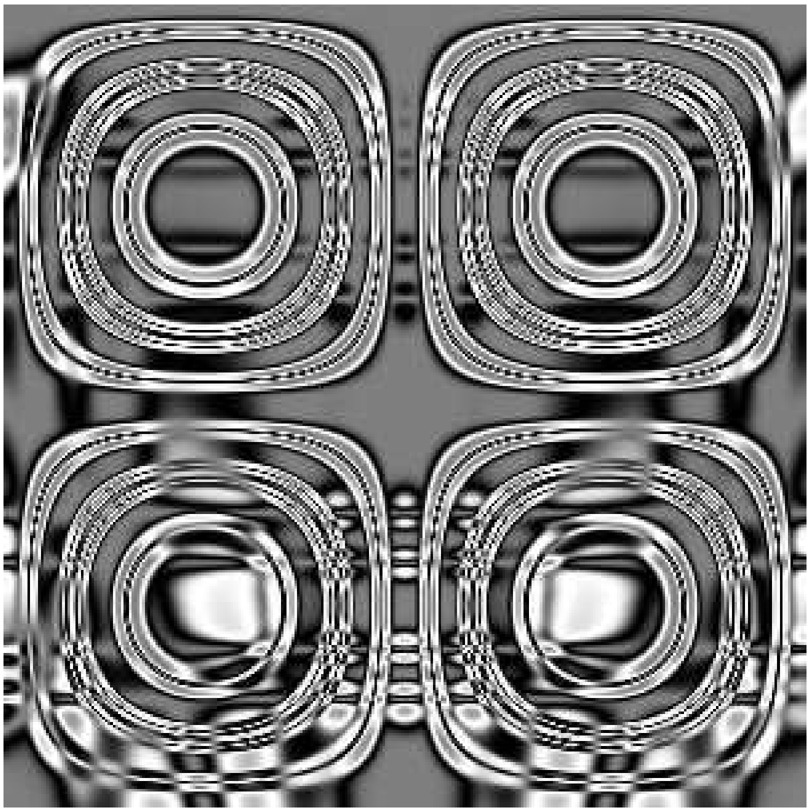
Computer Science 131, Fall 2001
Assignment 2: Random Art

Out: Thursday, September 13

Due: Thursday, September 20, 2:45pm



Overview

The primary goal of this assignment is to give you practice working with higher-order functions and modules in Standard ML. The secondary goal is to create interesting pictures!

Startup

To begin, you should make a copy of all the files from the directory

```
/cs/cs131/src/2
```

The files are as follows:

`lib-base-sig.sml`	Used by the random-number-generator; you should ignore it
`lib-base.sml`	ditto
`random.sml`	Implementation of `Random` structure; should also ignored
`random-sig.sml`	Interface of the `Random` structure
`expr-sig.sml`	`EXPR` signature, describing expressions in x and y
`dexpr.sml`	An implementation of expressions satisfying `EXPR`
`build-expr.sml`	Module which builds random expressions (incomplete)
`art.sml`	Driver routines for emitting pictures (mostly complete)
`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 re-compilation command:

```
CM.make();
```

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 this by submitting each of the files individually with `cs131submit`, or by running

   ```
   cs131submitall
   ```

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

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

   `/cs/cs131/bin/ppmtojpeg` *picture-filename* `>` *your-user-id*`.jpg`

Don't leave out the >. Despite the name of this command, it will work for both `.ppm` and `.pgm` files. Then copy this jpeg file to the following directory on turing:

```
/mnt/web/www/courses/current/cs131/assignments/pictures
```

Finally, make sure that this file is world readable:

```
chmod 644  your-user-id.jpg
```

You can look at everyone's pictures (or show yours to your friends) by either loading these files into `xv`, or pointing a web browser at this directory via the URL

```
http://www.cs.hmc.edu/courses/current/cs131/assignments/pictures
```

1 Finishing the Driver Code (10%)

The code in `art.sml` is the driver for the entire program; it includes the `emit` and `cemit` 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.

2 Generating the Random Expressions (30%)

The interface `EXPR` for a structure implementing expressions is given in the file `expr-sig.sml`. Read over this file. Since the expression values and expression-generating functions like `x` and `sin` are *not* defined to be datatype constructors, this interface does not permit you to use them in pattern-matching. You may also want to look at the file `dexpr.sml` which contains a structure `Expr` satisfying the `EXPR` signature by using datatypes to represent expressions.

