Homework 1

Due 3 October, 5PM

Handout 1
CS242: Autumn 2012
26 September

(Updated with corrections on October 1, 2012)

Reading

1. You may do this homework individually or you may do the entire assignment with a homework partner. If you work with another person, you may turn in one set of solutions for both of you.
2. Haskell code for questions that are marked “submit code” should be submitted electronically according to instructions that will be posted on the course web site. Solutions for all other questions should be turned in on paper for manual grading.
3. Read Douglas Crockford’s ”A Survey of the JavaScript Programming Language” at `http://javascript.crockford.com/survey.html`.
4. Read the revised Chapter 5, covering Haskell, that is posted in CourseWare.

Problems

1. .... JavaScript scripting features

JavaScript was designed to serve as a *scripting language*. Describe one feature of JavaScript that you believe is useful for scripting HTML pages, but undesirable for writing large programs in JavaScript. Explain.

2. .... Javascript higher-order functions

Functions `map` and `reduce` are standard functions from traditional functional programming that achieved broader recognition as a result of Google’s MapReduce method for processing and generating large data sets. While `map`, `reduce`, and a number of related functions are provided in many JavaScript implementations, `map` and `reduce` can also be defined relatively simply in JavaScript as follows:

```
function map (f, inarray) {
    var out = [];
    for(var i = 0; i < inarray.length; i++) {
      out.push( f(inarray[i]) )
    }
    return out;
  }

function reduce (f, inarray) {
    if(inarray.length <= 1) return;
    if(inarray.length == 2) return f(inarray[0],inarray[1]);
    r = inarray[0];
    for(var li = 1;li < inarray.length;li++){
        r = f(r, inarray[li]);
    }
    return r;
}
```

Function `map(f, inarray)` returns an array constructed by applying `f` to every element if `inarray`. Function `reduce(f, inarray)` applies the function `f` of two arguments to elements in the list, from left to right, until it reduces the list to a single element. For example:

```
js> map( function(x){return x+1}, [1,2,3,4,5])
2,3,4,5,6
js> reduce( function(x,y){return x+y}, [1,2,3,4,5])
15
js> reduce( function(x,y){return x*y}, [1,2,3,4,5])
120
```

These functions can be combined in various ways. For each of the following questions, you may use a JavaScript implementation to test your answer yourself, but turn your solution in as part of a written description for manual grading.

(a) Explain how to use `map` and `reduce` to compute the sum of the first five squares, in one line. (The sum of the first three squares is $1^2 + 2^2 + 3^2$.)

(b) Explain how to use `map` and `reduce` to count the number of positive numbers in an array of numbers.

(c) Explain how to use `map` and/or `reduce` to "flatten" an array of arrays of numbers, such as [[1,2],[3,4],[5,6],[7,8,9]], to an array of numbers. (Hint: Look for built-in JavaScript concatenation functions.)

3. .... Javascript "blocks"

This problem asks you to compare two sections of code. The first one has three declarations and a fourth statement that consists of an assignment and a function call inside curly braces:

```
var x = 5;
   function f(y) {return (x+y)-2};
      function g(h){var x = 7; return h(x)};
         {var x = 10; z=g(f)};
```

The second section of code is derived from the first by placing each line in a separate function, and then calling all the functions with empty argument lists. In effect, each "`(function (){`" begins a new block because the body of each JavaScript function is in a separate block. Each "`})()`" closes the function body and calls the function immediately so that the function body is executed.

```
(function (){
    var x = 5;
    (function (){
          function f(y) {return (x+y)-2};
          (function (){
                function g(h){var x = 7; return h(x)};
                (function (){
                    var x = 10; z=g(f);
                })()
           })()
     })()
})()
```

(a) What is the value of `g(f)` in the first code example?

(b) The call `g(f)` in the first code example causes the expression `(x+y)-2` to be evaluated. What are the values of `x` and `y` that are used to produce the value you gave in part (a)?

(c) Explain how the value of `y` is set in the sequence of calls that occur before `(x+y)-2` is evaluated.

(d) Explain why `x` has the value you gave in part (b) when `(x+y)-2` is evaluated.

(e) What is the value of `g(f)` in the second code example?

(f) The call `g(f)` in the second code example causes the expression `(x+y)-2` to be evaluated. What are the values of `x` and `y` that are used to produce the value you gave in part (e)?

(g) Explain how the value of `y` is set in the sequence of calls that occur before `(x+y)-2` is evaluated.

(h) Explain why `x` has the value you gave in part (f) when `(x+y)-2` is evaluated.

