Exercise Set 4 — 30 September

Fragments

4.1 Program equivalence

Prove that

```
map f . concat = concat . map (map f)
```

4.2 Program equivalence

Under what conditions do the following two list comprehensions deliver the same result?

```
[e | x <- xs, p x, y <- ys]
```

and

```
[e | x <- xs, y <- ys, p x]
```

Small programs

4.3 Lists

Define a function `disjoint :: (Ord a) => [a] -> [a] -> Bool` that takes two lists in ascending order and determines whether or not they have an element in common.

4.4 Implementing merge sort for lists

Define a recursive function `merge :: Ord a => [a] -> [a] -> [a]` that merges two sorted lists to give a single sorted list.

Define `halve :: [a] -> ([a],[a])` that splits a list into two halves whose lengths differ by at most one.

Then define a function `msort :: Ord a => [a] -> [a]` to carry out the usual merge-sort algorithm, by splitting a list in half, sorting the two halves, and then merging them; recursion stops at the empty list or singleton list.

4.5 Fancy function application

Define a function `fancyApp :: [a->b] -> [a] -> [b]` that applies the functions from its first argument to the elements of its second argument in round-robin fashion, so, for example, `fancyApp [(+1),(*2),(+10)] [1,2,3,4,5,6,7]` evaluates to `[2,4,13,5,10,16,8]`.

4.6 Evaluators

Given the type declaration `data Expr = Val Int | Add Expr Expr` define a higher-order function `folde :: (Int -> a) -> (a -> a -> a) -> Expr -> a` such that `folde f g` replaces each `Val` (value) constructor in an expression by the function `f` and each `Add` constructor by the function `g`.

Then use `folde` to define a function `eval :: Expr -> Int` that evaluates an expression to an integer value, a function `size :: Expr -> Int` that calculates the number of values in an expression, a function `ops :: Expr -> Int` that calculates the number of arithmetic operations in an expression, a function `fringe :: Expr -> [Int]` that collects the values in an expression into a list, a function `exprMap :: (Int -> Int) -> Expr -> Expr` that applies the given function to each value in an expression.

Finally, define a function `simplify :: Expr -> Expr` that will replace every addition with its result provided the result is small, specifically, less than 100.

4.7 Drawing

In this exercise, we develop a simple tool for drawing. A drawing is just a line drawing consisting of some number of polygons. A polygon is given as a list of vertices, and a vertex is simply a pair of real numbers for the x and y coordinates. For instance,

```
[[(100.0,100.0),(100.0,200.0),(200.0,100.0)],
 [(150.0,150.0),(150.0,200.0),(200.0,200.0),(200.0,150.0)]]
```

is an internal representation in Haskell of a drawing consisting of a triangle and a square.

Convert such a representation of a drawing into a simple page description in the PostScript language (more specifically, Encapsulated PostScript); i.e., write a function
`makeCommand :: [[(Float,Float)]] -> String`.

The result returned by `makeCommand` is a Haskell value of type String, which must contain valid PostScript commands for drawing the given polygons.

For instance, the expression

```
makeCommand [[(100.0,100.0),(100.0,200.0),(200.0,100.0)],
             [(150.0,150.0),(150.0,200.0),(200.0,200.0),(200.0,150.0)]]
```

should evaluate to the text:

```
%!PS-Adobe-3.0 EPSF-3.0
%%BoundingBox: 100.0 100.0 200.0 200.0

100.0 100.0 moveto
100.0 200.0 lineto
200.0 100.0 lineto
closepath
stroke

150.0 150.0 moveto
150.0 200.0 lineto
200.0 200.0 lineto
200.0 150.0 lineto
closepath
stroke

showpage
%%EOF
```

which would be printed by a PostScript printer as in Figure 1.

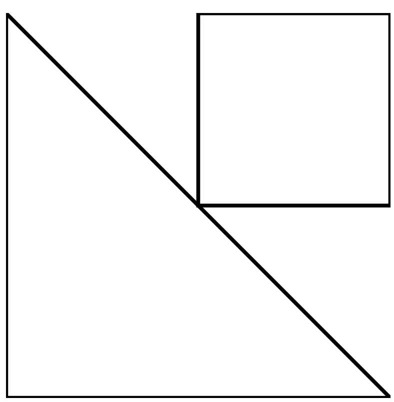

Figure 1: A triangle and a square.

Note that the bounding box, which you need to calculate, is the smallest upright rectangle such that no points of the drawing lie outside it; it is specified by giving the coordinates of its lower left and upper right corners, in our example $(100.0, 100.0)$ and $(200.0, 200.0)$.

Next, modify `makeCommand` such that any areas of overlap between polygons are shaded black.