Prog. Lang. Design and Implementation	Homework 4
ENGR 3599 Spring 2014 (Pucella)	Due Date: Friday, Mar 7

This homework is meant to be done in teams. You may discuss problems with fellow students, but all work must be entirely your own team's, 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, all the source code necessary to compile the homework (by running `CM.make "homework4.cm"`) should be zipped together in a file `homework4-sols.zip`.
- Your file `homework4-sols.zip` 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 `homework4-sols.zip` should contain a text file `readme.txt` listing the names of the people in your team, your email addresses, 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 `homework4.zip` available from the web site containing code you should 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.

A Functional Interpreter

This homework uses an interpreter similar to the one presented in the Functional Programming lectures. Everything is standard: the internal representation and its evaluation function, the lexer, and the parser.

The Internal Representation

The internal representation in module `InternalRepresentation` extends the internal representation for the simple functional language we saw in class with lists and records, which you will develop in this homework.

```
datatype value = VInt of int
               | VBool of bool
               | VClosure of string * expr * (string * value) list
               | VRecClosure of string * string * expr * (string * value) list
               | VList of value list
               | VRecord of (string * value) list

and expr = EVal of value
         | EFun of string * expr
         | EIf of expr * expr * expr
         | ELet of string * expr * expr
         | ELetFun of string * string * expr * expr
         | EIdent of string
         | EApp of expr * expr
         | EPrimCall1 of (value -> value) * expr
         | EPrimCall2 of (value -> value -> value) * expr * expr
         | ERecord of (string * expr) list
         | EField of expr * string
```

We could of course use rational numbers instead of integers, like we did in Homework 3.

Everything is as described in class, except for `VList` (Question 2), `VRecord` (Question 3), and `ERecord` and `EField` (Question 3).

The main functions in this module are

```
stringOfValue : value -> string
printValue : value -> unit
```

where `stringOfValue` converts a value into a nice human readable representation for printing purposes, and function `printValue` prints a value via `stringOfValue`.

The Lexer and Parser

Module `Parser` contains a straightforward lexer with the following tokens:

```
datatype token = T_LET                  (*  let  *)
               | T_IN                   (*  in   *)
               | T_SYM of string        (*  symbol, e.g. hello *)
               | T_INT of int           (*  integer, e.g. 10, ~3 *)
               | T_TRUE                 (*  true  *)
               | T_FALSE                (*  false  *)
               | T_EQUAL                (*  =  *)
               | T_LESS                 (*  <  *)
               | T_IF                   (*  if  *)
               | T_THEN                 (*  then  *)
               | T_ELSE                 (*  else  *)
               | T_LPAREN               (*  (   *)
               | T_RPAREN               (*  )   *)
               | T_PLUS                 (*  +   *)
               | T_MINUS                (*  -   *)
               | T_TIMES                (*  *   *)
               | T_BACKSLASH            (*  \   *)
               | T_RARROW               (*  ->  *)
               | T_LARROW               (*  <-  *)
               | T_DCOLON               (*  ::  *)
               | T_COMMA                (*  ,   *)
               | T_LBRACKET             (*  [   *)
               | T_RBRACKET             (*  ]   *)
               | T_LBRACE               (*  {   *)
               | T_RBRACE               (*  }   *)
               | T_DOT                  (*  .   *)
               | T_HASH                 (*  #   *)
               | T_DDOTS                (*  ..  *)
               | T_BAR                  (*  |   *)
               | T_MATCH                (*  match  *)
               | T_WITH                 (*  with   *)
```

Not all of those tokens are used or will be used. Call me conservative. As usual, 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.

There is also a recursive-descent parser (with backtracking) for the following grammar:

```
expr ::= eterm T_EQUAL eterm
         eterm T_LESS eterm
         eterm

eterm ::= cterm T_DCOLON cterm
          cterm

cterm ::= term T_PLUS term
          term T_MINUS term
          term

term :: = aterm aterm_list

aterm_list ::= aterm aterm_list
               <empty>

aterm ::= T_INT
          T_TRUE
          T_FALSE
          T_SYM
          T_BACKSLASH T_SYM T_RARROW expr
          T_LPAREN expr T_RPAREN
          T_IF expr T_THEN expr T_ELSE expr
          T_LET T_SYM T_EQUAL expr T_IN expr
          T_LET T_SYM sym_list T_EQUAL expr T_IN expr
```

The grammar is pretty much as you would expect, aside from the fact that it has those `eterm`, `cterm`, `term`, and `aterm` to manage the priority of operations: roughly, `=` and `<` bind more tightly than `::` which binds more tightly than `+` and `-` which bind more tightly than function application `f x y`. The atomic terms `aterm`s are where the action is: they include the integer and Boolean literals, and the rest of the expressions including anonymous functions and conditionals and lets.

