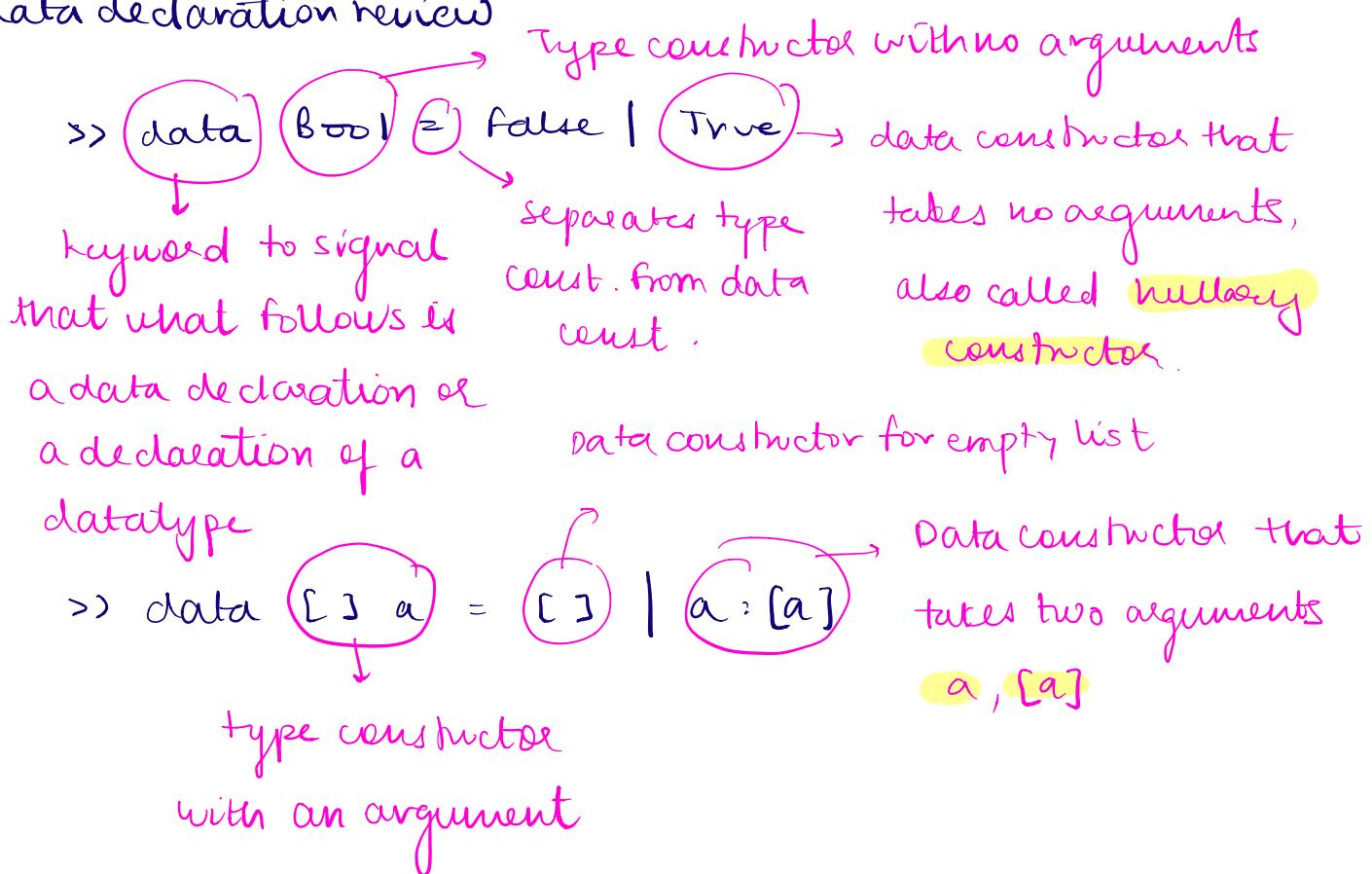


Chapter 11 : Algebraic Datatypes

*> A type can be thought of as an **enumeration** of constructors that have zero or more arguments

→ data declaration review



→ Data and type constructors

*> two kind of constructors in Haskell

→ data constructor
↳ type constructor

*> type constructor are used only at the **type level**, in type signatures and typeclass declarations & instances.
Types are static & resolve at compile time.

*> Data Constructors construct values at **term level**, i.e. values we can interact with at runtime. Called constructors because they define a means of creating or building

a type or value.

- *> Constants vs constructors : constants are type & data constructors that take zero arguments. They can store only a fixed type & amount of data. For example `Bool` is a type constant, since it isn't waiting for any additional information in the form of an argument in order to be fully realized as a type. It enumerates two values that are also constants.
- *> While we call `True` & `False` "data constructors", in fact since they don't take any arguments, their values are already established and not being constructed in any meaningful sense.
- *> Types and Data Constructors can be parameterized to allow flexibility to allow storage of different types & amount of data. When a constructor takes an arg, it operates like a fn, in the sense that it needs to be applied to get a concrete value.

