CS 510: Homework Assignment 3
Due: 24 October, 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 in implementing a series of extensions to the interpreter for the language called PROC 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.

Before addressing the extensions, we briefly recall the concrete and abstract syntax of PROC. 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 PROC that we shall study. These shall be presented in Section 3.

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

```
<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>  ::=   proc( <Identifier>) { <Expression> }
<Expression>  ::=   ( <Expression> <Expression>)
<Expression>  ::=   ( <Expression>)
```

Figure 1: Concrete Syntax of PROC

```
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
  | Proc of string*expr
  | App of expr*expr
```

3 Extensions to PROC

This section lists the extensions to PROC 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.

3.1 Abs

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

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

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

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

For example,

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

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

# interp "let x = 4
      in cons(x,
              cons(cons(x-1,
```

```
                            emptylist),
                      emptylist))";;
-: exp_val Proc.Ds.result = Ok (ListVal [NumVal 4; ListVal [NumVal 3]])

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

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

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

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

The additional production to the concrete syntax is:

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

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 : expr -> exp_val ea_result = fun e ->
  match e with
  | Int n         -> return (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}`

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
```

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(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
}";;
-: 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) -> (* update the definition given for lists to support trees *)
  | EmptyTree -> failwith "Implement me!"
```

4 Submission instructions

Submit a file named `HW3.zip` through Canvas. Your submission should have the same files as those in the stub. 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