Prog. Lang. Design and Implementation	Homework 3
ENGR 3599 Spring 2014 (Pucella)	Due Date: Friday, Feb 21

This homework is meant to be done individually. You may discuss problems with fellow students, but all work must be entirely your own, and should not be from any other course, present, past, or future. If you use a solution from another source, please cite it. If you consult fellow students, please indicate their names.

Submission information:

- For this problem set, solutions should be put in a file called `homework3.sml`.
- Your file `homework3.sml` should be emailed as an attachment before the beginning of class the day the homework is due at the following address:

  `homeworks.pldi.sp14@gmail.com`

- Your file `homework3.sml` should begin with a block comment that lists your name, your email address, and any remarks that you wish to make about the homework.

Notes:

- All code should compile in the latest release version of Standard ML of New Jersey.
- There is a file `homework3.sml` available from the web site containing code you can use, including code described in this write-up.
- All questions are compulsory.
- In general, you should feel free to define any helper function you need when implementing a given function. In many cases, that is the cleanest way to go.

Our Interpreter

In this homework, we put together everything we've done until now: the internal representation and its evaluation function, the lexer, and the parser, and produce a workable interpreter. Most of the code is given already, and it's really just a combination of everything you've done before. It's just been cleaned up a bit with some of the rough edges smoothed out.

Internal Representation

The internal representation is yet another variant of the intermediate representation we saw in class: it supports multiple-argument functions, it has rational numbers, and it has matrices. (No vectors—we can consider a matrix with a single row as a vector.) The intent is to approximate a language such as MATLAB.

```
datatype value = VRat of int * int                (* rational number   *)
               | VBool of bool                    (* Boolean           *)
               | VMat of value list list          (* matrix            *)

datatype expr = EVal of value                     (* value             *)
              | EAdd of expr * expr               (* addition          *)
              | ESub of expr * expr               (* subtraction       *)
              | EMul of expr * expr               (* multiplication    *)
              | EDiv of expr * expr               (* division          *)
              | EEq of expr * expr                (* equality          *)
              | EIf of expr * expr * expr         (* conditional       *)
              | ELet of string * expr * expr      (* let binding       *)
              | EIdent of string                  (* identifier        *)
              | ECall of string * expr list       (* function call     *)
              | EEye of expr                      (* identity - see Q1 *)
              | EMatrix of expr list list         (* matrix - see Q2   *)

datatype function = FDef of string list * expr
```

Note that there are no integer values: we use a rational number with denominator 1 to represent integers. That keeps things simple. We can always convert such rational numbers to *bona fide* integers when we print them.

Everything is as we have seen before, including matrix values `VMat m` represented by a list of rows, except that here a matrix can contain arbitrary values as entries. Thus,

```
VMat [[VRat (1,1), VRat (0,1)], [VRat (0,1), VRat (1,1)]]
```

is the 2×2 identity matrix $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$.

As far as operations are concerned, you can add/subtract/multiply/divide rational numbers, and you can add/subtract/multiply matrices (as long as their sizes are compatible). Additionally, you can add/subtract/multiply/divide matrices and rationals together, and like in MATLAB, the operation will be applied to every element of the matrix. Thus, adding $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ to 1 yields the matrix $\begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}$. Similarly, you can divide 1 by the matrix $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$ to get $\begin{pmatrix} 1 & 1/2 \\ 1/3 & 1/4 \end{pmatrix}$.

There is one new expression form completely implemented: `EMatrix`, which is an expression to construct a matrix out of expressions for its elements. For instance, the internal representation

```
EMatrix [[EVal (VRat (1,1)), EAdd (VRat (1,1), VRat (2,1))],
         [ESub (VRat (3,1), VRat (1,1)), EVal (VBool true)]]
```

evaluates to the value

```
VMat [[VRat (1,1), VRat (3,1)], [VRat (2,1), VBool true]]
```

(Yes, I know, there's a Boolean value in the matrix. Why not? It's only a problem if you try to add it to some other matrix.)

To evaluate `EMatrix`, you evaluate all the entries in the matrix to values, and create the matrix value from those values.

All of this has been implemented for you, essentially lifted from what we did in homework 1. (Except there, we were restricted to matrices of integers—now we have arbitrary values; have a look at the code to see how it's done.)

The main functions in that part of the code are:

```
eval : (string * function list) -> expr -> value
stringOfValue : value -> string
printValue : value -> unit
```

where `eval` is our standard evaluation function that takes a function environment and an expression in our internal representation and evaluates the expression to a value, and `stringOfValue` converts a value into a nice human readable representation for printing purposes. Function `printValue` prints a value via `stringOfValue`.

(There is also a `stringOfExpr` function that gives you a string representation of an expression of the internal representation, which might be useful for debugging purposes.)

Lexer and Parser

The lexer is a straightforward adaptation of the lexer we saw in class, with the following tokens:

```
datatype token = T_LET                    (*  let                        *)
               | T_SYM of string          (*  e.g. hello, world, abc123  *)
               | T_INT of int             (*  e.g. 1, 2, 3, ~4           *)
               | T_TRUE                   (*  true                       *)
               | T_FALSE                  (*  false                      *)
               | T_PLUS                   (*  +                          *)
               | T_TIMES                  (*  *                          *)
               | T_MINUS                  (*  -                          *)
               | T_SLASH                  (*  /                          *)
               | T_EQUAL                  (*  =                          *)
               | T_IF                     (*  if                         *)
               | T_LPAREN                 (*  (                          *)
               | T_RPAREN                 (*  )                          *)
               | T_COMMA                  (*  ,                          *)
               | T_EYE                    (*  eye   - see Q1             *)
               | T_SEMICOLON              (*  ;     - see Q2             *)
               | T_LBRACKET               (*  [     - see Q2             *)
               | T_RBRACKET               (*  ]     - see Q2             *)
               | T_DEF                    (*  def   - see Q3             *)
```

The main functions of the lexer are

```
lex : char list -> token list
lexString : string -> token list
```

both of which produce a list of tokens from their input string.