*> $\text{data } \text{Trivial} = \text{Trivial}' \rightarrow \text{constant data constructor}$.

(type constants)	\downarrow	$\text{constant at type level}$	\uparrow	$\text{data constructor with one argument}$
---------------------------	--------------	---------------------------------	------------	---

 $\text{data } \text{UnaryTypeCon } a = \text{UnaryValueCon } a$
 \downarrow
 $\text{type constructor of one argument. constructor awaiting a type constant to be applied to.}$
 \rightarrow
 $\text{this parameter allows for flexibility over the type of data being stored.}$

- * These don't behave like term level fns. in the sense of performing operations on data. (just placing values)
- * Not all type arguments to constructors have value-level witnesses, some are phantom. (not used at all)
- * Unary value **Con** could show up as different literal values at runtime, depending on the type of `a` it is applied to
 (list example)

$\text{Int} \rightarrow \text{Bool}$ concrete.
 types of values. or to be applied?

→ Type constructors and kinds

data [] **a** = [] | a : [a] → must be applied to a concrete type before we're a list

- * Kinds are types of types. Represented by `*'. We know something is **concrete type** or fully applied, when it is represented by `*'. When it is $* \rightarrow *$, like a fn, it is **waiting to be applied**. For example

>> let f = not True

→ doesn't take any arguments and is not waiting for application, to produce a value i.e. it has a concrete value.

>> :t f

f :: Bool

>> let f x = x > 3

>> :t f

f :: (Ord a, Num a) \Rightarrow a \rightarrow Bool

→ This on the other hand is awaiting appln. to an x to produce

a value, hence it has a function arrow in its type signature.

* we query the kind signature of a **type constructor** using `:k` in ghci.

`>> :k Bool` → note, that we're referring to the type & not the value. (concrete or fully applied)

`Bool :: *`

`>> :k [Int]` → type formed after applying Int argument to [a]. concrete

`[Int] :: *`

`>> :k []`

`[] :: * → *` → still needs to be applied to a concrete type.

→ Data constructors and values

`data PugType = PugData` → for any fn. that required a value of type `PugType`, we know the value will be `PugData`.

`data HuskyType a = HuskyData` → type variable doesn't occur as an argument to `HuskyData` or anywhere else after '`=`'.



`HuskyData` is a constant ← value.

This is what it means to have no witness or to be a phantom.

data DogueDeBordeaux doge = DogueDeBordeaux doge.

>> t DogueDeBordeaux

DogueDeBordeaux :- doge →

DogueDeBordeaux doge

Once doge is bound to a concrete type, then this will be a value of type DogueDeBordeaux doge.



>> t DogueDeBordeaux 'c'

DogueDeBordeaux 'c' ::

DogueBordeaux Chai

Not a value yet, but a defn. for how to construct a value of that type.

>> k DogueDeBordeaux

* → * → to be applied to get a concrete type

>> k DogueDeBordeaux Int

* → applied Int & got a concrete type

→ making a value of the type of each:

myPug = Pug Data.

myHusky = Husky Data

myDoge = DogueDeBordeaux 10 → agrees to the type variable being bound to

myDoge :: DogueDeBordeaux Int



```
data Doggies a =  
    Husky a  
    | Mastiff a  
deriving (Eq, Show)
```

*) The kind signature of the type constructor looks like a fn., and the type signature for either of its data constructors looks similar.

```
>> :k Doggies  
Doggies :: * → *  
>> :t Husky  
Husky :: a → Doggies a
```

*) So the behaviour of constructors is such that if they don't take any arguments, they behave like constants. If they do take arguments, they act like functions that don't do anything except getting applied.

→ Exercises: Dog Types.

Doggies : Type constructor

- :: k Doggies → * → *
- (:: t) Doggies String → *
- (:: t) Husky Int → Husky Int :: Num a ⇒ Doggies a
- (:: t) Husky (Int :: Integer) → Husky Int :: Doggies Integer
- (:: t) Mastiff "Scooby Doo" → Mastiff "Scooby Doo" :: Doggies [char]

:t DogueDeBordeaux → DogueDeBordeaux :: doge →
DogueDeBordeaux doge.

:t DogueDeBordeaux "doggie!" → DogueDeBordeaux "doggie!"
:: DogueDeBordeaux [Char]

→ What's type and what's data : types are static & resolve at compile time. Data is what we work with at runtime. Types circumscribe values & in that way, they describe which values are flowing through what parts of the program.

* Not everything referred to in a datatype is necessarily generated by that datatype itself. for example.

