Homework 4

Due Tuesday, 7 March

Handout 9

CSCI 334: Spring, 2012

Reading

1. **(Required)** Read Mitchell, Chapter 5.
2. **(As necessary)** Read Ullman, as needed for the programming questions.

Problems

1. (*10 points*) .... ML Map for Trees

   Mitchell, Problem 5.4

   You are not required to submit a program file with "turnin" for this question, but you may want to double check your answer by running your solution. If you do, be sure to include

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

   at the top of your file, so that datatypes print completely.

2. (*10 points*) .... Currying

   Mitchell, Problem 5.6

   You are not required to submit a program file with "turnin" for this question, but you may want to double check your answer by running your solution.

   Note that you MUST explain why the equations hold. One way to do this is to apply both sides of each equation to the same argument(s) and describe how each side evaluates to the same term. For example, show that

   $$\texttt{UnCurry}(\texttt{Curry}(\texttt{f}))(\texttt{s},\texttt{t}) = \texttt{f}(\texttt{s},\texttt{t})$$

   and

   $$\texttt{Curry}(\texttt{UnCurry}(\texttt{g}))\ \texttt{s}\ \texttt{t} = \texttt{g}\ \texttt{s}\ \texttt{t}$$

   for any `s` and `t`.

3. (*10 points*) .... Disjoint Unions

   Mitchell, Problem 5.7

   A quick summary of C unions for those who have not used them before:

   The declaration

```
union IntString {
  int i;
  char *s;
} x;
```

declares a variable `x` with type `union IntString`. The variable `x` may contain either an integer or a string value. (You may think of the type `char *` as being like `string` for this question.) To store an integer into `x`, you would write `x.i = 10`. To store a string, you would write `x.s = "moo"`. Similarly, we can read the value stored in `x` as an integer or a string with `num = x.i` and `str = x.s`, respectively. The expression `x.i` interprets and returns whatever value is in `x` as an integer, regardless of what was last stored to `x`, and similarly for `x.s`.

Also, the SML compiler by default doesn't generate the error message alluded to in the question write-up anymore. However, you should be able to reason about what it would be given what we know about datatypes...

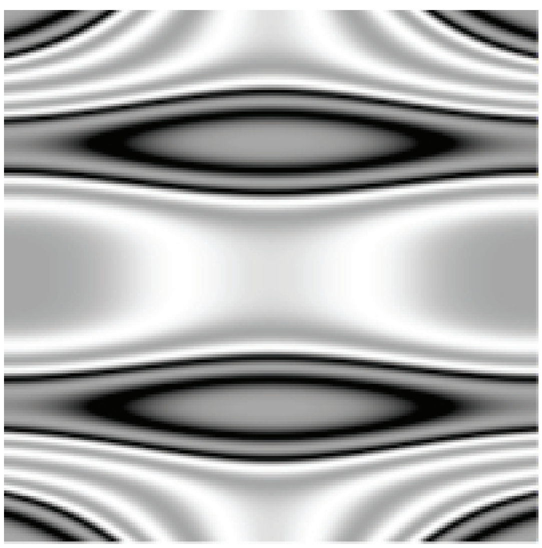
4. (*10 points*) .... Type Inference and Bugs

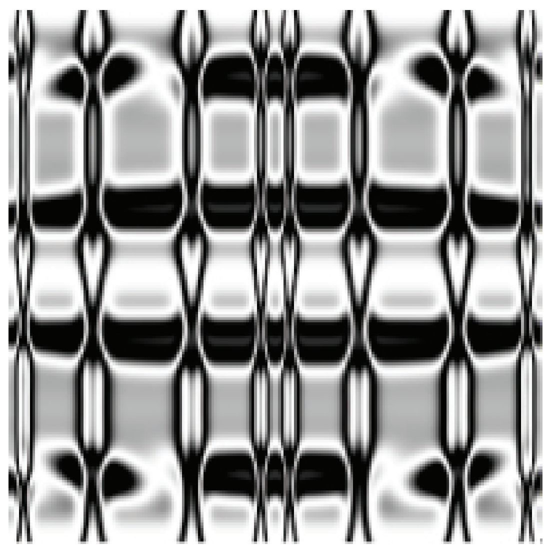
What is the type of the following ML function?

```
fun append(nil, l) = l
|   append(x::l, m) = append(l,m);
```

Write one or two sentences to explain succinctly and informally why `append` has the type you give. This function is intended to append one list onto another. However, it has a bug. How might knowing the type of this function help the programmer to find the bug?

