

## Comp 205: Comparative Programming Languages

### User-Defined Types

- Enumerated types
- Parameterised types
- Recursive types

Lecture notes, exercises, etc., can be found at:  
[www.csc.liv.ac.uk/~grant/Teaching/COMP205/](http://www.csc.liv.ac.uk/~grant/Teaching/COMP205/)

## Introducing Constants

Often it is useful to declare constants in programs:

- to make the program easier to read, and/or
- to group related definitions together

In Java, we might declare:

```
public static final NORTH = 0;  
public static final EAST  = 1;  
public static final SOUTH = 2;  
public static final WEST  = 3;
```

## Turning 90°

Because the constants are just names for integers, code can be hard to follow:

```
public int turn(int direction)  
{  
    direction++;  
    if (direction < 0) || (3 < direction)  
        return NORTH;  
    return direction;  
}
```

## Turning 90° Again

Not referring to the representation (int), makes the code easier to follow:

```
public int turn(int direction)  
{  
    switch (direction)  
    { case NORTH: return EAST;  
      case EAST : return SOUTH;  
      case SOUTH: return WEST;  
      default  : return NORTH;  
    }  
}
```

## Putting on the Style

Haskell allows the programmer to define types:

```
data CardinalPoint  
    = North | East | South | West
```

The interpreter treats these as new values of a new type:

```
Main> :t East  
East :: CardinalPoint  
Main> East  
ERROR - cannot find "show" function ...
```

## Deriving Classes

```
data CardinalPoint  
    = North | East | South | West  
    deriving (Eq, Show)
```

```
Main> East  
East  
Main> East == West  
False
```

## Constructor Review

"[]" and ":" are constructors for lists:

- they are primitive operators (i.e., not evaluated)
- all lists are built from these operators (and elements of the "parameter type")

## Some Constructors

`Bool` : `True` and `False`

`[a]` : `[]` and `:`

`(a,b)` : `(_,_)`

`Char` : `'a'`, `'b'`, ...

## Enumerated Types

Types defined to have only constants are referred to as **enumerated types**:

```
data Colours = Red | Orange | Yellow |
              Green | Blue | Purple

-- Bool is a built-in enumerated type:
--
-- data Bool = True | False
```

## True and False

The type `Bool` is a built-in enumerated type, with constructors `True` and `False`; these can be used in pattern-matching:

```
not :: Bool -> Bool

not True  = False
not False = True
```

## Constant Constructors

The constants in enumerated types are constructors, and can be used in pattern-matching:

```
turn :: CardinalPoint -> CardinalPoint

turn North = East
turn East  = South
turn South = West
turn West  = North
```

```
Main> turn East
South
```

## Constructors with Args

Often we need types where the constructors take arguments:

```
data Point = Pt (Int,Int)
           deriving (Show)
```

```
Main> Pt (3,6)
Pt (3,6)
Main> :t Pt (3,6)
Pt (3,6) :: Point
main> :t Pt
Pt :: (Int,Int) -> Point
```

## Constructors with Args

Constructors can be used in pattern-matching:

```
data Point = Pt (Int,Int)
  deriving (Show)
```

```
getX :: Point -> Int
getX (Pt (x,y)) = x
```

```
Main> getX (Pt (3,6))
3
```

## Constructor Curry

We can use currying for constructors that take more than one argument:

```
data IntString = IS Int [Char]
```

```
getString :: IntString -> [Char]
getString (IS n s) = s
```

```
Main> getString (IS 7 "prosper")
prosper
```

## Type Parameters

Types can be defined over arbitrary types using type variables:

```
data ThingAtPoint a = At (a,Point)
  deriving Show
```

```
Main> :t At(0::Int, Pt(2,4))
... :: ThingAtPoint Int
Main> :t At("begin", Pt(2,4))
... :: ThingAtPoint [Char]
```

## More Type Parameters

Types that have more than one type parameter use more than one type variable:

```
data ThingAndList a b = TL a [b]
  deriving Show
```

```
getList :: ThingAndList a b -> [b]
getList (TL t l) = l
```

```
Main> getList (TL 7 "prosper")
prosper
```

## Polymorphism

Parameterised definitions allow polymorphic functions:

```
data IntList a = IL Int [a]
  deriving Show
```

```
mapList :: (a -> b) ->
  IntList a -> IntList b
mapList f (IL n l) = IL n (map f l)
```

```
Main> mapList ord (IL 7 "abc")
NL 7 [97,98,99]
```

## Summary

- Users (programmers) can define types by specifying constructors and their types
- these types are distinct from all other types (and constructor names should not be reused)
- definitions can be parameterised over types

*Next: more user-defined types  
(trees and catamorphisms)*