Your next programming task is to complete the code in `buildexpr.sml`. The only function required by the interface of the `BuildExpr` structure is:

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

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 managable size; it's easy to write a naive expression generator which can generate incredibly enormous expressions. When you reach the cut-off point, you can simply return an expression with no sub-expressions, such as `Expr.x` or `Expr.y`.

To generate random numbers, you should use the functions in the `Random` structure; the implementation doesn't matter but the interface of this module is specified in `random-sig.sml`. Every time you extract a random number from a generator, which is a value of type `Random.rand`, the generator's internal state is automatically updated. The next time you ask for a random number from the same generator, you'll get a different answer, unless the same number just happened to randomly reoccur.

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 like `Expr.times(Expr.x,Expr.y)`. Since small expressions produce boring pictures, you should 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.emit : 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

```
Art.cemit : 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). Most of the pictures you saw in class used a depth of 10 or 11, but experiment to see what you think is best. The seeds for the random number generators determine the eventual picture, but are otherwise completely arbitrary.

You can view these files 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 you might need to first convert the file to jpeg format; see the submission information above for details.

3 An Alternate Representation (20%)

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` is 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. (The `ppexpr` function cannot be written successfully, since there's no way to convert an ML function to a string, so your implementation of this function should return something like the empty string `""` or `"unknown"`.) To test your code, remove the name `dexpr.sml` from the file `sources.cm` and replace it with the name `fexpr.sml`.

IMPORTANT: In this one file, you should not depend on type inference for the expression-creating functions. That is, write explicitly

```
val x : expr = ...
```

and

```
fun sin(e : expr) = ...
```

and so on.

4 Timings (10%)

Try generating the same picture (same depth and seeds) with both the datatype implementation of expressions and the function implementation. Remember, you can switch back and forth by changing the `sources.cm` file and re-compiling with `CM.make()`.

Time how long it takes to generate the same picture using the two representations. (You should use a depth large enough that a moderate speed difference would become clearly apparent.) Does one representation appear faster than the other? Explain in a comment in your `fexpr.sml` file your results, and suggest a reason for any signifiant difference you observed.

5 Functorizing (20%)

IMPORTANT: Make sure your code is working and you have take the timings in the previous step before making these changes! It is also a good idea to make a copy of all your source files before starting this step.

You have replaced the datatype implementation of expressions with an implementation in terms of functions. Unfortunately, as noted previously there's no good way to implement the new `Expr.ppexpr` function. We will address this problem, and practice using functors, by changing the code to generate each expression in both representations. The datatype representation will be used for display purposes, and the function representation will be used for computations.

- If we want to have both representations in the same program, the `dexpr.sml` and `fexpr.sml` files can no longer define structures with the same name. Change these files to name the structures `DExpr` and `FExpr`, respectively. Then make sure both of these files appear in your `sources.cm` file.

- We want to be able to build expressions in both representations. So, instead of `BuildExpr` being a structure that refers directly to the `Expr` structure (or directly to `FExpr` or `DExpr`), we must make it into a *functor* parameterized by a structure `Expr` structure matching the signature `EXPR`. Make this change in `buildexpr.sml`. Note that the signature of the the structure returned by the `BuildExpr` functor will now refer to the parameter's type `expr`.

- Finally, we need to modify `art.sml` appropriately. Add code to create two expression-building structures `DBuildExpr` and `FBuildExpr` by applying the `BuildExpr` functor to the two representations. Then, modify the `emit` and `cemit` functions to generate each expression (or group of expressions) twice, once for each representation, and to use the functional representation for evaluation and the datatype representation for printing. You will need *two* random number generators, both started with the same initial seeds.

6 Extensions (10%)

Extend the `EXPR` signature with at least three different expressions forms (or more if you feel like it), add the corresponding implementations to both the `dexpr.sml` and `fexpr.sml` files, and extend the `BuildExpr.build` code to use these new forms. The only requirements are (1) that these expression forms must return a value in the range $[-1.0, 1.0]$, assuming that all subexpressions (if any) return values in this range, and (2) your expressions should be purely mathematical and deterministic; no calling a random-number generator or loading in external bitmaps. There are no other constraints; the new functions need not even be continuous.

You may want to look at the structure `Real.Math` (which can also be referred to simply by the name `Math`) in the SML Basis Library (the last link on the main course page) to see what other mathematical functions are available.

Make sure to comment your extensions to the `EXPR` signature. Also, state whether you think there was a visible change in the sorts of pictures generated due to the extensions you added.