# 8 - Declaring Types and Classes

# Type Declarations

A new name for an existing type can be defined using a *type declaration*.

```
type String = [Char]
```

String is a synonym for the type [Char].

Type declarations can be used to make other types easier to read.

```
type Pos = (Int,Int)
```

we can define:

```
origin :: Pos
origin = (0,0)

left :: Pos → Pos
left (x,y) = (x-1,y)
```

Like function definitions, type declarations can also have *parameters*.

If

```
type Pair a = (a,a)
```

we can define:

```
mult :: Pair Int → Int
mult (m,n) = m*n

copy :: a → Pair a
copy x = (x,x)
```

Type declarations can also have multiple *parameters*.

If

```
type Assoc k v = [(k,v)]
```

we can define:

```
find :: Eq k ⇒ k → Assoc k v → v
find k t = head [v | (k‘,v)←t, k==k‘]
```

Type declarations can be nested:

```
type Pos = (Int,Int)

type Trans = Pos → Pos
```

However, they cannot be recursive:

```
type Tree = (Int,[Tree])
```

# Data Declarations

A completely new type can be defined by specifying its values using a *data declaration*.

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

Bool is a new type, with two new values False and True.

Note:

- The two values False and True are called the ***constructors*** for the type Bool.

- Type and constructor names must always begin with an upper-case letter.

- Data declarations are similar to context free grammars. The former specifies the values of a type, the latter the sentences of a language.

Values of new types can be used in the same ways as those of built in types. Given

```
data Answer = Yes | No | Unknown
```

we can define:

```
answers :: [Answer]
answers = [Yes,No,Unknown]

flip :: Answer → Answer
flip Yes     = No
flip No      = Yes
flip Unknown = Unknown
```

The constructors in a data declaration can also have parameters. Given

```
data Shape = Circle Float
           | Rect Float Float
```

we can define:

```
square :: Float → Shape
square n = Rect n n

area :: Shape → Float
area (Circle r) = pi * r^2
area (Rect x y) = x * y
```

Note:

- `Shape` has values of the form `Circle r` where `r` is a float, and `Rect x y` where `x` and `y` are floats.

- `Circle` and `Rect` can be viewed as *functions* that construct values of type Shape:

```
> :t Circle
Circle :: Float → Shape
> :t Rect
Rect :: Float → Float → Shape
```

Not surprisingly, data declarations themselves can also have parameters. Given

```
data Maybe a = Nothing | Just a
```

we can define:

```
safediv :: Int → Int → Maybe Int
safediv _ 0 = Nothing
safediv m n = Just (m `div` n)

safehead :: [a] → Maybe a
safehead [] = Nothing
safehead xs = Just (head xs)
```

# Recursive Types

New types can be declared in terms of themselves. That is, types can be ***recursive***.

```
data Nat = Zero | Succ Nat
```

Nat is a new type, with constructors Zero :: Nat and Succ :: Nat → Nat.

Note:

A value of type `Nat` is either `Zero`, or of the form `Succ n` where `n :: Nat`. That is, `Nat` contains the following infinite sequence of values:

```
Zero

Succ Zero

Succ (Succ Zero)

⋮
```

We can think of values of type `Nat` as *natural numbers*, where `Zero` represents 0, and `Succ` represents the successor function 1+.

```
Succ (Succ (Succ Zero))
```

represents the natural number

```
1 + (1 + (1 + 0))  =  3
```

Using recursion, it is easy to define functions that convert between values of type `Nat` and `Int`:

```
nat2int :: Nat → Int
nat2int Zero     = 0
nat2int (Succ n) = 1 + nat2int n


int2nat :: Int → Nat
int2nat 0 = Zero
int2nat n = Succ (int2nat (n-1))
```

Two naturals can be added by converting them to integers, adding, and then converting back:

```
add :: Nat → Nat → Nat
add m n = int2nat (nat2int m + nat2int n)
```

However, using recursion the function add can be defined without the need for conversions:

```
add Zero     n = n
add (Succ m) n = Succ (add m n)
```

For example:

```
  add (Succ (Succ Zero)) (Succ Zero)
=
  Succ (add (Succ Zero) (Succ Zero))
=
  Succ (Succ (add Zero (Succ Zero))
=
  Succ (Succ (Succ Zero))
```

Note:

The recursive definition for add corresponds to the laws 0+n = n and (1+m)+n = 1+(m+n).