As usual, the main functions of the recursive-descent parser correspond to the nonterminals:

```
parse_expr :         token list -> (expr * token list) option
parse_eterm:         token list -> (expr * token list) option
parse_cterm :        token list -> (expr * token list) option
parse_term :         token list -> (expr * token list) option
parse_aterm_list :   token list -> (expr list * token list) option
parse_aterm :        token list -> (expr * token list) option
parse :              token list -> expr
```

Note that while the parsing functions generally follow the structure I gave in class and in the last homework, I've optimized the parsing functions for `parse_expr`, `parse_eterm`, and `parse_cterm`, otherwise the backtracking was way too inefficient. I'll leave it up to you to figure out why, and how I fixed things up.

I've introduced the function `choice` I presented in class in Lecture 8 to simplify some of the code for the parser. Check it out.

If everything goes according to plan, the only parsing function you will have to modify is `parse_aterm`, since the additions to the parser you will make will all be to handle new rules for the `aterm` nonterminal.

The main parsing function `parse` function 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.

The Evaluator

Module `Evaluator` includes an environment-based evaluator for our functional language. It is call-by-value, as we saw in class, and there is nothing special going on there.

The main evaluation function is

```
eval :   (string * value) list -> expr -> value
```

where `eval env e` evaluates expression `e` in environment `env` and returns the resulting value.

Note that there is a definition for `initialEnv` towards the end of the module that defines the initial environment, including the primitive operations.

The Shell

The shell in module `Shell` is pretty much as described in the last homework. I did not implement top-level definitions, though they're easy to add; that's what you did in Question 3 of Homework 3, and the same approach would work here as well.

The main shell function is

```
run :    (string * value) list -> unit
```

where `run env` runs the shell in the initial environment *augmented* with `env`. So no matter what, you get the initial environment.

The one difference from the previous shell is that this one only evaluates. If you want to see the result of the parsing to debug your parser, then you can use the `:parse` shell instruction:

```
- Shell.run [];
Type . by itself to quit
Type :parse <expr> to see the parse of expression <expr>
Initial environment: add sub equal less cons nil hd tl map filter interval
homework4> :parse 10 + 20
Tokens = T_INT[10] T_PLUS T_INT[20]
EApp (EApp (EIdent "add",EVal (VInt 10)),EVal (VInt 20))
```

Question 1 (Warm Up: Curried Functions).

As we saw in class, functions all take a single argument. To deal with multiargument functions, we use currying. The way function application is parsed deals with curried functions correctly. If `max` is a curried function of two arguments, then

```
max 10 20
```

is interpreted as

```
(max 10) 20
```

and this works because `max` being curried means that it expects an argument and returns a new function that expects the second argument before computing the maximum of the two arguments.

Right now, currying has to be done by hand. For instance, function `max` above can be written as:

```
let max a = (\b -> if (a < b) then b else a)
  in max 10 20
```

(I've split the code over two lines, but of course our shell doesn't allow it.) Indeed:

```
- Shell.run [];
Type . by itself to quit
Type :parse <expr> to see the parse of expression <expr>
Initial environment: add sub equal less cons nil hd tl map filter interval
homework4> let max a = (\b -> if (a < b) then b else a) in max 10 20
20
```

It would be nicer if we could simply write:

```
let max a b = if (a < b) then b else a
  in max 10 20
```

like we do in Standard ML.

Guess what?

(a) Modify the parser to accept more than a single parameter to functions defined in a `let`, so that it produces a suitable internal representation for the resulting curried functions via a transformation.

