Reading

1. Read Mitchell, Chapter 5.
2. Read Ullman, as needed for the programming questions.

Problems

1. (10 points) ML Map for Trees
   Mitchell, Problem 5.4
   When you run your program, 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
   
   UnCurry(Curry(f))(s, t) = f(s, t)
   
   and
   
   Curry(UnCurry(g)) s t = g s 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 \( \text{num} = x.i \) and \( \text{str} = x.s \), 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 \( x.s \).

4. (15 points) ................................. Higher-Order Functions

One of the advantages of functional languages is the ability to write high-level functions which capture general patterns. For instance, in class we defined the “listify” function which could be used to make a binary operation apply to an entire list.

(a) Your assignment is to write a high-level function to support list abstractions. The language Miranda allows the user to write list abstractions of the form:

\[
[f(x) \mid x \leftarrow \text{startlist}; \text{cond}(x)]
\]

where \( \text{startlist} \) is a list of type \( 'a \), \( f : 'a \rightarrow 'b \) (for some type \( 'b \)), and \( \text{cond} : 'a \rightarrow \text{bool} \). This expression results in a list containing all elements of the form \( f(x) \), where \( x \) is an element in the list, \( \text{startlist} \), and \( \text{cond}(x) \) is true. For example, if \( \text{sqr}(x) = x \times x \) and \( \text{odd}(x) \) is true iff \( x \) is an odd integer, then

\[
[\text{sqr}(x) \mid x \leftarrow [1, 2, 5, 4, 3]; \text{odd}(x)]
\]

returns the list \([1, 25, 9]\) (that is, the squares of the odd elements of the list \(-1, 5, 3\)). Note that the list returned preserves the order of \( \text{startlist} \).

You are to write a function

\[
\text{listcomp} : ('a \rightarrow 'b) \rightarrow ('a \text{ list}) \rightarrow ('a \rightarrow \text{bool}) \rightarrow ('b \text{ list})
\]

so that

\[
\text{listcomp} f \text{ startlist cond} = [f(x) \mid x \leftarrow \text{startlist}; \text{cond}(x)].
\]

(Hint: One way to do this is to divide the function up into two pieces, the first of which calculates the list, \([x \leftarrow x \mid \text{startlist}; \text{cond}(x)]\), and then think how the “map” function can be used to compute the final answer. It’s also pretty straightforward to do it all at once.)

(b) Test your function by writing a function which extracts the list of all names of managerial employees over the age of 60 from a list of employee records, each of which has the following fields: “name” which is a string, “age” which is an integer, and “status” which has a value of managerial, clerical, or manual. (Be sure to define this function correctly. I’m always amazed at the number of people who miss this problem by carelessness!)

(c) Generalize your function in part a to

\[
\text{listcomp2} g \text{ slist1 slist2 cond} =
\]

\[
[g \times y \mid x \leftarrow \text{slist1}; y \leftarrow \text{slist2}; \text{cond} x y]
\]

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

This question will guide you through writing the ML code to generate random pictures like the following.
This problem brings together a number of topics we have studied, including grammars, parse
trees, and evaluation. You 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:

\[ e ::= x \mid y \mid \sin \pi e \mid \cos \pi e \mid (e + e)/2 \mid e \ast e \]

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:

```ml
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
\texttt{Sine} represents the application of the \texttt{sin} function to an argument multiplied by \(\pi\). 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.

\((\circ)\) **printing expressions:** The first two parts require that you edit and run only expr.sml.
First, write a function

\[
\text{exprToString} : \text{Expr} -> \text{string}
\]

to generate a printable version of an expression. For example, calling \texttt{exprToString} on the expression

\[
\text{Times}\left(\text{Sine}\left(\text{VarX}\right), \text{Cosine}\left(\text{Times}\left(\text{VarX}, \text{VarY}\right)\right)\right)
\]

should return a string similar to “\texttt{sin(pi*x)} * \texttt{cos(pi*x*y)}”. The exact details are left to you. (Remember that string concatenation is performed with the “\texttt{+}” operator.)
Test this function on a few sample inputs before moving to the next part.
(b) **expression evaluation:** Write the function

```plaintext
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 `expr.sml`, complete the definition of the function

```plaintext
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 \((e_1 ; e_2)\) 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:

```plaintext
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 declared above. How could you force it to have the specified type, and why might it be useful to do that?

(d) **Viewing Pictures:** You can view .pgm files, as well as the .ppm files described below, under X-windows with the `xv` program. (You get to the menu in this program by right-clicking on the picture.) To view the output from a non-Unix machine, or to post them on the web, etc., you might need to first convert the file to jpeg format with the following command:

```plaintext
ppmtojpeg picture-filename > moo.jpg
```

Despite the name of this command, it will work for both .ppm and .pgm files.

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

```plaintext
build(depth, rand) : int * (int*int->int) -> 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 value of type int * int -> int, which is a random-number-generator. 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.

You can test your code by running the function
doRandomGray : int * int * int -> unit

which, given a maximum depth and two seeds for the random number generator, generates a random file and emits it as the grayscale image art.pgm, or by running the function
doRandomColor : int * int * int -> unit

which, given a maximum expression depth and two seeds for the random number generator, creates three random functions 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.

Extensions: Extend the Expr datatype with at least four more expression forms, and add the corresponding cases to exprToString, eval, and build. The two requirements for this part are

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. it 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.

Turnin: 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.