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 Bool = False | True
```

- `data` : keyword to signal that what follows is a data declaration or a declaration of a datatype
- `Bool` : Type constructor with no arguments
- `=` : separates type const. from data const.
- `False | True` : data constructor that takes no arguments, also called nullary constructor.

```
>> data [] a = [] | a : [a]
```

- `[] a` : type constructor with an argument
- `[]` : data constructor for empty list
- `a : [a]` : Data constructor that takes two arguments a, [a]

→ 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 typed 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 donot 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.

*) data Trivial = Trivial' → constant data constructor.

term level ↑

Trivial: constant at type level (type constants)

*) data UnaryTypeCon a = UnaryValueCon a

UnaryTypeCon: type constructor of one argument. constructor awaiting a type constant to be applied to.

UnaryValueCon: data constructor with one argument.

*) These donot behave like term level fns. in the sense of performing operations on data.

*) Not all type arguments to constructors have value-level witnesses, Some are phantom.

*) UnaryValueCon could show up as different literal values at runtime, depending on the type of `a` it is applied to

Int → 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've 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 `* -> *`, like a fn, it is waiting to be applied. for example

```
>> let f = not True
>> :t f
f :: Bool
```

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

```
>> let f x = x > 3
>> :t f
f :: (Ord a, Num a) => a -> Bool
```

→ This on the other hand is awaiting applm. 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 requires 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 '='.

↓

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

HuskyData is a constant value. ←

```
data DogueDeBordeaux doge = DogueDeBordeaux doge.
```

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

↓

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

```
>> :t DogueDeBordeaux
DogueDeBordeaux :: doge → DogueDeBordeaux doge

>> :t DogueDeBordeaux 'c'
DogueDeBordeaux 'c' :: DogueBordeaux Char
```

```
>> :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 = PugData.

myHusky = HuskyData

myDoge = DogueDeBordeaux 10
```

myDoge :: DogueDeBordeaux Int

↑ → agrees to the type variable being bound to

```
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 donot do anything except getting applied.

→ Exercises: Dog Types.

```
Doggies : Type Constructor
:k Doggies → * -> *
:k Doggies String → *
:t Husky 10 → Husky 10 :: Num a => Doggies a
:t Husky (10 :: Integer) → Husky 10 :: 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.

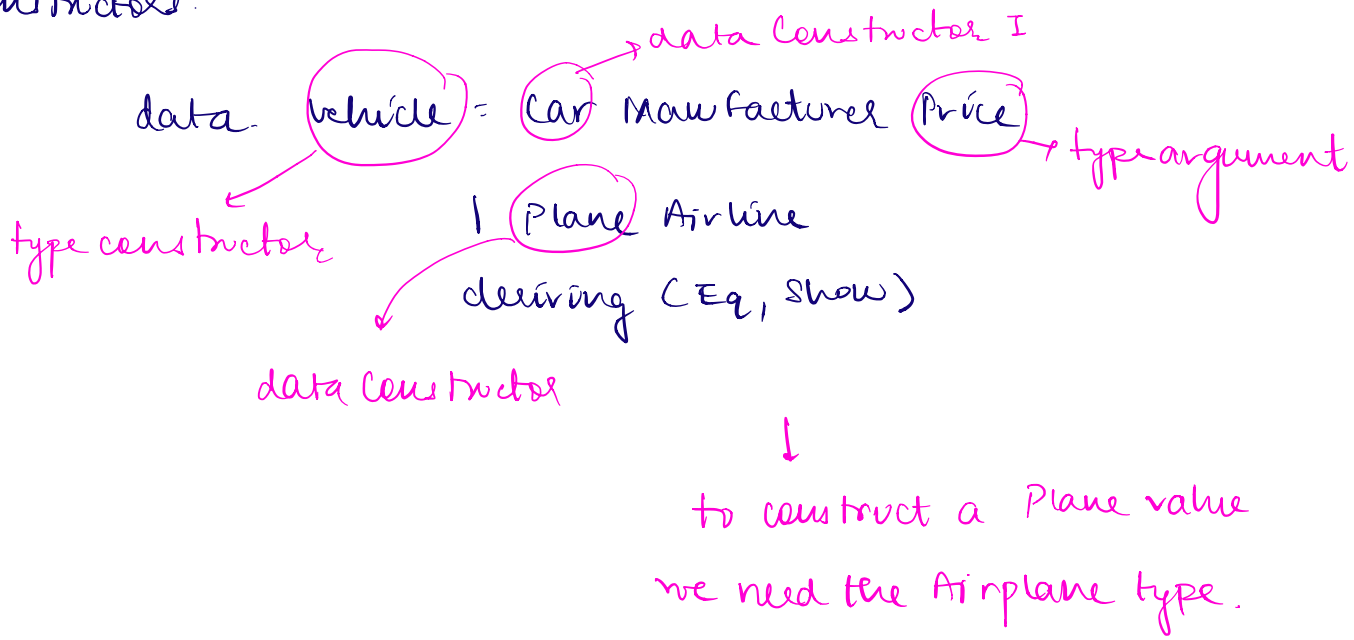
data Manufacturer = Mini | Mazda | Tata

deriving (Eq, Show)

data Airline = PapuAir | CatapultsR'Us | TakeYourChances

deriving (Eq, Show)

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



*) Here, the datatypes are generating the constructors

*) Data constructors take arguments, which are specific types, but not specific values

→ Exercises : Vehicles

(1)
```
isCar :: Vehicle → Bool
isCar (Car _ _) = True
isCar _ = False
```

(2)
```
isPlane :: Vehicle → Bool
isPlane (Plane _) = True
isPlane _ = False
```

(3)
```
areCars :: [Vehicle] → [Bool]
areCars xs = map isCar xs
```

④ getManu :: vehicle → Manufacturer

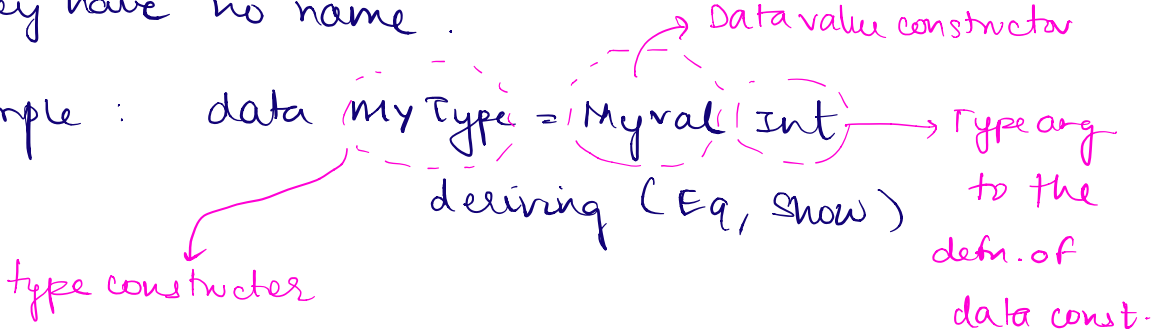
getManu (Car m _) = m

→ 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 type - and are considered the canonical product type; they are anonymous products because they have no name.

*) for example : data MyType = Myval Int deriving (Eq, Show)



→ 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 inhibit a type can help us reason about our programs.

*) Cardinality of Bool = 2 → 2 data constructors both nullary, hence only 2 possible values.

*) Cardinality of INT8 = 256 → values from (−128 to 127). INT8 is not included in the standard prelude & 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 & 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.

→ 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 & 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)
```

