Prog. Lang. Design and Implementation	Homework 2
ENGR 3599 Spring 2014 (Pucella)	Due Date: Friday, Feb 7

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 `homework2.sml`.
- Your file `homework2.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 `homework2.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 `homework2.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.

This homework uses a variant of the intermediate representation we saw in class that supports *single-argument functions* and in which primitive operations are built into the representation. (Yes, there is a reason why I restrict to single-argument functions; see Question 1.)

```
datatype value = VInt of int
               | VBool of bool
                   (* the new stuff *)
               | VPair of value * value
               | VList of value list
               | VFun of function

and expr = EVal of value
         | EAdd of expr * expr
         | ESub of expr * expr
         | EMul of expr * expr
         | ENeg of expr
         | EEq of expr * expr
         | EIf of expr * expr * expr
         | ELet of string * expr * expr
         | EIdent of string
         | ECall of string * expr
             (* the new stuff *)
         | EPair of expr * expr
         | EFirst of expr
         | ESecond of expr
         | ESlet of (string * expr) list * expr
         | ECons of expr * expr
         | EIsEmpty of expr
         | EHead of expr
         | ETail of expr
         | ECallE of expr * expr

and function = FDef of string * expr
```

A couple of things to note: we have both new value forms and new expression forms to create and work with those new values. The questions on this homework are about implementing the functionality of those expressions. Because of these additions, the datatypes are now mutually recursive, since they refer to each other in a cycle. (Can you spot the cycle?)

First, look and internalize the provided code—it is just an interpreter structured like every other interpreter weve seen. There are a few places where functions call a function `unimplemented` that simply raises an exception. Those are the places where I want you to do something, by replacing that function call by something that does the right thing.

Question 1 (Pairs).

This question concerns the new value form

```
| VPair of value * value
```

and the new expression forms

```
| EPair of expr * expr
| EFirst of expr
| ESecond of expr
```

This lets us work with pairs in the object language.

A value `VPair (`v_1`, `v_2`)` is a pair of value v_1 and value v_2. Values v_1 and v_2 can be arbitrary values, and need not be of the same type. For example,

```
VPair (VInt 1,
       VPair (VBool true,
              VInt 3))
```

is a perfectly fine pair whose first component is integer `1` and second component is itself a pair of Boolean `true` and integer `3`.

How do we create pairs in the object language? We can create them directly by using `EVal (VPair (`v_1`, `v_2`))`, or we can use expression form `EPair (`e_1`, `e_2`)` which should evaluate to the pair value made up of the values to which e_1 and e_2 evaluate, respectively. Thus,

```
EPair (EAdd (EVal (VInt 1),
             EVal (VInt 2)),
       EVal (VInt 4))
```

should evaluate to value `VPair (VInt 3, VInt 4)`.

We only have two operations on pairs, `EFirst` and `ESecond`, that are used to extract the first and second components of a pair. Thus,

```
EFirst (EPair (EVal (VInt 1),
               EVal (VInt 2)))
```

should evaluate to `VInt 1`, and

```
ESecond (EPair (EVal (VInt 1),
                EVal (VInt 2)))
```

should evaluate to `VInt 2`. It is an evaluation error to try to extract the first or second component of a value that is not a pair.

Once we have pairs, as mentioned in class, we can "fake" a two-arguments function by passing the arguments in a pair, a three-arguments function by passing the arguments in a pair whose first component is the first argument to the function and the second component is a pair of the second and third arguments; similarly for functions with more arguments.

For example, here is function `exp` from lecture, that expects two arguments `a` and `n` and computes a^n, modified to take its arguments in a pair:

```
("exp",
 FDef ("args",
       ELet ("a", EFirst (EIdent "args"),
             ELet ("n", ESecond (EIdent "args"),
                   EIf (EEq (EIdent "n", EVal (VInt 0)),
                        EVal (VInt 1),
                        EMul (EIdent "a",
                              ECall ("exp",
                                     EPair (EIdent "a",
                                            ECall ("pred",
                                                   EIdent "n")))))))))
```

(it relies on function `pred` given in lecture, and also provided in the sample code.)