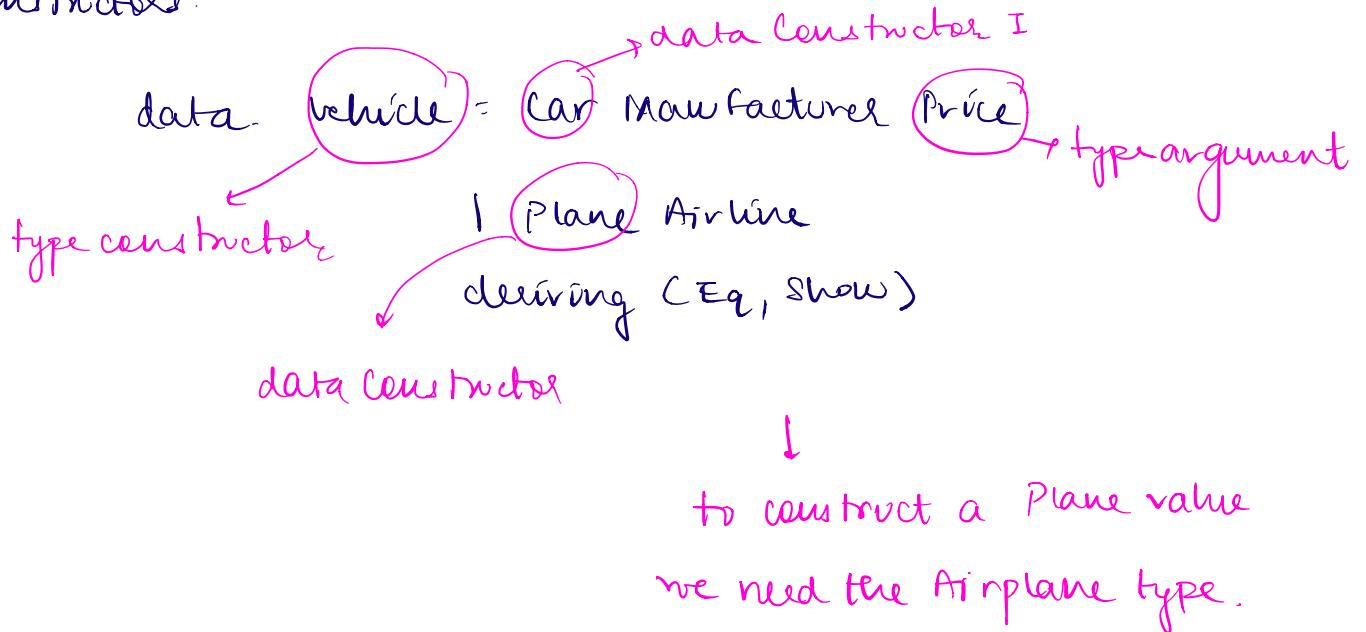
data Price = Price Integer deriving (Eq, Show)

Value Price does not depend solely on this datatype definition. It depends on the type Integer as well. If, for some reason, Integer wasn't in scope, we'd be unable to generate Price values.

data Manufacturer = Mini | Mazda | Tata
deriving (Eq, Show)

data Airline = PapuAir | CatapultsRUs | TakeYourChances
United
deriving (Eq, Show)

*) Since none of them take arguments , all are generated by their declarations and are more like constant values than constructs .



- *) Here, the datatypes are generating the constructors
 - *) Data constructors take arguments, which are specific types, but not specific values

→ Exercises : Vehicles

① local :: vehicle → Bool

`isCar (Car _)` = True

isCar = False

② is Plane :: vehicle \rightarrow Boat

isPlane (Plane-) = True

r's Plane - \Rightarrow False

③ onChase :: [vehicle] \rightarrow [bool]

one char r = map is car r

④ $\text{getMan} :: \text{vehicle} \rightarrow \text{Manufacturer}$

$\text{getMan}(\text{car } m) = m$

→ Data constructor Arities. : Arity refers to the no. of arguments a function or constructor takes.

* Data constructors that take more than one argument are called products.

* Tuples can take several arguments - as many there are inhabitants of each tuple - and are considered the canonical product type ; they are anonymous products because they have no name.

* for example : $\text{data MyType} = \text{MyVal}(\text{Int})$

Data value constructor

deriving (Eq, Show) to the defn. of data const.

type constructor

→ What makes these datatypes algebraic? Algebraic datatypes in Haskell are algebraic because we can describe the patterns of argument structures using two basic operations: sum & product.

* Cardinality of a datatype is the number of possible values it defines. This no. can be as small as 0 or as large as ∞ . Knowing how many possible values inhabit a type can help us reason about our programme.

- *) Cardinality of Bool = 2 \rightarrow 2 data constructors both nullary , hence only 2 possible values.
- *) Cardinality of INT8 = 256 \rightarrow values from (-128 to 127) . INT8 is not included in the standard prelude f needs to be imported using Data.Int .
- *) Having a single nullary value for a single type allows us to reason that any time we encounter the type in any type signature , we know the value we're looking at.
- *) for unary data constructor , the parameter will be a type f not a value . The value will be constructed at runtime from the argument we applied to. **Datatypes with unary constructor have cardinality as the type they contain.**
- \rightarrow newType : type that can only have a **single unary data constructor** . These declarations are different from type declaration (data keyword) & type synonyms (type keyword) . Cardinality of newtype based definitions is the same as the type involved in the constructor .
- *) Advantages over a vanilla data declaration f type alias .
 - i) no runtime overhead , as it **reuses the representation of the type it contains** . The difference between newType and the type it contains is gone by the

time the compiler generates the code. for ex.

tooManyGoats :: Int → Bool

tooManyGoats n = n > 42

i) would cause problems. If we were to confuse the no.s of goats & cows. Hence being to apply a typecheck would be quite useful. Hence,

newtype Goats = Goats Int deriving (Show, Eq)

newtype Cows = Cows Int deriving (Show, Eq)

↓

safes , pattern matching in the fn.

