CS 457/557 Functional Programming

Lecture 2

Haskell Basics; Data Types (Shapes)

Booleans

- Type **`Boolean`** has constructors (values) **`True`** and **`False`**.
- Relational operators return booleans:

  `<  >  <=  >=  ==  /=`

- Combinators on booleans:

  `&&  ||  not`

- Examples

```
Prelude> 5 > 7
False
Prelude> 1==4
False
Prelude> 5 < 7 && 1 !=4
True
```

Definition by Cases

File contents:

```
absolute x | x < 0  = -x
           | x >= 0 = x
```

Session:

```
Main> absolute 3
3
Main> absolute (- 5)
5
Main>
```

Definition By Patterns

- Example on booleans

```
myand True False = False
myand True True = True
myand False False = False
myand False True = False
```

- The order of definition matters
  - Variables in Patterns match anything

```
myand2 True True = True
myand2 x y = False
```

  - What happens if we reverse the order of the two equations above?

Patterns On Lists

- File Contents

```
hd (x:xs) = x
tl (x:xs) = xs
firstOf3 [x,y,z] = x
```

- Hugs Session

```
Main> hd [1,2,3]
1

Main> firstOf3 [1,2,3]
1

Main> firstOf3 [1,2,3,4]
Program error: {firstOf3 [1, 2, 3, 4]}
```

Rules for Patterns

- All the patterns (on the left) must have compatible types
- The cases should (but are not required to) be exhaustive
- There should usually be no ambiguity as to which case applies
  - Ordering resolves ambiguity if there is any
- A Pattern is:
  - A variable
  - A constructor applied to patterns
  - A constant, like: `3` or `[]`
  - A tuple of patterns

Recursion

- To obtain universal computing power, we need a way to repeat computations
- Most languages support loops, e.g.

```
while b do s
```

- These are completely useless in a purely functional program (why?)
- Functional programs use recursion instead of iteration.
  - Each recursive activation of function has new set of formal parameters
  - Function can compute something different each time it is called!

Some Recursive Functions

```
plus :: Int -> Int -> Int
plus x y = if x == 0 then y
           else 1 + (plus (x-1) y)

len :: [a] -> Int
len [] = 0
len (x:xs) = 1 + (len xs)

even, odd :: Int -> Bool
even 0 = True
even n = odd(n-1)
odd 0 = False
odd n = even(n-1)
```

Local Definitions

- Local Definitions: Where

```
len2 [] = 0
len2 (x:xs) = one + z
  where z = len2 xs
        one = 1
```

The order of (local) definitions does not matter - even if they depend on each other

- Indentation matters !

```
where
f x y = ....
  where
  g a = ......
  h a b = ....
k u v w = .....
```

Functions g and h are local to the definition of f. The function k is at the same level as f

- Rule of thumb
  - Definitions at the same scope level should be indented equally far

Larger example: Sorting

- Insertion sort

```
sort [] = []
sort (h:t) = insert h (sort t)
   where
   insert x [] = [x]
   insert x (h:t)
            | x <= h    = x:h:t
            | otherwise  = h:(insert x t)
```

Types of Errors

- Syntax errors (use of symbols in unexpected places)

```
f x = (x*0) = x
```

```
Main> :r
Reading file "lect01.hs":
ERROR "lect01.hs" (line 36): Syntax error in input
  (unexpected `=')
Main>
```

- Undefined variables

```
lastone2  = head . rev
```

```
Main> :r
Reading script file "lect01.hs":
ERROR "lect01.hs" (line 38): Undefined variable "rev"
Main>
```

Types of Errors (cont.)

Type errors

```
plusone :: Int -> Int
```

```
Main>  plusone [1,2]
ERROR: Type error in application
*** expression     : plusone [1,2]
*** term           : [1,2]
*** type           : [Int]
*** does not match : Int
```

Types of Errors (cont.)