↓

safer, 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 is 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 typeclass 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 we still have to define it separately.

*) Adding the pragma →

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

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
  tooMany (x, y) = tooMany x

instance TooMany (Int, Int) where
  tooMany (x, y) = tooMany $ x + y
```

→ Sum types : '|' 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 =

```
  Num Int8          -> 256
| BoolyBool Bool    -> 2
                       } 258
  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 NegativeLiterals extension. This works with

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

→ 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, of possibly different data types, in a single value.

*) data QuantumBool = QuantumTrue
| QuantumFalse
| QuantumBoth
deriving (Eq, Show)

Cardinality : 3

data TwoQs =
MkTwoQs QuantumBool QuantumBool
deriving (Eq, Show)

Cardinality : 9

*) we could also have written the TwoQs type using a type alias and the tuple data constructor. Note: Type alias create type constructors, not data constructors

type TwoQs = (QuantumBool, QuantumBool)

→ **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 &
ca = MkPerson "chris" 16
```

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

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

↓

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

value constructor fields with named row accessors.

value constructor

→ there are now named row field accessors.

↓

These are just fns - 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"
```

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

⇒ 2 * (3 + 4) → sum of products

⇒ 2 * 3 + 2 * 4 = 14

*) 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 expre. 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 & 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 AutorName
  | NonFiction AuthorName
  deriving (Eq, Show)
```

product types.

*) Another common example

```
data Expr =
    Number Int
  | Add Expr Expr
  | Minus Expr
  | Mult Expr Expr
  | Divide Expr Expr
```

Sum of products

↓

hence normal form

*) 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 appln. in problems where one is writing fns. or folds over representations of datatypes, such as with generics & metaprogramming.

→ Exercises : How does your Garden Grow?

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
| Daily 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)

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

newtype NumPig =
NumPig Int

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, ofcourse until the compiler chokes.

```
>> newType NumSheep =
       NumSheep Int
       deriving (Eq, Show)

>> data BigFarmHouse =
       BigfarmHouse NumCow NumPig NumSheep
       deriving (Eq, Show)

>> type BigFarmHouse' =
       Product NumCow (Product NumPig NumSheeps)
```

↓ 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 ( )

data Animal = Cow CowInfo
            | Pig PigInfo
            | Sheep SheepInfo
              deriving ( )

type Animal' = Sum CowInfo ( Sum PigInfo SheepInfo)
```

CowInfo.
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 "Pully" 5 True)
>> let pully = Second p' :: Animal'
```

The location it needs to end up in the second loc at Animal' → first of second : PigInfo. 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 SocialNetwork, 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 =
    GnuPlusLinux
  | OpenBSD
```

```
    | Mac
    | Windows
    deriving (Show)

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

data Programmer =
  Programmer { os :: OperatingSystem
             , lang :: ProgLang }
  deriving (Show)

>> 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.