# Exercises

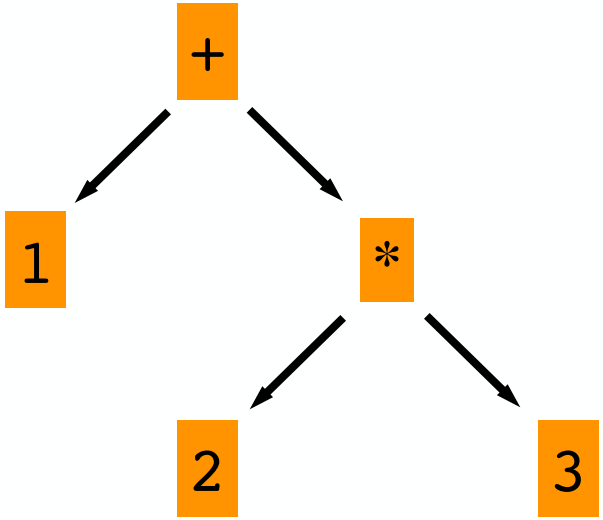
Using recursion and the function add, define a function that *multiplies* two natural numbers.

```
> mult (Succ (Succ Zero)) (Succ (Succ Zero))
Succ (Succ (Succ (Succ Zero)))
```

```
mult :: Nat -> Nat -> Nat
mult Zero     n = Zero
mult (Succ m) n = addp n (mult m n)
```

# Arithmetic Expressions

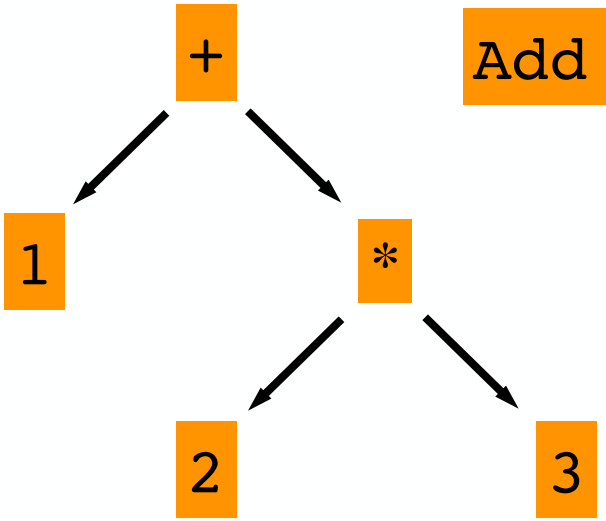
Consider a simple form of *expressions* built up from integers using addition and multiplication.



Using recursion, a suitable new type to represent such expressions can be declared by:

```
data Expr = Val Int
          | Add Expr Expr
          | Mul Expr Expr
```

The expression on the previous slide would be represented as follows:




```
Add (Val 1) (Mul (Val 2) (Val 3))
```

Define functions that process expressions.

```
size :: Expr → Int
```

```
size :: Expr → Int
size (Val n)   = 1
size (Add x y) = size x + size y
size (Mul x y) = size x + size y
```

```
eval :: Expr → Int
```

```
eval :: Expr → Int
eval (Val n)   = n
eval (Add x y) = eval x + eval y
eval (Mul x y) = eval x * eval y
```

Note:

The three constructors have types:

```
Val :: Int → Expr
Add :: Expr → Expr → Expr
Mul :: Expr → Expr → Expr
```