I've also included a recursive-descent parser (with backtracking), a small variant of the one we saw in class and that I gave you in sample code. Here is the grammar it implements:

```
        expr ::= T_IF expr T_THEN expr T_ELSE expr
                 T_LET T_SYM T_EQUAL expr T_IN expr
                 eterm T_EQUAL eterm
                 eterm

   expr_list :: = expr T_COMMA expr_list
                  expr

       eterm ::= term T_PLUS eterm
```

```
                term T_MINUS eterm
                term

       term ::= factor T_TIMES term
                factor T_SLASH term
                factor

     factor ::= T_INT
                T_TRUE
                T_FALSE
                T_SYM T_LPAREN expr_list T_RPAREN
                T_SYM
                T_LPAREN expr TRPAREN
```

So it's pretty much as you would expect. It is different than the one I presented in class in that it deals with function calls appropriately: `1 + double (x)` should parse now.

As presented in class, the main functions of the recursive-descent parser are

```
val parse_expr :      token list -> (expr * token list) option
val parse_expr_list : token list -> (expr list * token list) option
val parse_eterm :     token list -> (expr * token list) option
val parse_term :      token list -> (expr * token list) option
val parse_factor :    token list -> (expr * token list) option

val parse : token list -> expr
```

There is a parsing function for every nonterminal of the grammar, most of which return an `expr`, except for `parse_expr_list` which returns a list of `expr`.

You will probably only interact with the parser via the `parse` function, which takes a list of tokens and attempts to parse it as an expression of the internal representation; it returns the corresponding `expr` if it succeeds, and raises an exception otherwise.

You can evaluate an expression in the surface syntax by combining functions above in succession:

```
- val v = eval [] (parse (lexString "let x = 1/2 + 1/3 in x / 2"));
val v = VRat (5,12) : value
- printValue v;
5/12
val it = () : unit
```

A Small Shell

The only thing that's missing to make what we have done until now a complete system is a decent way to interact with the language. It's astonishingly straightforward to create a simple shell. Here's the whole code:

```
fun shell fenv = let
    fun prompt () = (print "pldi-hw3> "; TextIO.inputLine (TextIO.stdIn))
    fun pr l = print ((String.concatWith " " l)^"\n")
    fun read fenv =
        (case prompt ()
          of NONE => ()
           | SOME ".\n" => ()
           | SOME str => eval_print fenv str)
    and eval_print fenv str =
        (let val ts = lexString str
             val _ = pr (["Tokens ="] @ (map stringOfToken ts))
             val expr = parse ts
             val _ = pr ["Internal rep = ", stringOfExpr (expr)]
             val v = eval fenv expr
             val _ = pr [stringOfValue v]
         in
             read fenv
         end
         handle Parsing msg => (pr ["Parsing error:", msg]; read fenv)
              | Evaluation msg => (pr ["Evaluation error:", msg]; read fenv))
in
    print "Type . by itself to quit\n";
    read fenv
end
```

You give `shell` an initial function environment, and it spends its time in the two mutually recursive functions `read` and `eval_print`: `read` reads a line of input, and `eval_print` lexes, parses, and evaluates the result before printing it and then goes back to `read`ing a line of input.

The above shell actually dumps out information as it evaluates expressions you type: it prints the list of tokens obtained by the lexing phase, and also the expression obtained by the parsing phase. This is useful for debugging.

Here's a sample run that you can try out for yourself:

```
- shell [];
Type . by itself to quit
```

```
pldi-hw3> let x = 1/2 + 1/3 in x /2
Tokens = T_LET T_SYM[x] T_EQUAL T_INT[1] T_SLASH T_INT[2] T_PLUS T_INT[1]
   T_SLASH T_INT[3] T_SYM[in] T_SYM[x] T_SLASH T_INT[2]
Internal rep =  ELet ("x",EAdd (EDiv (EVal (VRat (1,1)),EVal (VRat (2,1))),
   EDiv (EVal (VRat (1,1)),EVal (VRat (3,1)))),EDiv (EIdent "x",EVal (VRat (2,1))))
5/12
pldi-hw3> .
val it = () : unit
```

The messiness in the code for `shell` is due to the need to print stuff, and to handle exceptions. Note the use of the `let val _ = <some expression with side effects>` idiom to sequence side effects like printing in the midst of evaluating SML expressions.

We can of course supply functions in the initial function environment:

```
- shell [("double", FDef (["n"], EMul (EIdent "n", EVal (VRat (2,1)))))];
Type . by itself to quit
pldi-hw3> double (2)
Tokens = T_SYM[double] T_LPAREN T_INT[2] T_RPAREN
Internal rep =  ECall ("double",EVal (VRat (2,1)))
4
pldi-hw3> double (10/3)
Tokens = T_SYM[double] T_LPAREN T_INT[10] T_SLASH T_INT[3] T_RPAREN
Internal rep =  ECall ("double",EDiv (EVal (VRat (10,1)),EVal (VRat (3,1))))
20/3
pldi-hw3> .
val it = () : unit
```

That's supremely clunky. But don't worry, we'll take care of that in Question 3.

