Homework 3

Due Tuesday, 28 Feb

Handout 7
CSCI 334: Spring, 2012

Reading

1. **(Required)** Read Mitchell, skim 4.4–4.5, 5, skim 6.1.
2. **(As necessary)** Read Ullman. (I will leave three copies in the Unix lab – please use them, but do not take them out of lab.)
3. **(Optional)** J. Backus, Can programming be liberated from the von Neuman style?, Comm. ACM 21, 8 (1978) 613-641. You can find this on the cs334 web site.

Problems

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

Mitchell, Problem 4.6

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

Mitchell, Problem 4.11

The function g should be defined as follows (there is a typo in the book):

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

Mitchell, Problem 5.1

4. (*10 points*) ..... ML Types

Mitchell, Problem 6.1

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

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

```
sml < example.ml
```

at the command line. As with Lisp, the ML compiler will process the program in the file and print the result. For example, if "example.ml" contains

```
(* double an integer *)
fun double (x) = x * x;

(* return the length of a list *)
fun listLength (nil) = 0
  | listLength (l::ls) = 1 + listLength ls
  ;

double (10);
listLength (1::[2,3,4]);
```

the command "`sml < example.sml`" will produce the following:

```
val double = fn : int -> int
val listLength = fn : 'a list -> int
val it = 100 : int
val it = 4 : int
```

You can also run "sml" and enter in declarations and expressions to evaluate at the prompt.

Start early on this part so you can see the TA or me if you have problems understanding the language. There are many valuable resources available to help you learn ML:

- The examples in the *Mitchell* book and in your notes.
- Ullman's *Elements of ML Programming* book. I will leave several copies in the Unix lab for your reference. **Do not remove these books from the lab.**
- Several very good tutorials listed on the `links` web page.

A few additional details:

- You can configure Emacs to provide auto-formatting and syntax highlighting while editing ML files. To do this, add the following line to your `.local_emacs` file in your home directory. (You may need to create that file if it isn't already there):

  ```
  (load-file "/home/faculty/freund/share/sml-mode/site-start.el")
  ```

- You may include all parts of this question in one file if you wish, or you may use a separate file for each part— either is fine. And be sure to comment! (Comments in ML appear inside `(*` and `*)` characters.)
- Put the following line at the top of your ML files to ensure that large data types and lists are fully printed:

  ```
  Control.Print.printDepth := 100;
  Control.Print.printLength := 100;
  ```

- You should use pattern matching where possible.
- To submit your work, use the turnin script: "turnin -c 334 foo.ml" will turn in the file foo.ml. **Hand in a print out of the code with the problem set on Tuesday as well.** There are several thought questions in the descriptions below. Please answer these questions in comments in the code.

(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)
val it = 30 : int
- sumSquares(5)
val it = 55 : int
```

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

```
- listDup("moo", 4);
val it = ["moo","moo","moo","moo"] : string list
- listDup(1, 2);
val it = [1,1] : int list
- listDup(listDup("cow", 2), 2);
val it = [["cow","cow"],["cow","cow"]] : string list list
```

Question: Your function will have a type like `'a * int -> 'a list`. What does this type mean? Why is it the appropriate type for your function.

(b) **Zipping and Unzipping**

Write the function `zip` to compute the product of two lists of arbitrary length. You should use pattern matching to define this function:

```
- zip [1,3,5,7] ["a","b","c","de"];
val it = [(1,"a"),(3,"b"),(5,"c"),(7,"de")]: (int * string) list
```

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 "match not exhaustive" warning from the compiler to indicate that you have not taken care of all possible patterns—this is fine.

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

```
- unzip [(1,"a"),(3,"b"),(5,"c"),(7,"de")];
val it = ([1,3,5,7], ["a","b","c","de"]): int list * string list
```

Write `zip3`, to zip three lists.

```
- zip3 [1,3,5,7] ["a","b","c","de"] [1,2,3,4];
val it = [(1,"a",1),(3,"b",2),(5,"c",3),(7,"de",4)]: (int * string * int) list
```

Question: 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. In other words, 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 `''a * ''a list -> int` 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:

```
- find(3, [1, 2, 3, 4, 5]);
val it = 2 : int
- find("cow", ["cow", "dog"]);
val it = 0 : int
- find("rabbit", ["cow", "dog"]);
val it = ~1 : int
```

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.

(d) **Trees**

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

```
datatype IntTree = LEAF of int | NODE of (IntTree * IntTree);
```

Write a function `sum:IntTree -> int` that adds up the values in the leaves of a tree:

```
- sum(LEAF 3);
val it = 3 : int
- sum(NODE(LEAF 2, LEAF 3));
val it = 5 : int
- sum(NODE(LEAF 2, NODE(LEAF 1, LEAF 1)));
val it = 4 : int
```

Write a function `height: IntTree -> int` that returns the height of a tree:

```
- height(LEAF 3);
val it = 1 : int
- height(NODE(LEAF 2, LEAF 3));
val it = 2 : int
- height(NODE(LEAF 2, NODE(LEAF 1, LEAF 1)));
val it = 3 : int
```

Write a function `balanced: IntTree -> bool` that returns true if a tree is balanced (ie, both subtrees are balanced and differ in height by at most one). You may use your `height` function in the definition of `balanced`.

```
- balanced(LEAF 3);
val it = true : bool
- balanced(NODE(LEAF 2, LEAF 3));
val it = true : bool
- balanced(NODE(LEAF 2, NODE(LEAF 3, NODE(LEAF 1, LEAF 1))));
val it = false : bool
```

Question: What is non-optimal about using the `height` function in the definition of `balanced`? Can you suggest a more efficient implementation? You need not write code, but describe in a sentence or two how you would do this.

(e) **Stack Operations**

Certain programming languages (and HP calculators) evaluate expressions using a stack. As I am sure many of you learned in cs136, 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:

```
datatype OpCode =
    PUSH of real
  | ADD
  | MULT
  | SUB
  | DIV
  | SWAP
  ;
```

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 = real list;
```

Write a recursive evaluation function with the signature

```
eval : OpCode list * Stack -> real
```

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:

```
fun eval (nil,a::st) = (* ... *)
  | eval (PUSH(n)::ops,st) = (* ... *)
  | (* ... *)
  | 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).

If you wrote a PostScript interpreter in cs136, compare that experience to this one. In particular, what advantages does ML offer for writing this type of program? (No need to write an answer to this question, but come ready to talk about it at the next lecture).

6. (*10 points*) ..... (Bonus Question) Klop's Combinator

Read the section Mitchell regarding fixed point operations. There is more than one fixed point operator. Here is another one. Let

$$A = \lambda abcdefghijklmnopqstuvwxyzr.r(thisisafixedpointcombinator)$$

where $\lambda abcdefghijklmnopqstuvwxyzr.r(...)$ is shorthand for $\lambda a.\lambda b.\lambda c.\ldots\lambda z.\lambda r.r(...)$. Also define $B = AAAAAAAAAAAAAAAAAAAAAAAAAA$, where A appears 26 times in B. Show that $Bf = f(Bf)$ for any f.