Homework: VarLang

Questions:

1. (4 pt) Write Varlang programs:
   (a) compute an area of a circle given the diameter $d = 10$
   (b) compute the temperature in Fahrenheit given a temperature $c = 30$ in Celsius

   Sol: (2pt each)

   (a) (let ((d 10) (r (/ d 2))) (* 3.14 r r)))
   (b) (let ((c 30)) (+ (* c (/ 9 5)) 32))

2. (4 pt) Get familiar with the Varlang syntax. Write the left-most derivations and construct parse trees for the following Varlang expressions
   (a) (let ((x 1)) (let ((x 4)) x))
   (b) (let ((c 3)(d 4)) d)

   Sol:

   (2pt each)

   (a) (let ((x 1)) (let ((x 4)) x))

```
program
⇒ exp
⇒ letexp
⇒ ’(’ ’let’ ’(’ (’(’ Identifier exp ’)’)+ ’)’ exp ’)’
⇒ ’(’ ’let’ ’(’ ’(’ Identifier exp ’)’ ’)’ exp ’)’
⇒ ’(’ ’let’ ’(’ ’(’ Letter exp ’)’ ’)’ exp ’)’
⇒ ’(’ ’let’ ’(’ ’(’ ’x’ exp ’)’ ’)’ exp ’)’
⇒ ’(’ ’let’ ’(’ ’(’ ’x’ numexp ’)’ ’)’ exp ’)’
⇒ ’(’ ’let’ ’(’ ’(’ ’x’ Number ’)’ ’)’ exp ’)’
⇒ ’(’ ’let’ ’(’ ’(’ ’x’ Digit ’)’ ’)’ exp ’)’
⇒ ’(’ ’let’ ’(’ ’(’ ’x’ 1 ’)’ ’)’ exp ’)’
⇒ ’(’ ’let’ ’(’ ’(’ ’x’ 1 ’)’ ’)’ letexp ’)’
⇒ ’(’ ’let’ ’(’ ’(’ ’x’ 1 ’)’ ’)’ ’(’ ’let’ ’(’ (’(’ Identifier exp ’)’)+ ’)’ exp ’)’ ’)’
⇒ ’(’ ’let’ ’(’ ’(’ ’x’ 1 ’)’ ’)’ ’(’ ’let’ ’(’ ’(’ Identifier exp ’)’ ’)’ exp ’)’ ’)’
⇒ ’(’ ’let’ ’(’ ’(’ ’x’ 1 ’)’ ’)’ ’(’ ’let’ ’(’ ’(’ Letter exp ’)’ ’)’ exp ’)’ ’)’
⇒ ’(’ ’let’ ’(’ ’(’ ’x’ 1 ’)’ ’)’ ’(’ ’let’ ’(’ ’(’ x exp ’)’ ’)’ exp ’)’ ’)’
⇒ ’(’ ’let’ ’(’ ’(’ ’x’ 1 ’)’ ’)’ ’(’ ’let’ ’(’ ’(’ x numexp ’)’ ’)’ exp ’)’ ’)’
⇒ ’(’ ’let’ ’(’ ’(’ ’x’ 1 ’)’ ’)’ ’(’ ’let’ ’(’ ’(’ x Number ’)’ ’)’ exp ’)’ ’)’
⇒ ’(’ ’let’ ’(’ ’(’ ’x’ 1 ’)’ ’)’ ’(’ ’let’ ’(’ ’(’ x Digit ’)’ ’)’ exp ’)’ ’)’
⇒ ’(’ ’let’ ’(’ ’(’ ’x’ 1 ’)’ ’)’ ’(’ ’let’ ’(’ ’(’ x 4 ’)’ ’)’ exp ’)’ ’)’
⇒ ’(’ ’let’ ’(’ ’(’ ’x’ 1 ’)’ ’)’ ’(’ ’let’ ’(’ ’(’ x 4 ’)’ ’)’ varexp ’)’ ’)’
⇒ ’(’ ’let’ ’(’ ’(’ ’x’ 1 ’)’ ’)’ ’(’ ’let’ ’(’ ’(’ x 4 ’)’ ’)’ Identifier ’)’ ’)’
⇒ ’(’ ’let’ ’(’ ’(’ ’x’ 1 ’)’ ’)’ ’(’ ’let’ ’(’ ’(’ x 4 ’)’ ’)’ Letter ’)’ ’)’
⇒ ’(’ ’let’ ’(’ ’(’ ’x’ 1 ’)’ ’)’ ’(’ ’let’ ’(’ ’(’ x 4 ’)’ ’)’ x ’)’ ’)’
```

