# Haskell Overview III (3A)

Young Won Lim
10/11/16

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using OpenOffice.

# Based on

Haskell Tutorial, Medak & Navratil
ftp://ftp.geoinfo.tuwien.ac.at/navratil/HaskellTutorial.pdf

Yet Another Haskell Tutorial, Daume
https://www.umiacs.umd.edu/~hal/docs/daume02yaht.pdf

# Type Inference

```
Prelude> 7 :: Int
7
Prelude> 7 :: Double
7.0
```

usually don't have to declare types

(type inference)

to declare types, use :: to do it.

https://wiki.haskell.org/Learn_Haskell_in_10_minutes

# Type Information Display

```
Prelude> :t False
False :: Bool
Prelude> :t 'A'
'A' :: Char
Prelude> :t "Hello, world"
"Hello, world" :: [Char]
```

:t Print type information

https://wiki.haskell.org/Learn_Haskell_in_10_minutes

# Type Classes

```
Prelude> :t 42
42 :: (Num t) => t
Prelude> :t 42.0
42.0 :: (Fractional t) => t
Prelude> :t gcd 15 20
gcd 15 20 :: (Integral t) => t
```

42 can be used as any **numeric** type

42.0 can be any **fractional** type

Gcd 15 20 can be any **integral** type

https://wiki.haskell.org/Learn_Haskell_in_10_minutes

# Type Classe Constraint

```
Prelude> :t 42
42 :: (Num t) => t
Prelude> :t 42.0
42.0 :: (Fractional t) => t
Prelude> :t gcd 15 20
gcd 15 20 :: (Integral t) => t
```