The `map` Function

I mentioned in class that SML functions can be passed around to other functions as arguments and put in data structures such as lists and tuples. In other words, functions are values just like any other. I will have more to say about it, but for now, let me just point out that it opens up a lot of programming vistas.

For instance, suppose that you have a list of values and you want to create a new list containing all the elements of the original list transformed in some way. For instance, let's say that you have a list of integers, and you want to create the list of all those integers but doubled: from `[1,2,3,4]`, produce `[2,4,6,8]`. There is an obvious recursive function to do that:

```
fun doubleList [] = []
  | doubleList (x::xs) = (2*x)::(doubleList xs)
```

and `doubleList [1,2,3,4]` returns `[2,4,6,8]`, as expected.

Suppose instead that you wanted to transform a list of strings and produce the list of their correspondings lengths: from `["goodbye","cruel","world"]`, produce `[7,5,5]`. Again, there's an obvious recursive function to do that:

```
fun sizesList [] = []
  | sizesList (x::xs) = (size x)::(sizesList)
```

(Function `size : string -> int` returns the length of a string.)

There is some functionality in common between `doubleList` and `sizesList`: all they do is apply a transformation to every element of the list, and cons up the results into a new list.

We can make this common functionality manifest by using a single function `map : ('a -> 'b) -> 'a list -> 'b list` that applies a given function to every element of a list, and returns the list of results.

Function `map` takes two argument: a function to apply to every element of the list, and a list to which to apply the function. Since the function can transform elements of the list into any elements of any type, the type of `map` is polymorphic.

Here are some sample interactions with `map` to show you how it works:

```
- fun double x = 2 * x;
val double = fn : int -> int
- map double [1,2,3,4];
val it = [2,4,6,8] : int list
- map double [10,20,30];
val it = [20,40,60] : int list
```

```
- map size ["goodbye", "cruel", "world"];
val it = [7,5,5] : int list
- map size ["RIP", "Philip", "Seymour", "Hoffman"];
val it = [3,6,7,7] : int list
```

Function `map` is built-in in SML, but if it weren't you could define it easily, pretty much as you would expect:

```
fun map f [] = []
  | map f (x::xs) = (f x)::(map f xs)
```

I've use `map` liberally in the code. Have a look. It is supremely useful when you need to transform lists.

You can define functions to *map* a function over any recursive data structure in that way. For instance, in the supplied code, I've provided a `mapMat` function that does what `map` does but over a list of lists instead of a simple list: mapping a given function over all the elements in the sublists while maintaining the overall structure. For example:

```
- fun double x = 2 * x;
val double = fn : int -> int
- mapMat double [[1,2], [3,4,5]];
val it = [[2,4],[6,8,10]] : int list list
```

Question 1 (Warm Up: Identity Matrices)

Right now, out internal representation has matrix values, but the only way to create them in the internal representation is by using `EVal`, and we have no surface syntax that lets us create matrices.

We're going to add a primitive operation to create identity matrices. The operation, inspired from MATLAB, is `eye`. The idea is that `eye (3)`, say, should produce the 3×3 identity matrix

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

I've already added the `EEye` expression to the internal representation. All we need is add support for it in the interpreter. We're going to do this in several steps: extend the evaluation function to deal with the new operation, extend the lexer to deal with it, and then extend the parser to deal with it.

(a) Code a function `applyEye` with type

```
value -> value
```

where `applyEye v` should return the identity matrix of size n (as a `value`) when value `v` represents integer n. Call `evalError` if the value provided is not an integer.

To help you, I've given you a function `identityMat : int -> value list list` that returns a suitable identity matrix of a given size. The function reports an error if the integer provided is less than or equal to zero, since our matrics have size at least 1.

```
- applyEye (VRat (1,1));
val it = VMat [[VRat (1,1)]] : value
- applyEye (VRat (2,1));
val it = VMat [[VRat (1,1),VRat (0,1)],[VRat (0,1),VRat (1,1)]] : value
- applyEye (VRat (4,1));
val it =
  VMat
    [[VRat (1,1),VRat (0,1),VRat (0,1),VRat (0,1)],
     [VRat (0,1),VRat (1,1),VRat (0,1),VRat (0,1)],
     [VRat (0,1),VRat (0,1),VRat (1,1),VRat (0,1)],
     [VRat (0,1),VRat (0,1),VRat (0,1),VRat (1,1)]] : value
- applyEye (VRat (3,2));
uncaught exception Evaluation
  raised at: solution3.sml:85.27-85.41
- applyEye (VMat [[VRat (1,1)]]);
uncaught exception Evaluation
  raised at: solution3.sml:85.27-85.41
```

(b) Extend the substitution function `subst` to account for substituting into an `EEye` expression.

```
- subst (EEye (EIdent "x")) "x" (EVal (VRat (1,1)));
val it = EEye (EVal (VRat (1,1))) : expr
- subst (EEye (EAdd (EIdent "x", EIdent "x"))) "x" (EVal (VRat (2,1)));
val it = EEye (EAdd (EVal (VRat (2,1)),EVal (VRat (2,1)))) : expr
- subst (EEye (EAdd (EIdent "y", EIdent "x"))) "x" (EVal (VRat (2,1)));
val it = EEye (EAdd (EIdent "y",EVal (VRat (2,1)))) : expr
```

