PROGRAMMING IN HASKELL

Chapter 10.1 - Declaring Types

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

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)
```

Type Declarations

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

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.

Type Declarations

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.

Type Declarations

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
```

Type Declarations

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
```

Type Declarations

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 <u>functions</u> that construct values of type Shape:

```
Circle :: Float → Shape

Rect :: Float → Float → Shape
```

Maybe

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

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

We use Maybe to handle errors. We can define:

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

Maybe

Most of what makes Haskell so powerful is based on the language being safe, predictable, and reliable. The traditional method of throwing an error is frowned upon in Haskell. Use of Maybe is one way to manage this:

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

Indicates an error

a is the type of the answer when no error is thrown

Either


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

Indicates an error and returns an a

b is the type of the answer when no error is thrown

We can similarly use Either to handle errors

```
safediv' :: Int -> Int -> Either String Int
safediv' n m = if m==0
               then Left "cannot divide by zero"
               else Right (n `div` m)
```

Either is increasingly being used in production Haskell

Handy function with Either

partitionEithers is a useful function in Haskell that separates a list containing Either values into two separate lists:

- A list of all the Left values
- A list of all the Right values

```
partitionEithers :: [Either a b] -> ([a], [b])
```

Example with partitionEithers


```
import Data.Either (partitionEithers)

exampleList :: [Either String Int]
exampleList = [Right 10, Left "Error A", Right 20, Left "Error B", Right 30]

main :: IO ()
main = do
    let (errors, results) = partitionEithers exampleList
    print errors
    print results
```

Results in

```
["Error A","Error B"]
[10,20,30]
```

Recursive Types

In Haskell, new types can be declared in terms of themselves. That is, types can be <u>recursive</u>.

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

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

Recursive Types

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)
```

⋮

Recursive Types

- 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`

Recursive Types

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))
```

Recursive Types

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)
```

Recursive Types

For example:

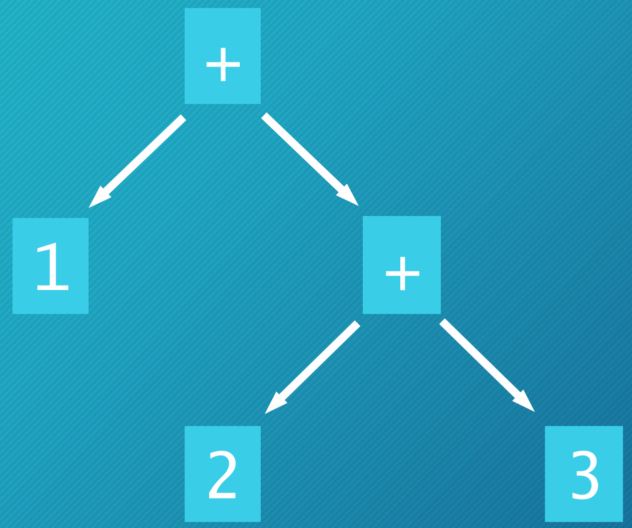
```
add (Succ (Succ Zero)) (Succ Zero)
=
Succ (add (Succ Zero) (Succ Zero))
=
Succ (Succ (add Zero (Succ Zero))
=
Succ (Succ (Succ Zero))
```

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.



Recursive Types

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

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

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

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

Recursive Types

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

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

Recursive Types

Note:

- The two constructors have types:

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

Recursive Types


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

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

```
value :: Expr → Int
value (Val v )    = n
value (Add x y)   = value x + value y
```

Recursive Types

For example the expression 1 + (2 + 3) is evaluated as follows:

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

```
value (Add (Val 1) (Add (Val 2) (Val 3))
= {applying value}
value (Val 1) + value (Add (Val 2) (Val 3))
= {applying the first value and  then second value}
  1 + value (Val 2) + value (Val 3)
= {applying values }
  1 + 2 + 3
```

Improving our expression evaluator

In this example, the order of evaluation of the expression was determined by Haskell (left to right).

We now look at encoding the controlling of which step should come next. We call this an

abstract machine

Improving our expression evaluator

We use a control stack for the abstract machine which contains the list of operations to be performed by the machine after the current evaluation has been completed.

```
type Cont = [Op]

data Op = EVAL Expr | ADD Int
```

We now define a function that evaluates an expression in the context of a control stack.