**type** t belongs to Num **type class**

**type** t belongs to Fractional **type class**

**type** t belongs to Integral **type class**

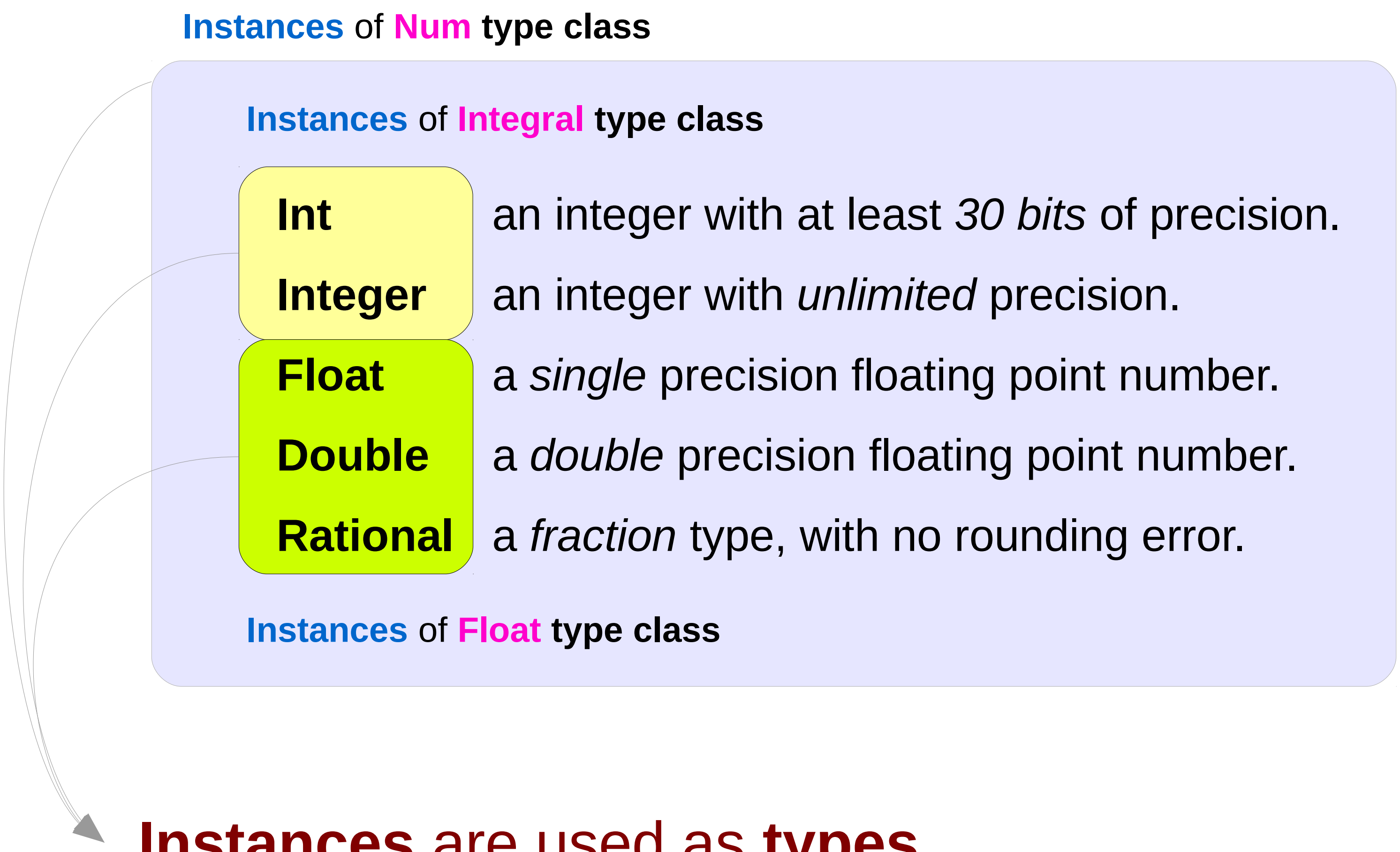
class constraint

the type t is *constrained* by the context
(Num t), (Fractional t), (Integral t)

| | |
|---|---|
| **(Num t) =>** | the **types** of t must be **Num type class** |
| **(Fractional t) =>** | the **types** of t must be **Fractional type class** |
| **(Integral t) =>** | the **types** of t must be **Integral type class** |

https://wiki.haskell.org/Learn_Haskell_in_10_minutes

# Instances



**Instances** of **Num** **type class**

**Instances** of **Integral** **type class**

| | |
|---|---|
| **Int** | an integer with at least *30 bits* of precision. |
| **Integer** | an integer with *unlimited* precision. |
| **Float** | a *single* precision floating point number. |
| **Double** | a *double* precision floating point number. |
| **Rational** | a *fraction* type, with no rounding error. |

**Instances** of **Float** **type class**

**Instances** are used as **types**

https://wiki.haskell.org/Learn_Haskell_in_10_minutes

# Type Class

a **type class** definition:

specifying
a set of
**functions** or **constants**,
together with their respective types,

that must be implemented
for every type that *should* *belong* to the **type class**

Like the Interface in Java

https://en.wikipedia.org/wiki/Type_class

# Type Class Definition

the **type class** Eq is intended to *include* those **types** that implement equality (==), (/=) functions

```
class Eq a where
      (==) :: a -> a -> Bool
      (/=)  :: a -> a -> Bool
```

a **type** **a** has an **instance** of the **class** **Eq** if there is an (overloaded) operation **==** and ***/*=** defined.

a **type** a *belongs* to the **type class** Eq

if  (**==**) and (***/*=**) functions are defined

https://en.wikipedia.org/wiki/Type_class

# Instance of a Class

**type class** Eq

*parameterized* **type** a

```
class Eq a where
      (==) :: a -> a -> Bool
      (/=)  :: a -> a -> Bool
```

a **type a** can be an **instance** of the **class Eq** if there is an (overloaded) operation **==** and **/=** defined.

The **type Integer** is an **instance** of the **class Eq**, whose **method** == and /= are defined

The **type Float** is an **instance** of the **class Eq**, whose **method** == and /= are defined

https://en.wikipedia.org/wiki/Type_class

# Instance Declaration

```
class Eq a where
    (==) :: a -> a -> Bool
```

| type class | type |
|---|---|
| Eq | a |

```
instance Eq Integer where
  x == y     = x `integerEq` y
```

```
instance Eq Float where
  x == y     = x `floatEq` y
```

| type class | instance |
|---|---|
| Eq | **Integer** |
| Eq | **Float** |

https://en.wikipedia.org/wiki/Type_class

# Default Method

```
class  Eq a  where
  (==), (/=)       ::  a -> a -> Bool
  x /= y           =   not (x == y)
```

If a method is not defined
in an instance declaration,
then the default implementation
defined in the class declaration,
if it exists, is used instead.