Many functions on expressions can be defined by replacing the constructors by other functions using a suitable **`fold`** function.

```
eval = folde id (+) (*)
```

# Exercises

Define a suitable function **`folde`** for expressions such that `eval = folde id (+) (*).`

```
> eval (Add (Val 1) (Mul (Val 2) (Val 3)))
7
```

```
folde :: (Int -> a) ->
         (a -> a -> a) ->
         (a -> a -> a) -> Expr -> a
folde f g h (Val n)   = f n
folde f g h (Add x y) = g (folde f g h x)
                          (folde f g h y)
folde f g h (Mul x y) = h (folde f g h x)
                          (folde f g h y)
```

# Exercises

Define the function **`nbExpr`** computing the number of values in an expression using `folde`

```
> nbExpr (Add (Val 1) (Mul (Val 2) (Val 3)))
3
```

```
nbExpr :: Expr -> Int
nbExpr =  folde (\_->1) (+) (+)
```

# Exercises

Define a type `Tree` a of binary trees built from `Leaf` values of type a using a `Node` constructor that takes two binary trees as parameters.

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

Define the functions

- `size`
- `occurs`
- `flatten`
- `occursST`

# Exercises

```
size :: Tree a -> Int
size (Leaf _) = 1
size (Node l _ r) = 1 + (size l) + (size r)
```

```
occurs :: Eq a => a → Tree a → Bool
occurs v (Leaf v') = v == v'
occurs v (Node l v' r) =   v == v'
                         || occurs v l
                         || occurs v r
```

# Exercises

```
flatten :: Tree a -> List a
flatten (Leaf v) = [v]
flatten (Node l v r) = flatten l
                     ++ [v]
                     ++ flatten r
```

```
occurs' :: Ord a => a → Tree a → Bool
occurs' v (Leaf v') = v == v'
occurs' v (Node l v' r) | v == v' = True
                        | v < v' = occurs' l
                        | v > v' = occurs' r
```

# Class and instance declarations

A ***class*** is like an interface that defines some behaviour: if a ***type*** is a part of a ***class***, it supports and implements the behaviour the class defines.

A type instance must support equality and inequality

```
data class Eq a where
  (==),(/=) :: a → a → Bool
  x /= y = ¬(x == y)
```

Default definition for inequality ⇒ an instance only requires a definition for ==

# Explicitly defined instances

We can made instances of a class explicitly.

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

```
instance Eq Bool where
    False == False = True
    True == True   = True
       _ == _      = False
```

Only types declared with `data` and `newtype` can be made instances of a class.

# Class extension

The class **Ord** of types whose values are totally ordered is declared in the standard library as an extension of the class **Eq**.

```
class  (Eq a) => Ord a  where
   (<), (<=), (>=), (>)  :: a -> a -> Bool
   max, min              :: a -> a -> a
   min x y | x <= y     = x
           | otherwise  = y
   max x y | x <= y     = y
           | otherwise  = x
```

# Instance declaration

An equality instance can be made an ordered instance by defining the corresponding operators.

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

```
instance Ord Bool where
    False < True = True
        _ < _    = False
        b <= c   = (b<c) || (b==c)
        b > c    = c < b
        b >= c   = c <= b
```

# Derived instances

New types can be made instances of `Eq, Ord, Show, Read`, using the **`deriving`** mechanism.

```
data Shape = Circle Float
           | Rect Float Float
     deriving (Eq, Ord, Show, Read)
```

Values are ordered lexicographically.

```
> Circle 1 == Circle 1
True
> Rect 1 5 < Rect 2 5
True
```

# Derived instances

For constructor types with arguments, the types of arguments should be instances of the derived class.

```
data Bool = False | True
     deriving (Eq, Ord, Show, Read)
```

All the functions in the derived classes can be used with values from `Bool`.

```
> False == False
True
> False < True
True
```