Improving our expression evaluator

We now define a function that evaluates an expression in the context of a control stack

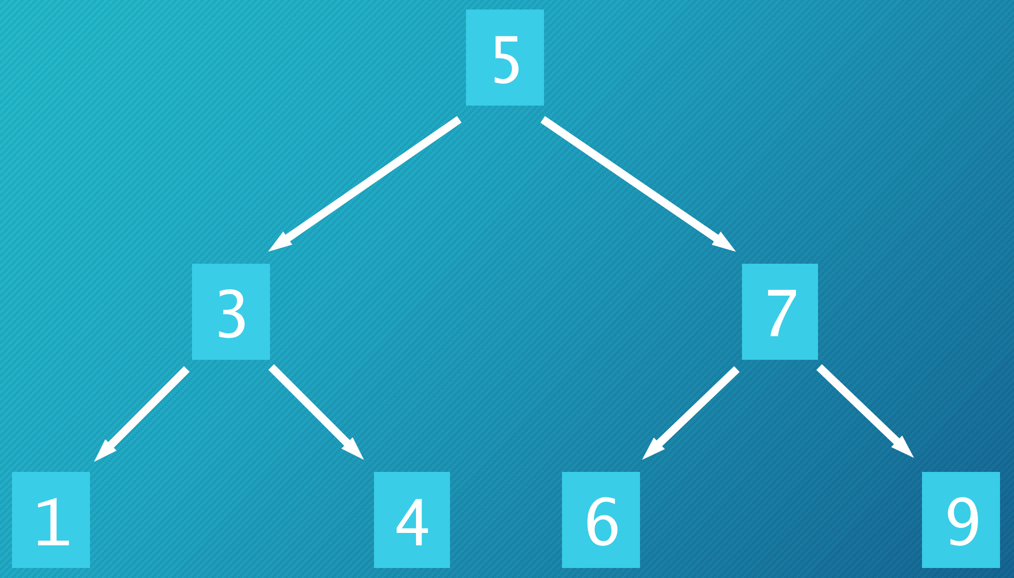
```
eval :: Expr -> Cont -> Int
eval (Val n)      c = exec c n
eval (Add x y)    c = exec x (EVAL y:c)
```

```
exec :: Cont -> Int -> Int
exec []               n   = n
exec (EVAL y : c)     n   = eval y (ADD n: c)
exec (ADD n: c)       m   = exec c (n + m)
```

```
value :: Expr-> Int
value e = eval e []
```

Binary Trees

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



Binary Trees

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))
```

Binary Trees

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.

Binary Trees

Now consider the function <u>flatten</u> 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 <u>search tree</u> 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].

Binary Trees

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.

Typeclasses in Haskell

- a means of defining the behaviour associated with a type separately from that type's definition.

- There are a number of typeclasses already defined in Haskell's base package.

- We look at
    - Eq
    - Ord
    - Num
    - Show (IO - later)
    - Read (IO later)

Typeclasses in Haskell - Eq

- All basic datatypes from Prelude except for functions and IO have instances of Eq.

- If a type instantiates Eq it means that we know how to compare two values for *value* or *structural* equality.

Typeclasses in Haskell - Eq

Required methods

```
(==) :: Eq a => a -a-> Boolean
(/=) :: Eq a => a -a-> Boolean
```

Typeclasses in Haskell - Ord

- If a type instantiates Ord it means that we know a "natural" ordering of values of that type.

Typeclasses in Haskell - Ord

Uses a custom algebraic data type:

```
data Ordering = LT | EQ | GT
```

Required methods:

```
compare :: Ord a => a -> a -> Ordering
```

or

```
(<=) :: Ord a => a -> a -> Boolean
```

(the standard's default *compare* method uses (<=) in its implementation)

Typeclasses in Haskell - Ord

Defines

```
compare :: Ord a => a-> a -> Ordering
(<) :: Ord a => a ->a-> Boolean
(>=) :: Ord a => a ->a-> Boolean
(<=) :: Ord a => a ->a-> Boolean
(>) :: Ord a => a ->a-> Boolean
max :: Ord a => a-> a -> a
min :: Ord a => a-> a -> a
```

Typeclasses in Haskell - Num

- The most general class for number types
- This class contains both
  - integral types **Int**, **Integer**, Word32 etc.) and
  - fractional types (**Double**, **Rational**, also complex numbers etc.)

Typeclasses in Haskell - Num

Defines

```
fromInteger :: Num a=> Integer -> a
(Converts an integer to the general Num type)
(+) :: Num a => a -> a-> a
(-) :: Num a => a -> a-> a
(*) :: Num a => a -> a-> a
negate :: Num a => a -> a
abs :: Num a => a -> a
signum :: Num a => a -> a
```

Typeclasses in Haskell - Num

See more at

https://wiki.haskell.org/Converting_numbers

Derived Instances

- When new types are declared, usually appropriate to make them instances of
  - Eq
  - Ord
  - Show
  - Read

  Using the deriving mechanism

Derived Instances

Show needed when you want to print values of the type to the terminal

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

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

```
> False < False
False
```


ANY
QUESTIONS?