Prog. Lang. Design and Implementation	**Homework 5**
ENGR 3599 Spring 2014 (Pucella)	**Due Date: Tuesday, Apr 8**

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 "homework5.cm"`) should be zipped together in a file `homework5-sols.zip`.
- Your file `homework5-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 `homework5-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 `homework5.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's the cleanest way to go.

An Imperative Interpreter

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

The internal representation in module `InternalRepresentation` extends the internal representation for the simple imperative language we saw in class with mutable lists and local variables.

```
datatype value = VInt of int
               | VBool of bool
               | VList of (value ref) list

and expr = EVal of value
         | ELet of string * expr * expr
         | EIf of expr * expr * expr
         | EIdent of string
         | ECall of string * expr list
         | EPrimCall of (value list -> value) * expr list

and stmt = SUpdate of string * expr
         | SIf of expr * stmt * stmt
         | SWhile of expr * stmt
         | SCall of string * expr list
         | SPrint of expr list
         | SBlock of stmt list
         | SPrimCall of (value list -> unit) * expr list
         | SVar of (string * expr * stmt)
```

Everything is as described in class, except for `VList` (Question 1) and `SVar` (Question 2).

Note that primitive operation calls occur both in the expression language (as `EPrimCall`) for primitive operations that returns a value, and in the statement language (as `SPrimCall`) for primitive operations that do not return values. Note also that primitive operations are defined to take their arguments as a list of values, so that we don't have to bother implementing primitive calls for 1 arguments, 2 arguments, 3 arguments, etc. like we did in Homework 4. See the implementation of `primPlus` and `primEq` in module `Evaluator` for examples.

Module `Evaluator` defines functions

```
eval :   (string * entry) list -> expr -> value
exec :   (string * entry) list -> stmt -> unit
```

to respectively evaluate expressions and execute statements. Both of these take as first argument an environment, which is a list of bindings associating to an identifier an *entry*, which is defined as:

```
datatype entry = Func of (string list * I.expr * (string * entry) list)
               | Proc of (string list * I.stmt * (string * entry) list)
               | Var of I.value ref
               | Con of I.value
```

representing, respectively, functions, procedures, variables, and constants.

Module `Evaluator` also defines functions to add functions, procedures, variables, and constants to the environment, which are used by the shell to extend the environment when you type in a declaration.

The parser in module `Parser` defines the following surface syntax:

```
<stmt> ::= if <expr> <stmt> else <stmt>
           if <expr> <stmt>
           while <expr> <stmt>
           <identifier> <- <expr>
           <identifier> ( <expr_list> )
           print ( <expr_list> )
           { }
           { <stmt_list> }

<stmt_list> ::= <stmt> ; <stmt_list>
                <stmt>

<expr> ::= <eterm> = <term>
           <eterm>

<eterm> ::= <term> + <term>
            <term>

<term> ::= <integer>
           true
           false
           <identifier> ( <expr_list> )
           <identifier>
           ( <expr> )
           if <expr> then <expr> else <expr>
           let <identifier> = <expr> in <expr>

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

```
<decl> ::= function <identifier> ( <sym_list> ) <expr>
           procedure <identifier> ( <sym_list> ) <stmt>
           var <identifier> = <expr>
           const <identifier> = <expr>
           <stmt>

<sym_list> ::= <identifier> , <sym_list>
               <identifier>
               <empty>
```

Have a look at module `Parser` for a list of the tokens used to implement the above grammar.

The interesting nonterminals are `<decl>`, which represents declarations that you can enter at the shell, `<stmt>`, which represents statements, and `<expr>`, which represents expressions.

One slight difference from how we've been doing parsing until now is that instead of an `expect_`T function for every token T, there is now a single `expect` function for all the tokens that do not return a value—`expect` takes a token `t` and a token list `ts` and checks if the first token of `ts` is `t`.

Question 1 (Mutable Lists)

The internal representation has a class of values for *mutable lists*, that is, lists where the elements of the lists can be changed. The representation of a mutable list is