(c) Extend the evaluation function `eval` to account for evaluating an `EEye` expression. Note that the language is call-by-value.

```
- eval [] (EEye (EVal (VRat (2,1))));
val it = VMat [[VRat (1,1),VRat (0,1)],[VRat (0,1),VRat (1,1)]] : value
- eval [] (EEye (EVal (VRat (4,1))));
val it =
  VMat
    [[VRat (1,1),VRat (0,1),VRat (0,1),VRat (0,1)],
     [VRat (0,1),VRat (1,1),VRat (0,1),VRat (0,1)],
     [VRat (0,1),VRat (0,1),VRat (1,1),VRat (0,1)],
     [VRat (0,1),VRat (0,1),VRat (0,1),VRat (1,1)]] : value
- eval [] (EEye (EAdd (EVal (VRat (3,1)), EVal (VRat (1,1)))));
val it =
  VMat
    [[VRat (1,1),VRat (0,1),VRat (0,1),VRat (0,1)],
     [VRat (0,1),VRat (1,1),VRat (0,1),VRat (0,1)],
     [VRat (0,1),VRat (0,1),VRat (1,1),VRat (0,1)],
     [VRat (0,1),VRat (0,1),VRat (0,1),VRat (1,1)]] : value
- eval [] (ELet ("x", EVal (VRat (4,2)), EEye (EMul (EIdent "x", EIdent "x"))))
   ;
val it =
  VMat
    [[VRat (1,1),VRat (0,1),VRat (0,1),VRat (0,1)],
     [VRat (0,1),VRat (1,1),VRat (0,1),VRat (0,1)],
     [VRat (0,1),VRat (0,1),VRat (1,1),VRat (0,1)],
     [VRat (0,1),VRat (0,1),VRat (0,1),VRat (1,1)]] : value
-
```

(d) The `token` datatype has a constructor `T_EYE` to represent the token corresponding to the keyword `eye`. Extend the `produceSymbol` function to make the lexer produce the `T_EYE` token when encountering the symbol `eye`.

```
- lexString "eye";
val it = [T_EYE] : token list
```

```
- lexString "Popeye";
val it = [T_SYM "Popeye"] : token list
- lexString "eyeball";
val it = [T_SYM "eyeball"] : token list
- lexString "poke out an eye";
val it = [T_SYM "poke",T_SYM "out",T_SYM "an",T_EYE] : token list
- lexString "eye (3)";
val it = [T_EYE,T_LPAREN,T_INT 3,T_RPAREN] : token list
```

(e) Extend the parser to correctly parse `eye (`*expr*`)` in the surface syntax as internal representation expression `EEye (`*e*`)`, where *e* is the result of parsing *expr*.

The best way to do this is to extend the grammar of `factor`:

```
factor ::= T_INT
           T_TRUE
           T_FALSE
           T_EYE T_LPAREN expr T_RPAREN          (* new *)
           T_SYM T_LPAREN expr_list T_RPAREN
           T_SYM
           T_LPAREN expr TRPAREN
```

Thus, you should modify the `parse_factor` function to take that new case into account, and write a function `parse_factor_EYE` to deal with that new case. (This is in keeping with the naming conventions for the recursive-descent parser functions that I've used in this homework. You can of course name the function whatever you want...)

Note that you'll probably want to write a function `expect_EYE` to go with the `T_EYE` token.

```
- parse_factor (lexString "eye (1)");
val it = SOME (EEye (EVal (VRat (1,1))),[]) : (expr * token list) option
- parse_factor (lexString "eye (2)");
val it = SOME (EEye (EVal (VRat (2,1))),[]) : (expr * token list) option
- parse (lexString "eye (2)");
val it = EEye (EVal (VRat (2,1))) : expr
- parse (lexString "1 + eye (2)");
val it = EAdd (EVal (VRat (1,1)),EEye (EVal (VRat (2,1)))) : expr
- parse (lexString "eye (1+2) + eye (2)");
val it =
  EAdd
    (EEye (EAdd (EVal (VRat (1,1)),EVal (VRat (2,1)))),
     EEye (EVal (VRat (2,1)))) : expr
```

Once you have the above implemented, you should be able to use this new syntax in the shell:

```
- shell [];
Type . by itself to quit
pldi-hw3> eye (1)
Tokens = T_EYE T_LPAREN T_INT[1] T_RPAREN
Internal rep =  EEye (EVal (VRat (1,1)))
[1]
pldi-hw3> eye (2)
Tokens = T_EYE T_LPAREN T_INT[2] T_RPAREN
Internal rep =  EEye (EVal (VRat (2,1)))
[1 0 ; 0 1]
pldi-hw3> eye (3)
Tokens = T_EYE T_LPAREN T_INT[3] T_RPAREN
Internal rep =  EEye (EVal (VRat (3,1)))
[1 0 0 ; 0 1 0 ; 0 0 1]
pldi-hw3> eye (3) + eye (3)
Tokens = T_EYE T_LPAREN T_INT[3] T_RPAREN T_PLUS T_EYE T_LPAREN T_INT[3] T_RPAREN
Internal rep =  EAdd (EEye (EVal (VRat (3,1))),EEye (EVal (VRat (3,1))))
[2 0 0 ; 0 2 0 ; 0 0 2]
pldi-hw3> let x = eye (3) + eye (3) in x * x
Tokens = T_LET T_SYM[x] T_EQUAL T_EYE T_LPAREN T_INT[3] T_RPAREN T_PLUS T_EYE
   T_LPAREN T_INT[3] T_RPAREN T_SYM[in] T_SYM[x] T_TIMES T_SYM[x]
Internal rep =  ELet ("x",EAdd (EEye (EVal (VRat (3,1))),EEye (EVal (VRat (3,1)))
   ),EMul (EIdent "x",EIdent "x"))
[4 0 0 ; 0 4 0 ; 0 0 4]
pldi-hw3> .
val it = () : unit
```

Question 2 (Matrices).