(a) Code a function `applyPair` with type

```
value -> value -> value
```

where `applyPair v1 v2` returns a pair value with `v1` and `v2` as components.

```
- applyPair (VInt 1) (VInt 2);
val it = VPair (VInt 1,VInt 2) : value
- applyPair (VInt 1) (VBool true);
val it = VPair (VInt 1,VBool true) : value
- applyPair (VInt 1) (VPair (VInt 1, VInt 2));
val it = VPair (VInt 1,VPair (VInt #,VInt #)) : value
```

(b) Code a function `applyFirst` with type

```
value -> value
```

where `applyFirst v` returns the first component of `v` if it is a pair, and returns an error otherwise.[1]

[1] Use function `evalError` provided in the code to create the error.

```
- applyFirst (VPair (VInt 1, VInt 2));
val it = VInt 1 : value
- applyFirst (VPair (VBool true, VBool false));
val it = VBool true : value
- applyFirst (VPair (VPair (VInt 1, VInt 2), VPair (VInt 3, VInt 4)));
val it = VPair (VInt 1,VInt 2) : value
- applyFirst (VInt 1);

uncaught exception Fail [Fail: Eval Error @ applyFirst]
  raised at: solution2.sml:43.28-43.54
- applyFirst (VBool true);

uncaught exception Fail [Fail: Eval Error @ applyFirst]
  raised at: solution2.sml:43.28-43.54
```

(c) Code a function `applySecond` with type

```
value -> value
```

where `applyFirst v` returns the second component of `v` if it is a pair, and returns an error otherwise.

```
- applySecond (VPair (VInt 1, VInt 2));
val it = VInt 2 : value
- applySecond (VPair (VBool true, VBool false));
val it = VBool false : value
- applySecond (VPair (VPair (VInt 1, VInt 2), VPair (VInt 3, VInt 4)));
val it = VPair (VInt 3,VInt 4) : value
- applySecond (VInt 1);

uncaught exception Fail [Fail: Eval Error @ applySecond]
  raised at: solution2.sml:43.28-43.54
- applySecond (VBool true);

uncaught exception Fail [Fail: Eval Error @ applySecond]
  raised at: solution2.sml:43.28-43.54
```

(d) Complete the cases `EPair`, `EFirst`, and `ESecond` of substitution function `subst`.

(e) Complete the cases `EPair`, `EFirst`, and `ESecond` of evaluation function `eval`. You will probably want to use the functions in (a)–(c).

Here is a sample interaction with the evaluation function, using examples from the supplied code.

```
- eval [] (EFirst (EPair (EVal (VInt 1),
                          EVal (VInt 2))));
val it = VInt 1 : value
- eval [] (ESecond (EPair (EVal (VInt 1),
                           EVal (VInt 2))));
val it = VInt 2 : value
- eval [] (EFirst (ELet ("x", EVal (VInt 1),
                         EPair (EIdent "x", EVal (VInt 2)))));
val it = VInt 1 : value
- exp;
val it = ("exp",FDef ("args",ELet (#,#,#))) : string * function
- pred;
val it = ("pred",FDef ("n",ESub (#,#))) : string * function
- eval [exp, pred] (ECall ("exp", EPair (EVal (VInt 4),
                                         EVal (VInt 9))));
val it = VInt 262144 : value
```

Question 2 (Simultaneous Bindings).

This question concerns the new expression form

```
    | ESlet of (string * expr) list * expr
```

This lets us define *simultaneous bindings*. In some kind of proto-syntax, a simultaneous binding is of the form:

```
slet x = ...
     y = ...
     z = ...
 in ...
```

(where of course the `...` stand for actual expressions).

A simultaneous binding first evaluates all the expressions to be bound to the identifiers, and *then* binds the resulting values to those identifiers. This means, in particular, that the following code (still in our proto-syntax) would evaluate to `(20,10)`:

```
let x = 10
 let y = 20
  slet x = y
       y = x
   in (x,y)
```

Spend a few seconds to see why that is different from the following code, where the bindings for `x` and `y` are *not* simultaneous:

```
let x = 10
 let y = 20
  let x = y
   let y = x
    in (x,y)
```

This last expression evaluates to `(20,20)`.

A simultaneous binding

```
slet x = 10
     y = 20
 in e
```

is expressed in our internal language as

```
ESlet ([("x", EVal (VInt 10)),
        ("y", EVal (VInt 20))],
       e)
```

Thus, you see that `ESlet` takes a list of bindings (each binding a pair of an identifier name as a string and an expression to evaluate and bind to the identifier) and an expression representing the body of the `slet`.

(a) Complete the case `ESlet` of substitution function `subst`.

Substituting into an `ESlet` requires some thought. Here is the pseudo-code for substituting into an `ESlet`. It is very similar to substituting into an `ELet`, except that we need to substitute into all the expressions in the bindings.