Hint: I expect you will modify the parser so that it implements the following *modified* rule of the grammar for the `aterm` nonterminal that deals with function definitions:

```
aterm ::= ...
          T_LET T_SYM sym_list T_EQUAL expr T_IN expr
```

along with the new nonterminal `sym_list`:

```
sym_list ::= T_SYM sym_list
             T_SYM
```

For example:

```
- Shell.run [];
Type . by itself to quit
Type :parse <expr> to see the parse of expression <expr>
Initial environment: add sub equal less cons nil hd tl map filter interval
homework4> let double x = x + x in double 10
20
homework4> :parse let double x = x + x in double 10
Tokens = T_LET T_SYM[double] T_SYM[x] T_EQUAL T_SYM[x] T_PLUS T_SYM[x] T_IN
    T_SYM[double] T_INT[10]
ELetFun ("double","x",EApp (EApp (EIdent "add",EIdent "x"),EIdent "x"),EApp (
    EIdent "double",EVal (VInt 10)))
homework4> let max a b = if (a < b) then b else a in max 10 20
20
homework4> :parse let max a b = if (a < b) then b else a in max 10 20
Tokens = T_LET T_SYM[max] T_SYM[a] T_SYM[b] T_EQUAL T_IF T_LPAREN T_SYM[a]
    T_LESS T_SYM[b] T_RPAREN T_THEN T_SYM[b] T_ELSE T_SYM[a] T_IN T_SYM[max]
    T_INT[10] T_INT[20]
ELetFun ("max","a",EFun (b,EIf (EApp (EApp (EIdent "less",EIdent "a"),EIdent "b
    "),EIdent "b",EIdent "a")),EApp (EApp (EIdent "max",EVal (VInt 10)),EVal (
    VInt 20)))
homework4> let sum3 a b c = a + (b + c) in sum3 10 20 30
60
homework4> :parse let sum3 a b c = a + (b + c) in sum3 10 20 30
Tokens = T_LET T_SYM[sum3] T_SYM[a] T_SYM[b] T_SYM[c] T_EQUAL T_SYM[a] T_PLUS
    T_LPAREN T_SYM[b] T_PLUS T_SYM[c] T_RPAREN T_IN T_SYM[sum3] T_INT[10] T_INT
    [20] T_INT[30]
ELetFun ("sum3","a",EFun (b,EFun (c,EApp (EApp (EIdent "add",EIdent "a"),EApp (
    EApp (EIdent "add",EIdent "b"),EIdent "c")))),EApp (EApp (EApp (EIdent "
    sum3",EVal (VInt 10)),EVal (VInt 20)),EVal (VInt 30)))
```

Question 2 (Lists).

In Homework 2, you implemented lists for the simple expression language we used then.

We'll do the same here, but push a bit further.

A list in the internal representation is implemented by the value constructor `VList vs` where `vs` is the list of values that makes up the list. Thus, `VList []` represents the empty list, `VList [VInt 1, VInt 2]` represent the list of integers 1 and 2, and `VList [VBool true, VInt 1]` represents the list of Boolean value `true` and integer 1.

Right now, there is no way to create such values from within the object language.

Let's fix that.

(a) Add the following primitive operations to the initial environment: `nil`, `cons`, `hd`, `tl`.

- `nil` is a value representing the empty list
- `cons` is a function taking two arguments `x` and `xs` where `xs` is a list, and returns a new list `ys` where the first element of `ys` is `x` and the rest of `ys` is `xs`.
- `hd` is a function taking a list `xs` and returning the first element of `xs`
- `tl` is a function taking a list `xs` and returning the list made up of all but the first element of `xs`.

```
- Shell.run [];
Type . by itself to quit
Type :parse <expr> to see the parse of expression <expr>
Initial environment: add sub equal less cons nil hd tl map filter interval
homework4> nil
[]
homework4> cons 1 nil
[1]
homework4> cons 1 (cons 2 nil)
[1,2]
homework4> 1::(2::nil)
[1,2]
homework4> hd (1::(2::nil))
1
homework4> tl (1::nil)
[]
homework4> tl (1::(2::nil))
[2]
homework4> :parse cons 1 nil
Tokens = T_SYM[cons] T_INT[1] T_SYM[nil]
EApp (EApp (EIdent "cons",EVal (VInt 1)),EIdent "nil")
```

```
homework4> :parse hd (1::nil)
Tokens = T_SYM[hd] T_LPAREN T_INT[1] T_DCOLON T_SYM[nil] T_RPAREN
EApp (EIdent "hd",EApp (EApp (EIdent "cons",EVal (VInt 1)),EIdent "nil"))
```

Note that you should not have to modify the parser to do this. You only need to add the suitable primitive operations to the initial environment. (You will of course need to provide implementations of those primitive operations within the evaluator–see the primitive operations `equal` and `less` for reference.)

