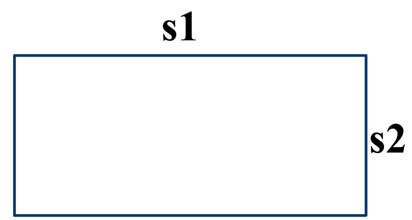
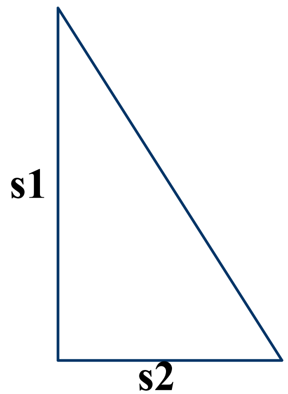
CS 457/557 Functional Programming

Lecture 6

Perimeters of Shapes

The Perimeter of a Shape





- To compute the perimeter we need a function with four equations (1 for each `Shape` constructor).
- The first three are easy …

```
perimeter :: Shape -> Float
perimeter (Rectangle  s1 s2) = 2*(s1+s2)
perimeter (RtTriangle s1 s2) =
                         s1 + s2 + sqrt (s1^2+s2^2)
perimeter (Polygon pts)      =
                         foldl (+) 0 (sides pts)
```

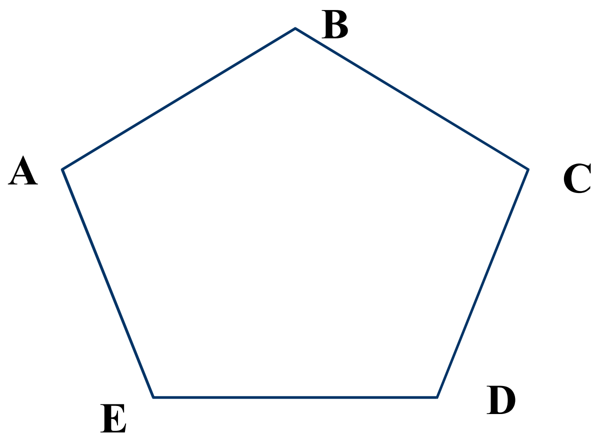
- This assumes that we can compute the lengths of the sides of a polygon. This shouldn't be too difficult since we can compute the distance between two points with `distBetween`.

Recursive Def'n of `Sides`

```
sides         :: [Vertex] -> [Side]
sides  []     = []
sides (v:vs) = aux v vs
  where
   aux v1 (v2:vs') = distBetween v1 v2 : aux v2 vs'
   aux vn []       = distBetween vn v  : []
-- aux vn []       = [distBetween vn v]
```

- But can we do better? Can we remove the direct recursion, as a seasoned functional programmer might?

Visualize What's Happening



- The list of vertices is: `vs = [A,B,C,D,E]`
- We need to compute the distances between the pairs of points `(A,B)`, `(B,C)`, `(C,D)`, `(D,E)`, and `(E,A)`.
- Can we compute these pairs as a list?
  `[(A,B),(B,C),(C,D),(D,E),(E,A)]`
- Yes, by "zipping" the two lists:
  `[A,B,C,D,E]` and `[B,C,D,E,A]`
  as follows:
  `zip vs (tail vs ++ [head vs])`

Zipping Lists

- The zip function (already in the library) can be written:

```
zip :: [a] -> [b] -> [(a,b)]
zip (x:xs) (y:ys) = (x,y):(zip xs ys)
zip _       _     = []
```

  – What happens if the lists are of unequal length?

- This leads to a new version of **sides**

```
sides       :: [Vertex] -> [Side]
sides vs = map d (zip vs (tail vs ++ [head vs]))
           where d (v1,v2) = distBetween v1 v2
```

- This is more elegant than the explicit recursion, but still verbose; in particular, the need to define **d** is sad. We can avoid this in at least two ways.

