CS 496: Homework Assignment 4
Due: 10 April, 11:55pm

1 Assignment Policies

Collaboration Policy. It is acceptable for students to collaborate in understanding the material but not in solving the problems or programming. Use of the Internet is allowed, but should not include searching for existing solutions.

Under absolutely no circumstances code can be exchanged between students. Excerpts of code presented in class can be used.

Assignments from previous offerings of the course must not be re-used. Violations will be penalized appropriately.

2 Assignment

This assignment has two parts. Part I consist in adding records with mutable fields to EXPLICIT-REFS to obtain EXPLICIT-REFS-MF. Part II consists in writing a series of programs in the language EXPLICIT-REFS-MF that make use of this new language construct.

2.1 Part I

As seen in class in OCaml, a mutable field in a record is one whose contents may be updated through assignment. One can declare a field to be mutable by using "`<=`" instead of "`=`". In the next example, a record is defined that holds two items of personal data, a social security number and the age. The `ssn` field in the example below is immutable but the `age` field is mutable; `age` is then updated to 31:

```
let p = {ssn = 10; age <= 30}
in begin
     p.age <= 31;  (* updates age to 31 *)
     p.age
   end
```

Evaluating this expression should produce `Ok (NumVal 31)`. Consider the evaluation of this other expression:

```
let p = {ssn = 10; age <= 30}
in begin
     p.age <= 31;
     p
   end
```

It should produce `Ok (RecordVal [("ssn", (false, NumVal 10)); ("age", (true, RefVal 1))])`. Notice two things here:

1. `false` indicates the field is immutable and `true` that it is mutable.
2. Mutable fields are always assigned a reference. In this example, the store will hold the expressed value `NumVal 31` at location 1.

Updating an immutable field should not be allowed. For example, the following expression should report an error `Error "Field not mutable"`:

```
let p = { ssn = 10; age <= 20}
in begin
    p.ssn <= 11;
    p.age
   end
```

EXPLICIT-REFS-MF also has a predicate to check whether a run-time value is a number or not. Here is how it should behave:

```
# interp "number?(2)";;
- : exp_val Explicit_refs.Ds.result = Ok (BoolVal true)
# interp "number?(2+1)";;
- : exp_val Explicit_refs.Ds.result = Ok (BoolVal true)
utop # interp "number?(zero?(0))";;
- : exp_val Explicit_refs.Ds.result = Ok (BoolVal false)
utop # interp "number?(proc(x) {x+1})";;
- : exp_val Explicit_refs.Ds.result = Ok (BoolVal false)
```

utop

The updated concrete syntax is:

```
<Expression>   ::=  { <Identifier> <FieldType> <Expression>^{+(;)} }
<Expression>   ::=  <Expression>. <Identifier>
<Expression>   ::=  <Expression>. <Identifier> <= <Expression>
<Expression>   ::=  number?( <Expression>)

<FieldType>    ::=  = | <=
```

The updated abstract syntax is as follows:

```
type expr =
  ...
  | Record of (string*(bool*expr)) list
  | Proj of expr*string
  | SetField of expr*string*expr
  | IsNumber of expr
```

For example,

```
# parse "
let p = {ssn = 10; age <= 30}
in begin
     p.age <= 31;
     p
   end";;
- : expr =
Let ("p", Record [("ssn", (false, Int 10)); ("age", (true, Int 30))],
 BeginEnd [SetField (Var "p", "age", Int 31); Var "p"])
```
utop

You are asked to implement the interpreter extension. The `RecordVal` constructor has added for you.

```
type exp_val =
  | NumVal of int
  | BoolVal of bool
  | ProcVal of string*Ast.expr*env
  | RecordVal of (string*(bool*exp_val)) list
```
ds.ml

As for `eval_expr`, the case for `Record` has already been implemented for you. You are asked to implement `Proj`, `SetField` and `IsNumber`:

```
let rec  eval_expr : expr -> exp_val ea_result = fun e ->
  match e with
  | IsNumber(e) ->
     failwith "implement"
  | Record(fs) ->
    sequence (List.map process_field fs) >>= fun evs ->
    return (RecordVal (addIds fs evs))
  | Proj(e,id) ->
     failwith "implement"
  | SetField(e1,id,e2) ->
     failwith "implement"
  | IsNumber(e) ->
     failwith "implement"
 and
 process_field (_id,(is_mutable,e))  =
  eval_expr e >>= fun ev ->
  if is_mutable
  then return (RefVal (Store.new_ref g_store ev))
  else return ev
```

where `sequence` is define here:

```
let rec sequence : ('a ea_result) list -> ('a list) ea_result  =
  fun cs ->
  match cs with
  | [] -> return []
  | c::t ->
    c >>= fun v ->
    sequence t >>= fun vs ->
    return (v::vs)
```
ds.ml

2.2 Part II: Implementing Binary Trees in EXPLICIT-REFS-MF

This part of the assignment asks you to implement binary tree operations in the language EXPLICIT-REFS-MF. Since we do not have a print operation, in order to test your code you will have to insert a `debug` instruction and inspect the contents of the environment and store. Following the discussion in class using OCaml, a binary tree will be implemented as a record with two fields `head` and `size`, both of which are mutable. The `head` field may either contain `0` for null or may be a node record. Node records have fields `data`, `left` and `right`, all of which are mutable. We will start with some examples. EXPLICIT-REFS-MF source code must be placed in text files with extension `.exr`.

