Chapter 10 - 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

\[
\text{type \ Pos = (Int,Int)}
\]

we can define:

\[
\begin{align*}
\text{origin} & \quad :: \text{Pos} \\
\text{origin} & \quad = (0,0) \\
\text{left} & \quad :: \text{Pos} \rightarrow \text{Pos} \\
\text{left} \ (x,y) & \quad = (x-1,y)
\end{align*}
\]
Like function definitions, type declarations can also have parameters. For example, given

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

we can define:

```haskell
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 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

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

we can define:

```plaintext
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

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

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:

\[
\begin{align*}
\text{Circle} &:: \text{Float} \rightarrow \text{Shape} \\
\text{Rect} &:: \text{Float} \rightarrow \text{Float} \rightarrow \text{Shape}
\end{align*}
\]
Not surprisingly, data declarations themselves can also have parameters. For example, given

\[
\text{data } \text{Maybe } a = \text{Nothing} \mid \text{Just } a
\]

we can define:

\[
\begin{align*}
\text{safediv} & \quad :: \text{Int} \to \text{Int} \to \text{Maybe } \text{Int} \\
\text{safediv} \_ \_0 & = \text{Nothing} \\
\text{safediv} \ m \ n & = \text{Just } (m \ `\text{div}` \ n)
\end{align*}
\]

\[
\begin{align*}
\text{safehead} & \quad :: [a] \to \text{Maybe } a \\
\text{safehead} \ [] & = \text{Nothing} \\
\text{safehead} \ \text{xs} & = \text{Just } (\text{head } \text{xs})
\end{align*}
\]
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 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+.

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:

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

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

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

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:

\[
\begin{align*}
\text{add (Succ (Succ Zero)) (Succ Zero)} & = \\
\text{Succ (add (Succ Zero) (Succ Zero))} & = \\
\text{Succ (Succ (add Zero (Succ Zero)))} & = \\
\text{Succ (Succ (Succ Zero))} & = 
\end{align*}
\]

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:

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

- The three constructors have types:

  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 = fold 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 = Leaf Int
          | Node Tree Int Tree
```

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

```hs
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 integer occurs in a binary tree:

\[
\text{occurs} :: \text{Int} \rightarrow \text{Tree} \rightarrow \text{Bool}
\]

\[
\text{occurs } m \ (\text{Leaf } n) = m == n
\]
\[
\text{occurs } m \ (\text{Node } l \ n \ r) = m == n
\]
\[
\quad \text{ll occurs } m \ l
\]
\[
\quad \text{ll occurs } m \ r
\]

But... in the worst case, when the integer does not occur, this function traverses the entire tree.
Now consider the function `flatten` that returns the list of all the integers contained in a tree:

\[
\begin{array}{l}
\text{flatten} :: \text{Tree} \rightarrow [\text{Int}]
\text{flatten (Leaf n)} = [n]
\text{flatten (Node l n r)} = \text{flatten l}
\quad + [n]
\quad + \text{flatten r}
\end{array}
\]

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:

\[
\begin{align*}
\text{occurs } m \ (\text{Leaf } n) & \quad = m == n \\
\text{occurs } m \ (\text{Node } l \ n \ r) \quad & \quad \text{if } m == n \quad = \ True \\
& \quad \text{if } m < n \quad = \ \text{occurs } m \ l \\
& \quad \text{if } m > n \quad = \ \text{occurs } m \ r
\end{align*}
\]

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.