overloaded method definition

The default definition can be overloaded in an instance declaration

https://en.wikipedia.org/wiki/Type_class

# Class Constraint

```
elem :: a -> [a] -> Bool
```

the function elem has
the type a -> [a] -> Bool

```
elem :: (Eq a) => a -> [a] -> Bool
```

the type a is *constrained*
by the context (Eq a)

the **types** of a must *belong*
to the Eq **type class**

=> : called as a 'class constraint'

https://en.wikipedia.org/wiki/Type_class

# Class Constraint Example

elem function definition

elem function determines whether an element is in a list

```
elem :: (Eq a) => a -> [a] -> Bool
elem y [ ]       =   False
elem y (x:xs)   =   (x == y) || elem y xs
```

https://en.wikipedia.org/wiki/Type_class

# Enumerated Data Types

Type Constructor Data Constructor

```
data    Bool    =    True | False
```

The type being defined here is **Bool**, and it has exactly two values: True and False.

```
True  :: Bool
False :: Bool
```

```
var1 :: Bool
var1 = True

var2 :: Bool
var2 = False
```

```
data    Color    =    Red | Green | Blue
```

```
Red    :: Color
Green  :: Color
Blue   :: Color
```

```
var3 :: Color
var3 = Red

var4 :: Color
var4 = Green

var5 :: Color
var5 = Blue
```

https://www.haskell.org/tutorial/goodies.html

# Type Names and Constructor Functions

A nullary constructor:
takes no arguments

A multi-constructor

```
data        Bool        =        True   |  False
```

Type Constructor

Data Constructor

**Type name** : Bool
The name of new data type

Usually it appears in the linea concerning type information ( :: )

**Constructor function**
: True, False

Usually it appears in the lines concerning application ( = )