Example 1 below creates a binary tree with three nodes.

```
(* Example 1 *)

let n_left = { data <= 12; left <= 0; right <= 0}    (* 0 in head signals null *)
in let n_right = { data <= 44; left <= 0; right <= 0}
in let n_root = { data <= 33; left <= n_left ; right <= n_right }
in let t1 = { root <= n_root ; size <= 3}
in debug(t1)
```

bt_tree.exr

You may run this in utop as follows:

```
# interpf "bt1";;
>>Environment:
n_left->{data <= RefVal (0); left <= RefVal (1); right <= RefVal (2)},
n_right->{data <= RefVal (3); left <= RefVal (4); right <= RefVal (5)},
n_root->{data <= RefVal (6); left <= RefVal (7); right <= RefVal (8)},
t1->{root <= RefVal (9); size <= RefVal (10)}
>>Store:
0->NumVal 12,
1->NumVal 0,
2->NumVal 0,
3->NumVal 44,
4->NumVal 0,
5->NumVal 0,
6->NumVal 33,
7->{data <= RefVal (0); left <= RefVal (1); right <= RefVal (2)},
8->{data <= RefVal (3); left <= RefVal (4); right <= RefVal (5)},
9->{data <= RefVal (6); left <= RefVal (7); right <= RefVal (8)},
10->NumVal 3
- : exp_val Explicit_refs.Ds.result = Error "Reached breakpoint"
```

utop

Example 2 below sums up all the numbers in a binary tree of numbers.

```
(* Example 2 *)

let n_left = { data <= 12; left <= 0; right <= 0}    (* 0 in head signals null *)
in let n_right = { data <= 44; left <= 0; right <= 0}
in let n_root = { data <= 33; left <= n_left ; right <= n_right }
in let t1 = { root <= n_root ; size <= 3}
in letrec sum_tree_helper (node) =
          if number?(node)
          then 0
          else node.data + (sum_tree_helper node.left) + (sum_tree_helper node.right)
in let sum_tree = proc (t) { (sum_tree_helper t.root) }
```

```
in begin
      (sum_tree t1)
   end
```
bt_sum.exr

You may run this in utop as follows:

```
# interpf "bt2";;
- : exp_val Explicit_refs.Ds.result = Ok (NumVal 89)
```
utop

You are asked to implement the following exercises. The involve implementing operations on binary search trees of numbers. The stubs are provided for you in the `src` folder. You may include helper functions. Note: you can use `=`, `<` and `>` for comparing numbers.

1. `bt_find.exr`. For example,

```
utop # interpf "bt_find";;
- : exp_val Explicit_refs.Ds.result = Ok (BoolVal true)
```
utop

2. `bt_max.exr`. For example,

```
utop # interpf "bt_max";;
- : exp_val Explicit_refs.Ds.result = Ok (NumVal 44)
```
utop

   You may assume the tree is not empty.

3. `bt_add.exr`. For example,

```
utop # interpf "bt_add";;
>>Environment:
n_left->{data <= RefVal (0); left <= RefVal (1); right <= RefVal (2)},
n_right->{data <= RefVal (3); left <= RefVal (4); right <= RefVal (5)},
n_root->{data <= RefVal (6); left <= RefVal (7); right <= RefVal (8)},
t1->{root <= RefVal (9); size <= RefVal (10)},
add_helper->...,
add_bt->...
>>Store:
0->NumVal 12,
1->NumVal 0,
2->{data <= RefVal (11); left <= RefVal (12); right <= RefVal (13)},
3->NumVal 44,
4->NumVal 0,
5->NumVal 0,
6->NumVal 33,
7->{data <= RefVal (0); left <= RefVal (1); right <= RefVal (2)},
8->{data <= RefVal (3); left <= RefVal (4); right <= RefVal (5)},
9->{data <= RefVal (6); left <= RefVal (7); right <= RefVal (8)},
10->NumVal 4,
11->NumVal 23,
12->NumVal 0,
13->NumVal 0
- : exp_val Explicit_refs.Ds.result = Error "Reached breakpoint"
```
utop

   If the key to be inserted is already in the tree, just return 0.

3 Submission instructions

Submit a file named `HW4.zip` through Canvas. Include the original stub provided. Please write the names of the members of the team in the as a comment at the top of the `interp.ml` file.