↓

tooManyGoats :: Goats → Bool

tooManyGoats (Goats n) = n > 42

↓

no chance of a mixup here.

ii) newtype is similar to type alias in that the representations of the named type and the type it contains are identical & any distinction b/w them is gone by compile time. So String → [char], and Goats → Int. using newtype & type synonyms can improve the readability of the code.

↓

One key diff. here is that we can define instances for newtype, but can't be done in the case of type

Synonyms. These instances can allow for a different behaviour than the one for the underlying type

↓

class TooMany a where

tooMany :: a → Bool

instance TooMany Int where

tooMany n = n > 42

↓

Now let's say that goat counting should have a different behaviour than that of its underlying Int type. We can define an instance for this, which isn't possible for type synonyms.

↓

newtype Goats = Goats Int deriving Show

instance TooMany Goats where

tooMany (Goats n) = n > 43

* Using the type class instances of the type our newtype contains. as free for the primitive ones., using the deriving clauses. for user-defined typeclasses we can use a language extension called

Generalized Newtype Deriving. Language extensions

are enabled in GHC using LANGUAGE pragma ., i.e . it tells the compiler to process data in ways beyond what the standard provides for.

*) Here the extension will tell the compiler to allow our newType to rely on a type class instance for the type it contains.

*) Without the extension, the process is as follows :-

```
class TooMany a where  
    tooMany :: a → Bool
```

```
instance TooMany Int where
```

```
    tooMany n = n > 42
```

```
newtype Goats =
```

```
    Goats Int deriving (Eq, Show)
```

```
instance TooMany Goats where
```

```
    tooMany (Goats n) = tooMany n
```

→ Goats instance
will do the same
thing as the
Int instance, but

*) Adding the Pragma →

```
newtype Goats =
```

```
    Goats Int deriving (Eq, Show,  
                        TooMany)
```

we still have to
define it separately.

Here we don't have to define an instance of TooMany for Goats that's identical to the Int instance.

→ Exercises : logic gates

Instance TooMany (Int, String) where

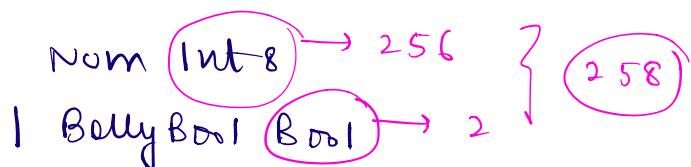
$$\text{tooMany } (x, y) = \text{tooMany } x$$

instance TooMany (Int, Int) where

$$\text{tooMany } (x, y) = \text{tooMany } \$ x + y$$

→ Sum types : 'I' in the data declaration represents the 'sum' in algebraic datatypes. The cardinality of sum types is the sum of cardinality of its individual data constructors. For Bool, hence the cardinality is the sum of the cardinality of True, & False i.e. 2.

for example : data NumberOrBool =



deriving (Eq, Show)

Detour : If we choose (-128) for a value precisely, we get a warning, even though it is absolutely in the range of Int8. Here (-128) desugars into (negate 128). The compiler is not okay with 128 being used for Int8, and hence throws a warning, even though we're working with its -ve counterpart. One way to get over this is to use Negative literals extension. This works with

the `'-'` operator, but doesn't work well with negative.

→ Product types : A product type's cardinality is the product of the cardinalities of its argument types. More on the lines of struct in C used to carry multiple values around in a single data constructor. Any data constructor with two or more type arguments is a product.

*) Tuples → anonymous products . Gives us a way to encapsulate two pieces of data . , or possibly different data types , in a single value .

*) data QuantumBool = QuantumTrue

- | Quantum False
- | Quantum Both

Cardinality : 3

delivering (Eq, show)

data TwoQs =

Mk Two as QuantumBoo1 QuantumBoo1

deriving (Eq, show)

Cardinality : 9

* we could also have written the TwoAs type using a type alias and the tuple data constructor. Note:

Type aliases create type constructors, not data constructors

`type TwoQs = (QuantumBool, QuantumBool)`

hence used at type level.

→ Record Syntax : product types with additional syntax to provide convenient accessors to fields within the record.

```
>> data Person =  
    MkPerson String Int  
    deriving (Eq, Show)
```

JK = MkPerson "Julie" 26 8

CA = MkPerson "Chris" 16

The contents of this can be unpacked while using in functions.

```
name :: Person → String  
name (MkPerson s _) = s
```

↓
value constructor

data Person =
 Person { name :: String,
 age :: Int }
 deriving (Eq, Show)

value constructor fields. with named row accessors.
→ There are now named row field accessors.

