Homework 3

Due Friday, 2/11/10

Please turn in your homework solutions online at `http://www.dci.pomona.edu/tools-bin/cs131upload.php` before the beginning of class on the due date.

1. (10 points) **Translation into Lambda Calculus**

   Please do problem 4.6 from Mitchell, page 84.

2. (20 points) **Lazy Evaluation and Parallelism**

   Please do problem 4.11 from Mitchell, page 87.

   The function g should be defined as follows (there may be a typo in the book, depending on the printing):

```
fun g(x, y) = if x = 0
              then 1
              else if x + y = 0
                   then 2
                   else 3
```

3. (5 points) **Algol 60 Procedure Types**

   Please do problem 5.1 from Mitchell, page 122.

4. (10 points) **Haskell types**

   Explain the Haskell type for each of teh following declarations:

   (a) a(x,y) = x+2*y
   (b) b(x,y) = x+y/2.0
   (c) c(f) = \y → f y
   (d) d(f,x) = f(f(x))
   (e) e(x,y,b) = if b(y) then x else y

   Because you can simply type these expressions into ghci (with a `let` in front if you type them directly or as is if you load them from a file) to determine their type, be sure to write a short explanation to show that you understand why the function has the type you give.

5. (50 points) **Haskell Programming**

   For this problem, use the ghci interpreter on the Unix machines in the computer lab. To run the program in the file "example.hs", type

```
-> ghci
Prelude> :l example.hs
Prelude> :set +t       -- gives more info when you type an expression
```

at the command line.

The Haskell compiler will process the program in the file and then wait for the user to type an expression. For example, if "example.hs" contains

```
-- double an integer
double x = x + x;

-- return the length of a list *)
listLength [] = 0
listLength (l:ls) = 1 + listLength ls
```

You can test the program by typing the following:

```
*Main> double 10
20
it :: Integer
*Main> listLength (1:[2,3,4])
4
it :: Integer
```

Start early on this part so you can see the TA or me if you have problems understanding the language. Looking at the examples in the on-line tutorials and text and in your notes will help a great deal in understanding how to use Haskell. Use pattern matching where possible.

Comments in Haskell appear from – to the end of the line.

(a) **Basic Functions**

Define a function `sumSquares` that, given a nonnegative integer n, returns the sum of the squares of the numbers from 1 to n:

```
- sumSquares 4;
30
- sumSquares 5;
55
```

Define a function `listDup` that takes a pair of an element, e, of any type, and a non-negative number, n, and returns a list with n copies of e:

```
> listDup("moo", 4);
["moo","moo","moo","moo"]
it :: [[Char- listDup(1, 2);
[1,1]
it :: [Integer]
> listDup(listDup("cow", 2), 2)
[["cow","cow"],["cow","cow"]]
it :: [[[Char]]]
```

Your function will have a type like `(Num t) => (a, t) -> [a]`. What does this type mean? Why is it the appropriate type for your function.

(b) **Zipping and Unzipping**

The function `zip` to compute the pairwise interleaving of two lists of arbitrary length is predefined, but I'd like you to write it from scratch anyway (calling it `zip'`. You should use pattern matching to define this function:

```
-> *Main> zip' [1,3,5,7] ["a","b","c","de"]
[(1,"a"),(3,"b"),(5,"c"),(7,"de")]
it :: [(Integer, [Char])]
```

Note: If the lists don't have the same length, you may decide how you would like the function to behave. If you don't specify any behavior at all you will get a warning from the compiler that you have not taken care of all possible patterns— this is fine.

Write the inverse function, `unzip'`, which behaves as follows:

```
*Main> unzip' [(1,"a"),(3,"b"),(5,"c"),(7,"de")]
([1,3,5,7],["a","b","c","de"])
it :: ([Integer], [[Char]])
```

Again, `unzip` is built-in, but you will write your own `unzip'`.

Write `zip3'`, to zip three lists.

```
*Main> zip3' [1,3,5,7] ["a","b","c","de"] [1,2,3,4]
[(1,"a",1),(3,"b",2),(5,"c",3),(7,"de",4)]
it :: [(Integer, [Char], Integer)]
```

Once again, `zip3` is built-in, but you will write your own `zip3'`.

