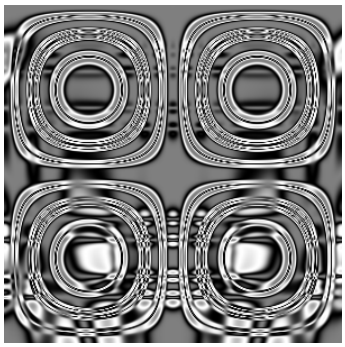
Assignment 2

Random Art

Code Due: 11:59 PM, Wednesday, September 21, 2005



Goals

1. To practice working with higher-order functions
2. To illustrate the difference between interface and implementation
3. To create interesting pictures!

Startup

To begin, make a copy of all the files from the directory `/cs/cs131/src/ass2`:

`expr-sig.sml`	EXPR signature, describing expressions in x and y
`dexpr.sml`	An implementation of expressions satisfying EXPR (partial)
`art.sml`	Main routines for creating pictures (partial)
`sources.cm`	The makefile-equivalent for the Compilation Manager

Because this assignment involves a number of files, instead of loading code with `use` you should use the SML/NJ Compilation Manager. This system automatically figures out what files need to be recompiled and, quite usefully, in what order. The file `sources.cm` contains the list of files to be loaded, and is read by the recompilation command,

```
CM.make "sources.cm";
```

Submission

The submission process has *two* parts:

1. You must submit all of the `.sml` and `.cm` code files, even those you haven't changed, so that the graders can easily compile and run your program. You can do so by submitting each of the files individually with `cs131submit`, or by logging on to turing and running

   ```
   cs131submitall
   ```

   which submits *all* the `.sml` and `.cm` files in the current directory.

2. **Important:** To receive credit for this assignment, you must also provide your favorite output file, but not via submit. To save space and improve portability, first convert your picture to JPEG format, using the command

   `pnmtojpeg` *picture-filename* `>` *picture-name*`.jpg`

   Note that you'll need to have `/cs/cs131/bin` in your path to run this command. Also, don't leave out the `>`.

   Then submit your JPEG file by logging on to turing and running

   `submitjpg` *picture-filename*`.jpg`

You can look at your pictures before submitting them using the `display` command on turing, wilkes and knuth, and using Preview on Macs (`display`, which is part of the ImageMagick suite, can view `.ppm` and `.pgm` files directly but Preview needs to have the file converted to a `.jpg` first—there are also numerous other applications you might also be able to use to view these images on your own computers, including `xv` and `eog` on LINUX and Graphic Converter on the Mac).

You can view everyone's submitted pictures using your Web browser, via the URL `http://www.cs.hmc.edu/cs131/homework/pictures`.

1 Expression Evaluation (5%)

The code in `dexpr.sml` is an almost-complete implementation satisfying the `EXPR` signature (interface), shown in Figure 1. In this implementation, expressions are represented as values from a `datatype`. Finish this code by writing the function

```
eval : expr -> real*real -> real
```

that evaluates 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 by `Expr` as `Sin(X)` is supposed to correspond to the mathematical expression $\sin(\pi x)$, and so the `eval` function must be defined appropriately.)

Remember that since the definitions are now inside a module (unlike the code you wrote for the first assignment), outside the module (e.g., when testing at the command line) you have to say `Expr.eval` or `Expr.build_avg`).

> *Note:* Putting "`Expr.`" on the front of the functions you're using can be tedious when debugging. You can, of course, type `open Expr;` at the SML prompt, to allow you refer to `eval` or `build_avg` directly, but if you do, to will need to *remember* to reopen the module if you recompile it. If you forget, you may end up running an older version of your code and get very confused.

```
signature EXPR =
  sig
    type expr           (* type whose values represent expressions
                           in x and y *)

    (* Expression creation *)
    (* All expressions should return values in the range [-1,1] when
       x and y also range over [-1,1]. *)

    val build_x      : expr          (* returns the expression for "x" *)
    val build_y      : expr          (* returns the expression for "y" *)
    val build_sin    : expr -> expr  (* creates   sine (PI * given expr) *)
    val build_cos    : expr -> expr  (* creates   cosine (PI * given expr) *)
    val build_avg    : expr * expr -> expr  (* average of the two exprs. *)
    val build_times  : expr * expr -> expr  (* product of the two exprs. *)

    (* Expression evalation *)
    val eval : expr -> real*real -> real

    (* Pretty-printer for expressions *)
    val ppexpr : expr -> unit

    (* A sample reasonably complicated expression *)
    val sampleExpr : expr
  end
```

Figure 1: `expr-sig.sml`

2 Finishing the Driver Code (5%)

The code in `art.sml` is the driver for the entire program; it includes the `doGray` and `doColor` 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. The bad news is that SML does not include `for` loops. The good news is that we can use higher-order functions to implement them.