↓
These are just fn's that go from the product type to a member of product.

```
>> :t name  
name :: Person → String  
>> :t age  
age :: Person → Int  
>> let papu = Person "Papu" 5  
>> age Papu  
5  
>> name Papu ≈ papu.name  
"Papu"
```

→ normal form : All existing algebraic rules for products, and some apply in type systems., and that includes the distributive property .

$$\begin{aligned} &\Rightarrow 2^*(3+4) \xrightarrow{\text{sum of products}} \\ &\Rightarrow 2^*3 + 2^*4 = 14 \end{aligned}$$

- *) In normal arithmetic, the expression is in normal form when it's been reduced to a final result .
- *) If we think of the numerals in the above expr. as representations of set cardinality , then the sum of products expression is in normal form .
- *) for example : data Friction = Fiction deriving Show
data Nonfriction = NonFiction deriving Show

data BookType = FictionBook Fiction | NonFictionBook NonFiction deriving Show

Types f
not values .

- *) Keeping in mind , that when we use a type, we are working with the possibility of encountering any of their values. i.e. , we can't only the True value of Bool type , we must admit the possibility of either Bool value i.e. False .

*) type AuthorName = String $a^*(b+c)$

data Author = Author (AuthorName, BookType)

↓
can be written as the follows:

data Author =

Fiction AuthorName

| Nonfiction AuthorName

deriving (Eq, Show)

product types.

*) Another common example

data Expr =

Number Int

Sum of products

| Add Expr Expr

↓

| Minus Expr

hence normal form

| Mult Expr Expr

| Divide Expr Expr

*) A stricter interpretation of normal form would require representing products with tuples & sum with Either. Then

type Number = Int

type Add = (Expr, Expr)

type Minus = (Expr)

type Mult = (Expr, Expr)

type Divide = (Expr, Expr)

data Expr =

Either Number

(Either Add (

either minus (

either Mult Divide)))

*) This repr. finds appm. in problems where one is writing fun. or folds over representations of data types, such as with generics & metaprogramming.

→ Exercises : How does your Garden know ?

1. data FlowerType = Gardenia
| Daisy
| Rose
| Lilac
deriving (Show)

type Gardener = String

data Garden =
Garden Gardener FlowerType.
deriving (Show)
↓ SOP form

data Garden = Gardenia Gardener
| Daisy Gardener
| Rose Gardener
| Lilac Gardener
deriving (Show)

→ Constructing & Deconstructing values.

- *) Essentially two things that can be done with a value:
we can generate it or we can match on it & consume it.
- *) Data in Haskell are immutable, so values carry with them the info. about how they were created. This info can be used when we consume or deconstruct the value.
- *) data Guesswhat = Chickenbutt deriving (Eq, Show)

data Id a = MKId a deriving (Eq, Show)

data Product a b = Product a b deriving (Eq, Show)

data Sum a b = First a
| Second b
deriving (Eq, Show)

Here Sum & Product
Used to repr.
arbitrary sums and
products.

data RecordProduct a b =

RecordProduct { pfirst :: a
, psecond :: b }
deriving (Eq, Show)

*) newtype NumCow =
NumCow Int
deriving (Eq, Show)

newtype NumPig =
NumPig Int

If we have two values in
a product, then the
conversion to using
product is straightforward.

}

deriving (Eq, Show)

data Farmhouse =

Farmhouse NumCow NumPig
deriving (Eq, Show)

'2'

} Product type

type Farmhouse' = Product NumCow NumPig

*) Farmhouse & Farmhouse' are the same.

*) for an example with three values in a product instead of two, we must begin to take advantage of the fact that Product takes two values., one of which itself can be a product of types.

This type of nesting can be done as much as required, of course until the compiler chooses.

>> newType NumSheep =

NumSheep Int

deriving (Eq, Show)

>> data BigFarmHouse =

BigfarmHouse NumCow NumPig NumSheep

deriving (Eq, Show)

>> type BigFarmHouse' =

Product NumCow (Product NumPig NumSheep)

↓ something similar can be done using sum

type Name = String

type Age = Int

type LovesMud = Bool

type PoundsOfWool = Int

data CowInfo = CowInfo Name Age

deriving (Eq, Show)

data PigInfo = PigInfo Name Age LovesMud

deriving ()

data SheepInfo = SheepInfo Name Age PoundsOfWool

deriving ()

CowInfo.

data Animal = Cow CowInfo

PigInfo SheepInfo.

| Pig PigInfo

| Sheep SheepInfo

deriving ()

type Animal' = Sum CowInfo (Sum PigInfo SheepInfo)

/ \

first CowInfo | second (first PigInfo)

(second SheepInfo)

>> let annie' = (SheepInfo "Annie" 5 70)

>> let annie = second (second annie') :: Animal'

>> :t annie

annie :: Animal'

```

>> let annie = Second (Second annie')
>> :t annie
annie :: Sum a1 (Sum a2 SheepInfo)
>> let p' = First (PigInfo "Polly" 5 True)
    let p' = Second p' :: Animal'

```

The location it

needs to end up in → first of Second : PigInfo.

the second loc at Animal'

first : CowInfo

Second of Second : SheepInfo.

```
>> let elmo' = Second (SheepInfo "Elmo" 5 5)
```

```
>> let elmo = First elmo' :: Animal'
```

↓

gives an error, that we are trying to fit (Sum a0
SheepInfo on CowInfo).

