A datatype declaration defines a type constructor and data constructors.

↓

values of a particular type; they are also fncs. that let us create data, or values, of a particular type.

The need for types: Type systems in logic & mathematics have been designed to impose constraints that enforce correctness. A type system defines the association between different parts of a program and checks that all parts fit together in a logically consistent, provable correct way.

Static type system enable many errors to be caught at compile time.

Good type systems also enable compiler optimizations, because the compiler can know and predict certain things about the execution of a program based on the types

→ Reading type signatures

values → fully applied functions

(*) When we query the types of numeric values, we see typeclass information instead of a concrete type., because the compiler doesn't know which specific numeric type a value is until the type is either declared or the compiler is forced to infer a specific type based on a function.

```
:t 13
13 :: Num a => a
```

constrained polymorphic

→ Understanding the function type

(*) `→` is the type constructor for functions. It's a type constructor, very much like Bool, except the `→` type constructor takes arguments and has no data constructors.

```
:info (->)
data (->) a b
```

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

Like the tuple type constructor which needs to be applied to two arguments, a fn. must have two arguments — one input & a result. But the fn. type constructor does not have data constructors.

↓

The value that shows up at term level is the function, hence function are values.

↓

Arrow is an infix operator that has two parameters and associates to the right. (although the fn. applⁿ. is left associative)

```
fst :: (a,b) -> a
```

two parameters

the function type

→ Typeclass - constrained type variables

```
:t (+)
(+) :: Num a => a -> a -> a
:t (/)
(/) :: Fractional a => a -> a -> a
```

Compiler gives the least specific & more general type it can.

*) Instead of limiting these fns to a concrete type, we get a typeclass - constrained polymorphic type variable.

*) When a typeclass constraints a typeclass this way, the variable could represent any of the concrete types that have instances of that typeclass so that specific operations on which the fn. depends are defined for that type.

↓

Using typeclasses allows us to generalise an operation over multiple types.

*) Multiple type constraints: (Ord a, Num a) => a -> a ->

Ordering

→ Currying: As in lambda calculus, all fns. in Haskell take one argument and return one result. There is no built in support for multiple arguments in Haskell. Instead there are syntatic conveniences that construct curried fns. by default. Currying refers to the nesting of multiple fns, each accepting one argument & returning one

result, to allow the illusion of multiple-parameter fns.

clearly evident from the defn. of the 'function type'

↓

Each arrow in a type signature represents one argument and a result, with the final type being the final result.

↓

for example: (+) :: Num a ⇒ a → a → a → final result

↙ typeclass constraint, Addition is defined in that typeclass

↓ represents successive function applications

↓

The function on the top is returning another function that accepts the next argument.

This is equivalent to (+) :: Num a ⇒ a → (a → a)

↓

this kind of definition makes querying the default behaviour in Haskell.

*) like in lambda expression, applying the expression to one argument returns a partially applied fn. which awaits application to a second argument. After we apply it to the second argument, we have the final result.

*) one argument, one result (no matter how many nested lambdas)

*) explicit parenthesization, as when an input parameter is itself a fn., may be used to indicate order of evaluation.

→ Partial Application : The ability to apply only some of a function's arguments is called partial application.

addStuff :: Integer → (Integer → Integer)

↓

Applying addStuff to one argument gives us the function addTen, which is the return function of addStuff. Now applying addTen to an argument gives us a return value, and no more function arrows left.

→ manual currying and uncurrying

↓

f a b = f (a,b)
f (a,b) = f a b.

means un-nesting the functions and replacing the two functions with a tuple of two values

↓

(+) :: Num a ⇒ (a,a) → a (better fits the description of addition)

↓

an example of manual currying using lambda fns.

anonNested :: Integer → Bool → Integer

anonNested = (\i -> (\b -> i + nonsense b))

anonNested 10 -> \b -> 10 + nonsense b.

→ Sectioning : **partial application** of **infix operators**.

↓

allows us to choose whether the argument we're partially applying the operator to, is the first or second argument.

↓

argument order plays an important role with noncommutative operators.