Note also that the syntax `e1::e2` in the object language will automatically transform into a call to `cons`.

(b) Modify function `primEq` so that it accurately checks whether two lists are equal.

```
- Shell.run [];
Type . by itself to quit
Type :parse <expr> to see the parse of expression <expr>
Initial environment: add sub equal less cons nil hd tl map filter interval
homework4> nil = nil
true
homework4> nil = 1
false
homework4> nil = 1::nil
false
homework4> 1::nil = 1::nil
true
homework4> 1::nil = 2::nil
false
homework4> 1::nil = 1::(2::nil)
false
```

(c) Modify the parser so that it recognizes an expression of the form

$$[e_1, e_2, e_3, \ldots, e_k]$$

and implement a transformation to produce the following internal representation:

$$\texttt{cons}\ e_1\ \texttt{(cons}\ e_2\ \texttt{(cons}\ e_3\ \texttt{(cons ...\ (cons}\ e_4\ \texttt{nil))))}$$

```
- Shell.run [];
Type . by itself to quit
Type :parse <expr> to see the parse of expression <expr>
Initial environment: add sub equal less cons nil hd tl map filter interval
homework4> []
[]
```

```
homework4> [1]
[1]
homework4> [1,2]
[1,2]
homework4> [1,2,3]
[1,2,3]
homework4> [10+20, 30+40, 40+50]
[30,70,90]
homework4> :parse []
Tokens = T_LBRACKET T_RBRACKET
EVal (VList [])
homework4> :parse [1]
Tokens = T_LBRACKET T_INT[1] T_RBRACKET
EApp (EApp (EIdent "cons",EVal (VInt 1)),EVal (VList []))
homework4> :parse [1,2]
Tokens = T_LBRACKET T_INT[1] T_COMMA T_INT[2] T_RBRACKET
EApp (EApp (EIdent "cons",EVal (VInt 1)),EApp (EApp (EIdent "cons",EVal (VInt
   2)),EVal (VList [])))
homework4> :parse [1+1,3]
Tokens = T_LBRACKET T_INT[1] T_PLUS T_INT[1] T_COMMA T_INT[3] T_RBRACKET
EApp (EApp (EIdent "cons",EApp (EApp (EIdent "add",EVal (VInt 1)),EVal (VInt 1)
   )),EApp (EApp (EIdent "cons",EVal (VInt 3)),EVal (VList [])))
```

Hint: The easiest way to do this is to extend the grammar with a new rule for the `aterm` nonterminal:

```
aterm ::= ...
          T_LBRACKET expr_list T_RBRACKET
```

and a new nonterminal:

```
expr_list ::= expr T_COMMA expr_list
              expr
              <empty>
```

Parsing the above rule for `aterm` should perform the transformation and produce the suitable internal representation expression.

(d) Modify the parser so that it recognizes an expression of the form

$$\texttt{match } e_1 \texttt{ with [] -> } e_2 \texttt{ | } s_1\texttt{::}s_2 \texttt{ -> } e_3$$

and implement a transformation to produce the following internal representation:

```
if e1 = nil then
   e2
else
  let s1 = hd e1 in
    let s2 = tl e1 in
       e3
```

```
- Shell.run [];
Type . by itself to quit
Type :parse <expr> to see the parse of expression <expr>
Initial environment: add sub equal less cons nil hd tl map filter interval
homework4> match nil with [] -> 1 | x::xs -> 2
1
homework4> match (1::nil) with [] -> 1 | x::xs -> 2
2
homework4> match nil with [] -> 0 | x::xs -> x + (hd xs)
0
homework4> match (1::(2::nil)) with [] -> 0 | x::xs -> x + (hd xs)
3
homework4> let sum xs = match xs with [] -> 0 | x::xs -> x + sum xs in sum
    [1,2,3]
6
homework4> :parse match (1::(2::nil)) with [] -> 0 | x::xs -> x + (hd xs)
Tokens = T_MATCH T_LPAREN T_INT[1] T_DCOLON T_LPAREN T_INT[2] T_DCOLON T_SYM[
    nil] T_RPAREN T_RPAREN T_WITH T_LBRACKET T_RBRACKET T_RARROW T_INT[0] T_BAR
     T_SYM[x] T_DCOLON T_SYM[xs] T_RARROW T_SYM[x] T_PLUS T_LPAREN T_SYM[hd]
    T_SYM[xs] T_RPAREN
EIf (EApp (EApp (EIdent "equal",EApp (EApp (EIdent "cons",EVal (VInt 1)),EApp (
    EApp (EIdent "cons",EVal (VInt 2)),EIdent "nil"))),EVal (VList [])),EVal (
    VInt 0),ELet ("x",EApp (EIdent "hd",EApp (EApp (EIdent "cons",EVal (VInt 1)
    ),EApp (EApp (EIdent "cons",EVal (VInt 2)),EIdent "nil"))),ELet ("xs",EApp
    (EIdent "tl",EApp (EApp (EIdent "cons",EVal (VInt 1)),EApp (EApp (EIdent "
    cons",EVal (VInt 2)),EIdent "nil"))),EApp (EApp (EIdent "add",EIdent "x"),
    EApp (EIdent "hd",EIdent "xs")))))
```