→ Constructing values

data Id a = MkId a deriving (Eq, Show)

Id has an argument, so we have to apply it to
Something before we can construct a value of that type

```
>> :t MkId
```

MkId :: a → Id a

```
>> idInt :: Id Integer
```

idInt = MkId 10

```
>> idString :: Id [char]  
idString = MkId "2563" ↗
```

```
type Awesome = Bool  
type Name = String  
person :: Product Name Awesome  
person = Product "simon" True  
data Twitter = Twitter deriving (Eq, Show)  
data Askfm = Askfm deriving (Eq, Show)
```

```
SocialNetwork :: Sum Twitter Askfm
```

```
SocialNetwork = first Twitter ↘
```

we get to use only
one value out of the
two possible ones.

↓

Also, we have to use one of the
data constructors generated by the
definition of sum to indicate which
of the possibilities in the disjunction,
we mean to express.

```
>> First Askfm :: Sum Twitter Askfm
```

↓

gives an error, since mismatch - Twitter is
supposed to be first & Askfm supposed to be
second.

*) The type signature Sum Twitter Askfm asserts the ordering which can be done as follows too :

```
data SocialNetwork =  
    Twitter  
    | Askfm  
deriving (Eq, Show)
```

→ Here Twitter & Askfm are direct inhabitants of Social Network where before they inhabited the sum type.

*) Avoid using type synonyms with unstructured data like text or binary. Type synonyms are best used when we want something lighter weight than newtypes but also want type signatures to be more explicit.

→ Record syntax :

```
myRecord :: RecordProduct Integer Float
```

```
myRecord =  
    RecordProduct { pfirst = 42  
                  , psecond = 0.00001 }
```

•) Using this is more compelling, when we have domain-specific names for things.

```
data OperatingSystem =  
    CmplusLinux  
    | OpenBSD
```

```
| Mac  
| windows  
deriving ()
```

```
data ProgLang =  
    Haskell  
    | Agda  
    | Idris  
    | Purescript  
deriving ()
```

```
data Programmer =  
    Programmer { os :: OperatingSystem  
                , lang :: ProgLang }
```

```
deriving ()
```

```
>> nineToFive :: Programmer
```

```
nineToFive = Programmer { os = Mac  
                           , lang = Haskell }
```

```
>> feelingWizardly :: Programmer
```

```
feelingWizardly = Programmer { lang = Haskell,  
                               , os = Mac }
```

→ Accidental bottoms from records. : either define the whole record at once or not at all. Partial application of the data constructor suffices to handle this. Percolate values through your programs, not bottoms.

```

data ThereYet = There Float Int Bool deriving (Eq, Show)

nope :: Float -> Int -> Bool -> ThereYet
nope = There

notYet :: Int -> Bool -> ThereYet
notYet = nope 25.5

notQuite :: Bool -> ThereYet
notQuite = notYet True

yussss :: ThereYet
yussss = notQuite True

```

→ Deconstructing values.

`newtype Name = Name String` `der()`

`newtype Acres = Acres Int` `der()`

`data FarmerType = DairyFarmer`
 | WheatFarmer
 | SoybeanFarmer
`deriving Show`

`data Farmer = Farmer Name Acres FarmerType`
`deriving Show`

`i's DairyFarmer :: Farmer -> Bool`

`i's DairyFarmer (Farmer _ _ DairyFarmer) = True`

`i's DairyFarmer _ = False.`

`data FarmerRec =`
`FarmerRec { name :: Name`
`, acres :: Acres`
`, farmerType :: FarmerType }`
`deriving Show`

isDairyFarmerRec :: FarmerRec → Bool

isDairyFarmerRec farmer = fetching the record from
case farmerType farmer OF the argument.

Dairy Farmer → True

- → False

*>) Another way of unpacking & extracting contents of a product type.

→ Accidental bottoms from records.

data Automobile = Null

| Car { make :: String }

| model :: String

| year :: Integer }

driving()

↓

Terrible thing to do. First maybe should be employed in the place of null. Second, consider the case where one has a null value, but we have used one of the record accessors.

↓

Whenever we have a product that uses record accessors, we should keep them separate from any sum type that is wrapping it. Split out the product into its independent type with its own constructor instead of only as an inline data

constructor product .

