Chapter 2

A Module of Shapes: Part I

Defining New Datatypes

- The ability to define new data types in a programming language is important.
- Kinds of data types:
  - *enumerated types*
  - *records* (or *products*)
  - *variant records* (or *sums*)
  - *recursive types*
- Haskell's `data` declaration provides these kinds of data types in a uniform way that abstracts away from their implementation details, by providing an abstract interface to the newly defined type.
- Before looking at the example from Chapter 2, let's look at some simpler examples.

The Data Declaration

- Example of an *enumeration* data type:

```
data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
     deriving Show
```

- The names **Sun through Sat** are constructor constants (since they have no arguments) and are the *only* elements of the type.
- For example, we can define:

```
valday :: Integer -> Day
valday 1 = Sun
valday 2 = Mon
valday 3 = Tue
valday 4 = Wed
valday 5 = Thu
valday 6 = Fri
valday 7 = Sat

Hugs> valday 4
```

Constructors & Patterns

- Constructors can be matched by *patterns*.
- For example:

```
dayval :: Day -> Integer
dayval Sun = 1
dayval Mon = 2
dayval Tue = 3
dayval Wed = 4
dayval Thu = 5
dayval Fri = 6
dayval Sat = 7

Hugs> dayval Wed
4
```

Other Enumeration Data Type Examples

```
data Bool = True | False  -- predefined in Haskell
     deriving Show
data Direction = North | East | South | West
     deriving Show
data Move = Paper | Rock | Scissors
     deriving Show

beats :: Move -> Move
beats Paper    = Scissors
beats Rock     = Paper
beats Scissors = Rock

Hugs> beats Paper
Scissors
```

Variant Records

More complicated data types:

```
data Tagger = Tagn Integer | Tagb Bool
            deriving Show
```

- These constructors are not constants – they are functions:

```
Tagn :: Integer -> Tagger
Tagb :: Bool    -> Tagger
```

- As for all constructors, something like "**`Tagn 12`**"
  - Cannot be simplified
    (and thus, as discussed in Chapter 1, it is a *value*).
  - Can be used in patterns.

Example functions on **Tagger**

```
number  (Tagn n) = n
boolean (Tagb b) = b

isNum (Tagn _) = True
isNum (Tagb _) = False

isBool x = not (isNum x)

Hugs> :t number
number :: Tagger -> Integer

Hugs> number (Tagn 3)
3

Hugs> isNum (Tagb False)
False
```

Another Variant Record Data Type

```
data Temp = Celsius  Float
          | Fahrenheit  Float
          | Kelvin  Float
```

- We can use patterns to define functions over this type:

```
toKelvin (Celsius c)    = Kelvin (c + 272.0)
toKelvin (Fahrenheit f) =
      Kelvin ( 5/9*(f-32.0) + 272.0 )
toKelvin (Kelvin k)     = Kelvin k
```

Finally: the **Shape** Data Type from the Text

- The **Shape** data type from Chapter 2 is another example of a variant data type:

```
data Shape = Rectangle Float Float
           | Ellipse Float Float
           | RtTriangle Float Float
           | Polygon [(Float,Float)]
deriving Show
```

- The last line – "**deriving Show**" – tells the system to build a **show** function for the type **Shape** (more on this later).
- We can also define functions yielding refined shapes:

```
circle, square :: Float -> Shape
circle radius = Ellipse radius radius
square side   = Rectangle side side
```

Functions over **shape**

- Functions on shapes can be defined using pattern matching.

```
area :: Shape -> Float
area (Rectangle s1 s2)  =  s1*s2
area (Ellipse r1 r2)    =  pi*r1*r2
area (RtTriangle s1 s2) = (s1*s2)/2
area (Polygon (v1:pts)) =  polyArea pts
   where polyArea :: [(Float,Float)] -> Float
         polyArea (v2:v3:vs) = triArea v1 v2 v3 +
                               polyArea (v3:vs)
         polyArea _          = 0
```

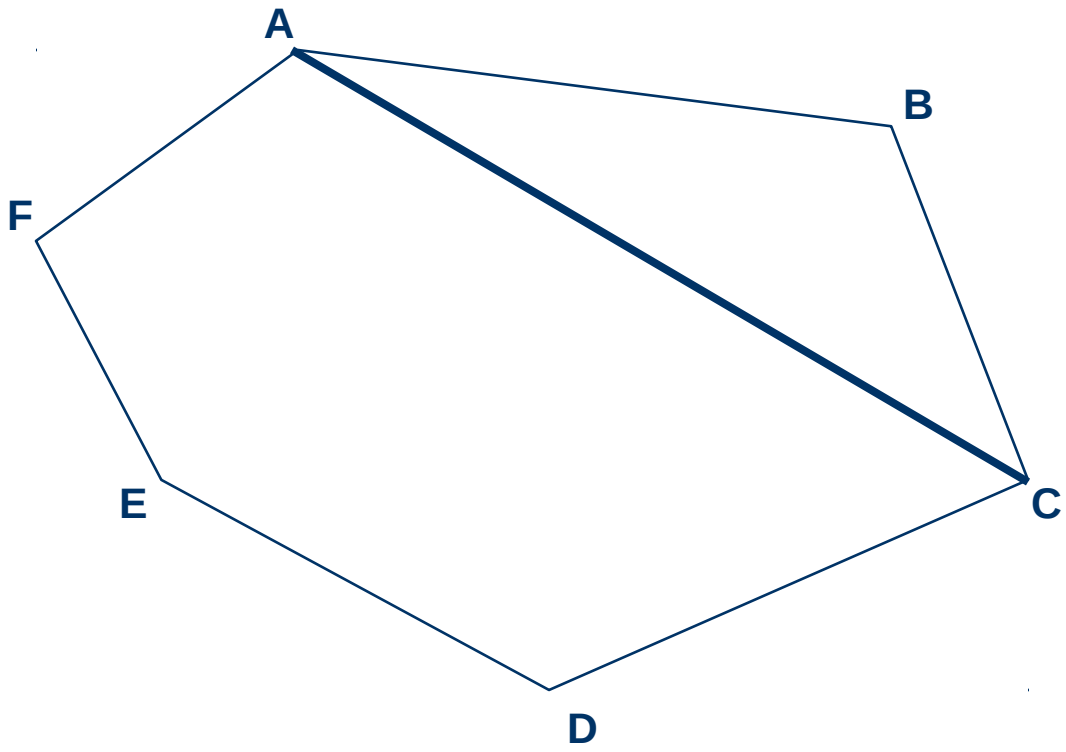
Note use of auxiliary function.

Note use of nested patterns.

Note use of wild card pattern (which matches anything).

Algorithm for Computing Area of Polygon

totalArea = area (triangle [A,B,C]) + area (polygon[A,C,D,E,F])



TriArea

```
triArea v1 v2 v3 =
  let a = distBetween v1 v2
      b = distBetween v2 v3
      c = distBetween v3 v1
      s = 0.5*(a+b+c)
  in sqrt (s*(s-a)*(s-b)*(s-c))


distBetween (x1,y1) (x2,y2)
   = sqrt ((x1-x2)^2 + (y1-y2)^2)
```