CS 510 – Quiz 2

Exercise 1

Consider the language LET+TUPLES, an extension of the LET-language with pairs. Its concrete syntax is given below:

<Program>	::=	<Expression>
<Expression>	::=	<Number> |<Identifier> |<Expression> - <Expression>
<Expression>	::=	zero? (<Expression>)
<Expression>	::=	if <Expression> then <Expression> else <Expression>
<Expression>	::=	let <Identifier> = <Expression> in <Expression>
<Expression>	::=	tuple(<Expression>$^{*/,}$)
<Expression>	::=	untuple (<Identifier>$^{*/,}$) = <Expression> in <Expression>

Only the last two productions in the grammar are new, the others are part of LET.

- The `tuple` keyword constructs a tuple with the values of its arguments.
- The expression `untuple (x1,...,xn)=e1 in e2` 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 xi is bound to vi.

Examples of programs in LET+TUPLES are:

1. `tuple (2,3,4)`
2. `tuple (2,3,zero?(0))`
3. `tuple (tuple(7,9),3)`
4. `tuple(zero?(4),11-x)`
5. `untuple (x,y,z)= tuple(3, tuple(5 , 12),4 )in x` is a program that evaluates to `NumVal 3`.
6. The program `let x = 34 in untuple (y,z)=tuple(2,x)in z` evaluates to `NumVal 34`.

You are asked to address the following requests.

1. Extend the expressed values of LET so that now pairs of expressed values may be produced as a result of evaluating a program (see the examples above). This requires adding a new constructor to the definition of `exp_val`. Use the tag `PairVal` as the name of your constructor.

```
type exp_val =
  | NumVal of int
  | BoolVal of bool
  (* complete below *)
```

2. Extend the interpreter for LET to LET+TUPLES, so `eval_expr` is capable of executing expressions involving `pair` and `unpair`. You may define helper functions.

```
let rec eval_expr (en:env) (e:expr) :exp_val =
  match e with
  | Int n           -> NumVal n
  | Var id           ->
    (match apply_env en id with
    | None -> failwith @@ "Variable "^id^" undefined"
    | Some ev -> ev)
  | Tuple(es) ->  (* complete *)
  | LetTuple(ids,e1,e2) -> (* complete *)
```