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 …

```haskell
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 \textbf{Sides}

\begin{verbatim}
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]
\end{verbatim}

• 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: \( \text{vs} = [A, B, C, D, E] \)
- 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:  
  \(\text{zip vs (tail vs ++ [head vs])}\)
Zipping Lists

• The zip function (already in the library) can be written:
  
  \[
  \text{zip} :: [a] \rightarrow [b] \rightarrow [(a,b)]
  \]
  
  \[
  \text{zip} \ (x:xs) \ (y:ys) = (x,y) : (\text{zip} \ xs \ ys)
  \]
  
  \[
  \text{zip} _ \ _ _ _ _ _ _ _ = []
  \]
  
  – What happens if the lists are of unequal length?

• This leads to a new version of \textit{sides}

  \[
  \text{sides} :: [\text{Vertex}] \rightarrow [\text{Side}]
  \]

  \[
  \text{sides} \ vs = \text{map} \ d \ (\text{zip} \ vs \ (\text{tail} \ vs \ ++ \ [\text{head} \ vs]))
  \]

  \[
  \text{where} \ d \ (v1,v2) = \text{distBetween} \ v1 \ v2
  \]

• This is more elegant than the explicit recursion, but still verbose; in particular, the need to define \textit{d} is sad. We can avoid this in at least two ways.
More variants of \texttt{sides}

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

\[
\text{uncurry} :: (a \to b \to c) \to (a,b) \to c \\
\text{uncurry } f (x,y) = f x y
\]

allowing us to write

\[
\text{sides vs} = \text{map (uncurry distBetween)} \\
\quad (\text{zip vs (tail vs ++ [head vs])})
\]

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

\[
\text{zipWith (+) [1,2,3] [4,5,6] = [5,7,9]}
\]

So we can write

\[
\text{sides vs} = \text{zipWith distBetween} \\
\quad \text{vs (tail vs ++ [head vs])}
\]
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 $r_1 > r_2$:

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

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

$$s_i = s_{i-1} (2i-1)(2i-3) \ e^2$$

for $i \geq 1$

$$4i^2$$

$$e = \sqrt{r_1^2 - r_2^2} / r_1$$

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)

Now we want to compute \([s_1, s_2, s_3, \ldots]\).
To fix e, let’s define:

\[
\text{aux } s \ i = \text{nextEl } e \ s \ i
\]

So, we would like to compute:

\[
[s_1, \\
  s_2 = f s_1 2, \\
  s_3 = f s_2 3 = f (f s_1 2) 3, \\
  s_4 = f s_3 4 = f (f (f s_1 2) 3) 4, \\
  \ldots \\
]
\]

Can we capture this pattern?
Scanl (scan from the left)

• Yes, using the predefined function `scanl`:

```haskell
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:

```haskell
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:

```haskell
s = scanl aux s1 [2 ..]
```
Sample Series Values

\[
\begin{align*}
  s_1 &= 0.122449, \\
  s_2 &= 0.0112453, \\
  s_3 &= 0.00229496, \\
  s_4 &= 0.000614721, \\
  s_5 &= 0.000189685, \\
  \ldots
\end{align*}
\]

Note how quickly the values in the series get smaller ...
How far to go?

- It may seem worrisome that
  \[ s = \text{scanl aux s1 [2 ..]} \]
  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

  \[
  \text{significant :: Float -> Bool}
  \]
  \[
  \text{significant x = x > 0.0001 -- for example}
  \]

- Can use this handy pre-defined function

  \[
  \text{takeWhile :: (a -> Bool) -> [a] -> [a]}
  \]
  \[
  \text{takeWhile p []} = []
  \]
  \[
  \text{takeWhile p (x:xs)} \mid \text{p x} \quad = \quad x : \text{takeWhile p xs}
  \]
  \[
  \text{otherwise} = []
  \]
Putting it all Together

\[
\text{perimeter (Ellipse } r_1 \ r_2) \\
| \ r_1 > r_2 \quad = \text{ellipsePerim } r_1 \ r_2 \\
| \ \text{otherwise} = \text{ellipsePerim } r_2 \ r_1 \\
\text{where ellipsePerim } r_1 \ r_2 \\
= \text{let } e = \sqrt{(r_1^2 - r_2^2) / r_1} \\
s = \text{scanl aux (0.25*}e^2) [2..] \\
\text{aux } s \ i = \text{nextEl } e \ s \ i \\
\text{significant } x = x > \text{epsilon} \\
sSum = \text{sum (takeWhile significant } s) \\
in 2*\pi*r_1*(1 - sSum)
\]