Advanced Programming

Handout 4

Introductions

- Me: Benjamin C. Pierce

  (known as Benjamin, or, if you prefer, Dr. Pierce, but *not* Ben or Professor)

- You?

Review

- What are the types of these functions?

```
f x = [x]

g x = [x+1]

h [] = 0
h (y:ys) = h ys + 1
```

Review

- How about these?

```
f1 x y = [x] : [y]

f2 x [] = x
f2 x (y:ys) = f2 y ys

f3 [] ys = ys
f3 xs [] = xs
f3 (x:xs) (y:ys) = f3 ys xs
```

Review

- How about these?

```
foo x y = x (x (x y))

bar x y z = x (y z)

baz x (x1:x2:xs) = (x1 `x` x2) : baz xs
baz x _          = []
```

What does **baz** do?

Review

- Recall that `map` is defined as:

```
map :: (a->b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

- What does this function do?

```
mystery f l = map (map f) l
```

Review

- Recall that **foldr** is defined as:

```
foldr :: (a->b->b) -> b -> [a] -> b

foldr op init []      = init
foldr op init (x:xs) =
    x `op` foldr op init xs
```

N.b.: This *was* part of HW 2

- Challenge: Use **foldr** to define a function **maxList :: [Integer] -> Integer** that returns the maximum element from its argument.

- Challenge 2: Use **foldr** to define **map**

Review

- Recall that the function

```
zip :: [a] -> [b] -> [(a,b)]
```

  takes a pair of lists and returns a list of pairs of their corresponding elements:

```
zip [1,2,3] [True,True,False]
➔ [(1,True), (2,True), (3,False)]
```

- What is its definition?

Review

- The function

  `zipWith :: (a->b->c) -> [a] -> [b] -> [c]`

  generalizes `zip`:

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

  ➔ `[5,7,9]`

- What is its definition?

- Can `zip` be defined in terms of `zipWith`?

- Can `zip` be defined in terms of `foldr` or `foldl`?

A Quick Footnote

(We're all in this together...)

Clarification

- Handout 3 said:

  "When we write `(1,2,3,4)` we really mean `(1,(2,(3,4)))`."

- **This is "morally true" but misleading: tuple types in Haskell are n-ary, so `(Integer,Integer,Integer,Integer)` and `(Integer,(Integer,(Integer,Integer)))` are distinct types and expressions like `(1,2,3,4)==(1,(2,(3,4)))` are not legal.**

Infinite Lists

Infinite Lists

- Lists in Haskell need not be finite. E.g.:

```
list1 = [1..]           -- [1,2,3,4,5,6,...]

f x = x:(f(x+1))
list2 = f 1             -- [1,2,3,4,5,6,...]

list3 = 1:2:list3       -- [1,2,1,2,1,2,...]
```

Working with Infinite Lists

- Of course, if we try to perform an operation that requires consuming *all* of an infinite list (such as finding its length), our program will loop.
- However, a program that only consumes a *finite part* of an infinite list will work just fine.

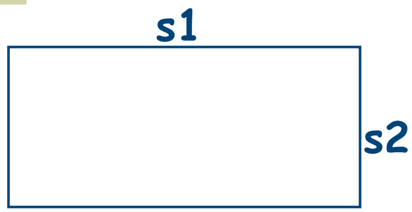
```
take 5 [10..]  ➔  [10,11,12,13,14]
```

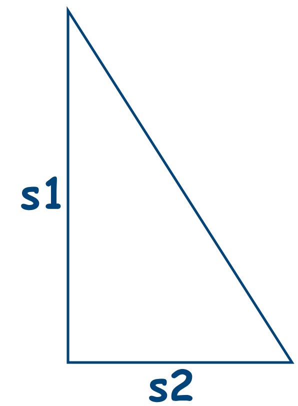
Lazy Evaluation

- The feature of Haskell that makes all this work is *lazy evaluation*.
- Only the portion of a list that is actually needed by other parts of the program will actually be constructed at run time.
- We will discuss the mechanics of lazy evaluation in much more detail later in the course. Today, let's look at a more interesting example of its use...

Shapes III: Perimeters of Shapes (Chapter 6)

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)
                      -- or: sumList (sides pts)
```

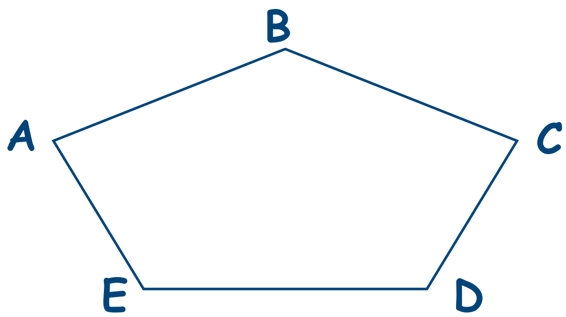
- This assumes that we can compute the lengths of the sides of a polygon. This shouldn't be too difficult since we can compute the distance between two points with **`distBetween`**.

