Homework 6 — ML module language — assigned Wednesday 10 March — due Tuesday 30 March

All ML code in this assignment must be encapsulated in appropriate modules. Be careful to distinguish internal (auxiliary or abstract) component from those that should be visible to the outside, and write the signatures accordingly. Document the purpose of signatures and structures. Document how you chose between opaque and transparent ascription of signatures.

Reading assignment

Read Chapter 8 of *ML for the Working Programmer*, focusing on input-output; explore available input-output primitives in the online SML Basis Library documentation. Read about the mutable store in ML (type constructors `ref` and `array`), but do not use it in your work.

Read Chapter 7 of *ML for the Working Programmer*, but ignore all references to `abstype`, which is obsolete.

In preparation for lectures on 24 March and 29 March, read Chapter 9 of *ML for the Working Programmer*.

6.1 Input and output (10pts)

In exercises 5.1 and 4.1, we wrote the functions `parse: string -> expr` and `eval: expr -> int`, where `expr` is:

```
datatype expr = Num of int
              | Add of expr * expr
              | Mul of expr * expr
```

Learn how to read a string from standard input, and how to write an integer to standard output. Write a function `calc: unit -> unit` that accepts from the user a string supposed to contain an arithmetic expression, parses it using `parse`, evaluates the expression using `eval`, and prints the value. If there is a syntax error in the input, the error should be reported to the user.

6.2 Real functions (90pts)

Warning: In this exercise, the word *function* refers both to ML functions and to mathematical functions. The intended meaning should be clear from the context.

In this exercise we use the following representation for a class of mathematical functions that is a subset of $\mathbf{R} \to \mathbf{R}$:

```
datatype expr = Num of real
              | IntNum of int
              | ConstE
              | ConstPi
              | Var of string
              | Let of {var: string, value: expr, body: expr}
              | Neg of expr
              | Add of expr * expr
              | Sub of expr * expr
              | Mul of expr * expr
              | Div of expr * expr
              | Sin of expr
              | Cos of expr
              | Tan of expr
              | Arctan of expr
              | Exp of expr
              | Ln of expr
              | Power of expr * expr

type env = string -> real
exception Unbound of string
val emptyEnv: env = fn s => raise (Unbound s)
fun extendEnv oldEnv s n s' = if s' = s then n else oldEnv s'

exception FunctionUndefinedAtArgument of string
```

Note that we have an explicit `Let` construct, similar to a *where* clause in ordinary mathematical usage. The environment type `env` can be used to supply values for the free variables of an expression.

Usage example: the mathematical expression $\sqrt{2\pi}\frac{\sin x}{x}$ can be represented as

```
Mul (Power (Mul (IntNum 2, ConstPi), Div (IntNum 1, IntNum 2)),
     Div (Sin (Var "x"), Var "x")
    ) : expr
```

6.2.1 Evaluator (10pts)

Write the ML function:

```
evalExpr: {
           func: expr,
           env: env
          } -> real
```

which evaluates the expression `func` in the environment `env`. If there exists some variable that is free in the expression `func` but not defined in the environment `env`, then the ML function `evalExpr` should raise an exception.

Usage example: evaluating the mathematical function $x \mapsto e^{-x^2}$ at the point $x = 2.3$ can be represented as

```
evalExpr {func = Exp (Neg (Power (Var "x", IntNum 2))),
          env = extendEnv emptyEnv "x" 2.3
         }
```

which should evaluate to $5.04 \cdot 10^{-3}$.

6.2.2 Drawing (40pts)

Write the ML function:

```
plotExprs: {
            fies: {func: expr, indep: string, env: env} list,
            interval: {lower: real, upper: real},
            numPoints: int
           } -> string
```

which is a combination of an expression evaluator and a function plotter that generates PostScript output (exactly as in Homework Exercise 3.3). The ML function `plotExprs` should plot each supplied function expression over the given interval with the given number of internal points. The `indep`s provided are the variables that are to be considered as the independent variables of the given `func`s. The value of each such independent variable ranges over the same given `interval`; note that the particular variable name may be different in each of the function expressions. The `env`s should provide the values for any other free variables of each of the `func`s.