- More type errors

```
Prelude> :t head
head :: [a] -> a

Prelude> head True
ERROR: Type error in application
*** expression     : hd True
*** term           : True
*** type           : Bool
*** does not match : [a]

Prelude> head 1
ERROR - Illegal Haskell 98 class constraint in inferred type
*** Expression : head 1
*** Type       : Num [a] => a
```

Overloading and Classes

- If we omitted the type contraint on difference:

```
difference x y = if x>=y then x-y else y-x
```

```
Main> :type difference
difference :: (Ord a, Num a) => a -> a -> a

Main> difference 3 4
-1

Main> difference 4.5 7.8
-3.3
Main>
```

The classes Ord and Num represent sets of types

Defining New Datatypes

- Ability to add new datatypes in a programming language is important.
- Some important kinds of datatypes
  - enumerated types
  - records (or products)
  - variant records (or sums)
- Haskell's **`data`** declaration provides many of these kinds of types in a uniform way which abstracts from their implementation details.
- The **`data`** declaration defines a new **type** and a set of **constructors** for that type. Constructors can be used to create new instances of the type and to pattern-match against existing instances.

Records Example: Complex Numbers

- Complex numbers in rectangular format

```
data Complex = Complex Float Float
   deriving Eq

add (Complex r1 i1) (Complex r2 i2) =
     Complex (r1+r2) (i1+i2)
mul (Complex r1 i1) (Complex r2 i2) =
     Complex (r1*r2-i1*i2) (r1*i2+r2*i1)
```

- Note that **`Complex`** is used both to construct new values and to pattern-match existing ones.
- Deriving clause automatically generates a structural equality operator

```
(Complex 2.0 3.0 == Complex 2.0 3.0)  ==> True
```

Enumerations Example

```
data Month = Jan | Feb | Mar | Apr | May | Jun |
             Jul | Aug | Sep | Oct | Nov | Dec
     deriving (Eq,Enum,Show)

daysInMonth Sep = 30
daysInMonth Apr = 30
daysInMonth Jun = 30
daysInMonth Nov = 30
daysInMonth Feb = 28
daysInMonth _   = 31

Main> [Jun .. Sep]
[Jun,Jul,Aug,Sep]
```

Variant Records

```
data Temp = F Float  -- Fahrenheit
          | C Float  -- Celsius
          | K Float  -- Kelvin

freezing :: Temp -> Bool
freezing (F t) = t <= 32.0
freezing (C t) = t <= 0.0
freezing (K t) = t <= 273.0

kelvenize :: Temp -> Temp
kelvenize (K t) = (K t)
kelvenize (C t) = (K (273.0 + t))
kelvenize (F t) = kelvenize (C ((t-32.0)/9.0*5.0))
```

Maybe is another useful variant

```
data Maybe a = Just a
             | Nothing

lookup :: a -> [(a,b)] -> Maybe b
lookup k []                          = Nothing
lookup k ((k',v):rest) | k == k'     = Just v
                       | otherwise = lookup k rest
```

Shape types from the Text

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

- Deriving Show
  - tells the system to build a **show** function for the type **Shape**
- Using Shape - Functions returning shape objects

```
circle radius = Ellipse radius radius
square side = Rectangle side side
```

- Is there anything unsatisfactory about this definition?

