Introductions

• Me: Benjamin C. Pierce
  (known as Benjamin, or, if you prefer, Dr. Pierce, but not Ben or Professor)

• You?
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?

\[ f_1 \ x \ y = [x] : [y] \]
\[ f_2 \ x \ [] = x \]
\[ f_2 \ x \ (y:ys) = f_2 \ y \ ys \]
\[ f_3 \ [] \ ys = ys \]
\[ f_3 \ xs \ [] = xs \]
\[ f_3 \ (x:xs) \ (y:ys) = f_3 \ ys \ xs \]
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?
Recall that \texttt{map} is defined as:

\[
\texttt{map} :: (a\rightarrow b) \rightarrow [a] \rightarrow [b]
\]

\[
\texttt{map} \ f \ [] = []
\]

\[
\texttt{map} \ f \ (x:xs) = f \ x : \texttt{map} \ f \ xs
\]

What does this function do?

\[
\texttt{mystery} \ f \ l = \texttt{map} \ (\texttt{map} \ f) \ l
\]
Recall that \texttt{foldr} is defined as:

\[
\texttt{foldr} :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b
\]

\[
\texttt{foldr \ op \ init \ [\] = init}
\]

\[
\texttt{foldr \ op \ init \ (x:xs) = x \ `op` \ foldr \ op \ init \ xs}
\]

\textbf{Challenge}: Use \texttt{foldr} to define a function \texttt{maxList :: [Integer] \rightarrow Integer} that returns the maximum element from its argument.

\textbf{Challenge 2}: Use \texttt{foldr} to define \texttt{map}
Recall that the function

$$\text{zip} :: [a] \to [b] \to [(a,b)]$$

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

$$\text{zip} [1,2,3] [\text{True}, \text{True}, \text{False}] \Rightarrow [(1,\text{True}), (2,\text{True}), (3,\text{False})]$$

What is its definition?
The function

\[
\text{zipWith} :: (a \to b \to c) \to [a] \to [b] \to [c]
\]

generalizes \text{zip}:

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

What is its definition?

Can \text{zip} be defined in terms of \text{zipWith}?

Can \text{zip} be defined in terms of \text{foldr} or \text{foldl}?
A Quick Footnote

(We’re all in this together...)
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

\((\text{Integer, Integer, Integer, Integer, Integer})\) and
\((\text{Integer, (Integer, (Integer, Integer)))}\) are distinct types and expressions like
\((1,2,3,4)==(1,(2,(3,4)))\) are not legal.
Infinite Lists
Lists in Haskell need not be finite.

E.g.:

```haskell
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.

  \[
  \text{take 5 } [10..] \Rightarrow [10,11,12,13,14]
  \]
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)
To compute the perimeter we need a function with four equations (1 for each \texttt{Shape} constructor).

The first three are easy …

\begin{verbatim}
perimeter :: Shape \rightarrow 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)
\end{verbatim}

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 \texttt{distBetween}. 

\textbf{The Perimeter of a Shape}
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?
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?

\[
[A, B, C, D, E, (A, B), (B, C), (C, D), (D, E), (E, A)]
\]

Yes, by “zipping” the two lists:

\[
[A, B, C, D, E] \text{ and } [B, C, D, E, A]
\]

as follows:

\[
\text{zip vs (tail vs ++ [head vs])}
\]
New Version of `sides`

This leads to:

```
sides  :: [Vertex] -> [Side]
sides vs = zipWith distBetween
            vs
            (tail vs ++ [head vs])
```
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 $r_2$:

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

where $s_1 = \frac{1}{4} e^2$

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

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

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

**Perimeter of an Ellipse**
Computing the Series

\[ \text{nextEl} :: \text{Float} \to \text{Float} \to \text{Float} \to \text{Float} \]
\[ \text{nextEl} \ e \ s \ i = s \cdot (2 \cdot i - 1) \cdot (2 \cdot i - 3) \cdot (e^2) / (4 \cdot i^2) \]

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:
\[
\begin{align*}
    s_1 &= \text{aux} \ s_1 \ 2, \\
    s_2 &= \text{aux} \ s_2 \ 3 = \text{aux} \ \text{aux} \ s_1 \ 2 \ 3, \\
    s_3 &= \text{aux} \ s_3 \ 4 = \text{aux} \ \text{aux} \ \text{aux} \ s_1 \ 2 \ 3 \ 4, \\
    \vdots
\end{align*}
\]

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

- Yes, using the predefined function \texttt{scanl}:

\[
\text{scanl} :: (a \to b \to b) \to b \to [a] \to [b] \\
\text{scanl} \ f \ \text{seed} \ [\] = \text{seed} : [\] \\
\text{scanl} \ f \ \text{seed} \ (x:xs) = \text{seed} : \text{scanl} \ f \ \text{newseed} \ xs \\
\text{where} \ newseed = f \ x \ \text{seed}
\]

- For example:

\[
\text{scanl} \ (+) \ 0 \ [1,2,3] \\
\implies [0, \\
1 = (+) 0 \ 1, \\
3 = (+) 1 \ 2, \\
6 = (+) 3 \ 3] \\
\implies [0, 1, 3, 6]
\]

- Using \texttt{scanl}, the result we want is:

\[
\text{scanl aux s1 [2 ..]}
\]
Sample Series Values

\[s_1 = 0.122449,\]
\[s_2 = 0.0112453,\]
\[s_3 = 0.00229496,\]
\[s_4 = 0.000614721,\]
\[s_5 = 0.000189685,\]
\[\ldots\]

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)
```