https://www.haskell.org/tutorial/goodies.html

# Data Constructor

Data constructors group values together and tag alternatives

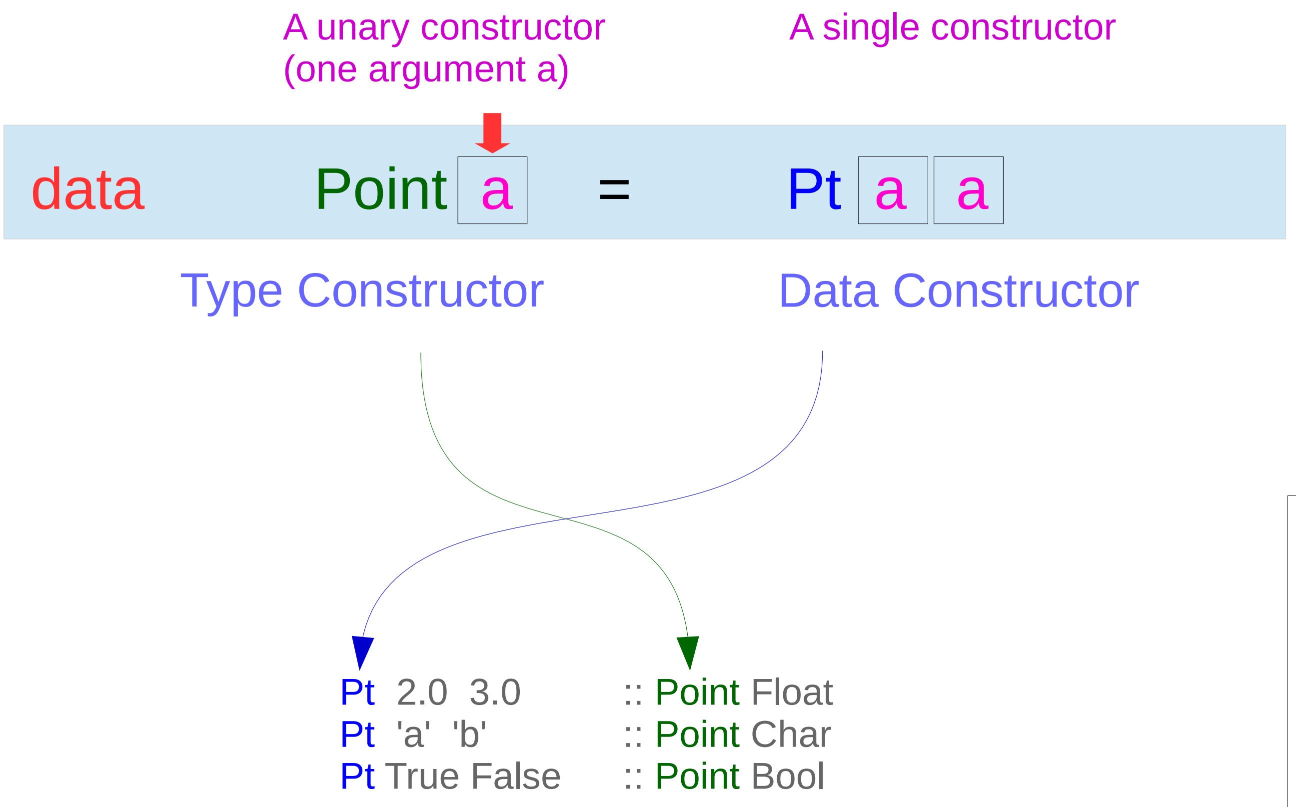
Deconstructing data constructors

- What a data constructor does is holding values together
- Have to separate them in order to use them.
- pattern matching **( )**

Data constructors are not types but values

https://wiki.haskell.org/Constructor

# Parameterized Data Type Definition



A unary constructor
(one argument a)

A single constructor

```
data        Point a   =   Pt a a
```

Type Constructor

Data Constructor

| | |
|---|---|
| Pt 2.0 3.0 | :: Point Float |
| Pt 'a' 'b' | :: Point Char |
| Pt True False | :: Point Bool |

```
v1    :: Point Float
v1    = Pt  2.0  3.0

v2    :: Point Char
v2    = Pt  ‘a’  ‘b’

v3    :: Point Bool
v3    = Pt True False
```

```
Pt :: a -> a -> Point a
```

https://www.haskell.org/tutorial/goodies.html

**Data constructors group values together and tag alternatives**

https://wiki.haskell.org/GADTs_for_dummies

# Polynom Data Type (1)

```
roots :: (Float, Float, Float) -> (Float, Float)
roots    (a,b,c)                    = if d < 0 then error "sorry" else (x1, x2)
      where   x1 = e + sqrt d / (2 * a)
              x2 = e - sqrt d / (2 * a)
              d = b * b - 4 * a * c
              e = - b / (2 * a)

real ::   (Float, Float, Float) -> Bool
real      (a,b,c)                   = (b*b - 4*a*c) >= 0

p1 = (1.0,2.0,1.0) :: (Float, Float, Float)
p2 = (1.0,1.0,1.0) :: (Float, Float, Float)
ps = [p1,p2]
newPs       = filter real ps
rootsOfPs   = map roots newPs
```

# Polynom Data Type (2)

```
data      Polynom      =   Poly    Float Float Float
```

| | |
|---|---|
| data | the keyword |
| Polynom | the name of the data type |
| Poly | the constructor function (:t Poly)<br>`Poly :: Float -> Float ->  Float ->  Polynom` |
| Float | the three arguments to the Poly constructor |

# Polynom Data Type (3)

```
data        Polynom        =   Poly    Float Float Float
```

```
roots' :: Float Float Float -> (Float, Float)
roots' a b c          = … function definition …
```

```
roots2  :: Polynom -> (Float, Float)
roots2  (Poly a b c) = … function definition …
```

(Float, Float) tuple

(Poly a b c) **pattern matching**

```
p1, p2   :: Polynom
p1 = Poly 1.0, 2.0, 3.0
p2 = Poly 1.0, 3.0, (-5.0)
```

# Recursive Definition of Lists

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

Any type is ok but
The type of every element in
the list must be the same

```
List         =        [ ]  |  (a : List)
```