↓

```
data Car = Car { make :: String  
                , model :: String  
                , year :: Integer }  
deriving (Eq, Show)
```

↓

```
data Automobile = Null | Automobile Car  
deriving ()
```

↓

>> make Null → Couldn't match expected type
'Car' with actual type 'Automobile'

*) In Haskell, we want the typechecker to catch us doing things wrong, so we can fix it before problems multiply and things go wrong at runtime .

→ Function type is exponential : In the arithmetic of calculating inhabitants of types , function type is exponential operator . Given a fn , $a \rightarrow b$, inhabitants = b^a .

So if $\text{Bool} \rightarrow 3$ type . $3^2 = 9$ inhabitants and thus has nine possible implementations . i.e. i/p + o/p value combination .

$$a \rightarrow b \rightarrow c :: (c^b)^a \Rightarrow c^{(b+a)}$$

data Quantum = Yes
| No
| Both
der ()

Either a b = left a |
right b .

quantumSum1 :: Either Quantum Quantum

(3+3)

quantumSum1 = Right Yes.

6 different
possible values.

quantumSum2 :: Either Quantum Quantum

quantumSum2 = Left Both.

quantProd1 :: (Quantum, Quantum)

Quant Prod1 = (Yes, Yes)

Quant Prod2 :: (Quantum, Quantum)

3*3
possible
values.

Quant Prod2 = (Yes, No)

Quant Prod3 :: (Quantum, Quantum)

(product type)

quantProd3 = (Both, Yes)

→ function type: each unique implementation of the function is an inhabitant.

quantFlip1 :: Quantum → Quantum

quantFlip1 Yes = Yes

quantFlip1 No = Yes

quantFlip1 Both = Yes.

→ Higher kinded datatypes : kinds are types of type constructors, primarily encoding the no. of arguments they take.

$* \rightarrow *$: needs one argument to be fully applied
 $* \rightarrow * \rightarrow *$: needs two arguments to be fully applied. This is known as higher - kinded type .

```
data Silly a b c d =  
    MkSilly a b c d der()
```

>> : kind Silly

Silly :: $* \rightarrow * \rightarrow * \rightarrow * \rightarrow *$

>> : kind Silly Int

Silly Int :: $* \rightarrow * \rightarrow * \rightarrow *$

>> : kind Silly Int String Bool

Silly Int String Bool :: $* \rightarrow *$

concrete value

>> : kind Silly Int String Bool String

↑

Silly Int String Bool String :: $*$ (Fully applied)

>> : kind (,, ,)

(,, ,) :: $* \rightarrow * \rightarrow * \rightarrow * \rightarrow *$

>> : kind (Int, String, Bool, String)

(Int, String, Bool, String) :: $*$

* Getting comfortable with higher kinded types is important as type arguments provide a generic way to express a hole to be filled by consumers of your datatype later.

* For example:

```
data EsResult found a =  
    EsResult found { _version :: DocVersion  
                    / _source :: a  
    } deriving ()
```

We know that this particular response from Elasticsearch will include a **DocVersion value**, so that's been assigned a type.

↓

On the other hand `_source` has a type **a** because we have no idea what the structure of the documents they're pulling from Elasticsearch look like.

↓

In practice, we do need to be able to do something with that value of type `a`. The thing we will want to do with it - the way we consume or use that data - will usually be from a **fromJSON typeclass instance for deserializing JSON data into a Haskell datatype**.

↓

But in Haskell, we don't conventionally want to

constrain that polymorphic a in the datatype.

↓

The FromJSON typeclass (assuming that is what's needed in the given context) will likely constrain the variable in the type signature(s) for the function(s) that will process this data.

↓

Accordingly, the FromJSON typeclass instance for EsResultFound requires a FromJSON instance for that a:

```
instance (FromJSON a) =>
    FromJSON (EsResultFound a) where
        parseJSON (Object v) =
            EsResultFound .
                <$> v .: "version"
                <*> v .: "source"
        parseJSON _ = empty
```

↓

By not fully applying the type - by having it higher-kinded - space is left for the type of the response to vary, for the "hole" to be filled in by the end user.

→ Lists are polymorphic

```

data [ ] a = [ ] | a : [ a ]
                  ↓
      type of value
      our list contains.
      ↓
single type argument
to [ ]
  
```

marks the end of the list

rest of our list

$\gg : k []$

$\gg [] :: * \rightarrow *$

infix data constructor.

product of a $f[a]$

*) infix type + data constructors : Any operator that starts with a colon must be an infix type or data constructor.
All infix data constructors must start with a colon.

*) data $\text{list } a = \text{Nil} | \text{Cons}(a \text{ list } a)$

```

list type
constructor
  ↓
cons
constructor
  ↓
single value of
type a in the
Cons product
  ↓
  
```

→ The rest of
our list

*) let nil = Nil.

$\gg : t \text{ nil}$

$\text{nil} :: \text{list } a$ → type parameters isn't applied because Nil by itself doesn't tell the type inference what the list contains.