5. (*50 points*) .... Random Art





This problem brings together a number of topics we have studied, including grammars, parse trees, and evaluation. Your job is to write an ML program to construct and plot randomly generated functions. The language for the functions can be described by a simple grammar:

$$e ::= x \mid y \mid \sin \pi e \mid \cos \pi e \mid (e+e)/2 \mid e * e$$

Any expression generated by this grammar is a function over the two variables x and y. Note that any function in this category produces a value between -1 and 1 whenever x and y are both in that range.

We can characterize expressions in this grammar with the following ML datatype:

```
datatype Expr =
    VarX
  | VarY
  | Sine     of Expr
  | Cosine   of Expr
  | Average  of Expr * Expr
  | Times    of Expr * Expr;
```

Note how this definition mirrors the formal grammar given above; for instance, the constructor `Sine` represents the application of the sin function to an argument multiplied by π. Interpreting abstract syntax trees is much easier than trying to interpret terms directly.

To begin, you should copy the `expr.sml` and `art.sml` starter files from the handouts web page.

(a) **Printing Expressions:** The first two parts require that you edit and run only `expr.sml`. First, write a function

```
exprToString : Expr -> string
```

to generate a printable version of an expression. For example, calling `exprToString` on the expression

```
Times(Sine(VarX),Cosine(Times(VarX,VarY)))
```

should return a string similar to "`sin(pi*x)*cos(pi*x*y)`". The exact details are left to you. (Remember that string concatenation is performed with the "^" operator.)

Test this function on a few sample inputs before moving to the next part.

(b) **Expression Evaluation:** Write the function

```
eval : Expr -> real*real -> real
```

to evaluate the given expression at the given (x, y) location. You may want to use the functions `Math.cos` and `Math.sin`, as well as the floating-point value `Math.pi`. (Note that an expression tree represented, e.g., as `Sine(VarX)` corresponds to the mathematical expression $\sin(\pi x)$, and the `eval` function must be defined appropriately.)

Test this function on a few sample inputs before moving on to the next part. Here are a few sample runs:

```
- eval (Sine(Average(VarX,VarY))) (0.5,0);
val it = 0.707106781187 : real
- eval sampleExpr (0.1,0.1);
val it = 0.569335014033 : real
```

(c) **Driver Code:** The `art.sml` file includes the `doRandomGray` and `doRandomColor` functions, which generate grayscale and color bitmaps respectively. These functions want to loop over all the pixels in a (by default) 301 by 301 square, which naturally would be implemented by nested for loops. In `art.sml`, complete the definition of the function

```
 for : int * int * (int -> unit) -> unit
```

The argument triple contains a lower bound, an upper bound, and a function; your code should apply the given function to all integers from the lower bound to the upper bound, inclusive. If the greater bound is strictly less than the lower bound, the call to `for` should do nothing. Implement this function using imperative features. In other words, use a `ref` cell and the while construct to build the `for` function.

Note: It will be useful to know that you can use the expression form (e1 ; e2) to execute expression e1, throw away its result, and then execute e2. Thus, inside an expression a semicolon acts exactly like comma in C or C++. Also, the expression "`()`" has type `unit`, and can be used when you want to "do nothing".

Test your code with a call like the following:

```
for (2, 5, (fn x => (print ((Int.toString(x)) ^ "\n"))));
```

It should print out the numbers 2,3,4, and 5.

Now produce a grayscale picture of the expression `sampleExpr`. You can do this by calling the `emitGrayscale` function. Look at `doRandomGray` to see how this function is used.

If you get an `uncaught exception Chr` error while producing a bitmap, that is an indication that your `eval` function is returning a number outside the range [-1,1].

Note: The type assigned to your `for` function may be more general than the type described above. How could you force it to have the specified type, and why might it be useful to do that? (You don't need to submit an answer to this, but it is worth understanding.)

(d) **Viewing Pictures:** You can view pgm files, as well as the ppm files described below, on our Linux computers with the `eog` program. To view the output from a non-Unix machine, or to post them on a web, etc., you might need to first convert the file to jpeg format with the following command:

```
convert art.pgm art.jpg
```

The `convert` utility will work for both .ppm and .pgm files.