Hint: The easiest way to do this is to extend the grammar with a new rule for the `aterm` terminal:

```
aterm ::= ...
          T_MATCH expr T_WITH T_LBRACKET T_RBRACKET T_RARROW
               expr T_BAR T_SYM T_DCOLON T_SYM T_RARROW expr
```

Parsing the above rule for `aterm` should perform the transformation and produce the suitable internal representation expression.

(e) Add a function `interval` to the initial environment, where `interval i j` produces the list `[i, i+1, i+2, ..., j]`. (The list produced should be empty if `j` < `i`.)

```
- Shell.run [];
Type . by itself to quit
Type :parse <expr> to see the parse of expression <expr>
Initial environment: add sub equal less cons nil hd tl map filter interval
homework4> interval 1 1
[1]
homework4> interval 1 10
[1,2,3,4,5,6,7,8,9,10]
homework4> interval 3 0
[]
homework4> interval (1+2) (4+5)
[3,4,5,6,7,8,9]
homework4> :parse interval 1 10
Tokens = T_SYM[interval] T_INT[1] T_INT[10]
EApp (EApp (EIdent "interval",EVal (VInt 1)),EVal (VInt 10))
```

You should not have to modify the parser to answer this question.

(f) Modify the parser so that it recognizes an expression of the form

$$\texttt{[}\ e_1\ \texttt{..}\ \ e_2\ \texttt{]}$$

and implement a transformation to produce the following internal representation:

$$\texttt{interval}\ e_1\ e_2$$

```
- Shell.run [];
Type . by itself to quit
Type :parse <expr> to see the parse of expression <expr>
Initial environment: add sub equal less cons nil hd tl map filter interval
homework4> interval
<function interval (i,EFun (j,...))>
homework4> [0..0]
[0]
homework4> [1..2]
[1,2]
homework4> [1..10]
[1,2,3,4,5,6,7,8,9,10]
homework4> [10..15]
[10,11,12,13,14,15]
```

```
homework4> [10..0]
[]
homework4> [(1+2)..(4+5)]
[3,4,5,6,7,8,9]
homework4> :parse [1..10]
Tokens = T_LBRACKET T_INT[1] T_DDOTS T_INT[10] T_RBRACKET
EApp (EApp (EIdent "interval",EVal (VInt 1)),EVal (VInt 10))
```

Hint: The easiest way to do this is to extend the grammar with a new rule for the `aterm` terminal:

```
aterm ::= ...
          T_LBRACKET expr T_DDOTS expr T_RBRACKET
```

Parsing the above rule for `aterm` should perform the transformation and produce the suitable internal representation expression.

(g) Add a function `map` to the initial environment, where `map f xs` produces the list obtained by applying function `f` to every element in `xs`.

```
- Shell.run [];
Type . by itself to quit
Type :parse <expr> to see the parse of expression <expr>
Initial environment: add sub equal less cons nil hd tl map filter interval
homework4> map
<function map (f,EFun (xs,...))>
homework4> map (\x -> x) [1,2,3,4]
[1,2,3,4]
homework4> map (\x -> x + 1) [1,2,3,4]
[2,3,4,5]
homework4> map (\x -> x + x) [1,2,3,4]
[2,4,6,8]
homework4> map (add 10) [1,2,3,4,5]
[11,12,13,14,15]
```

You should not have to modify the parser to answer this question.