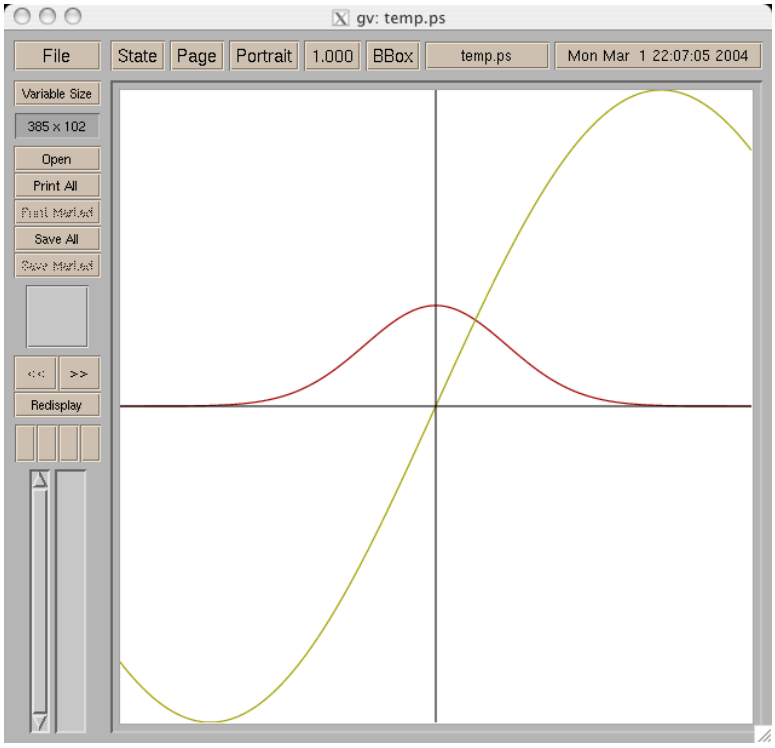
The ML function `plotExprs` should handle singularities gracefully and still produce an appropriate plot. Hint: we suggest two approaches: one is building into the evaluator certain knowledge of the behavior of elementary real functions, such as, e.g., that the function ln is defined only for positive arguments; and the other is relying on the behavior of SML Basis Library functions from the structure `Real` and inspecting their results.

The drawing area of the resulting PostScript program should be *consistent* for functions with arbitrary domains and ranges; to make this precise, we insist that the bounding box in the generated PostScript must be 0 0 500 500, and the plots should fully use the drawing area.

Usage example: to plot the mathematical functions $x \mapsto e^{-x^2}$ and $t \mapsto A\sin\omega t$ with parameter values $A = \pi$ and $\omega = 0.7$ on the interval $[-\pi, \pi]$, invoke

```
plotExprs {
           fies =
             [
              {
               func = Exp (Neg (Power (Var "x", IntNum 2))),
               indep = "x",
               env = emptyEnv
              },
              {
               func = Mul (Var "A", Sin (Mul (Var "omega", Var "t"))),
               indep = "t",
               env = extendEnv (extendEnv emptyEnv "A" Math.pi) "omega" 0.7
              }
             ],
           interval = {lower= ~Math.pi, upper= Math.pi},
           numPoints = 1000
          }
```

The result should look like:



6.2.3 Symbolic differentiation (40pts)

Write the ML function:

```
diffExpr : string -> expr -> expr
```

which takes a string representing a variable *x* and an expression `e` representing a real function f, and returns another expression `e'` such that `e'` represents the real function $\frac{df}{dx}$, the derivative of f with respect to *x*.

Differentiating `expr`s is a matter of translating the well-known formulas of calculus into transformations over the datatype `expr`. Particular attention, however, should be given to handling the `Let` expression correctly.

Usage example:

```
diffExpr "t" (Mul (Var "A", Sin (Mul (Var "omega", Var "t"))))
```

should evaluate to:

```
Mul (Var "A", Mul (Var "omega", Cos (Mul (Var "omega", Var "t"))))
```

(or something algebraically equivalent).

Extra credit: wherever possible, simplify the result, to replace constant expressions (such as `Add (Num 3.0, Num 4.0)`) with number constants (in this example, `Num 7.0`); also simplify additions by zero and multiplications by zero and one, and expressions raised to the zeroth or first power.

6.3 Using lists for arithmetic: extra credit

This is an extension of exercises 2.4 and 3.2.

1. Use an appropriate datatype to represent integers of arbitrary size, including negative integers, and repeat exercises 2.4 and 3.2.
2. Use an appropriate datatype to represent floating-point numbers of arbitrary size and precision, and repeat exercises 2.4 and 3.2. Represent a floating-point number as a sign, an integer mantissa (of arbitrary size) and an integer exponent (which may be restricted to the range representable by the ML `int` type).
3. Implement a square-root function $\sqrt{x}$ for these floating-point numbers.
4. Implement an exponentiation function e^x for these floating-point numbers.
5. Using these floating-point numbers, evaluate e to 300 digits of precision. Check your result.
6. Using these floating-point numbers, evaluate π to 300 digits of precision. Check your result.
7. Using these floating-point numbers, evaluate $e^{\pi\sqrt{163}}$ to 300 digits of precision. Ramanujan said this number was nearly an integer. Check.

How to turn in

Make sure that you have thoroughly tested your code, and include all your test runs!

Turn in your code by running

~clint/handin your-file

on a regular UNM CS machine. You should use whatever filename is appropriate in place of your-file.

Include the following statement with your submission, signed and dated:
I pledge my honor that in the preparation of this assignment I have complied with the University of New Mexico Board of Regents' Policy Manual, including Section 4.8, Academic Dishonesty.