an empty list

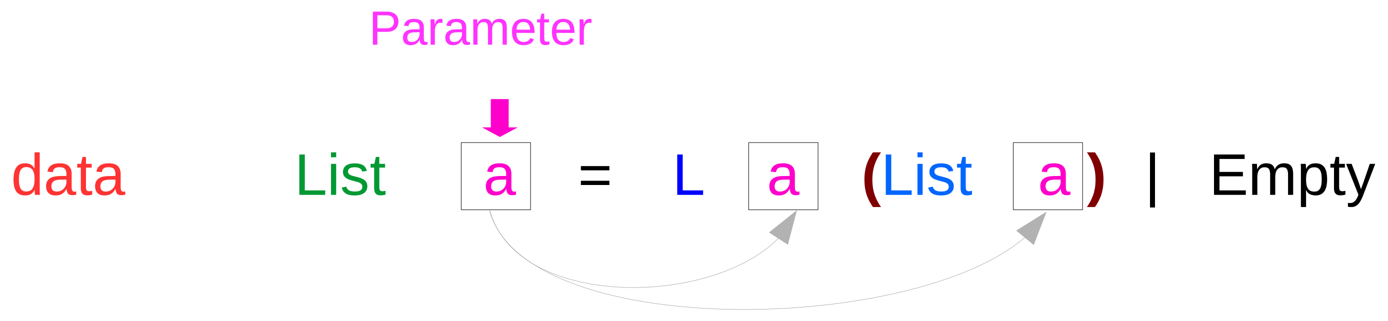
a list with at least one element

[ ]

(x:xs)

https://www.haskell.org/tutorial/goodies.html

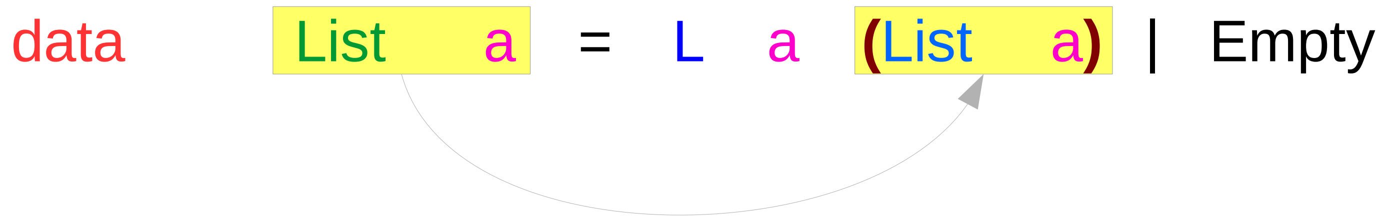
# List Type Definition : Parameterized & Recursive



Parameter

data List a = L a (List a) | Empty

Bool
Int
Integer
Float
Double
Char
String



data List a = L a (List a) | Empty

***Recursive Definition***

# List Type Definition : Constructors

Type Constructor

Data Constructor with two parameters

```
data    List  a  =  L  a  (List  a) | Empty
```

Data Constructors

```
Empty
L  a  (List  a)
```

```
L :: a -> List a -> List a
```

Head : element

Tail : list

# List Type Definition : Examples

```
data List a = L a (List a) | Empty
```

```
L1, L2, L3 :: List Integer
L1 = Empty
L2 = L 1 L1
L3 = L 5 L2
```

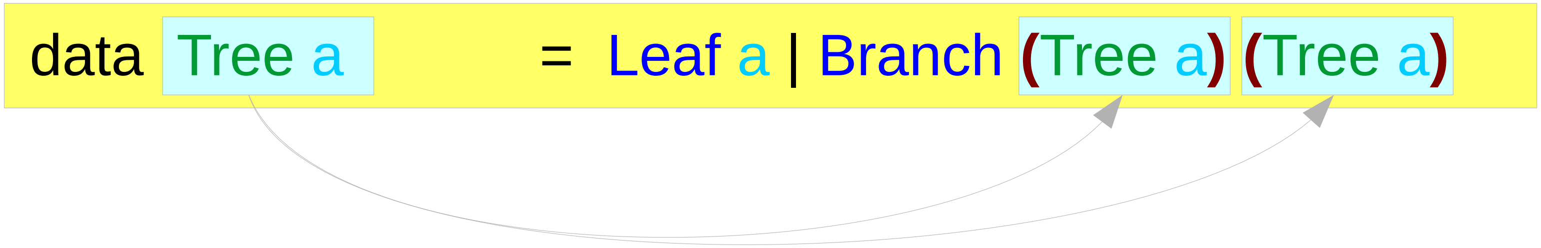
type constructor List

```
L4 = L 1.5 Empty :: List Double
```

type constructor List

data constructor
L a (List a)