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  : []
    -- i.e. 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])`

New Version of `sides`

This leads to:

```
sides    :: [Vertex] -> [Side]
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 and r_2:

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

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

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

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

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)
```

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

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:

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

Can we capture this pattern?

Scanl (scan from the left)

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

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

- For example:

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

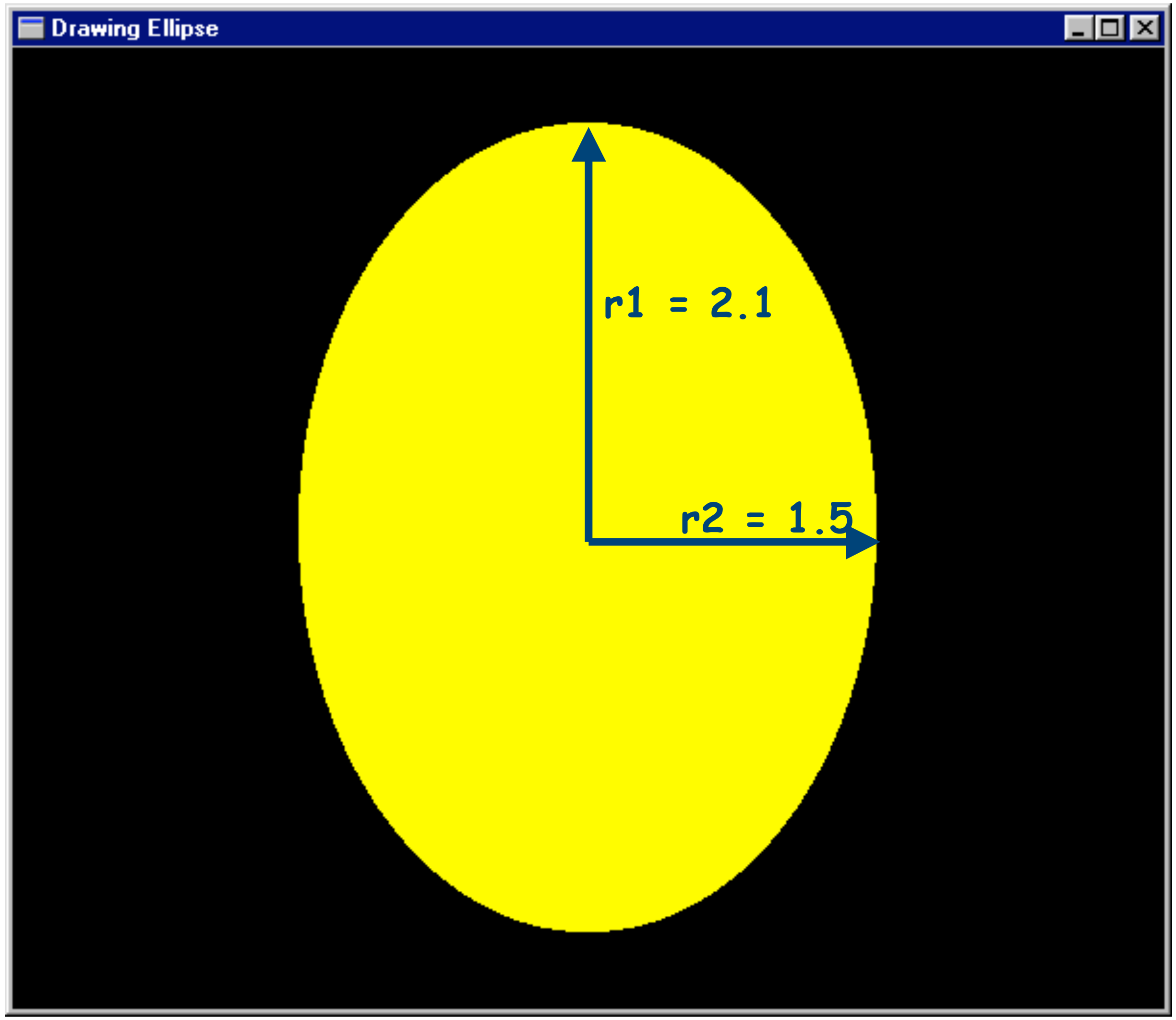
- Using **`scanl`**, the result we want is:

```
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 ...



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)
                            (map intToFloat [2..])
                aux s i = nextEl e s i
                test x = x > epsilon
                sSum = foldl (+) 0 (takeWhile test s)
            in 2*r1*pi*(1 - sSum)
```