Lecture 3B:Part 2
Algebraic data types

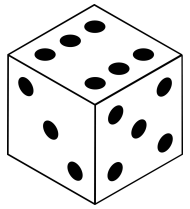
COMP 1100

Acknowledgement of Country

- ✓ *I wish to acknowledge the traditional custodians of the land we are meeting on, the Ngunnawal people. I wish to acknowledge and respect their continuing culture and the contribution they make to the life of this city and this region. I would also like to acknowledge and welcome any other Aboriginal and Torres Strait Islander people who are enrolled in our courses.*

Introducing algebraic types

- ✓ Algebraic data types (ADT) definition are introduced by the keyword `data`, followed by the type and and equals sign and then information about how elements are **constructed**.
- ✓ There names of the type and the constructors begin with capital letters.
- ✓ **Enumerated (or Sum) types**




```
data Die     = S1 | S2 | S3 | S4 | S5 | S6

data Coin    = Tail | Head
```

- ✓ **Product types**

```
data Outcome = Outcome Die Coin
```

```
S1 Tail  S2 Tail  S3 Tail  S4 Tail  S5 Tail  S6 Tail
S1 Head  S2 Head  S3 Head  S4 Head  S5 Head  S6 Head
```

Data declarations: Enumerated types

- ✓ The type `Bool` from the standard prelude

```
data Bool = False | True
```

The symbol | is read as *or*, and the new values of the type are called *constructors*.

- ✓ Equivalently

```
data A = B | C
```

The names are not relevant! The meaning is assigned by the programmer, via the functions that they define on new types.

Data declarations : Enumerated types

- ✓ New types can be used in the same way as built-in types
- ✓ **Example**: Given the declaration

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

functions that apply a move to a position

```
move :: Move -> Pos -> Pos
move North (x, y) = (x, y + 1)
move South (x, y) = (x, y - 1)
move East  (x, y) = (x + 1, y)
move West  (x, y) = (x - 1, y)
```

```
> a = (0, 0) :: Pos
> a
(0,0)
> b = move North a
> b
(0,1)
> b = move West (move North a)
> b
(-1,1)
```

Data declarations : Enumerated types

- ✓ The constructors in a data declaration can have arguments

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

The type `Shape` has values of the form `Circle r`, and `Rect h w`, where `r`, `x`, and `y` are floating-point numbers.

```
area :: Shape -> Float
area (Rect h w) = h * w
```

Data declarations : Enumerated types

- ✓ Define functions on shape such as to produce a square of a given size, and to calculate the area of a shape

```
square :: Float -> Shape
square n = Rect n n
```

```
> s = square 5
> area s
25.0
```

- ✓ The difference between normal functions and constructor functions is that the latter have no defining equations, and exists for the purposes of building pieces of data.

Data declarations : Enumerated types

- ✓ **QUIZ**: Implement a `area` function for `Circle`.

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

square :: Float -> Shape
square n = Rect n n

area :: Shape -> Float
area (Rect x y) = x * y
area (Circle r) = pi * r^2
```

```
> c = Circle 5
> :t c
c :: Shape
> area c
78.53982
```

Data declarations : Enumerated types

- When new types are declared, it is appropriate to make them into instance of a number of built-in classes.

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

- The ordering on the constructors of a type is determined by their position in its declaration. `False` appears before `True`.
- **Example**

```
> False == False
True
> False < True
True
> show False
"False"
> read "False" :: Bool
False
```

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

vs

```
data Shape = Circle Float
             | Rect Float Float
             deriving Show
```

```
> square 5 :: Shape
<interactive>:1:1: error:
• No instance for (Show Shape)
arising from a use of ‘print’
• In a stmt of an interactive
GHCi command: print it
```

```
> square 5 :: Shape
Rect 5.0 5.0
```

Data declarations: Maybe

- ✓ Data declarations themselves can also be parameterized.
- ✓ Standard prelude type

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

- ✓ We can think of values of type `Maybe` as being values of type `a` that may either fail or succeed.
- ✓ **Example**

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

```
> safediv 1 0
Nothing
> safediv 5 0
Nothing
> safediv 5 2
Just 2
```

Data declarations: Maybe

Data declarations : Product types

- ✓ Instead of using a tuple, a data type with a number of components can be defined as

```
data People = Person Name Age deriving (Eq, Show)
type Name = String
type Age = Int
```

- ✓ This type is often called a product type as it takes an element of Age × Name.
- ✓ Say $a \in$ `Age` and $n \in$ `Name`, then the element of `People` formed from a and n will be `Person`.

```
> p1 = Person "Ronnie" 14
> p1
Person "Ronnie" 14
> :t p1
p1 :: People
```

Data declarations : Product types

- An alternative definition of the type people is given by

```
type People = (Name, Age)
```

- Advantages of ADT
  - Each object of the type carries an explicit label of the purpose of the element (e.g. `Person`)
  - It is not possible to treat a pair consisting of a string and a number a a person. (should be constructed by the `Person` constructor)
  - The type will appear in any error message due to mis-typing (`type` might be expanded and disappear from an error message)
- Advantages of a tuple type
  - Using a tuple in polymorphic functions such as `fst`, `snd`, that take first and second element out of the tuple.

Quiz

- **Problem 1**: What are the types of the following functions

```
second xs = head (tail xs)
swap x y  = (y, x)
pair x y  = (x, y)
double x  = x*2
palindrome xs = reverse xs == xs
twice f x = f (f x)
```

HINT: take case to include the necessary class constraints in the types if the functions are defined using overloaded operators