*) let oneItem = (Cons "wohoo!", Nil)

*) Type constructors are functions, one level up, structuring things that cannot exist at runtime - it's purely static

and describes the structure of types.

→ Binary Tree

(Binary Search Tree)

data Binary Tree a =

Leaf

| Node (Binary Tree a) a (BinaryTree a)
deriving ()

*) inserting into trees: first thing to be aware of is that we need **Ord** in order to have enough information about our values to know how to arrange them in our tree. This will allow us to follow the insertion algorithm for BST.

*) Important to remember that data is immutable in Haskell. Each time we want to insert a value into a tree, we build the whole tree.

insert' :: Ord a => a → Binary Tree a → Binary Tree a

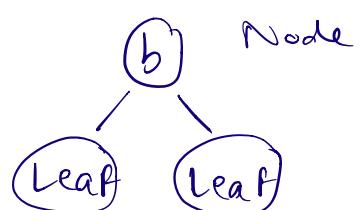
insert' b Leaf = Node Leaf(b) Leaf

insert' b (Node left a right)

value ↙ | b == a = Node left a right

ptr reference ↙ | b > a = Node left a (insert' b right)

 | b < a = Node (insert' b left) a right



*) let t1 = insert 0' Leaf

>> t1

Node Leaf 0 Leaf

0

>> let t2 = insert' 3 t1
 value
 reference ptr
 >> t2
 Node Leaf 0 (Node Leaf 3 Leaf)
 >> let t3 = insert' 2 t2
 >> t3
 Node Leaf 0 (Node (Node Leaf 2 Leaf) 3 Leaf)

$\text{mapTree} :: (a \rightarrow b)$
 → binaryTree a
 → binaryTree b.

$\text{mapTree} - \text{Leaf} = \text{Leaf}$

$\text{mapTree } f \text{ (Node left a right)} =$ mapping for current
 element
 ↑
 Node (mapTree f left) (f a) (mapTree f right)
 calling on left part

convert binary trees to lists

preorder
NLR

preorder :: binaryTree a → [a]

postorder

preorder Leaf = []

LRN

preorder (Node left a right) =

inorder

a : [(preOrder left) ++

(preOrder right)]

inorder :: binaryTree a → [a]

inorder Leaf = []

inorder (Node left a right) =

(inorder left) ++ [a] ++ (inorder

Right)

postOrder :: Binary Tree a → [a]

postorder Leaf = []

postorder (Node left right) =

(postorder left) ++ (postorder right)
++ [a]

→ foldr for Binary Tree

foldTree :: $(a \rightarrow b \rightarrow b) \rightarrow b \rightarrow \text{binary Tree } a \rightarrow b$

↓ ↓
 fn. to apply on current binary
 two arguments tree

leaf value

values to be returned

foldTree f z a = foldr f z (inorder a)

*> vigenere cipher

MEETATDOWN ALLY

$(\text{mod}((\text{ord } x - 97) + (\text{ord } i - 97))$
 $26), + 97$
 $(mppz ae oywz)$

vigenere [] - - = []

vigenere (x:xs) p i = (f x (p !! i)) :

vigenere xs p (mod (i+1) (length p))

f x i = chr \$(mod ((ord x - 97) + (ord i - 97)) 26)
+ 97

*) as - patterns : pattern match on part of something & still refer to the entire original value.

$f :: \text{Show } a \Rightarrow (a, b) \rightarrow \text{IO}(a, b)$

$f t @ (a, -) = \text{do}$

print a

return t

used @ to introduce a binding named 't' in order to refer to the entire original value.

① isSubseqOF :: (Eq a) $\Rightarrow [a] \rightarrow [a] \rightarrow \text{Bool}$

isSubseqOF [] _ = True dummyIndex

isSubseqOF _ [] = False data.maybe

isSubseqOF w @ (x:xs) (y:ys) data.list

| x == y = isSubseqOF xs ys

| otherwise = isSubseqOF x ys.

② capitalizeWords :: String $\rightarrow [(String, String)]$

capitalizeWords x = map capitalize \$ words x

capitalize :: String $\rightarrow (String, String)$

capitalize w @ (x:xs) = (w, toUpper x : xs)

① capitalizeWord :: String \rightarrow String

capitalizeWord (x:xs) = toUpper x : xs

② capitalizeParagraph :: String → String

"blah. woot ha."

"blah." → "Blah."

last

"woot" → "woot"

"ha." →

"blah.", "woot", "ha."

capitalizeParagraph x = concat \$ capitalize' \$ words x

capitalize' :: String → String

capitalize' w @ (x:xs)

| last w == '.' = toupper x:xs

| otherwise = w

capitalizeParagraph

[blah] [woot ha.]
↓
woot ha

takeWhile (== '.')

split' [] = []

split' x = takeWhile (/= '.') x :

split' (drop1 \$ left)

where left = dropWhile (/= '.') x

"blah. woot ha."

@ take

takewhile ($\ell = \cdot, \cdot$) \times

"blah"

"woot ha \emptyset "

"woot ha" : split