

# CS 496: Homework Assignment 3

Due: 25 February, 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.

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

```

2  type expr =
   | Var of string
   | Int of int
4  | Sub of expr*expr
   | Let of string*expr*expr
6  | 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 adding these extensions to your local copy of the repository of the LET language. Complete the implementation of the function `eval_expr` in the file `PLaF/src/let/lib/interp.ml`. Any helper functions must be placed in `PLaF/src/let/lib/ds.ml`. No other files should be modified.

#### 3.1 Lists

Extend the interpreter to be able to handle the operators listed below. Whenever you are requested to produce an error, no specific error string is required (use any reasonable one).

- `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:

```

2  type exp_val =
   | NumVal of int
   | BoolVal of bool
4  | ListVal of exp_val list

```

Examples of evaluation of expressions involving the new constructs are:

```

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

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

6

# interp "let x = 4
8     in cons(x,
10         cons(cons(x-1,
12             emptylist()),
                emptylist()))";
- : exp_val Let.Ds.result = Ok (ListVal [NumVal 4; ListVal [NumVal 3]])

14

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

16

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

20

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

22

# interp "cons(cons(1,emptylist()),emptylist());";
24 - : exp_val Let.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:

```

2  type expr =
   ...
   | Cons of expr*expr
4  | Hd of expr
   | Tl of expr
6  | IsEmpty of expr
   | Emptylist of texpr option

```

Here is the stub for the interpreter:

```

1  let rec eval_expr : expr -> exp_val ea_result =
   fun e ->
3  match e with
   | Int n      -> return (NumVal n)
5
   ...
7

```

```

9 | Cons(e1, e2) -> failwith "Implement me!"
  | Hd(e) -> failwith "Implement me!"
  | Tl(e) -> failwith "Implement me!"
11 | IsEmpty(e) -> failwith "Implement me!"
   | Emptylist(_t) -> failwith "Implement me!"

```

## 3.2 Tuples

Extend the interpreter to be able to handle the operators

- `<e1,...,en>` that creates a tuple with the values of each of the `ei`.
- `let <id1,...,idn> = e1 in e2` that evaluates `e1`, makes sure it is a tuple, extracts each of the `n` components of the tuple and binds them to the identifiers `id1`, ..., `idn`, respectively, and then evaluates `e2` under the extended environment.

Examples of programs in this extension are:

1. `<2,3,4>`
2. `<2,3,zero?(0)>`
3. `<<7,9>,3>` (note the use of a space)
4. `let x=2 in <zero?(4),11-x>`
5. `let <x,y,z> = <3, <5 , 12>,4> in x`
6. `let x = 34 in let <y,z>=<2,x> in z` evaluates to `Ok (NumVal 34)`.

```

# interp "<2,3,4>";;
2 - : exp_val Let.Ds.result = Ok (TupleVal [NumVal 2; NumVal 3; NumVal 4])

# interp "<2,3,zero?(0)>";;
4 - : exp_val Let.Ds.result = Ok (TupleVal [NumVal 2; NumVal 3; BoolVal true])

# interp "<<7,9>,3>";;
8 - : exp_val Let.Ds.result =
Ok (TupleVal [TupleVal [NumVal 7; NumVal 9]; NumVal 3])

# interp "let x=2 in <zero?(4),11-x>";;
12 - : exp_val Let.Ds.result = Ok (TupleVal [BoolVal false; NumVal 9])

# interp "let <x,y,z> = <3, <5 , 12>,4> in x";;
14 - : exp_val Let.Ds.result = Ok (NumVal 3)

# interp "let x = 34 in let <y,z>=<2,x> in z";;
16 - : exp_val Let.Ds.result = Ok (NumVal 34)

# interp "let <x,y> = <1,2,3> in x";;
20 - : exp_val Let.Ds.result =
22 Error "extend_env_list: Arguments do not match parameters!"

# interp "let <x,y,z> = <1,2> in x";;
24 - : exp_val Let.Ds.result =

```

```

26 Error "extend_env_list: Arguments do not match parameters!"
28 # interp "let <x,y,z> = 1 in x";;
- : exp_val Let.Ds.result = Error "Expected a tuple!"

```

The additional production to the concrete syntax is:

```

<Expression> ::= <<Expression>*(,)>
<Expression> ::= let <<Identifier>*(,)> = <Expression> in <Expression>

```

The  $*(,)$  above the nonterminal indicates zero or more copies separated by commas. The angle brackets construct a tuple with the values of its arguments. An expression of the form `let <x1,...,xn>=e1 in e2` first evaluates `e1`, makes sure it is a tuple of `n` values, say `v1` to `vn`, and then evaluates `e2` in the extended environment where each identifier `xi` is bound to `vi`.

The abstract syntax node for this extension is as follows:

```

1 type expr =
  ...
3 | Tuple of expr list
  | Untuple of string list * expr*expr

```

Here is the stub for the interpreter:

```

let rec eval_expr : expr -> exp_val ea_result =
2   fun e ->
    match e with
4   | Int n -> return (NumVal n)
    ...
6   ...
8   | Tuple(es) -> failwith "Implement me!"
    | Untuple(ids,e1,e2) -> failwith "Implement me!"
10  and
    eval_exprs : expr list -> (exp_val list) ea_result =
12    fun es ->
        match es with
14    | [] -> return []
    | h::t -> eval_expr h >>= fun i ->
16        eval_exprs t >>= fun l ->
            return (i::l)

```

## 4 Submission instructions

Hand in the `interp.ml` and `ds.ml` files ONLY, as one zip file called `hw3.zip`. 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