Question 1 gives us some surface syntax for creating matrices, but it's restricted.

We're going to add surface syntax, inspired by MATLAB, for creating arbitrary matrices. The surface syntax expression

```
[1 2 3 4; 5 6 7 8; 9 10 11 12]
```

will create the 3×4 matrix $\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix}$. Similarly,

```
[1 2 3 4]
```

will create the 1×4 matrix $\begin{pmatrix} 1 & 2 & 3 & 4 \end{pmatrix}$, which might as well be considered a vector.

Of course, we should be able to use arbitrary expressions to represent entries in the matrix, so that

```
let x = 10 in [(1+2) 3; (x-5) x]
```

will create the 2×2 matrix $\begin{pmatrix} 3 & 3 \\ 5 & 10 \end{pmatrix}$

This surface syntax will parse down to the internal representation expression `EMatrix` described in the introduction. The above expression `let x = 10 in [(1+2) 3; (x-5) x]` should parse down to

```
ELet ("x",EVal (VRat (10,1)),
      EMatrix [[EAdd (EVal (VRat (1,1)),EVal (VRat (2,1))),
                EVal (VRat (3,1))],
               [ESub (EIdent "x",EVal (VRat (5,1))),
                EIdent "x"]])
```

Makes sense?

Everything is already implemented in the internal representation to evaluate `EMatrix`. The only thing you need to do is modify the lexer and the parser to provide the surface syntax above.

(a) The `token` datatype already contains token definitions `T_LBRACKET`, `T_RBRACKET`, and `T_SEMICOLON` for `[`, `]`, and `;`. Extend the lexer to produce these tokens.

Note: the regular expression for `T_LBRACKET` is `"\\["`, for `T_RBRACKET` is `"\\]"`, and for `T_SEMICOLON` is `";"`.

```
- lexString "[";
val it = [T_LBRACKET] : token list
- lexString "]";
val it = [T_RBRACKET] : token list
- lexString ";";
val it = [T_SEMICOLON] : token list
- lexString "[];";
val it = [T_LBRACKET,T_RBRACKET,T_SEMICOLON] : token list
- lexString "[1 2 3]";
val it = [T_LBRACKET,T_INT 1,T_INT 2,T_INT 3,T_RBRACKET] : token list
- lexString "[1 2 3; 4 5 6]";
val it =
  [T_LBRACKET,T_INT 1,T_INT 2,T_INT 3,T_SEMICOLON,T_INT 4,T_INT 5,T_INT 6,
   T_RBRACKET] : token list
```

(b) Here are the changes to the grammar that will add the required surface syntax. First, we add a new rule to the `factor` nonterminal rules:

```
factor ::= T_INT
           T_TRUE
           T_FALSE
           T_EYE T_LPAREN expr T_RPAREN          (* added in Q1 *)
           T_SYM T_LPAREN expr_list T_RPAREN
           T_SYM
           T_LPAREN expr TRPAREN
           T_LBRACKET expr_rows T_RBRACKET       (* new *)
```

Then we introduce two new nonterminals `expr_rows` and `expr_row` with rules as follows:

```
expr_rows ::= expr_row T_SEMICOLON expr_rows
              expr_row

 expr_row ::= expr expr_row
              expr
```

Your job is to extend the parser to deal with the above modifications to the grammar. That will require adding a function that you might want to call `parse_factor_MATRIX` to deal with the new rule added to `factor`, and new functions `parse_expr_rows` and `parse_expr_row` to parse the two new nonterminals, at least:

```
parse_expr_rows:  token list -> (expr list list * token list) option
parse_expr_row:  token list -> (expr list * token list) option
```