Functions over Shape

- Functions over **Shape** 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
```

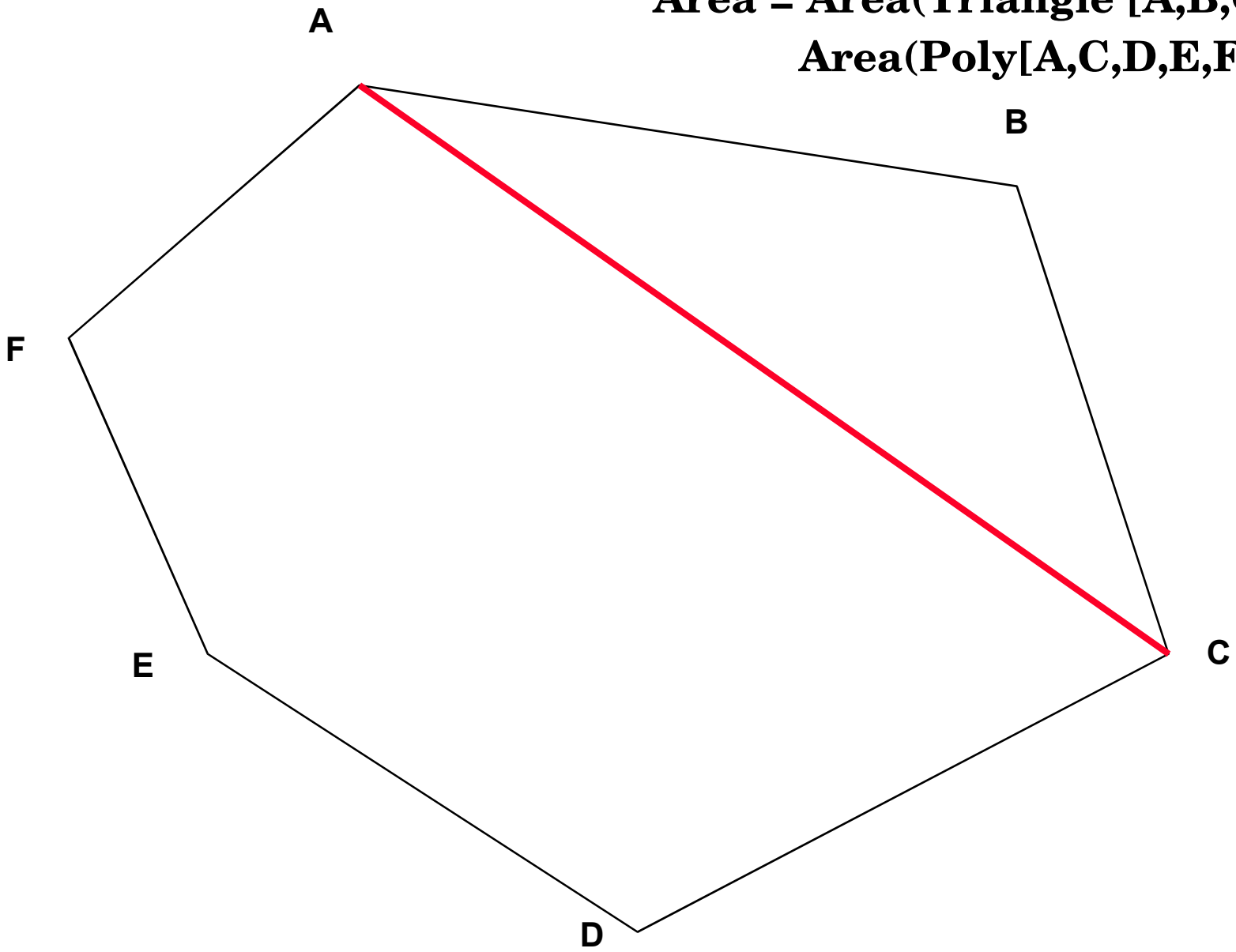
Note use of prototypes

Note use of nested patterns

Note use of wild card pattern (matches anything)

Poly = [A,B,C,D,E,F]

Area = Area(Triangle [A,B,C]) + Area(Poly[A,C,D,E,F])



Calculating areas

```
area (Polygon [A,B,C,D,E,F]) ==>
polyArea [B,C,D,E,F] where v1 = A ==>
triArea A B C +
    (polyArea [C,D,E,F] where v1 = A) ==>
triArea A B C +
    (triArea A C D +
         (polyArea [D,E,F] where v1 = A)) ==>
triArea A B C +
    (triArea A C D +
         (triArea A D E +
             (polyArea [E,F] where v1 = A))) ==>
triArea A B C +
     (triArea A C D +
          (triArea A D E +
              (triArea A E F +
                  (polyArea [F] where v1 = A)))) ==>
triArea A B C + (triArea A C D + (triArea A D E +
                                  (triArea A E F + 0)))
```

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)
```