(h) Modify the parser so that it recognizes an expression of the form

$$\texttt{[} e_1 \texttt{ | } s \texttt{ <- } e_2 \texttt{]}$$

and implement a transformation to produce the following internal representation:

$$\texttt{map (\}s \texttt{ -> } e_1\texttt{) } e_2$$

```
- Shell.run [];
Type . by itself to quit
Type :parse <expr> to see the parse of expression <expr>
Initial environment: add sub equal less cons nil hd tl map filter interval
homework4> [ x | x <- [1,2,3,4]]
[1,2,3,4]
homework4> [ x + 1 | x <- [1,2,3,4]]
[2,3,4,5]
homework4> [ x + x | x <- [1,2,3,4]]
[2,4,6,8]
homework4> [ add 10 x | x <- [1,2,3,4]]
[11,12,13,14]
```

Hint: The easiest way to do this is to extend the grammar with a new rule for the `aterm` terminal:

```
aterm ::= ...
          T_LBRACKET expr T_BAR T_SYM T_LARROW expr T_RBRACKET
```

Parsing the above rule for `aterm` should perform the transformation and produce the suitable internal representation expression.

(i) Add a function `filter` to the initial environment, where `filter p xs` produces the list obtained by keeping only the elements of `xs` for which predicate `p` returns `true`.

```
- Shell.run [];
Type . by itself to quit
Type :parse <expr> to see the parse of expression <expr>
Initial environment: add sub equal less cons nil hd tl map filter interval
homework4> filter
<function filter (p,EFun (xs,...))>
homework4> filter (\x -> true) [1,2,3,4]
[1,2,3,4]
homework4> filter (\x -> false) [1,2,3,4]
[]
homework4> filter (\x -> x < 3) [1,2,3,4]
[1,2]
homework4> filter (\x -> 3 < x) [1,2,3,4]
[4]
homework4> filter (\x -> x = 0) [1,2,0,3,4,0,5,6,0]
[0,0,0]
```

You should not have to modify the parser to answer this question.

(j) Modify the parser so that it recognizes an expression of the form

$$\texttt{[} e_1 \texttt{ | } s \texttt{ <- } e_2\texttt{, } e_3 \texttt{]}$$

and implement a transformation to produce the following internal representation:

$$\texttt{map (\}s \texttt{ -> } e_1\texttt{) (filter (\}s \texttt{ -> } e_3\texttt{) } e_2\texttt{)}$$

```
- Shell.run [];
Type . by itself to quit
Type :parse <expr> to see the parse of expression <expr>
Initial environment: add sub equal less cons nil hd tl map filter interval
homework4> [ x | x <- [1,2,3,4], true]
[1,2,3,4]
homework4> [ x | x <- [1,2,3,4], false]
[]
homework4> [ x | x <- [1,2,3,4], x < 3]
[1,2]
homework4> [ x + x | x <- [1,2,3,4], x < 3]
[2,4]
```

Hint: The easiest way to do this is to extend the grammar with a new rule for the `aterm` terminal:

```
aterm ::= ...
          T_LBRACKET expr T_BAR T_SYM T_LARROW expr T_COMMA expr T_RBRACKET
```

Parsing the above rule for `aterm` should perform the transformation and produce the suitable internal representation expression.

To think about: Parts (h) and (j) above are examples of a more general form of syntax called *list comprehension*. If you've used Python before, chances are you're familiar with list comprehension. Can you generalize the above translations so that you can transform an arbitrary list comprehension? A list comprehension is an expression of the form

$$\texttt{[} e \texttt{ | } q_1\texttt{, } q_2\texttt{, } \ldots\texttt{, } q_k\texttt{]}$$

where e is an expression and each q_i is a qualifier, either

- a generator s `<-` e, or
- a filter e.

Each generator samples an element from the corresponding list e and binds it to s, while each filter filters out elements for which expression e is false.

Thus, `[ x + y | x <- xs, y <- ys, x + y < 10]` returns the list of all sums of elements from `xs` and `ys` that add up to less than 10. A qualifier can use identifiers bound in generators earlier in the qualifier list.