Note that you'll probably want to write functions `expect_LBRACKET`, `expect_RBRACKET`, and `expect_SEMICOLON` to go with the new tokens.

```
- parse_factor (lexString "[ 1 2 ]");
val it = SOME (EMatrix [[EVal (VRat (1,1)),EVal (VRat (2,1))]],[])
  : (expr * token list) option
- parse_factor (lexString "[ 1 2 ; 3 4]");
val it =
  SOME
    (EMatrix
       [[EVal (VRat (1,1)),EVal (VRat (2,1))],
        [EVal (VRat (3,1)),EVal (VRat (4,1))]],[])
  : (expr * token list) option
- parse_factor (lexString "[ 1 (2 + 3)]");
val it =
  SOME
    (EMatrix [[EVal (VRat (1,1)),EAdd (EVal (VRat (2,1)),EVal (VRat (3,1)))]],
     []) : (expr * token list) option
- parse_expr_rows (lexString "1 2 ; 3 4");
val it =
  SOME
    ([[EVal (VRat (1,1)),EVal (VRat (2,1))],
      [EVal (VRat (3,1)),EVal (VRat (4,1))]],[])
  : (expr list list * token list) option
- parse_expr_rows (lexString "1 2");
val it = SOME ([[EVal (VRat (1,1)),EVal (VRat (2,1))]],[])
  : (expr list list * token list) option
- parse_expr_row (lexString "1 2");
val it = SOME ([EVal (VRat (1,1)),EVal (VRat (2,1))],[])
  : (expr list * token list) option
- parse_expr_row (lexString "1");
val it = SOME ([EVal (VRat (1,1))],[]) : (expr list * token list) option
- parse (lexString "[1 2; 3 4]");
val it =
  EMatrix
    [[EVal (VRat (1,1)),EVal (VRat (2,1))],
     [EVal (VRat (3,1)),EVal (VRat (4,1))]] : expr
- parse (lexString "1 + [1 2; 3 4]");
val it =
  EAdd
    (EVal (VRat (1,1)),
     EMatrix
       [[EVal (VRat (1,1)),EVal (VRat (2,1))],
        [EVal (VRat (3,1)),EVal (VRat (4,1))]]) : expr
- parse (lexString "[66 99] * [1 2; 3 4]");
```

```
val it =
  EMul
    (EMatrix [[EVal (VRat (66,1)),EVal (VRat (99,1))]],
     EMatrix
       [[EVal (VRat (1,1)),EVal (VRat (2,1))],
        [EVal (VRat (3,1)),EVal (VRat (4,1))]]) : expr
```

Once you have the above implemented, you should be able to use this new syntax in the shell:

```
- shell [];
Type . by itself to quit
pldi-hw3> [ 1 2 ]
Tokens = T_LBRACKET T_INT[1] T_INT[2] T_RBRACKET
Internal rep =  EMatrix [[EVal (VRat (1,1)),EVal (VRat (2,1))]]
[1 2]
pldi-hw3> [ 1 2 ; 3 4]
Tokens = T_LBRACKET T_INT[1] T_INT[2] T_SEMICOLON T_INT[3] T_INT[4] T_RBRACKET
Internal rep =  EMatrix [[EVal (VRat (1,1)),EVal (VRat (2,1))],[EVal (VRat (3,1))
    ,EVal (VRat (4,1))]]
[1 2 ; 3 4]
pldi-hw3> [ 1 2 ; 3 4] + 100
Tokens = T_LBRACKET T_INT[1] T_INT[2] T_SEMICOLON T_INT[3] T_INT[4] T_RBRACKET
    T_PLUS T_INT[100]
Internal rep =  EAdd (EMatrix [[EVal (VRat (1,1)),EVal (VRat (2,1))],[EVal (VRat
    (3,1)),EVal (VRat (4,1))]],EVal (VRat (100,1)))
[101 102 ; 103 104]
pldi-hw3> [ 1 2 ; 3 4] * 100
Tokens = T_LBRACKET T_INT[1] T_INT[2] T_SEMICOLON T_INT[3] T_INT[4] T_RBRACKET
    T_TIMES T_INT[100]
Internal rep =  EMul (EMatrix [[EVal (VRat (1,1)),EVal (VRat (2,1))],[EVal (VRat
    (3,1)),EVal (VRat (4,1))]],EVal (VRat (100,1)))
[100 200 ; 300 400]
pldi-hw3> [ 1 2 ; 3 4] * [10 20; 30 40]
Tokens = T_LBRACKET T_INT[1] T_INT[2] T_SEMICOLON T_INT[3] T_INT[4] T_RBRACKET
    T_TIMES T_LBRACKET T_INT[10] T_INT[20] T_SEMICOLON T_INT[30] T_INT[40]
    T_RBRACKET
Internal rep =  EMul (EMatrix [[EVal (VRat (1,1)),EVal (VRat (2,1))],[EVal (VRat
    (3,1)),EVal (VRat (4,1))]],EMatrix [[EVal (VRat (10,1)),EVal (VRat (20,1))],[
    EVal (VRat (30,1)),EVal (VRat (40,1))]])
[70 100 ; 150 220]
pldi-hw3> let x = [1 2; 3 4] * [5 6; 7 8] in x + x
Tokens = T_LET T_SYM[x] T_EQUAL T_LBRACKET T_INT[1] T_INT[2] T_SEMICOLON T_INT[3]
     T_INT[4] T_RBRACKET T_TIMES T_LBRACKET T_INT[5] T_INT[6] T_SEMICOLON T_INT[7]
     T_INT[8] T_RBRACKET T_SYM[in] T_SYM[x] T_PLUS T_SYM[x]
```

```
Internal rep =  ELet ("x",EMul (EMatrix [[EVal (VRat (1,1)),EVal (VRat (2,1))],[
   EVal (VRat (3,1)),EVal (VRat (4,1))]],EMatrix [[EVal (VRat (5,1)),EVal (VRat
   (6,1))],[EVal (VRat (7,1)),EVal (VRat (8,1))]]),EAdd (EIdent "x",EIdent "x"))
[38 44 ; 86 100]
pldi-hw3> let x = 10 + 20 in [ (x) (x+1) (x+2) (x+3) ]
Tokens = T_LET T_SYM[x] T_EQUAL T_INT[10] T_PLUS T_INT[20] T_SYM[in] T_LBRACKET
   T_LPAREN T_SYM[x] T_RPAREN T_LPAREN T_SYM[x] T_PLUS T_INT[1] T_RPAREN T_LPAREN
    T_SYM[x] T_PLUS T_INT[2] T_RPAREN T_LPAREN T_SYM[x] T_PLUS T_INT[3] T_RPAREN
   T_RBRACKET
Internal rep =  ELet ("x",EAdd (EVal (VRat (10,1)),EVal (VRat (20,1))),EMatrix [[
   EIdent "x",EAdd (EIdent "x",EVal (VRat (1,1))),EAdd (EIdent "x",EVal (VRat
   (2,1))),EAdd (EIdent "x",EVal (VRat (3,1)))]])
[30 31 32 33]
pldi-hw3> .
val it = () : unit
```

Question 3 (Defining Functions).