```
VList of (value ref) list
```

A mutable list is represented as an SML list where the elements are reference cells that can hold values.

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

- `nil` is a value representing the empty mutable list
- `cons` is a function taking two arguments `x` and `xs` where `xs` is a mutable list, and returns a list `ys` where the first element of `ys` is `x` and the rest of `ys` is `xs`. The reference cells in `xs` should be the ones that appear in `ys`, so that any update to `xs` should be reflected in `ys`. (See examples in (b) below.)
- `hd` is a function taking a mutable list `xs` and returning the value of the first element of `xs`
- `tl` is a function taking a mutable list `xs` and returning the mutable list `ys` made up of all but the first element of `xs`. The reference cells in `ys` should be the same that appear in `xs`, so that any update to any but the first element of `xs` should be reflected in `ys`. (Again, see examples in (b) below.)

```
- Shell.run [];
Type . by itself to quit
Type ? by itself for environment information
Type :parse <stmt> to show parsing information
homework5> print (nil)
[]
homework5> print (cons (1, cons (2,nil)))
[1,2]
homework5> print (hd (cons (1, cons (2, nil))))
1
homework5> print (tl (cons (1, cons (2, nil))))
[2]
homework5> print (cons (33, cons (true, cons (cons (99,nil), nil))))
[33,true,[99]]
```

(b) Add the following primitive operation to the initial environment: `updateHd`.

- `updateHd` is a *procedure* taking two arguments `xs` and `x` where `xs` is a mutable list and `x` is a value, and which *updates* the first element of `xs` so that it has value `x`.

You should not have to modify the parser or the evaluator for this exercise.

```
homework5> const xs = cons (1,cons (2, cons (3, nil)))
Constant xs added to environment
homework5> print (xs)
[1,2,3]
homework5> updateHd (xs,33)
homework5> print (xs)
[33,2,3]
homework5> updateHd (tl (xs), 66)
homework5> print (xs)
[33,66,3]
homework5> updateHd (tl (tl (xs)), 99)
homework5> print (xs)
[33,66,99]
homework5> const ys = cons (0,xs)
Constant ys added to environment
homework5> print (ys)
[0,33,66,99]
homework5> updateHd (xs,99)
homework5> print (ys)
[0,99,66,99]
```

(c) Extend the parser so that it recognizes a statement of the form

$$\texttt{hd (}e_1\texttt{) <- }e_2$$

(where e_1 and e_2 are expressions) and produces for it the internal representation corresponding to the statement

$$\texttt{updateHd (}e_1\texttt{,}e_2\texttt{)}$$

```
homework5> const xs = cons (1,cons (2,cons (3,nil)))
Constant xs added to environment
homework5> print (xs)
[1,2,3]
homework5> hd (xs) <- 33
homework5> print (xs)
[33,2,3]
homework5> hd (tl (xs)) <- 66
homework5> print (xs)
[33,66,3]
homework5> hd (tl (tl (xs))) <- 99
homework5> print (xs)
[33,66,99]
homework5> :parse hd (tl (tl (xs))) <- 99
SCall ("updateHd",[ECall ("tl",[ECall ("tl",[EIdent "xs"])]),EVal (VInt 99)])
```

WARNING: *You may be tempted to define a token for* **`hd`**. *If you do that, then you might very well lose the ability to use* **`hd`** *as a function to access the head of a list. (Why?) There are a few ways around that issue. I'll let you figure out one way that works for you.*

Question 2 (Local Variables).

The internal representation

```
SVar of string * expr * stmt
```

is used to represent a local variable. Intuitively, this acts like an `ELet`, but creates a variable instead of a constant value. (Also, it is a statement, and not an expression.) Thus, executing `SVar (name,e,s)` executes statement `s` in an environment that has been extended with `name` bound to a new reference cell with initial value the value to which `e` evaluates. Variable `name` can be referred to and updated freely in `s`, and shadows any other identifier with the same name that may already appear in the environment.

(a) Complete the implementation of function `exec` in `Evaluator` for the `SVar` case.