More variants of `sides`

I. The predefined `uncurry` function converts any curried binary function or operator to a single-argument version on pairs:

```
uncurry :: (a -> b -> c) -> (a,b) -> c
uncurry f (x,y) = f x y
```

allowing us to write

```
sides vs = map (uncurry distBetween)
               (zip vs (tail vs ++ [head vs]))
```

II. There is a predefined function `zipWith` that is just like `zip` except that it applies its first argument (a curried function) to each pair of values. For example:

```
zipWith (+) [1,2,3] [4,5,6] = [5,7,9]
```

So we can write

```
sides vs = zipWith distBetween
                   vs (tail vs ++ [head vs])
```

Perimeter of an Ellipse

There is one remaining case: the *ellipse*. The perimeter of an ellipse is given by the summation of an infinite series. For an ellipse with radii $r_1 > r_2$:

$$p = 2\pi r_1(1 - \Sigma\, s_i)$$

where $s_1 = 1/4\, e^2$

$$s_i = \frac{s_{i-1}\,(2i-1)(2i-3)\, e^2}{4i^2} \quad \text{for } i >= 1$$

$$e = \mathrm{sqrt}\,(r_1^2 - r_2^2) / r_1$$

Given s_i, it is easy to compute s_{i+1}.

Computing the Series

```
nextEl:: Float -> Float -> Float -> Float
nextEl e s i = s*(2*i-1)*(2*i-3)*(e^2) / (4*i^2)
```

$$s_{i+1} = \frac{s_i\,(2i\text{-}1)(2i\text{-}3)\,e^2}{4i^2}$$

Now we want to compute $[s_1, s_2, s_3, \ldots]$.

To fix **e**, let's define:

```
aux s i = nextEl e s i
```

So, we would like to compute:

$$
\begin{array}{l}
[s_1, \\
\ \ s_2 = f\ s_1\ 2, \\
\ \ s_3 = f\ s_2\ 3 = f\ (f\ s_1\ 2)\ 3, \\
\ \ s_4 = f\ s_3\ 4 = f\ (f\ (f\ s_1\ 2)\ 3)\ 4, \\
\ \ \ldots \\
]
\end{array}
$$

Can we capture this pattern?

Scanl (scan from the left)

- Yes, using the predefined function **scanl**:

```
scanl :: (a -> b -> a) -> a -> [b] -> [a]
scanl f seed  []     = seed : []
scanl f seed (x:xs) = seed : scanl f newseed xs
      where newseed =  f seed x
```

- For example:

```
  scanl (+) 0 [1,2,3]
  = [ 0,
      (+) 0 1, -- = 1
      (+) 1 2, -- = 3
      (+) 3 3 ] -- = 6
  = [ 0, 1, 3, 6 ]
```

