↓

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

Note: set theory is a precursor to type theory

↓

used in design of programming langs.

define new datatypes but not necessarily data values.

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

:) Type constructor : name of the type - Capitalized

↓

used when reading or writing type signatures (only)

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

term level, rather than type level.

For example `data Bool = False | True`

- `Bool` → type constructor for Bool, i.e. name of the type and shows up in type signatures
- `False` | `True` → indicates a sum type or a logical disjunction i.e OR
  - `False` → Data constructor for False
  - `True` → Data constructor for True

`not True`
`False`
→ data constructor or value

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

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 → represents ratio of two integer values. is the x/y format in simplest form.

iv) Scientific → coefficient as integer and exponent as int

↓

not available by default, needed to be included

Arbitrary precision (not limited precision like float, double)

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

↓

Way of adding functionality to types that is reusable across all types that have instances of that typeclass.

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

type.

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.

```
⇒ :t (/)
(/) :: Fractional a ⇒ a → a → 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 on 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

↓

using arbitrary-precision types as a matter of course

↓

Here 'Num' is a superclass for 'Fractional'

Num

↓

fractional

→ comparing numbers : not equal → /=

:t (/=)

⇒ /= :: Eq a ⇒ a → a → Bool

⇒ :t (<)

⇒ < :: Ord a ⇒ a → a → Bool

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

Ord : types 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.

if cond
then expression_A
else expression_B

Basic structure

no implicit typecasting to bool.

↓

The expression should evaluate to a bool.

types of these two expressions must be the same.

→ 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 (,)

```
: info (,)
data (,) a b = (,) a b.
```

→ product type (AND)

↓

need to supply both arguments.

↳ two parameters

↓

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 level to express list values.

c) no. of values in the list isn't specified in the type.

→ Names and Variables : In Haskell seven categories of entities that have names : functions, term-level variables, data constructors, type variables, type constructors, type classes and modules.