CS 496: Homework Assignment 3
Due: 3 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. 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 of 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 and the parser that converts the concrete syntax into the abstract syntax is already provided for you (`Interp.parse`). Your task is to complete the definition of the interpreter, that is, the function `eval`[1] so that it is capable of handling the new language features.

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.

[1] This function has sometimes been referred to as `eval_expr` in class.

<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>

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. You should do so by completing the implementation of the function `eval` in the file `interp.ml` of the supporting files of the stub.

3.1 Abs

Extend the interpreter to be able to handle an `abs` operator. For example,

```
# Interp.interp "abs((-5)) - 6";;
- : Ds.exp_val = Ds.NumVal (-1)
# Interp.interp "abs(7) - 6";;
- : Ds.exp_val = Ds.NumVal 1
```

Note that negative numbers must be written inside parentheses. The additional production to the concrete syntax is:

<Expression> ::= abs (<Expression>)

The abstract syntax node for this extension is as follows:

```
type expr =
  ...
  | Abs of expr
```

You are asked to extend the definition of `eval` so that it is capable of handling these new forms of expressions. In particular, it should be able to handle the abstract syntax representation of `abs((-5)) - 6` which is:

```
Ast.Sub (Ast.Abs (Ast.Sub (Ast.Int 0, Ast.Int 5)), Ast.Int 6)
```

Here is the stub for the interpreter:

```ocaml
let rec eval (en:env) (e:expr):exp_val =
  match e with
  | Int n      -> NumVal n

  ...

  | Abs(e1) -> failwith("Implement me!")
```

3.2 Lists

Extend the interpreter to be able to handle the operators

- `emptylist` (creates an empty list)
- `cons` (adds an element to a list; if the second argument is not a list, it should produce an error)
- `hd` (returns the head of a list; if the list is empty it should produce an error)
- `tl` (returns the tail of a list; if the list is empty it should produce an error)
- `empty?` (checks whether a list is empty or not; if the argument is not a list it should produce an error)

Note that in order to implement these extensions, the set of *expressed values* must be extended accordingly. It now becomes:

$$\text{ExpVal} = \text{Int} + \text{Bool} + \text{List of ExpVal}$$

The corresponding implementation of expressed values in OCaml is:

```ocaml
type exp_val =
  | NumVal of int
  | BoolVal of bool
  | ListVal of exp_val list
```

The additional production to the concrete syntax is:

<Expression>	::=	emptylist
<Expression>	::=	hd (<Expression>)
<Expression>	::=	tl (<Expression>)
<Expression>	::=	empty? (<Expression>)
<Expression>	::=	cons (<Expression>, <Expression>)

For example,

```
# Interp.interp "cons(1,emptylist)";;
- : Ds.exp_val = Ds.ListVal [Ds.NumVal 1]

# Interp.interp "cons(cons(1,emptylist),emptylist)";;
- : Ds.exp_val = Ds.ListVal [Ds.ListVal [Ds.NumVal 1]]

# Interp.interp "let x = 4
      in cons(x,
              cons(cons(x-1,
                             emptylist),
                   emptylist))";;
- : Ds.exp_val = Ds.ListVal [Ds.NumVal 4; Ds.ListVal [Ds.NumVal 3]]

# Interp.interp "empty?(emptylist)";;
- : Ds.exp_val = Ds.BoolVal true

# Interp.interp "empty?(tl(cons(cons(1,emptylist),emptylist)))";;
- : Ds.exp_val = Ds.BoolVal true

# Interp.interp "tl(cons(cons(1,emptylist),emptylist))";;
- : Ds.exp_val = Ds.ListVal []

# Interp.interp "cons(cons(1,emptylist),emptylist)";;
- : Ds.exp_val = Ds.ListVal [Ds.ListVal [Ds.NumVal 1]]
```

The abstract syntax node for this extension is as follows:

```
type expr =
  ...
  | Cons of expr*expr
  | Hd of expr
  | Tl of expr
  | Empty of expr
  | EmptyList
```

Here is the stub for the interpreter:

```
let rec eval (en:env) (e:expr):exp_val =
  match e with
  | Int n         -> NumVal n

  ...

  | Cons(e1, e2) -> failwith("Implement me!")
  | Hd(e1 )       -> failwith("Implement me!")
  | Tl(e1)        -> failwith("Implement me!")
  | Empty(e1)      -> failwith("Implement me!")
  | EmptyList     -> failwith("Implement me!")
```

3.3 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.

- `caseT e1 of { emptytree -> e2, node(id1,id2,id3) -> e3}`. First it evaluates the target expressions `e1` to obtain a tree – an error should be produced if it is not. Then it either evaluates the branch `e2` or `e3` depending on whether the tree is empty or not. In the case that the tree is not empty (i.e. that it is of the form `node(ev1,ev2,ev3)`), before evaluating the `e3` branch, it extends the environment with new associations: `id1` goes to `ev1`, `id2` goes to `ev2`, and `id3` goes to `ev3`.

Note that in order to implement these extensions, the set of *expressed values* must be extended accordingly. It now becomes:

$$\text{ExpVal} = \text{Int} + \text{Bool} + \text{List of ExpVal} + \text{Tree of ExpVal}$$

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
```

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> }

For example,

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

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

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

# Interp.interp("
let t = node(emptylist,
             node(cons(5, cons(2, cons(1, emptylist))),
                  emptytree,
                  node(emptylist,
                       emptytree,
```

```
                       emptytree
                  )
             ),
             node(tl(cons(5, emptylist)),
                  node(cons(10, cons(9, cons(8, emptylist))),
                       emptytree,
                       emptytree
                  ),
                  node(emptylist,
                       node(cons(9, emptylist),
                            emptytree,
                            emptytree
                       ),
                       emptytree
                  )
             )
         )
in
caseT t of {
  emptytree -> 10,
  node(a,l,r) ->
    if empty?(a)
    then caseT l of {
            emptytree -> 21,
            node(b,ll,rr) -> if empty?(b)
                             then 4
                             else if zero?(hd(b))
                                  then 22
                                  else 99
         }
    else 5
}
");;
- : Ds.exp_val = Ds.NumVal 99
```

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 (en:env) (e:expr):exp_val =
  match e with
  | Int n        -> NumVal n

  ...

  | CaseT(e1,e2,id1,id2,id3,e3) -> failwith("Implement me!")
  | Node(e1,e2,e3)              -> failwith("Implement me!")
  | Empty(e1)                   -> (* update the definition given for lists to
  support trees *)
  | EmptyTree                   -> failwith("Implement me!")
```

4 Submission instructions

Submit a file named `HW3_<SURNAME>.zip` through Canvas. Include only the supporting files uploaded into Canvas but where `interp.ml` has been completed, as described in this document. Please write your name in the source code using comments. Your grade will be determined as follows, for a total of 100 points:

Section	Grade
3.1	20
3.2	30
3.3	50