```
- structure E = Evaluator;
structure E : ...
- structure I = InternalRepresentation;
structure I : ...
- E.exec [] (I.SVar ("x", I.EVal (I.VInt 10),
                        I.SPrint [I.EIdent "x"]));
10
val it = () : unit
- E.exec [] (I.SVar ("x", I.ELet ("y", I.EVal (I.VInt 20),
                                     I.EIdent "y"),
                        I.SPrint [I.EIdent "x"]));
20
val it = () : unit
- E.exec [] (I.SVar ("x", I.EVal (I.VInt 33),
                        I.SBlock [I.SPrint [I.EIdent "x"],
                                  I.SUpdate ("x", I.EVal (I.VInt 66)),
                                  I.SPrint [I.EIdent "x"]]));
33
66
val it = () : unit
- E.exec [] (I.SVar ("x", I.EVal (I.VInt 33),
                        I.SBlock [I.SPrint [I.EIdent "x"],
                                  I.SVar ("x", I.EVal (I.VInt 99),
                                             I.SUpdate ("x", I.EVal (I.VInt 66)
   )),
                                  I.SPrint [I.EIdent "x"]]));
33
33
val it = () : unit
```

(b) Extend the parser so that it recognizes a statement of the form

`letvar` x `=` e `in` s

(where x is an identifier, e is an expression, and s is a statement) and produces for it the internal representation

`SVar (`x`,` e`,` s`)`

```
homework5> letvar x = 10 in { print (x); x <- 20; print (x) }
10
20
homework5> letvar x = 10 in { letvar x = 20 in x <- 30 ; print (x) }
10
homework5> letvar x = 10 + 20 in { letvar y = x + x in x <- y ; print (x) }
60
homework5> :parse letvar x = 10 in x <- 30
SVar ("x",EVal (VInt 10),SUpdate ("x",EVal (VInt 30)))
homework5> :parse letvar x = 10 in { x <- 30; print (x) }
SVar ("x",EVal (VInt 10),SBlock [SUpdate ("x",EVal (VInt 30)),SPrint [EIdent "x
    "]])
```

(c) Extend the parser so that instead of writing

```
letvar x = 10 in { x <- 20; print (x) }
```

you can write instead

```
{ var x = 10 ; x <- 20; print (x) }
```

and produce the same internal representation.

More generally, in a block, `var` x `=` e creates a variable x with initial value e that can be used in the rest of that block.

You may want to study carefully the following sample output.

```
homework5> { var x = 10; x <- 20; print (x) }
20
homework5> :parse { var x = 10; x <- 20; print (x) }
SBlock [SVar ("x",EVal (VInt 10),SBlock [SUpdate ("x",EVal (VInt 20)),SPrint [
    EIdent "x"]])]
homework5> { var x = 10 ; { var x = 20 ; x <- 30 } ; print (x) }
10
homework5> :parse { var x = 10 ; { var x = 20 ; x <- 30 } ; print (x) }
SBlock [SVar ("x",EVal (VInt 10),SBlock [SBlock [SVar ("x",EVal (VInt 20),
    SBlock [SUpdate ("x",EVal (VInt 30))])],SPrint [EIdent "x"]])]
```

```
homework5> { var x = 10 ; var x = 20 ; x <- 30; print (x) }
30
homework5> :parse { var x = 10 ; var x = 20 ; x <- 30; print (x) }
SBlock [SVar ("x",EVal (VInt 10),SBlock [SVar ("x",EVal (VInt 20),SBlock [
   SUpdate ("x",EVal (VInt 30)),SPrint [EIdent "x"]])])]
homework5> { print (10); var x = 20; x <- 30; print (x) }
10
30
homework5> :parse { print (10); var x = 20; x <- 30; print (x) }
SBlock [SPrint [EVal (VInt 10)],SVar ("x",EVal (VInt 20),SBlock [SUpdate ("x",
   EVal (VInt 30)),SPrint [EIdent "x"]])]
homework5> { var x = 10; var y = x; x <- 20; print (y) }
10
homework5> :parse {var x = 10; var y = x ; x <- 20; print (y) }
SBlock [SVar ("x",EVal (VInt 10),SBlock [SVar ("y",EIdent "x",SBlock [SUpdate
   ("x",EVal (VInt 20)),SPrint [EIdent "y"]])])]
```

Note that there are many ways of achieving that effect. Some are easier to implement than others, though.