(e) **Generating Random Expressions:** Your next programming task is to complete the definition of

```
build(depth, rand) : int * RandomGenerator -> Expr
```

The first parameter to `build` is a maximum nesting depth that the resulting expression should have. A bound on the nesting depth keeps the expression to a manageable size; it's easy to write a naive expression generator which can generate incredibly enormous expressions. When you reach the cut-off point (i.e., `depth` is 0), you can simply return an expression with no sub-expressions, such as `VarX` or `VarY`. If you are not yet at the cut-off point, randomly select one of the forms of `Expr` and recursively create its subexpressions.

The second argument to `build` is a function of type `RandomGenerator`. As defined at the top of `art.sml`, the type `RandomGenerator` is simply a type abbreviation for a function that takes two integers and returns an integer:

```
type RandomGenerator = int * int -> int
```

Call `rand(l,h)` to get a number in the range `l` to `h`, inclusive. Successive calls to that function will return a sequence of random values. Documentation in the code describes how to make a `RandomGenerator` function with `makeRand`. You may wish to use this function while testing your `build` function.

Once you have completed `build`, you can generate pictures by calling the function

```
doRandomGray : int * int * int -> unit
```

which, given a maximum depth and two seeds for the random number generator, generates a grayscale image for a random image in the file `art.pgm`. You may also run

```
doRandomColor : int * int * int -> unit
```

which, given a maximum expression depth and two seeds for the random number generator, creates three random functions (one each for red, green, and blue), and uses them to emit a color image `art.ppm`. (Note the different filename extension).

A few notes:

- The build function should not create values of the `Expr` datatype directly. Instead, use the build functions `buildX`, `buildY`, `buildSine`, etc. that I have provided in `expr.sml`. This provides a degree of modularity between the definition of the `Expr` datatype and the client. We will look at how to enforce this separation with the ML module system in a few more weeks.
- A depth of 8 – 12 is reasonable to start, but experiment to see what you think is best.

- If every sort of expression can occur with equal probability at any point, it is very likely that the random expression you get will be either `VarX` or `VarY`, or something small like `Times(VarX,VarY)`. Since small expressions produce boring pictures, you must find some way to prevent or discourage expressions with no subexpressions from being chosen "too early". There are many options for doing this— experiment and pick one that gives you good results.
- The two seeds for the random number generators determine the eventual picture, but are otherwise completely arbitrary.

(f) **Extensions:** Extend the `Expr` datatype with at least three more expression forms, and add the corresponding cases to `exprToString`, `eval`, and `build`. The two requirements for this part are that:

  i. these expression forms must return a value in the range [-1,1] if all subexpressions have values in this range, and
  ii. at least one of the extensions must be constructed out of 3 subexpressions, ie. one of the new `build` functions must have type `Expr * Expr * Expr -> Expr`.

  There are no other constraints; the new functions need not even be continuous. Be creative! Make sure to comment your extensions.

(g) **Turnin:** To submit your work, use the turnin script: "`turnin -c 334` *file*". Submit:

  - `expr.sml`,
  - `art.sml`, and
  - your two favorite pictures generated by the program.

  We'll put these up on the web page. Feel free to submit more if you wish.

Include printouts of `expr.sml` and `art.sml` when you hand in your problem set.

6. *(10 points)* .... Expression Representation (Bonus Question)

This question explores an alternative way of representing expressions in the random art program.

Create a new file `expr-func.sml` which, like the file `expr.sml`, defines the `Expr` representation and basic operations on it. In this version, the definition of the type `Expr` should be not a datatype, but:

```
type Expr =  real * real -> real
```

That is, instead of the symbolic representation used by `expr.sml`, this implementation will represent each function in x and y directly as an SML function of two `real` arguments. Redefine the following operations on the new type:

- `exprToString`
- `eval`
- `buildX`, `buildY`, `buildSine`, etc.

The `eval` function in particular becomes much, much simpler than in `expr.sml`, but the `exprToString` function cannot be written successfully, since there is no way to convert an ML function to a string. Thus, your implementation of this function can return something like `"<function>"` or `"unknown"`. To test your code, replace

```
use "expr.sml";
```

at the top of `art.sml` with

```
use "expr-func.sml";
```

Use `turnin` to turn in `expr-func.sml`, and include a printout with your solutions to the other problems.