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

\[
\text{type Pair} \ a = (a, a)
\]

we can define:

\[
\text{mult :: Pair} \ \text{Int} \rightarrow \text{Int}
\]
\[
\text{mult} \ (m, n) = m \times n
\]
\[
\text{copy :: a} \rightarrow \text{Pair} \ a
\]
\[
\text{copy} \ x = (x, x)
\]
Type declarations can be nested:

```haskell
type Pos = (Int,Int)
type Trans = Pos → Pos
```

However, they cannot be recursive:

```haskell
type Tree = (Int,[Tree])
```
Data Declarations

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

\[
data \text{ Bool} = \text{False} \mid \text{True}
\]

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

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

we can define:

```haskell
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 \rightarrow Shape
square n = Rect n n

area :: Shape \rightarrow Float
area (Circle r) = \pi \times r^2
area (Rect x y) = x \times 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

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

we can define:

```haskell
safediv :: Int -> Int -> Maybe Int
safediv _ 0 = Nothing
safediv m n = Just (m `div` n)
```

```haskell
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.

\[
\text{data Nat} = \text{Zero} \mid \text{Succ Nat}
\]

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)

\ldots
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

\[ \text{Succ (Succ (Succ Zero))} \]

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:

\[
\begin{align*}
nat2int :: & \text{ Nat } \rightarrow \text{ Int} \\
nat2int & \text{ Zero} = 0 \\
nat2int \ (\text{Succ} \ n) & = 1 + nat2int \ n
\end{align*}
\]

\[
\begin{align*}
\text{int2nat ::} & \text{ Int } \rightarrow \text{ Nat} \\
\text{int2nat} & \ 0 = \text{ Zero} \\
\text{int2nat} \ n & = \text{ Succ} \ (\text{int2nat} \ (n-1))
\end{align*}
\]
Two naturals can be added by converting them to integers, adding, and then converting back:

\[
\text{add} :: \text{Nat} \to \text{Nat} \to \text{Nat} \\
\text{add} \ m \ n = \text{int2nat} \ (\text{nat2int} \ m + \text{nat2int} \ n)
\]

However, using recursion the function \text{add} can be defined without the need for conversions:

\[
\text{add} \ Zero \ n = n \\
\text{add} \ (\text{Succ} \ m) \ n = \text{Succ} \ (\text{add} \ m \ n)
\]
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 $(1+m)+n = 1+(m+n)$. 
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:

```haskell
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:

\[
\begin{align*}
\text{size} & : \text{Expr} \rightarrow \text{Int} \\
\text{size} (\text{Val} \ n) & = 1 \\
\text{size} (\text{Add} \ x \ y) & = \text{size} \ x + \text{size} \ y \\
\text{size} (\text{Mul} \ x \ y) & = \text{size} \ x + \text{size} \ y \\
\end{align*}
\]

\[
\begin{align*}
\text{eval} & : \text{Expr} \rightarrow \text{Int} \\
\text{eval} (\text{Val} \ n) & = n \\
\text{eval} (\text{Add} \ x \ y) & = \text{eval} \ x + \text{eval} \ y \\
\text{eval} (\text{Mul} \ x \ y) & = \text{eval} \ x \times \text{eval} \ y \\
\end{align*}
\]
Note:

- The three constructors have types:

\[
\text{Val} :: \text{Int} \rightarrow \text{Expr} \\
\text{Add} :: \text{Expr} \rightarrow \text{Expr} \rightarrow \text{Expr} \\
\text{Mul} :: \text{Expr} \rightarrow \text{Expr} \rightarrow \text{Expr}
\]

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

\[
\text{eval} = \text{fold}_\text{de} \text{id} (+) (*)
\]
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:

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

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

```hs
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:

```haskell
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 \texttt{flatten} that returns the list of all the values contained in a tree:

\begin{align*}
\text{flatten} &:: \text{Tree } a \rightarrow [a] \\
\text{flatten} (\text{Leaf } x) & = [x] \\
\text{flatten} (\text{Node } l \ x \ r) & = \text{flatten } l \\
& \quad + [x] \\
& \quad + \text{flatten } r
\end{align*}

A tree is a \textit{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:

\[
\begin{align*}
\text{occurs } x \ (\text{Leaf } y) &= x == y \\
\text{occurs } x \ (\text{Node } l \ y \ r) \ | \ x == y &= \text{True} \\
| \ x < y &= \text{occurs } x \ l \\
| \ x > y &= \text{occurs } x \ r
\end{align*}
\]

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

(1) Using recursion and the function add, define a function that multiplies two natural numbers.

(2) Define a suitable function fold for expressions, and give a few examples of its use.

(3) A binary tree is complete if the two sub-trees of every node are of equal size. Define a function that decides if a binary tree is complete.