In `art.sml`, fix the definition of the function

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

The argument triple contains a lower bound, and 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. For example,

```
Art.for (1,5, fn x => print (Int.toString x))
```

should print `12345` (without any spaces or newlines).

Test your code by producing a grayscale picture of the expression `Expr.sampleExpr`. (Hint: look at the function `Art.emitGray`.)

> *Note:* There are two natural ways in SML to do something, throw away its result, and then do something else. You can either say `val _ =` e_1 inside a `let` to evaluate an expression e_1 and throw away its result, or you can use the expression form `(`e_1 `;` e_2`)`. Inside an expression a semicolon acts exactly like the comma operator in C or C++ (namely, it evaluates the first expression, throws away the result, and then returns the result of the last expression).

3 Generating the Random Expressions (40%)

Your next programming task is to fix the definition of

```
build : int * Random.rand -> Expr.expr
```

in `art.sml`, which you may implement in any way you like. The first parameter to `build` is a maximum nesting depth that the resulting expression should have, and the second parameter is a random-number generator. (A bound on the nesting depth keeps the expression to a manageable size; it's easy to write a naïve expression generator that generates incredibly enormous expressions.) When you reach the cutoff point, you can simply return an expression with no sub-expressions, such as `Expr.x` or `Expr.y`.

The second argument, a value of type `Random.rand`, is a random-number generator (which might be thought of as an infinite source of random numbers). You can use the functions in the `Random` structure to pull out new random numbers from this generator. The implementation of `Random` shouldn't matter to you at all, but you will want to look at its interface, shown in Figure 2.

Now, 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 `Expr.x` or `Expr.y`, or something small, such as `Expr.times(Expr.x,Expr.y)`. Bacause small expressions produce boring pictures, you must find some way to prevent or discourage expressions with no subexpressions from being chosen "too early".

Once you have successfully recompiled with `CM.make`, you can test your code by running the function

```
Art.doGray : int * int * int -> unit
```

which, given a maximum depth and two integers that together form the seed for the random-number generator, generates an image of a random function and emits it as the grayscale image `art.pgm`, or by running the function

```
Art.doColor : int * int * int -> unit
```

which takes the same arguments as `doGray`, but instead creates *three* random functions and uses them to emit a color image `art.ppm` (note the different filename extension). A depth of 10 or 11 is reasonable to start, but experiment to see what you think

```
signature RANDOM =
  sig

    type rand
        (* the internal state of a random number generator *)

    val rand : (int * int) -> rand
        (* create rand from initial seed *)

    val toString : rand -> string
    val fromString : string -> rand
        (* convert state to and from string
         * fromString raises Fail if its argument
         * does not have the proper form.
         *)

    val randInt : rand -> int
        (* generate ints uniformly in [minInt,maxInt] *)

    val randNat : rand -> int
        (* generate ints uniformly in [0,maxInt] *)

    val randReal : rand -> real
        (* generate reals uniformly in [0.0,1.0) *)

    val randRange : (int * int) -> rand -> int
        (* randRange (lo,hi) generates integers uniformly [lo,hi].
         * Raises Fail if hi < lo.
         *)

  end; (* RANDOM *)
```

Figure 2: The interface provided by SML/NJ's `Random` structure

works best. The two seeds for the random number generators determine the eventual picture, but are otherwise completely arbitrary.

You can view these files under X Window System with the `display` program. (You get to the menu in this program by right-clicking on the picture.) To view the output from a non-Unix machine you might need to first convert the file to JPEG format; see the submission information on page 2.

4 An Alternate Representation (30%)

Create a new file, `fexpr.sml`, which, like the file `dexpr.sml` that you were given, contains a structure `Expr` satisfying the `EXPR` signature. 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 `dexpr.sml`, this implementation will represent each function in x and y directly as an SML function of two `real` arguments. You should define all the functions required by the `EXPR` interface. In particular, the `eval` function becomes much, much simpler than in `dexpr.sml`, but the `ppexpr` function cannot be written successfully, since there's no way to convert an ML function to a string. Thus, your implementation of this function can return something like the the string `"<function>"` or `"unknown"`. To test your code, remove the name `dexpr.sml` from the file `sources.cm` and replace it with the name `fexpr.sml`, and recompile.

5 Extensions (10%)

Extend the `EXPR` signature with three more expressions forms (be creative!); add the corresponding implementations to both the `dexpr.sml` and `fexpr.sml` files; and extend the `Art.build` code to use these new forms.

The two requirements on the new expression forms that you create

1. Each mush must return a value in the range [−1.0,1.0] when its arguments are within the range [−1.0,1.0]

2. At least one of the three must be constructed out of three subexpressions, i.e., one of the new `build` functions should 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 in the `EXPR` signature.