> To substitute `id` with `e` in `ESlet (bdgs,body)`:
>
> *substitute* `id` *with* `e` *in every expression in bindings* `bdgs`
>
> *if* `id` *is not one of the identifiers bound in* `bdgs`*:*
>
> *substitute* `id` *with* `e` *in* `body`

(b) Complete the case `ESlet` of evaluation function `eval`.

Here is a sample interaction with the evaluation function, using examples from the supplied code.

```
- eval [] (ESlet ([("x", EVal (VInt 1)),
                   ("y", EVal (VInt 2))],
                  EAdd (EIdent "x", EIdent "y")));
val it = VInt 3 : value
- eval [] (ESlet ([("x", EVal (VInt 20)),
                   ("y", EVal (VInt 10))],
                  ESlet ([("x", EAdd (EIdent "x", EIdent "y")),
                          ("y", ESub (EIdent "x", EIdent "y"))],
                         EPair (EIdent "x", EIdent "y"))));
val it = VPair (VInt 30,VInt 10) : value
- swap;
val it = ("swap",FDef ("args",ELet (#,#,#))) : string * function
- eval [swap] (ECall ("swap", EVal (VPair (VInt 10, VInt 20))));
val it = VPair (VInt 20,VInt 10) : value
- eval [] (ESlet ([("x", EVal (VInt 1)),
                   ("y", EIdent "x")],
                  EAdd (EIdent "x", EIdent "y")));

uncaught exception Fail [Fail: Eval Error @ eval/EIdent - x]
  raised at: solution2.sml:43.28-43.54
```

Question 3 (Lists).

This question concerns the new value form

```
    | VList of value list
```

and the new expression forms

```
    | ECons of expr * expr
    | EIsEmpty of expr
    | EHead of expr
    | ETail of expr
```

This lets us work with lists in the object language.

A value `VList` L is a list L of values. Those values in L can be arbitrary, and need not be of the same type. For example,

```
VList [VInt 1,
       VBool true,
       VList [],
       VList [VInt 2]]
```

is a perfectly fine list of four elements, the last two of which are themselves lists.

How do we create lists in the object language? We can create them directly by using `EVal (VList` L`)`, or we can use expression form `ECons (`e_1`,` e_2`)` which roughly corresponds to the `::` operation in Standard ML: it creates a new list in which the result of evaluating e_1 is the first element, and the rest of the elements are from the list obtained by evaluating e_2. Thus,

```
EPCons (EAdd (EVal (VInt 1),
              EVal (VInt 2)),
        EVal (VList [VInt 1, VInt 2]))
```

should evaluate to value `VList [VInt 3, VInt 1, VInt 2]`

There are three operations on lists. Operation `EIsEmpty` checks if a list is empty. It takes an expression, and if the result of evaluating the expression is an empty list, it returns true, otherwise it returns false (as values in the object language, obviously). Thus,

```
EIsEmpty (ECons (EVal (VInt 1),
                 EVal (VList [])))
```

should evaluate to `VBool false`.

Operations `EHead` and `ETail` extract out the first element of a list and the list made up of all but the first element, respectively. Thus,

```
EHead (EVal (VList [VInt 1, VInt 2, VInt 3]))
```

should evaluate to `VInt 1`, while

```
EHead (EVal (VList [VInt 1, VInt 2, VInt 3]))
```

should evaluate to `VList [VInt 2, VInt 3]`.

Here is a sample recursive function to compute the length of a list:

```
("length",
 FDef ("xs",
       EIf (EIsEmpty (EIdent "xs"),
            EVal (VInt 0),
            EAdd (EVal (VInt 1),
                  ECall ("length", ETail (EIdent "xs"))))))
```

Here is another that appends two lists, using pairs from Question 1 to pass the two lists as arguments:

```
("append",
 FDef ("args",
       ELet ("xs",EFirst (EIdent "args"),
             ELet ("ys",ESecond (EIdent "args"),
                   EIf (EIsEmpty (EIdent "xs"),
                        EIdent "ys",
                        ECons (EHead (EIdent "xs"),
                               ECall ("append",
                                      EPair (ETail (EIdent "xs"),
                                             EIdent "ys")))))))))
```

(a) Code a function `applyCons` with type

```
value -> value -> value
```

where `applyCons v1 v2` returns a list value with `v1` as first element of the list followed by the elements of `v2`. It should return an error if the second argument is not a list.

```
- applyCons (VInt 1) (VList [VInt 2, VInt 3, VInt 4]);
val it = VList [VInt 1,VInt 2,VInt 3,VInt 4] : value
- applyCons (VBool true) (VList [VInt 1, VBool false, VInt 2]);
val it = VList [VBool true,VInt 1,VBool false,VInt 2] : value
- applyCons (VList []) (VList []);
val it = VList [VList []] : value
- applyCons (VList [VInt 1, VInt 2]) (VList [VInt 3, VInt 4]);
val it = VList [VList [VInt #,VInt #],VInt 3,VInt 4] : value
- applyCons (VInt 1) (VInt 2);

uncaught exception Fail [Fail: Eval Error @ applyCons]
  raised at: solution2.sml:43.28-43.54
```