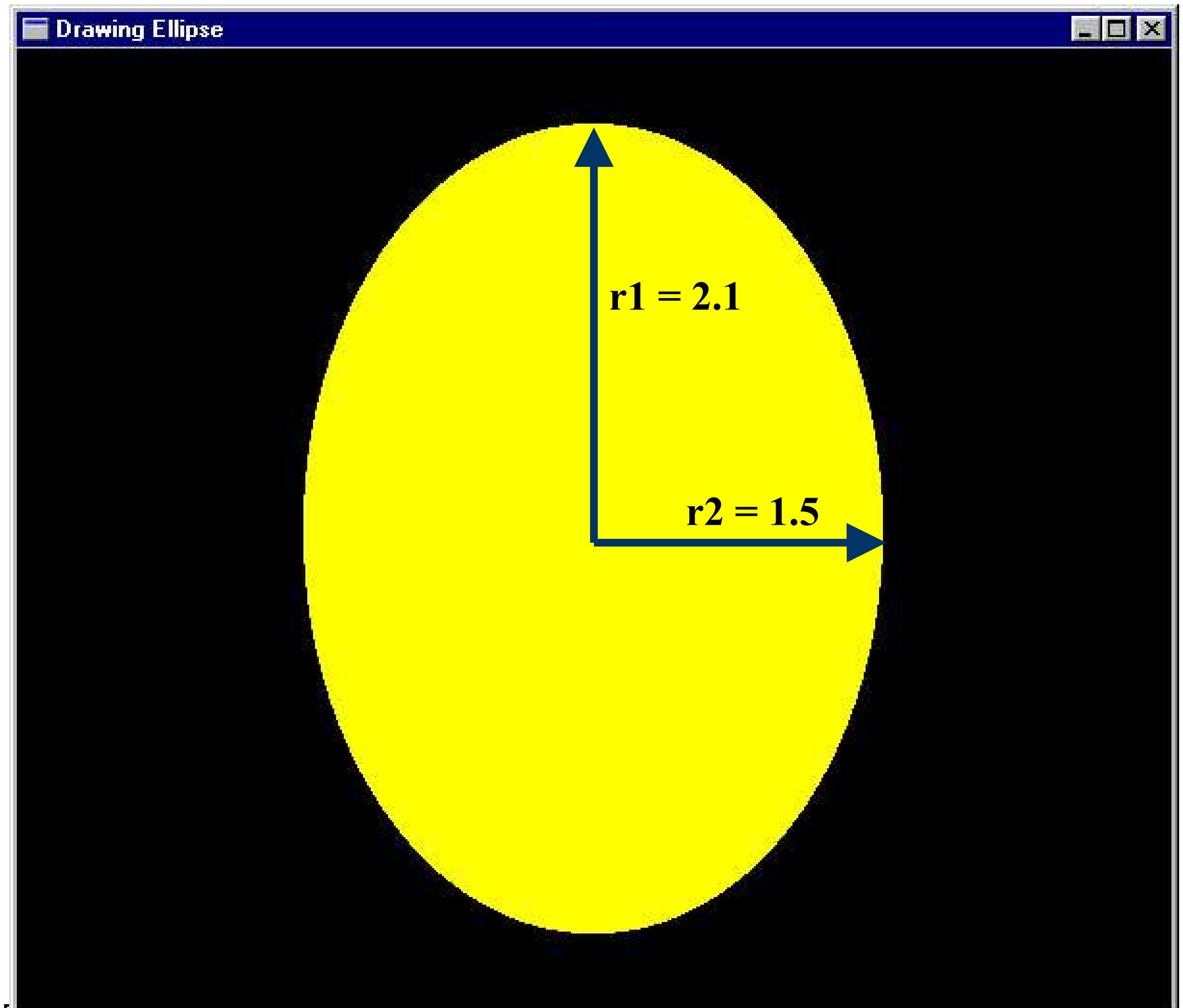
- Using **scanl**, the result we want is:

```
s = scanl aux s1 [2 ..]
```

Sample Series Values

```
[s1 = 0.122449,
 s2 = 0.0112453,
 s3 = 0.00229496,
 s4 = 0.000614721,
 s5 = 0.000189685,
 ...]
```

Note how quickly the values in the series get smaller ...



How far to go?

- It may seem worrisome that

  ```
  s = scanl aux s1 [2 ..]
  ```

  is an infinite list (because `[2 ..]` is)
- But that's no problem so long as we only ever examine a *finite prefix* of the list.
- How many should we take? Only as many as contribute significantly to the answer, e.g., only as long as they pass the significance test

  ```
  significant :: Float -> Bool
  significant x = x > 0.0001 -- for example
  ```

- Can use this handy pre-defined function

  ```
  takeWhile :: (a -> Bool) -> [a] -> [a]
  takeWhile p []                   = []
  takeWhile p (x:xs) | p x         = x : takeWhile p xs
                     | otherwise = []
  ```

Putting it all Together

```
perimeter (Ellipse r1 r2)
   | r1 > r2    = ellipsePerim r1 r2
   | otherwise = ellipsePerim r2 r1
   where ellipsePerim r1 r2
            = let e = sqrt (r1^2 - r2^2) / r1
                  s = scanl aux (0.25*e^2) [2..]
                  aux s i = nextEl e s i
                  significant x = x > epsilon
                  sSum = sum (takeWhile significant s)
              in 2*r1*pi*(1 - sSum)
```