The shell I gave you in the supplied code is useful and a good starting point, but it is severely limited in functionality. In particular, it does not allow you to add function definitions to the environment: the function environment has to be given as an argument to the shell, and it never changes.

If you think about a shell like the SML/NJ shell, or the Python shell, they all give you a way to define functions. In SML/NJ, you can write

```
- fun double x = 2 * x;
val double = fn : int -> int
- double 3;
val it = 6 : int
```

In Python:

```
>>> def double (x): return 2 * x
...
>>> double (3)
6
```

This is the sort of interaction I'd like you to implement:

```
- shell_wdef [];
Type . by itself to quit
pldi-hw3> def double (x) = x * 2
Tokens = T_DEF T_SYM[double] T_LPAREN T_SYM[x] T_RPAREN T_EQUAL T_SYM[x] ...
pldi-hw3> double (3)
Tokens = T_SYM[double] T_LPAREN T_INT[3] T_RPAREN
Internal rep =  ECall ("double", [EVal (VRat (3,1))])
6
pldi-hw3> double ([1 2 3; 4 5 6])
Tokens = T_SYM[double] T_LPAREN T_LBRACKET T_INT[1] T_INT[2] T_INT[3] ...
Internal rep =  ECall ("double", [EMatrix [[EVal (VRat (1,1)), ...
[2 4 6 ; 8 10 12]
pldi-hw3> def exp (a,n) = if n = 0 then 1 else a * exp (a,n-1)
Tokens = T_DEF T_SYM[exp] T_LPAREN T_SYM[a] T_COMMA T_SYM[n] T_RPAREN ...
pldi-hw3> exp (2,5)
Tokens = T_SYM[exp] T_LPAREN T_INT[2] T_COMMA T_INT[5] T_RPAREN
Internal rep =  ECall ("exp", [EVal (VRat (2,1)),EVal (VRat (5,1))])
32
pldi-hw3> exp (2,10)
```

```
Tokens = T_SYM[exp] T_LPAREN T_INT[2] T_COMMA T_INT[10] T_RPAREN
Internal rep =  ECall ("exp", [EVal (VRat (2,1)),EVal (VRat (10,1))])
1024
pldi-hw3> exp ([2 0; 0 2],10)
Tokens = T_SYM[exp] T_LPAREN T_LBRACKET T_INT[2] T_INT[0] T_SEMICOLON ...
Internal rep =  ECall ("exp", [EMatrix [[EVal (VRat (2,1)), ...
[1024 0 ; 0 1024]
pldi-hw3> .
val it = () : unit
```

So how do we go about it?

First off: take a minute and think about how you might approach the problem, given what we already know. In the shell, after reading a line of input, we'll want to (1) notice we have a function definition instead of an expression to evaluate, (2) convert the function definition into something suitable to add to the function environment, and (3) actually add the function definition to the function environment.

The easiest solution is to simply extend our grammar.

Right now, our grammar parses expressions. We are going to extended it so that it parses *shell declarations*. A shell declaration is either a function definition, or an expression to evaluate. Here is the extension to our existing grammar:

```
         decl ::= T_DEF T_SYM T_LPAREN symbol_list T_RPAREN T_EQUAL expr
                  expr

  symbol_list ::= T_SYM T_COMMA symbol_list
                  T_SYM
```

This requires adding a new token `T_DEF` to the lexer, which is produced by the symbol `def`.

We can of course extend our existing recursive-descent parser to parse the above nonterminals, via two new parsing functions `parse_decl` and `parse_symbol_list` (among others).

What results should those parsing function return? For `parse_symbol_list`, it makes sense for it to return a list of strings. For `parse_decl`, though, it's a bit more interesting. Parsing a `decl` gives you back either a function definition, or an expression. The best way to deal with that is to define a new datatype to represent the result. I've done that for you:

```
datatype decl = DeclDefinition of string * (string list) * expr
              | DeclExpression of expr
```

Thus, the `parse_decl` function should have type

```
token list -> (decl * token list) option
```

where the `decl` returned is either a function definition `DeclDefinition (name,params,body)` or an expression `DeclExpression e`. Note that the function definition `DeclDefinition (name,params,body)` contains all the information needed to construct the value that you need to put in the function environment. (Recall that the function environment is a list of pairs, each pair a name and a value of type `function`. Confused yet? Take a break, and make sure you understand function environments.)

Now that we have this shiny new parsing function `parse_decl`, what do we do with it? In the shell, when we parse the input string, we can now parse it as a shell declaration: if we get back a `DeclDefinition`, we add that function to the function environment, and if it's a `DeclExpression`, we evaluate it and print the result.

Onward!