(b) Code a function `applyIsEmpty` with type

```
value -> value
```

where `applyIsEmpty v` returns `VBool true` if `v` is an empty list, and `VBool false` otherwise.

```
- applyIsEmpty (VList []);
val it = VBool true : value
- applyIsEmpty (VList [VInt 1]);
val it = VBool false : value
- applyIsEmpty (VList [VInt 1, VInt 2]);
val it = VBool false : value
- applyIsEmpty (VInt 1);
val it = VBool false : value
- applyIsEmpty (VBool true);
val it = VBool false : value
```

(c) Code a function `applyHead` with type

```
value -> value
```

where `applyHead v` returns the first element of `v` if it is a non-empty list, and returns an error otherwise.

```
- applyHead (VList [VInt 1, VInt 2, VInt 3]);
val it = VInt 1 : value
- applyHead (VList [VBool true, VBool false]);
val it = VBool true : value
- applyHead (VList []);
```

```
uncaught exception Fail [Fail: Eval Error @ applyHead]
  raised at: solution2.sml:43.28-43.54
- applyHead (VInt 1);

uncaught exception Fail [Fail: Eval Error @ applyHead]
  raised at: solution2.sml:43.28-43.54
```

(d) Code a function `applyTail` with type

```
value -> value
```

where `applyTail v` returns the list made up of all the elements of `v` except the first if `v` is a non-empty list, and returns an error otherwise.

```
- applyTail (VList [VInt 1, VInt 2, VInt 3]);
val it = VList [VInt 2,VInt 3] : value
- applyTail (VList [VInt 1]);
val it = VList [] : value
- applyTail (VList [VBool true, VBool false]);
val it = VList [VBool false] : value
- applyTail (VList []);

uncaught exception Fail [Fail: Eval Error @ applyTail]
  raised at: solution2.sml:43.28-43.54
- applyTail (VInt 1);

uncaught exception Fail [Fail: Eval Error @ applyTail]
  raised at: solution2.sml:43.28-43.54
```

(e) Complete the cases `ECons`, `EIsEmpty`, `EHead`, and `ETail` of substitution function `subst`.

(f) Complete the cases `ECons`, `EIsEmpty`, `EHead`, and `ETail` of evaluation function `eval`. You will probably want to use the functions in (a)–(d).

Here is a sample interaction with the evaluation function, using examples from the supplied code.

```
- eval [] (ECons (EVal (VInt 1),
                  EVal (VList [])));
val it = VList [VInt 1] : value
- eval [] (ECons (EVal (VInt 1),
                  EVal (VList [VInt 2])));
```

```
val it = VList [VInt 1,VInt 2] : value
- eval [] (EHead (ECons (EVal (VInt 1),
                         EVal (VList [VInt 2]))));
val it = VInt 1 : value
- eval [] (ETail (ECons (EVal (VInt 1),
                         EVal (VList [VInt 2]))));
val it = VList [VInt 2] : value
- eval [] (EIsEmpty (ECons (EVal (VInt 1),
                            EVal (VList [VInt 2]))));
val it = VBool false : value
- eval [] (EIsEmpty (ETail (ECons (EVal (VInt 1),
                                   EVal (VList [])))));
val it = VBool true : value
- eval [] (ELet ("x", EVal (VInt 1),
                 ELet ("ys", EVal (VList [VInt 2, VInt 3]),
                       ECons (EIdent "x", EIdent "ys"))));
val it = VList [VInt 1,VInt 2,VInt 3] : value
- length;
val it = ("length",FDef ("xs",EIf (#,#,#))) : string * function
- eval [length] (ECall ("length",
                        EVal (VList [VInt 10, VInt 20, VInt 30])));
val it = VInt 3 : value
- append;
val it = ("append",FDef ("args",ELet (#,#,#))) : string * function
- eval [append] (ECall ("append",
                        EVal (VPair (VList [VInt 1, VInt 2, VInt 3],
                                     VList [VBool true, VBool false]))));
val it = VList [VInt 1,VInt 2,VInt 3,VBool true,VBool false] : value
```

Question 4 (Functions as Arguments).