```haskell
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 10

yussss :: ThereYet
yussss = notQuite True
```

→ Deconstructing values.

newtype Name = Name String deriving()

newtype Acres = Acres Int deriving()

data FarmerType = DairyFarmer
| WheatFarmer
| SoybeanFarmer
deriving Show

data Farmer = Farmer Name Acres FarmerType
deriving Show

isDairyFarmer :: Farmer → Bool
isDairyFarmer (Farmer _ _ DairyFarmer) = True
isDairyFarmer _ = False.

---

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

```
isDairyFarmerRec :: FarmerRec → Bool

isDairyFarmerRec farmer =
    case farmerType farmer of
        DairyFarmer → True
        _           → False
```

fetching the record from the argument.

*) Another way of unpacking & unstructuring 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, $a \to b$, inhabitants = b^a.

So if Bool → 3 type. $3^2 = 9$ inhabitants and thus has nine possible implementations. i.e. i/p & o/p value combination.

$a \to b \to c \;\; \therefore \;\; (c^b)^a \Rightarrow c^{(b*a)}$

```
data Quantum = Yes
             | No
             | Both
             deriving ()
```

```
Either a b = Left a | Right b.
```

```
quantumSum1 :: Either Quantum Quantum
quantumSum1 = Right Yes.

quantumSum2 :: Either Quantum Quantum
quantumSum2 = Left Both.
```

(3 + 3) 6 different possible values.

```
quantProd1 :: (Quantum, Quantum)
quantProd1 = (Yes, Yes)

quantProd2 :: (Quantum, Quantum)
quantProd2 = (Yes, No)

quantProd3 :: (Quantum, Quantum)
quantProd3 = (Both, Yes)
```

3 * 3 possible values. (product type)

→ 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.

* → * : needs one argument to be fully applied

* → * → * : 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 deriving()
```

```
>> : kind Silly
Silly :: * → * → * → * → *

>> : kind Silly Int
Silly Int :: * → * → * → *

>> : kind Silly Int String Bool
Silly Int String Bool :: * → *

>> : kind Silly Int String Bool String
Silly Int String Bool String :: *

>> : kind (,,,)
(,,,) :: * → * → * → * → *

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

concrete value ↑ (fully applied)

*) 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 EsResultFound a =
    EsResultFound { _version :: DocVersion
                  , _source :: a
    } deriving ( )
```

we know that this particular response from Elastic search 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 donot 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]
```

- `[]` marks the end of the list
- `a` — type of value our list contains. ↓ single type argument to `[]`
- `:` — infix data constructor. product of a & [a]
- `[a]` → rest of our list

```
>> :k []
>> [] :: * → *
```

*) 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 List a = Nil | Cons a (List a)`

- `List` — list type constructor
- `Cons` — Cons Constructor
- `a` — single value of type a in the Cons product
- `(List a)` → The rest of our list

*) `let nil = Nil`

```
>> :t nil
nil :: List a
```

→ type parameter 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.

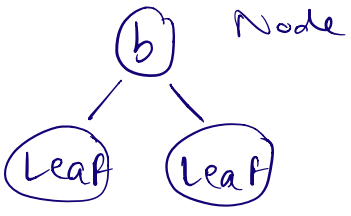
→ Binary Tree (Binary Search Tree)

```
data BinaryTree a =
    Leaf
  | Node (BinaryTree 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 -> BinaryTree a -> BinaryTree a
insert' b Leaf = Node Leaf b Leaf
insert' b (Node left a right)
    | b == a = Node left a right
    | 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
```

```
>> let t2 = insert' 3 t1
>> t2
Node Leaf 0 (Node Leaf 3 Leaf)
>> let t3 = insert' 2 t2
>> t3
Node Leaf 0 (Node (Node Leaf 2 Leaf) 3 Leaf)
```

```
mapTree :: (a -> b)
        -> binaryTree a
        -> binaryTree b

mapTree f
```

```
mapTree _ Leaf = Leaf
mapTree f (Node left a right) =
        Node (mapTree f left) (f a) (mapTree f right)
```

Convert binary trees to lists

preorder
NLR

postorder
LRN

inorder
LNR

```
preorder :: BinaryTree a -> [a]
preorder Leaf = []
preorder (Node left a right) =
        a : [(preOrder left) ++
             (preOrder right)]
```

```
inorder :: BinaryTree a -> [a]
inorder Leaf = []
inorder (Node left a right) =
        (inorder left) ++ [a] ++ (inorder
```

Right)

```
postOrder :: BinaryTree a -> [a]
postOrder Leaf = []
postOrder (Node left a right) =
        (postOrder left) ++ (postOrder right)
                ++ [a]
```

→ Folds for Binary Tree

```
foldTree :: (a -> b -> b) -> b -> binaryTree a -> b
```