Why can't you write a function `zip_any` that takes a list of any number of lists and zips them into tuples? From the first part of this question it should be pretty clear that for any fixed n, one can write a function `zipn`. The difficulty here is to write a single function that works for all n! I.e., can we write a single function `zip_any` such that `zip_any [list1,list2,...,listk]` returns a list of k-tuples no matter what k is?

(c) **find**

Write a function find with type `(Eq a) => (a, [a])` → `Integer` that takes a pair of an element and a list and returns the location of the first occurrence of the element in the list. For example:

```
*Main> find(3, [1, 2, 3, 4, 5])
2
*Main> find("cow", ["cow", "dog"])
0
*Main> find("rabbit", ["cow", "dog"])
-1
```

First write a definition for find where the element is guaranteed to be in the list. Then, modify your definition so that it returns `-1` if the element is not in the list.

[When the system is asked for the type of my solution for find, it gives me the even more general: `(Num a, Eq t) => (t, [t]) -> a`.]

(d) **Trees**

Here is the datatype definition for a binary tree storing integers at the leaves:

```
data IntTree = Leaf Integer | Interior (IntTree,IntTree) deriving Show
```

Write a function `treeSum:IntTree → Integer` that adds up the values in the leaves of a tree:

```
*Main> treeSum(Leaf 3)
3
*Main> treeSum(Interior (Leaf 2, Leaf 3))
5
*Main> treeSum(Interior(Leaf 2, Interior(Leaf 1, Leaf 1)))
4
```

Write a function `height : IntTree → Integer` that returns the height of a tree:

```
*Main> height(Leaf 3);
1
*Main> height(Interior(Leaf 2, Leaf 3));
2
*Main> height(Interior(Leaf 2, Interior(Leaf 1, Leaf 1)));
3
```

Write a function `balanced: IntTree → Bool` that returns true if a tree is balanced (i.e., both subtrees are balanced and differ in height by at most one). You may use your height function above.

```
*Main> balanced(Leaf 3);
True
*Main> balanced(Interior(Leaf 2, Leaf 3));
True
*Main> balanced(Interior(Leaf 2, Interior(Leaf 3, Interior(Leaf 1, Leaf 1))));
False
```

Is your implementation as efficient as possible? What is wrong with using the `height` function in the definition of `balanced`? How would you write `balanced` to be more efficient? (You need not write code, but describe how you would do this.)

(e) **Stack Operations**

Certain programming languages (and HP calculators) evaluate expressions using a stack. As some of you may know, PostScript is a programming language of this ilk for describing images when sending them to a printer. We are going to implement a simple evaluator for such a language. Computation is expressed as a sequence of operations, which are drawn from the following data type:

```
data OpCode = Push Float | Add | Mult | Sub | Div | Swap deriving Show
```

The operations have the following effect on the operand stack. (The top of the stack is shown on the left.)

OpCode	Initial Stack	Resulting Stack
`Push(r)`	...	r ...
`Add`	a b ...	(b + a) ...
`Mult`	a b ...	(b * a) ...
`Sub`	a b ...	(b - a) ...
`Div`	a b ...	(b / a) ...
`Swap`	a b ...	b a ...

The stack may be represented using a list for this example, although we could also define a stack data type for it.

```
type Stack = [Float]
```

Write a recursive evaluation function with the signature

```
eval :: ([OpCode], Stack) -> Float
```

It takes a list of operations and a stack. The function should perform each operation in order and return what is left in the top of the stack when no operations are left. For example,

```
eval([Push 2.0, Push 1.0, Sub],[])
```

returns 1.0. The `eval` function will have the following basic form:

```
eval ([],          a:rest)   = --
eval ((Push n):ops, rest)     = --
--
eval (_,            _)       = 0.0;
```

You need to fill in the blanks and add cases for the other opcodes.

The last rule handles illegal cases by matching any operation list and stack not handled by the cases you write. These illegal cases include ending with an empty stack, performing addition when fewer than two elements are on the stack, and so on. You may ignore divide-by-zero errors for now (or look at exception handling in Ullman– we will cover that topic in a few weeks).