This question concerns the new value form

```
| VFun of function
```

and the new expression form

```
| ECallE of expr * expr
```

Together, these provide a way to call functions passed as arguments, the way we can do it in Standard ML (and other languages).

A value `VFun` f represents a function. Not a function name—a bona fide function. For example,

```
VFun (FDef ("n", EAdd (EIdent "n",
                       EVal (VInt 1))))
```

is the function that takes an argument `n` and adds `1` to it.

How do we create such function values in the object language? We can create them directly using `VFun` f as above, or we can look them up in the function environment. The idea is that an idenfier (an expression `EIdent`) that has not been substituted for evaluates down to a function definition, if that identifier's name is a function name in the function environment. For example, if we have a function environment

```
[ ("pred", FDef ("n", ESub (EIdent "n",
                            EVal (VInt 1)))) ]
```

then the expression

```
EIdent "pred"
```

should evaluate to `VFun (FDef ("n", ESub (EIdent "n", EVal (VInt 1))))`.

How do we use such functions? We use expression `ECallE`, which intuitively works just like `ECall`, except that instead of determining which function to call by looking up the name of the function in the function environment, it simply calls the function to which its first argument evaluates. Thus, for example, and in the context of the sample function environment above containing `pred`,

```
ECallE (EIdent "pred",
        EVal (VInt 10))
```

should evaluate to `VInt 9`.

More interestingly, this new expression form allows us to write functions that expect functions as arguments and apply them to other values. For instance, the following function `twice`, described in lecture, takes a function f as argument and a value v and returns $f\ (f\ v)$:

```
("twice",
 FDef ("args",
       ELet ("f", EFirst (EIdent "args"),
             ELet ("x", ESecond (EIdent "args"),
                   ECallE (EIdent "f",
                           ECallE (EIdent "f",
                                   EIdent "x"))))))
```

Note that it uses a pair to accept two arguments, as seen in Question 1.

The following function takes a function and a list as arguments and produces a new list made up of the results of applying the function to every element of the list.

```
("mapf",
 FDef ("args",
       ELet ("f", EFirst (EIdent "args"),
             ELet ("xs", ESecond (EIdent "args"),
                   EIf (EIsEmpty (EIdent "xs"),
                        EVal (VList []),
                        ECons (ECallE (EIdent "f",
                                       EHead (EIdent "xs")),
                               ECall ("mapf",
                                      EPair (EIdent "f",
                                             ETail (EIdent "xs")))))))))
```

(a) Complete the case `ECallE` of substitution function `subst`.

(b) Complete the case `ECallE` and `EIdent` of evaluation function `eval`.

Here is a sample interaction with the evaluation function, using examples from the supplied code.

```
- succ;
val it = ("succ",FDef ("n",EAdd (#,#))) : string * function
- eval [succ] (EIdent "succ");
val it = VFun (FDef ("n",EAdd (#,#))) : value
- eval [] (EIdent "nonexistent");
```

```
uncaught exception Fail [Fail: Eval Error @ eval/EIdent - nonexistent]
  raised at: solution2.sml:43.28-43.54
- eval [succ] (ECallE (EIdent "succ", EVal (VInt 10)));
val it = VInt 11 : value
- eval [succ] (ECallE (EIdent "succ",
                       ECallE (EIdent "succ", EVal (VInt 10))));
val it = VInt 12 : value
- eval [succ] (ELet ("f", EIdent "succ",
                      ECallE (EIdent "f", EVal (VInt 15))));
val it = VInt 16 : value
- twice;
val it = ("twice",FDef ("args",ELet (#,#,#))) : string * function
- eval [twice, succ] (ECall ("twice", EPair (EIdent "succ",
                                             EVal (VInt 25))));
val it = VInt 27 : value
- pred;
val it = ("pred",FDef ("n",ESub (#,#))) : string * function
- eval [twice, pred] (ECall ("twice", EPair (EIdent "pred",
                                             EVal (VInt 25))));
val it = VInt 23 : value
- mapf;
val it = ("mapf",FDef ("args",ELet (#,#,#))) : string * function
- eval [mapf,succ]
     (ECall ("mapf",
             EPair (EIdent "succ",
                    EVal (VList [VInt 10, VInt 20, VInt 30]))));
val it = VList [VInt 11,VInt 21,VInt 31] : value
- eval [mapf]
     (ECall ("mapf",
             EPair (EVal (VFun (FDef ("n",
                                      EMul (EIdent "n", EVal (VInt 2))))),
                    EVal (VList [VInt 10, VInt 20, VInt 30]))));
val it = VList [VInt 20,VInt 40,VInt 60] : value
```