↓

not limited to arithmetic only

↓

for example `let celebrate = (++ " woot!")`

```
>> celebrate "naptime"
=> "naptime woot!"
```

↓

This syntax can be used with prefix based fcns by making them infix.

↓

```
9 `elem` [1..20]
```

↓

```
let c = (`elem` [1..10])
c 9 -> True
```

```
let x = 5
let y = (2^)
let z = (^2)
y x -> 32
z x -> 25
```

[..] → shorthand to create a list of all elements between the first & last values.

Sectioning syntax exists to allow some freedom in which arg. of a binary op. we apply a func. to.

Exercises : type arguments

```
let f :: a → a → a → a ; f = undefined
let x :: Char ; x = undefined
: t f x
```

⇒ Char → Char → Char

↓

we can check the type of things that haven't been implement yet, as long as we give it a undefined value in ghci.

→ polymorphism : polymorphic type variables gives us the ability to implement expressions that can accept arguments & return results of different types.

↓

Broadly speaking, type signatures may have three kinds of types : concrete, constrained polymorphic & parametrically polymorphic.

↓

fully polymorphic

↓

typeclass constrained

↓

decreases the type of values it can take, but increase the no. of operations can be safely used on the type

*) By default, type variables are resolved at the left-most part of the type signature and are fixed once sufficient

information to bind them to a concrete type becomes available.

*) Typeclass constraints limit the set of potential types (and thus, potential values) while also passing along the common fns. that can be used with those values.

*) Concrete types have even more flexibility, due to the additive nature of typeclasses, that more operations & fns. become available to them

*) polymorphic constants : Numeric literals are polymorphic and stay so until given a more specific type. The literals are given maximum polymorphism, until they are forced into a concrete type

↓

we can force the compiler to be more specific by declaring the type. example :

let x = 5 :: Int

*) working around constraints : length → Int

>> 6 / length [1,2,3]

>> :t (/)

(/) :: Fractional a ⇒ a → a → a

>> here 6 is of Num type and hence can be broadcasted down to fractional, but Int is neither a part of fractional typeclass, or a super typeclass to fractional, hence does not work.

↓

work around is available for these problems.

that deals with converting the numeric value from a specific type to a more generic type.

↓

Done using fromIntegral function that takes an Integral constrained value & lifting it to Num type.

↓

```
>> :t fromIntegral
>> fromIntegral :: (Integral a, Num b) => a -> b
```

↓

```
>> 6 / fromIntegral (length [1, 2, 3])
```

→ Type Inference : an algorithm for determining the types of expressions and Haskell uses it. It will infer the most generally applicable (polymorphic) type that is still correct.

→ Asserting type for declarations : we can assert a type to our declaration using :: , for example.

```
>> let triple x = x * 3 :: Integer
>> :t triple
triple :: Integer -> Integer
```

*) Generally type signatures are laid out for top level bound functions.

*) monomorphism restriction : top-level declarations by default will have a concrete type if any can be determined, for source files. It avoids the construct of assigning

the most general polymorphic type to an argument. To avoid that we can use

{-# LANGUAGE NOMonomorphismRestriction #-}

---

1. b) 0, "doge" :: Num a ⇒ (a, [Char])

c) 0 :: Integer, "doge" :: (Integer, [Char])

d) False :: Bool

3. x = 5

y = x + 5 ⇒ y = 10

z y = y * 10

>> :t z

>> z :: Num a ⇒ a → a

4. x = 5

y = x + 5

f = 4 / y ⇒ f = 0.4

>> :t f

f :: Fractional a ⇒ a

---

*) Another way of explaining type inference : Type inference is a faculty some programming languages, most notably Haskell & ML have to infer principal types from terms without needing explicit type annotations.

↓

With respect to Haskell, principal type is the most generic type which still typechecks, Principal type property holds for a type system, if a type can be found for a term in an environment for which all other types for that terms are instances of the principal type

↓

(Ord a, Num a) => a → principal type

Integer

*) Type variable : used to refer to unspecified type or set of types in a type signature.

id :: (a) → a

↓

type variable