# Tree Data Type : Recursive Definition



```
data Tree a        = Leaf a | Branch (Tree a) (Tree a)
```

*Recursive Definition*

**(Tree a) pattern matching**

https://www.haskell.org/tutorial/goodies.html

# Tree Data Type : Constructors

```
data Tree a        = Leaf a | Branch (Tree a) (Tree a)
```

Type Constructor : Tree a

Data Constructor 1: Leaf a

Leaf :: a -> Tree a

Data Constructor 2: Branch (Tree a) (Tree a)

Branch :: Tree a -> Tree a -> Tree a

Branch :: (~~Tree a~~) -> (~~Tree a~~) -> Tree a

https://www.haskell.org/tutorial/goodies.html

# Tree Data Type : Constructors

```
data Tree a          = Leaf a | Branch (Tree a) (Tree a)
```

```
fringe                     :: Tree a -> [a]
fringe (Leaf x)            =  [x]
fringe (Branch left right) =  fringe left ++ fringe right
```

https://www.haskell.org/tutorial/goodies.html

# Recursive Data Type Example (1)

```
data Bus = Start | Next (Bus) deriving Show

myBusA = Start

myBusB = Next (Next (Next (Start)))

myBusC = Next myBusB


plus :: Bus -> Bus ->  Bus

plus a Start = a

plus a (Next b) = Next (plus a b)


testBus :: Bus

testBus = plus myBusC myBusB
```

(Next b)
**parenthesis for pattern matching**

# Recursive Data Type Example (2)

```
data Bus = Start | Next (Bus) deriving Show
myBusA = Start
myBusB = Next (Next (Next (Start)))
myBusC = Next myBusB
          = Next ( Next (Next (Next (Start))) )
```

```
plus :: Bus -> Bus ->  Bus
plus a Start = a
plus a (Next b) = Next (plus a b)
```

```
plus myBusC myBusB
plus Next (Next (Next (Start))) Next (Next (Next (Next (Start))))
Next (plus Next (Next (Next (Start))) Next (Next (Next (Start))) )
Next (Next (plus Next (Next (Next (Start))) Next (Next (Start)) ))
Next (Next (Next (plus Next (Next (Next (Start))) Next (Start) )))
Next (Next (Next (Next (plus Next (Next (Next (Start)))  Start ))))
Next (Next (Next (Next (Next (Next (Next (Start))) ))))
```

# Recursive Data Type Example (3)

```
howFar :: Bus -> Int
howFar Start = 0
howFar (Next r) = 1 + howFar r

testInt :: Int
testInt = (+) (howFar myBusC) (howFar myBusB)
```

# Recursive Data Type Example (4)

**testInt = (+) (howFar myBusC) (howFar myBusB)**

**howFar** myBusC

**howFar** Next (Next (Next (Start)))
1 + **howFar** Next (Next (Start))
2 + **howFar** Next (Start)
3 + **howFar** Start
3

**howFar** :: Bus -> Int

**howFar** Start = 0

**howFar** (Next r) = 1 + **howFar** r

**howFar** myBusB

**howFar** Next (Next (Next (Next (Start))))
1 + **howFar** Next (Next (Next (Start)))
2 + **howFar** Next (Next (Start))
3 + **howFar** Next (Start)
4 + **howFar** Start
5

(+) 3 5

8

(Next r) **parens for pattern matching**

**(howFar** myBusC**) (howFar** myBusB**)**
**unnecessary parens in function call for readability**

# Anniversary Data Type (1)

```
data Anniversary =    Birthday String Int Int Int
                      | Wedding String String Int Int Int
```

```
Birthday    String            Int      Int               Int
--          name,             year,    month,            day
Wedding     String                     String                     Int      Int      Int
--          spouse name 1,             spouse name 2,             year,    month,   day
```

https://en.wikibooks.org/wiki/Haskell/Type_declarations

# Anniversary Data Type (2)

```
johnSmith :: Anniversary
johnSmith = Birthday "John Smith" 1968 7 3

smithWedding :: Anniversary
smithWedding = Wedding "John Smith" "Jane Smith" 1987 3 4

anniversariesOfJohnSmith :: [Anniversary]
anniversariesOfJohnSmith = [johnSmith, smithWedding]

anniversariesOfJohnSmith =
[Birthday "John Smith" 1968 7 3, Wedding "John Smith" "Jane Smith" 1987 3 4]
```

https://en.wikibooks.org/wiki/Haskell/Type_declarations

# Anniversary Data Type (3)

```
showDate :: Int -> Int -> Int -> String

showDate y m d = show y ++ "-" ++ show m ++ "-" ++ show d
```

```
showAnniversary :: Anniversary -> String

showAnniversary (Birthday name year month day) =
   name ++ " born " ++ showDate year month day

showAnniversary (Wedding name1 name2 year month day) =
   name1 ++ " married " ++ name2 ++ " on " ++ showDate year month day
```