Can you come up with a generalization of the translation above that can deal with arbitrary list comprehensions?

```
- Shell.run [];
Type . by itself to quit
Type :parse <expr> to see the parse of expression <expr>
Initial environment: add sub equal less cons nil hd tl map filter interval
   flatMap
homework4> [ x + y | x <- [1,2,3], y <- [10,20,30]]
[11,21,31,12,22,32,13,23,33]
homework4> [ x + y | x <- [1,2,3], y <- [10,20,30], x + y < 20]
[11,12,13]
homework4> [ x + y | x <- [1,2,3], y <- [10,20,30], x + y < y + y]
[11,21,31,12,22,32,13,23,33]
homework4> [ [x,y] | x <- [[1,2],[3,4],[5,6]], y <- x]
[[[1,2],1],[[1,2],2],[[3,4],3],[[3,4],4],[[5,6],5],[[5,6],6]]
homework4> [ y | x <- [[1,2],[3,4],[5,6]], y <- x]
[1,2,3,4,5,6]
homework4> let flatten xs = [ y | x <- xs , y <- x] in flatten
   [[1,2,3,4],[],[5,6,7]]
[1,2,3,4,5,6,7]
homework4> [ [x,y] | x <- [1,2,3], y <- [1,2,3]]
[[1,1],[1,2],[1,3],[2,1],[2,2],[2,3],[3,1],[3,2],[3,3]]
homework4> let cross xs = [ [x,y] | x <- xs, y <- xs] in cross [10,20,30,40]
[[10,10],[10,20],[10,30],[10,40],[20,10],[20,20],[20,30],[20,40],[30,10],[30,20],
   [30,30],[30,40],[40,10],[40,20],[40,30],[40,40]]
```

Question 3 (Records).

A *record* is like a tuple but where elements are not accessed by position but rather by name. In Standard ML, for instance, we can create a record as follows:

```
- {a=10,b=20,c=true};
val it = {a=10,b=20,c=true} : {a:int, b:int, c:bool}
```

and access *fields* of the record by name:

```
- val r = {a=10,b=20,c=true};
val r = {a=10,b=20,c=true} : {a:int, b:int, c:bool}
- #a r;
val it = 10 : int
- #b r;
val it = 20 : int
- #c r;
val it = true : bool
```

Our object language has some support for records. The internal representation can represent records. A record value is represented by `VRecord fvs` where `fvs` is a list of pairs `(s,v)` where `s` is the field name and `v` is the value associated with the field. Thus, the record {`a=10,b=20,c=true`} can be represented by:

```
VRecord [("a",VInt 10), ("b",VInt 20), ("c",VBool true)]
```

As with lists in Question 2, there is no way to create such values from within the object language, though.

The internal representation provides an expression for constructing records, `ERecord (fs)` and an expression for accessing the field of a record, `EField (e,s)`:

- `ERecord (fs)` takes a list `fs` of pairs of the form `(s,e)` where `s` is a field name and `e` is an expression, and creates a record value where each field of the record is associated with the value obtained by evaluating the corresponding expression.

  For instance, `ERecord [("a",EVal (VInt 10)), ("b",EApp (EApp (EIdent "add", EVal (VInt 1)), EVal (VInt 2)))]` evaluates to `VRecord [("a", VInt 10), ("b", VInt 3)]`.

- `EField (e,s)` takes an expression `e` and a field name `s` and returns the value associated with field name `s` in the record obtained by evaluating `e`. (If `e` does not evaluate to a record, then an evaluation error is reported.)

  For instance, `EField (EVal (VRecord [("a",VInt 1),("b",VInt 2)]), "b")` evaluates to `VInt 2`.

(a) Complete the implementation of evaluation function `eval` so that it interprets `ERecord` and `EField` correctly.

```
- structure I = InternalRepresentation;
structure I : ...
- Evaluator.eval [] (I.ERecord []);
val it = VRecord [] : InternalRepresentation.value
- Evaluator.eval [] (I.ERecord [("a",I.EVal (I.VInt 10))]);
val it = VRecord [("a",VInt 10)] : InternalRepresentation.value
- Evaluator.eval Evaluator.initialEnv (I.ERecord [("a",I.EApp (I.EApp (I.EIdent
    "add", I.EVal (I.VInt 10)), I.EVal (I.VInt 20)))]);
val it = VRecord [("a",VInt 30)] : InternalRepresentation.value
- Evaluator.eval [] (I.ERecord [("a",I.EVal (I.VInt 10)),("b",I.EVal (I.VBool
   true))]);
val it = VRecord [("a",VInt 10),("b",VBool true)]
  : InternalRepresentation.value
```

Hint: function `lookup`, oddly enough, might be useful.