(a) The `token` datatype already contains token definition `T_DEF` for symbol `def`. Extend the lexer to produce that token.

```
- lexString "def";
val it = [T_DEF] : token list
- lexString "define";
val it = [T_SYM "define"] : token list
- lexString "fundef";
val it = [T_SYM "fundef"] : token list
- lexString "def double (x) = 2 * x";
val it =
  [T_DEF,T_SYM "double",T_LPAREN,T_SYM "x",T_RPAREN,T_EQUAL,T_INT 2,T_TIMES,
   T_SYM "x"] : token list
```

(b) Code functions `parse_decl` and `parse_symbol_list` with types

```
parse_decl:  token list -> (decl * token list) option
parse_symbol_list:  token list -> (string list * token list) option
```

that extend the recursive-descent parser with ways to parse the `decl` and `symbol_list` nonterminals in the grammar extension describe above.

Note that you'll probably want to write a function `expect_DEF` to go with the new `T_DEF` token.

```
- parse_decl (lexString "def double (x) = 2 * x");
val it =
  SOME
    (DeclDefinition ("double",["x"],EMul (EVal (VRat (2,1)),EIdent "x")),[])
  : (decl * token list) option
- parse_decl (lexString "double (10)");
val it = SOME (DeclExpression (ECall ("double",[EVal (VRat (10,1))])),[])
```

```
  : (decl * token list) option
- parse_decl (lexString "def sum (a,b,c) = a + b + c");
val it =
  SOME
    (DeclDefinition
       ("sum",["a","b","c"],EAdd (EIdent "a",EAdd (EIdent "b",EIdent "c"))),
     []) : (decl * token list) option
- parse_decl (lexString "a + b + c");
val it =
  SOME (DeclExpression (EAdd (EIdent "a",EAdd (EIdent "b",EIdent "c"))),[])
  : (decl * token list) option
- parse_decl (lexString "1 + sum (10,20,30)");
val it =
  SOME
    (DeclExpression
       (EAdd
          (EVal (VRat (1,1)),
           ECall
             ("sum",
              [EVal (VRat (10,1)),EVal (VRat (20,1)),EVal (VRat (30,1))]))),
     []) : (decl * token list) option
```

(c) Code function `parse_wdef` (for *parse with definitions*) with type

```
token list -> decl
```

where `parse_wdef ts` tries to parses token list `ts` as a shell declaration and returns the result. It calls `parseError` to report an error if the parsing fails. (This is just a variation of `parse` that returns shell declarations instead of just expressions; you can inspire yourself from `parse`, obviously.)

```
- parse_wdef (lexString "def double (x) = 2 * x");
val it = DeclDefinition ("double",["x"],EMul (EVal (VRat (2,1)),EIdent "x"))
  : decl
- parse_wdef (lexString "double (10)");
val it = DeclExpression (ECall ("double",[EVal (VRat (10,1))])) : decl
- parse_wdef (lexString "def sum (a,b,c) = a + b + c");
val it =
  DeclDefinition
    ("sum",["a","b","c"],EAdd (EIdent "a",EAdd (EIdent "b",EIdent "c")))
  : decl
- parse_wdef (lexString "def broken x = x");
uncaught exception Parsing
  raised at: solution3.sml:86.28-86.39
```

(d) Code function `shell_wdef` (for *shell with definitions*) with type

```
(string * function) list -> unit
```

where `shell_wdef fenv` creates a shell that can not only evaluate expressions typed in, but also add functions to the function environment that can be used when evaluating expressions. Note that the function environment is always passed around explicitly as an argument to functions that need it.

You should probably copy/paste the definition of `shell` for the new `shell_wdef` function and then modify that, rather than writing it from scratch. (Although you are welcome to write it from scratch as well, of course.)

Sample outputs should be as above in the motivational section.

Question 4 (Roll Your Own).

This is an open-ended question. I want you to add something interesting to the interpreter we have. To get full points, your addition must require a change to the parser and to the internal representation and evaluation. Try to keep with the spirit of what we have done, but that's not strictly required.

Here are some suggestions:

- Add some syntax and internal representation support for extracting the entry of a matrix. You should be able to refer to the entry position using arbitrary expressions, including identifiers.
- Add some syntax and internal representation support for extracting the row of a matrix, or perhaps a submatrix of the matrix. (This is notation `A(1:2,2:4)` in MATLAB—you do not have to mimic that notation though.)
- Add one or more interesting operations to the language, such matrix inversion. Once you have matrix inversion, you can implement division for matrices.
- Add a new data type and corresponding operations to the language. Implement interactions between the corresponding new values and existing values.
- Implement your favorite MATLAB operations, as long as they're compatible with our language. (I.e., no variable update—we'll deal with those later.)[1]
- One possibility is to add enough infrastructure to do your favorite MATLAB thing in our little toy language. For instance, Wikipedia has the following MATLAB code for computing magic squares for odd values of n:

  ```
  [J,I] = meshgrid(1:n);
  A = mod(I + J - (n + 3) / 2, n);
  B = mod(I + 2 * J - 2, n);
  M = n * A + B + 1;
  ```

  We don't have a `mod` function, but that's easy to add. We don't have a `meshgrid` function, and I'm not sure what it does, but I'm certain it's possible to write a specialized operation to do the `[J,I] = meshgrid(1:n)` bit, and then the rest could be written

  ```
  let A = mod (I+J-(n+3)/2, n)  in
  let B = mod (I+2*J-2, n)
   in  n * A + B + 1
  ```

[1] You can find some inspiration here at `http://en.wikibooks.org/wiki/MATLAB_Programming/Introduction_to_array_operations` or `http://en.wikibooks.org/wiki/MATLAB_Programming/Basic_vector_operations`. Language `APL` also has some interesting vector/matrix operations, like *reduce*.