**Deconstructing Types**

**( )** around the constructor name and the bound variables are mandatory

the expression inside **( )** is not a call to the constructor function

https://en.wikibooks.org/wiki/Haskell/Type_declarations

# Anniversary Data Type (4)

```
type Name = String

data Anniversary =
   Birthday Name Date
   | Wedding Name Name Date

data Date = Date Int Int Int   -- Year, Month, Day

johnSmith :: Anniversary
johnSmith = Birthday "John Smith" (Date 1968 7 3)

smithWedding :: Anniversary
smithWedding = Wedding "John Smith" "Jane Smith" (Date 1987 3 4)

type AnniversaryBook = [Anniversary]

anniversariesOfJohnSmith :: AnniversaryBook
anniversariesOfJohnSmith = [johnSmith, smithWedding]

showDate :: Date -> String
showDate (Date y m d) = show y ++ "-" ++ show m ++ "-" ++ show d

showAnniversary :: Anniversary -> String
showAnniversary (Birthday name date) =
   name ++ " born " ++ showDate date
showAnniversary (Wedding name1 name2 date) =
   name1 ++ " married " ++ name2 ++ " on " ++ showDate date
```

https://en.wikibooks.org/wiki/Haskell/Type_declarations

# Polymorphic Type

types that are universally quantified in some way over all types

essentially describe families of types

(forall a) [a] is the family of types consisting of,

for every type a, the type of lists of a.

- lists of integers (e.g. [1,2,3])
- lists of characters (['a','b','c'])
- lists of lists of integers, etc.
- [2,'b'] is not a valid example

https://www.haskell.org/tutorial/goodies.html

# Subset Polymorphism

```
roots      ::     (Floating a)   =>   (a, a, a)  ->    (a, a)
```

https://www.haskell.org/tutorial/goodies.html

# Parameterized Polymorphism

```
plus ::    a    ->   a    ->   a,
plus ::    Int  ->   Int  ->   Int,
plus ::    Rat  ->   Rat  ->   Rat,
```

```
data List a = L a (List a) | Empty
```

```
listlen :: List a -> Int
listlen Empty = 0
listlen (L _ list) = 1 + listlen list
```

(L _ list) **pattern matching**

_ : match with any element

https://www.haskell.org/tutorial/goodies.html

# ExplicitForAll

| | |
|---|---|
| **Just** :: a -> Maybe a | **Just** :: forall a. a -> Maybe a |
| **Nothing** :: Maybe a | **Nothing** :: forall a. Maybe a |
| **reverse** :: [a] -> [a] | **reverse** :: forall a. [a] -> [a] |
| **map** :: (a -> b) -> [a] -> [b] | **map** :: forall a b. (a -> b) -> [a] -> [b] |
| **show** :: (Show a) => a -> String | **show** :: forall a. (Show a) => a -> String |

to **explicitly** specify the universal quantification

in polymorphic type signatures.

https://www.schoolofhaskell.com/school/to-infinity-and-beyond/pick-of-the-week/guide-to-ghc-extensions/explicit-forall

# Type function X

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

```
Left :: a -> Either

Right :: b -> Either
```

```
isLeft (Left a) = True

isLeft (Right b) = False
```

```
type X a = Either a a
```

https://wiki.haskell.org/GADTs_for_dummies

# First Class Values

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

Data constructors are first class values in Haskell and actually have a type.

the type of the Left constructor of the Either data type is:

```
Left :: forall b a. a -> Either a b
```

first class values:

- may be passed to functions
- may make a list
- may be data elements of other algebraic data types
- and so forth

https://wiki.haskell.org/Constructor

# Show Class

Class Show

the instances of Show are those types
that can be converted to character strings.
(information about the class)

The function show

show :: (Show a) => a -> String

Similar to the toString() method in Java

https://www.haskell.org/tutorial/goodies.html