

# PROGRAMMING IN HASKELL



## Chapter 8 - Declaring Types and Classes

# Type Declarations

In Haskell, 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. For example, given

```
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. For example, given

```
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 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. For example, 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. For example, 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:

```
Circle :: Float → Shape
```

```
Rect :: Float → Float → Shape
```

Not surprisingly, data declarations themselves can also have parameters. For example, 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

In Haskell, 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  
 $\text{Zero} :: \text{Nat}$  and  $\text{Succ} :: \text{Nat} \rightarrow \text{Nat}$ .

Note:

- A value of type `Nat` is either `Zero`, or of the form `Succ n` where  $n :: \text{Nat}$ . 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  $1 +$ .

□ For example, the value

`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:

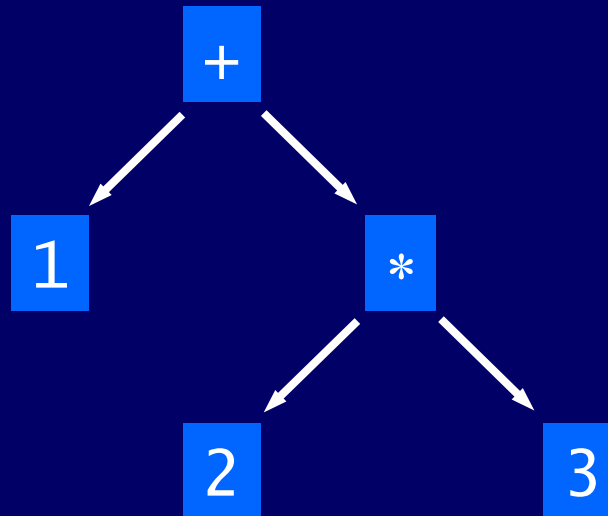
$$\begin{aligned} & \text{add (Succ (Succ Zero)) (Succ Zero)} \\ = & \text{Succ (add (Succ Zero) (Succ Zero))} \\ = & \text{Succ (Succ (add Zero (Succ Zero)))} \\ = & \text{Succ (Succ (Succ Zero))} \end{aligned}$$

Note:

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

# 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
```

For example, the expression on the previous slide would be represented as follows:

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

Using recursion, it is now easy to define functions that process expressions. For example:

```
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 (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:

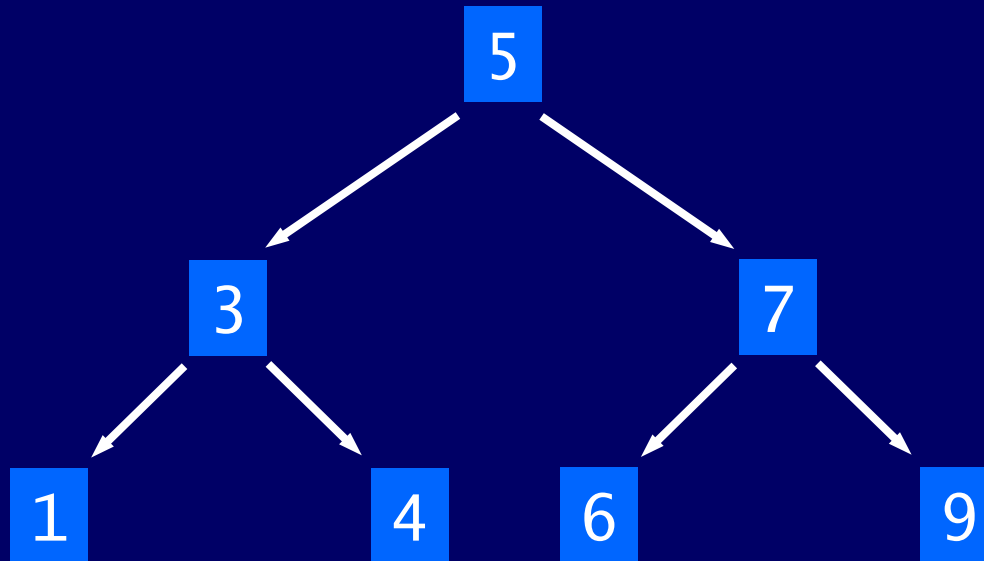
```
Val  :: Int → Expr
Add  :: Expr → Expr → Expr
Mul  :: Expr → Expr → Expr
```

- Many functions on expressions can be defined by replacing the constructors by other functions using a suitable fold function. For example:

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

# Binary Trees

In computing, it is often useful to store data in a two-way branching structure or binary tree.



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

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

For example, the tree on the previous slide would be represented as follows:

```
t :: Tree Int
t = Node (Node (Leaf 1) 3 (Leaf 4)) 5
        (Node (Leaf 6) 7 (Leaf 9))
```

We can now define a function that decides if a given value occurs in a binary tree:

```
occurs :: Ord a => a -> Tree a -> Bool
occurs x (Leaf y)      = x == y
occurs x (Node l y r) = x == y
                        || occurs x l
                        || occurs x r
```

But... in the worst case, when the value does not occur, this function traverses the entire tree.



Now consider the function flatten that returns the list of all the values contained in a tree:

```
flatten :: Tree a → [a]
flatten (Leaf x)      = [x]
flatten (Node l x r) = flatten l
                      ++ [x]
                      ++ flatten r
```

A tree is a search tree if it flattens to a list that is ordered. Our example tree is a search tree, as it flattens to the ordered list [1,3,4,5,6,7,9].

Search trees have the important property that when trying to find a value in a tree we can always decide which of the two sub-trees it may occur in:

```
occurs x (Leaf y)                = x == y
occurs x (Node l y r) | x == y = True
                      | x < y  = occurs x l
                      | x > y  = occurs x r
```

This new definition is more efficient, because it only traverses one path down the tree.