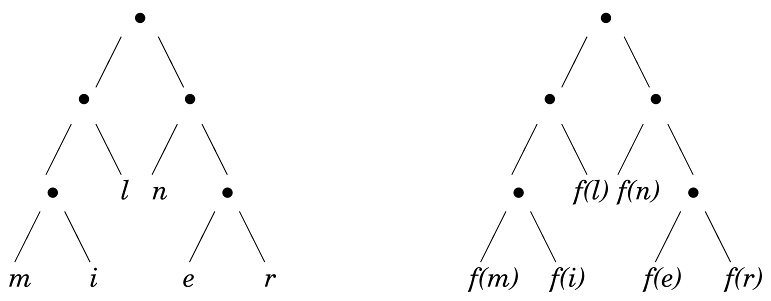
4. .... Haskell Map for Trees

(a) (Submit code) The binary tree data type

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

describes a binary tree for any type, but does not include the empty tree (i.e., each tree of this type must have at least a root node).

Write a function `maptree` that takes a function as an argument and returns a function that maps trees to trees by mapping the values at the leaves to new values, using the function passed in as a parameter. In more detail, if `f` is a function that can be applied to the leaves of tree `t` and `t` is the tree on the left, then `maptree f t` should result in the tree on the right:



For example, if `f` is the function `f x = x + 1` then

```
maptree f (Node (Node (Leaf 1) (Leaf 2))  (Leaf 3))
```

should evaluate to `Node (Node (Leaf 2) (Leaf 3)) (Leaf 4)`.

(b) (Submit text) Explain your definition in one or two sentences.

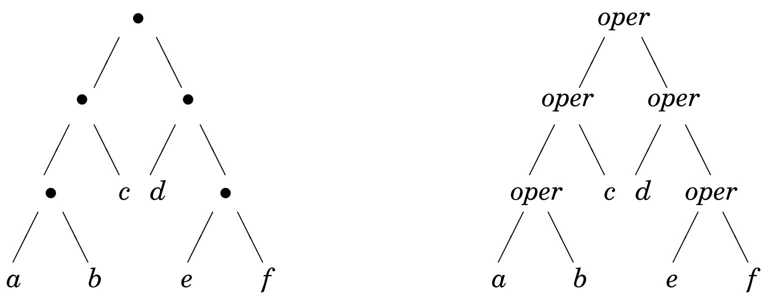
(c) (Submit text) What is the type Haskell gives to your function?
Why is it not the type `(t -> t) -> Tree t -> Tree t`?

5. .... Haskell Reduce for Trees

The binary tree datatype

```
data Tree a  =  Leaf a  |  Node (Tree a) (Tree a)
```

describes a binary tree for any type, but does not include the empty tree (i.e., each tree of this type must have at least a root node).



(a) (Submit code) Write a function

```
reduce ::  (a -> a -> a) -> Tree a -> a
```

that combines all the values of the leaves using the binary operation passed as a parameter. In more detail, if `oper :: a -> a -> a` and `t` is the nonempty tree on the left in this picture, then `reduce oper t` should be the result obtained by evaluating the tree on the right. For example, if `f` is the function

```
f :: Int -> Int -> Int
f x y = x + y
```

then `reduce f (Node (Node (Leaf 1) (Leaf 2)) (Leaf 3)) = (1+2)+3 = 6`

(b) (Submit text) Explain your definition of `reduce` in one or two sentences. Assume that `oper` is a commutative and associative operator. Also for the case when the tree has just a single `Leaf` node, the `reduce` function should simply return the contents of this leaf node.

6. .... Currying

This problem asks you to show that the Haskell types `a -> b -> c` and `(a,b) -> c` are essentially equivalent.

(a) (Submit code) Define higher-order Haskell functions

```
curry :: ((a,b) -> c) -> (a -> (b -> c))
```

and

```
uncurry ::  (a -> (b -> c)) -> ((a,b) -> c)
```

(b) (Submit text) For all functions `f :: (a,b) -> c` and `g ::a -> (b -> c)`, the following two equalities should hold (if you wrote the right functions):

```
uncurry(curry f) = f
curry(uncurry g) = g
```

Explain why each is true for the functions you have written. Your answer can be three or four sentences long. Try to give the main idea in a clear, succinct way. (We are more interested in insight than in number of words.) Be sure to consider termination behavior as well.

7. .... Infinite Data Structures

Recall that the following Haskell declaration

```
naturals :: [Int]
naturals = [ x | x <- [1..]]
```

uses a set comprehension to define an infinite list of positive integers.

(a) (Submit code) Define Haskell values denoting the list of all even positive integers and the list of all odd positive integers.

```
evens :: [Int]
evens =
```

and

```
odds :: [Int]
odds =
```

(b) (Submit code) Define a `merge` function that takes two ordered lists and returns the resulting merged list, in sorted order

```
merge :: [Int] -> [Int] -> [Int]
```

(c) (Submit text) Does the call `merge evens odds` terminate? Explain why or why not in a few sentences.