(b) Modify the parser so that it recognizes a record expression

$$\{\ s_1\ =\ e_1,\ s_2\ =\ e_2,\ \ldots,\ s_k\ =\ e_k\}$$

that creates records. The above expression should parse to an internal representation expression `ERecord`.

```
- Shell.run [];
Type . by itself to quit
Type :parse <expr> to see the parse of expression <expr>
Initial environment: add sub equal less cons nil hd tl map filter interval
   flatMap
homework4> {}
{}
homework4> {a=10}
{a=10}
homework4> {a=10,b=20}
{a=10,b=20}
homework4> {a=10, b=20, c=true}
{a=10,b=20,c=true}
homework4> :parse {a=10,b=20}
Tokens = T_LBRACE T_SYM[a] T_EQUAL T_INT[10] T_COMMA T_SYM[b] T_EQUAL T_INT[20]
    T_RBRACE
ERecord [(a,EVal (VInt 10)),(b,EVal (VInt 20))]
homework4> {a=10+20,c=40}
{a=30,c=40}
homework4> :parse {a=10+20,c=40}
```

```
Tokens = T_LBRACE T_SYM[a] T_EQUAL T_INT[10] T_PLUS T_INT[20] T_COMMA T_SYM[c]
    T_EQUAL T_INT[40] T_RBRACE
ERecord [(a,EApp (EApp (EIdent "add",EVal (VInt 10)),EVal (VInt 20))),(c,EVal (
    VInt 40))]
```

Hint: The easiest way to do this is to extend the grammar with a new rule for the `aterm` terminal:

```
aterm ::= ...
          T_LBRACE fields T_RBRACE
```

and a new nonterminal:

```
fields ::= T_SYM T_EQUAL expr T_COMMA fields
           T_SYM T_EQUAL expr
           <empty>
```

Parsing the above rule for `aterm` should produce the suitable internal representation expression.

(c) Modify the parser so that it recognizes a field-access expression

$$\texttt{\#}s\ \ e$$

that returns the value associated with field s in record e. The above expression should parse to an internal representation expression `EField`.

```
- Shell.run [];
Type . by itself to quit
Type :parse <expr> to see the parse of expression <expr>
Initial environment: add sub equal less cons nil hd tl map filter interval
    flatMap
homework4> #a {a=10,b=20}
10
homework4> #b {a=10,b=20}
20
homework4> #c {a=10}
Evaluation error: failed lookup for c
homework4> #a (let x = 10 in let y = 20 in {a=x,b=y})
10
homework4> #b (let x = 10 in let y = 20 in {a=x,b=y})
20
homework4> :parse #a {a=10,b=20}
Tokens = T_HASH T_SYM[a] T_LBRACE T_SYM[a] T_EQUAL T_INT[10] T_COMMA T_SYM[b]
    T_EQUAL T_INT[20] T_RBRACE
```

```
EField (ERecord [(a,EVal (VInt 10)),(b,EVal (VInt 20))],"a")
homework4> :parse #b {a=10,b=20}
Tokens = T_HASH T_SYM[b] T_LBRACE T_SYM[a] T_EQUAL T_INT[10] T_COMMA T_SYM[b]
    T_EQUAL T_INT[20] T_RBRACE
EField (ERecord [(a,EVal (VInt 10)),(b,EVal (VInt 20))],"b")
```

Hint: The easiest way to do this is to extend the grammar with a new rule for the `aterm` terminal:

```
aterm ::= ...
          T_HASH T_SYM expr
```

Parsing the above rule for `aterm` should produce the suitable internal representation expression.

To think about: The `#` notation for field access is obviously borrowed from Standard ML. A very common notation for field access is $e.s$, which access field s from record e. Could you support such a notation? Try to implement it as an alternate notation for field access. (It should parse into an `EField` as well.) What problem do you run into? How would you fix it?

Another thing: unlike everything else we've done on this homework, records seem to require support from the internal representing in the form of specific expression. Why? Is that really necessary? Could we get away with putting definitions in the initial environment instead?