CS 496: Homework Assignment 3
Due: 13 March, 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 from different teams. 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 consists in implementing a series of extensions to the interpreter for the language called LET that we saw in class. The concrete syntax of the extensions, the abstract syntax of the extensions (`ast.ml`) and the parser that converts the concrete syntax into the abstract syntax is already provided for you. Your task is to complete the definition of the interpreter, that is, the function `eval_expr` so that it is capable of handling the new language features and also implementing any helper functions in `ds.ml`.

Before addressing the extensions, we briefly recall the concrete and abstract syntax of LET. The concrete syntax is given by the grammar in Fig. 1. Each line in this grammar is called a *production* of the grammar. We will be adding new productions to this grammar corresponding to the extensions of LET that we shall study. These shall be presented in Section 3.

Next we recall the abstract syntax of LET, as presented in class. We shall also be extending this syntax with new cases for the new language features that we shall add to LET.

<Program>	::=	<Expression>
<Expression>	::=	<Number>
<Expression>	::=	<Identifier>
<Expression>	::=	<Expression> - <Expression>
<Expression>	::=	zero? (<Expression>)
<Expression>	::=	if <Expression> then <Expression> else <Expression>
<Expression>	::=	let <Identifier> = <Expression> in <Expression>
<Expression>	::=	(<Expression>)

Figure 1: Concrete Syntax of LET

```
type expr =
  | Var of string
  | Int of int
  | Sub of expr*expr
  | Let of string*expr*expr
  | IsZero of expr
  | ITE of expr*expr*expr
```

3 Extensions to LET

This section lists the extensions to LET that you have to implement. This must be achieved by completing the stub, namely by completing the implementation of the function `eval_expr` in the file `interp.ml` of the supporting files and also implementing any helper functions in `ds.ml`.

3.1 Binary Trees

Extend the interpreter to be able to handle the operators

- `emptytree` (creates an empty tree)
- `node(e1,e2,e3)` (creates a new tree with data `e1` and left and right subtrees `e2` and `e3`; if the second or third argument is not a tree, it should produce an error)
- `empty?(e)` that returns a boolean indicating whether the `e` is an empty binary tree (should return an error if `e` does not evaluate to binary tree)
- `caseT e1 of { emptytree -> e2, node(id1,id2,id3) -> e3}`

Note that in order to implement these extensions, the set with *expressed values* must be extended accordingly (see the examples below).

The corresponding implementation of expressed values in OCaml is:

```
type 'a tree = Empty | Node of 'a * 'a tree * 'a tree

type exp_val =
  | NumVal of int
```

```
  | BoolVal of bool
  | ListVal of exp_val list
  | TreeVal of exp_val tree
```

For example,

```
# interp "emptytree";;
- : exp_val Proc.Ds.result = Ok (TreeVal Empty)

# interp "node(5, node(6, emptytree, emptytree), emptytree)";;
- : exp_val Proc.Ds.result =
Ok (TreeVal (Node (NumVal 5, Node (NumVal 6, Empty, Empty), Empty)))

# interp "
caseT emptytree of {
  emptytree -> emptytree,
  node(a,l,r) -> l
}";;
- : exp_val Proc.Ds.result = Ok (TreeVal Empty)

# interp "
let t = node(0,
             node(521,
                  emptytree,
                  node(0,
                       emptytree,
                       emptytree
                  )
             ),
             node(5-4,
                  node(104,
                       emptytree,
                       emptytree
                  ),
                  node(0,
                       node(9,
                            emptytree,
                            emptytree
                       ),
                       emptytree
                  )
             )
         )
in
caseT t of {
  emptytree -> 10,
  node(a,l,r) ->
    if zero?(a)
    then caseT l of {
            emptytree -> 21,
            node(b,ll,rr) -> if zero?(b)
                             then 4
                             else 99
         }
    else 5
}";;
-: exp_val Proc.Ds.result = Ok (NumVal 99)
```

The additional production to the concrete syntax is:

<Expression> ::= emptytree
<Expression> ::= node(<Expression>, <Expression>, <Expression>)
<Expression> ::= caseT <Expression> of
{ emptytree -> <Expression> ,
node(<Id>, <Id>, <Id>) -> <Expression> }

The abstract syntax node for this extension is as follows:

```
type expr =
  ...
  | EmptyTree
  | Node of expr*expr*expr
  | CaseT of expr*expr*string*string*string*expr
```

Here is the stub for the interpreter:

```
let rec eval : expr -> exp_val ea_result= fun e ->
  match e with
  | Int n -> return (NumVal n)

  ...

  | CaseT(e1,e2,id1,id2,id3,e3) -> failwith "Implement me!"
  | Node(e1,e2,e3)  -> failwith "Implement me!"
  | Empty(e1) -> failwith "Implement me!"
  | EmptyTree -> failwith "Implement me!"
```

3.2 Records

Extend the interpreter to be able to handle the operators

- {`fname1=e1; ...; fnamen=en`} creates a record with n fields. Field i is assigned the expressed value resulting from evaluating expression `ei`. Reports an error if there are duplicate fields.
- `e.id` projects field `id` from the record resulting from evaluating `e`. Reports an error if `e` does not evaluate to a record or does not have a field named `id`.

Examples of programs in this extension are:

1. {`age=2; height=3`}
2. `let person = {age=2; height=3} in let student = {pers=person; cwid=10} in student`
3. {`age=2; height=3`}`.age`
4. {`age=2; height=3`}`.ages`
5. {`age=2; age=3`}

Examples of evaluating expressions in this extension are:

```
# interp "{age=2; height=3}";;
- : exp_val Proc.Ds.result =
Ok (RecordVal [("age", NumVal 2); ("height", NumVal 3)])

# interp "let person = {age=2; height=3}
in let student = {pers=person; cwid=10}
in student";;
- : exp_val Proc.Ds.result =
Ok
 (RecordVal
   [("pers", RecordVal [("age", NumVal 2); ("height", NumVal 3)]);
    ("cwid", NumVal 10)])

# interp "{age=2; height=3}.age";;
- : exp_val Proc.Ds.result = Ok (NumVal 2)

 # interp "{age=2; height=3}.weight";;
- : exp_val Proc.Ds.result = Error "Proj: field does not exist"

# interp "{age=2; age=3}";;
- : exp_val Proc.Ds.result = Error "Record: duplicate fields"

# interp "2.age";;
- : exp_val Proc.Ds.result = Error "Expected a record"
```

The additional production to the concrete syntax is:

$$\begin{array}{lll} \langle\text{Expression}\rangle & ::= & \{\ \langle\text{Identifier}\rangle = \langle\text{Expression}\rangle\ ^{*(;)}\} \\ \langle\text{Expression}\rangle & ::= & \langle\text{Expression}\rangle.\ \langle\text{Identifier}\rangle \end{array}$$

The $*(;)$ above the nonterminal indicates zero or more fields separated by semi-colons.

Make sure you extend the set of expressed values so that records may be produced as a result of evaluating a program (see the examples above). The `expr` type encoding the AST is has already been extended for you:

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

```
let rec eval : expr -> exp_val ea_result= fun e ->
  match e with
  | Int n -> return (NumVal n)

  ...
  | Record(fs) -> failwith "Implement me!"
  | Proj(e,id) -> failwith "Implement me!"
```

The following `sequence` function will be very handy for the `Tuple(es)` case. Place it in the file `ds.ml`. It grabs a list of fruit basket processors (using the idiom seen in class) and builds a **unique** fruit basket processor that applies each one in turn, returning the list of their results:

```
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)
```

4 Submission instructions

Type `make zip` in the top-level folder. This will create a zip file `hw3_[DATE_AND_TIME].zip` containing the files that are required for grading. Submit that file (**and no other!**) through Canvas. Please write the names of the members of your group as a comment in `interp.ml` Your grade will be determined as follows, for a total of 100 points:

Section	Grade
3.1	60
3.2	40