   (b) (let ((c 3)(d 4)) d)

```
program
⇒ exp
⇒ letexp
⇒ ’(’ ’let’ ’(’ (’(’ Identifier exp ’)’)+ ’)’ exp ’)’
⇒ ’(’ ’let’ ’(’ (’(’ Identifier exp ’)’ ’(’ Identifier exp ’)’ )+ ’)’ exp ’)’
⇒ '(' 'let' '(' ('(' Identifier exp ')' '(' Identifier exp ')' )+ ')' exp ')'
⇒ '(' 'let' '(' '(' Identifier exp ')' '(' Identifier exp ')' ')' exp ')'
⇒ '(' 'let' '(' '(' Letter exp ')' '(' Identifier exp ')' ')' exp ')'
⇒ '(' 'let' '(' '(' 'c' exp ')' '(' Identifier exp ')' ')' exp ')'
⇒ '(' 'let' '(' '(' 'c' numexp ')' '(' Identifier exp ')' ')' exp ')'
⇒ '(' 'let' '(' '(' 'c' Number ')' '(' Identifier exp ')' ')' exp ')'
⇒ '(' 'let' '(' '(' 'c' Digit ')' '(' Identifier exp ')' ')' exp ')'
⇒ '(' 'let' '(' '(' 'c' 3 ')' '(' Identifier exp ')' ')' exp ')'
⇒ '(' 'let' '(' '(' 'c' 3 ')' '(' Letter exp ')' ')' exp ')'
⇒ '(' 'let' '(' '(' 'c' 3 ')' '(' 'd' exp ')' ')' exp ')'
⇒ '(' 'let' '(' '(' 'c' 3 ')' '(' 'd' numexp ')' ')' exp ')'
⇒ '(' 'let' '(' '(' 'c' 3 ')' '(' 'd' Number ')' ')' exp ')'
⇒ '(' 'let' '(' '(' 'c' 3 ')' '(' 'd' Digit ')' ')' exp ')'
⇒ '(' 'let' '(' '(' 'c' 3 ')' '(' 'd' 4 ')' ')' exp ')'
⇒ '(' 'let' '(' '(' 'c' 3 ')' '(' 'd' 4 ')' ')' varexp ')'
⇒ '(' 'let' '(' '(' 'c' 3 ')' '(' 'd' 4 ')' ')' Identifier ')'
⇒ '(' 'let' '(' '(' 'c' 3 ')' '(' 'd' 4 ')' ')' Letter ')'
⇒ '(' 'let' '(' '(' 'c' 3 ')' '(' 'd' 4 ')' ')' 'd' ')'
```

3. (4 pt) List free and bound variables for the following Varlang expressions:

   (a) (let ((c 5) (z a)) (let ((x c) (y b)) (* c (- z (+ x y)))))

   (b) (let ((a b)(y 10)) (let ((x y)) (+ c y x (- g a))))

   Sol (2 pt each)

   (a) Bound: c z x y, Free: a b

   (b) Bound: a y x, Free: b c g

4. (4 pt) Write 2 different 'let' expressions using the syntax of Varlang such that each expression evaluates to value 21. Each expression must include at least 2 nested let expressions and there must be a "hole in one of the scopes."(The definition of this is seen in section 3.3 of Rajan-PL book pg 48), For example:

   (let ((x 2) (y 4)) (let ((x 5) (z 10)) (let ((z 1)) (+ (* x y) z))))
   $ 21

   Sol (2 pt each)

   - (let ((x 5) (y 2)) (let ((x (* x 2))) (let ((z (* x y))) (+ z 1)))) — the expression (let ((x (* x 2))) ...) overrides the outer-scoped x and creates a hole and outer scope
   - (let ((x 5) (y 2)) (let ((z (* x y))) (let ((y (* z y))) (+ y 1)))) — similar to the previous expression, but now the last let expression (let ((y (* x y))) ...) overrides the y from the outer-scope. See the the expression (let ((z (* x y))) ...) still uses the outer scope value defined for y

5. (6 pt) Extend the interpreter: understand the idea of initial environments in programming languages. Extend the Varlang interpreter such that Varlang starts with an environment with predefined values

   $ course
   342
   $ id
   500
   $ version
   1.0

Sol

(6pt)

Add the `Evaluator.initialize` method, add the call to initialize in `valueOf` method.

```
public class Evaluator implements Visitor<Value> {
  Env intialize(Env e){
    Env new_env = new ExtendEnv(e, "course", new NumVal(342));
    new_env = new ExtendEnv(new_env, "id", new NumVal(500));
    new_env = new ExtendEnv(new_env, "version", new NumVal(1.0));
    return new_env;
  }
  Value valueOf(Program p) {
    Env env = new EmptyEnv();
    env= intialize(env);
    Value result = (Value) p.accept(this, env);
    return result;
  }
}
```

6. (8 pt) Extend the interpreter: The let expression is useful to define multiple variables at the same time. However, one might want to refer to the previous defined variables in the same let expression when defining later variables. For example, in the evaluation of `(let ((a 3) (b a) (c (+ a b))) c)`, we created three variables, a, b and c, where a=3, b=a, c=a+b. Currently the Varlang intepreter will report "No binding found for name: a" This problem asks you to modify existing let evaluator to eliminate such errors and support this behavior. In the above example, *b* will get the value 3, c will get the value 6.

   Sol

   Evaluator.java (8 pt):

```
public class Evaluator implements Visitor<Value> {
        public Value visit( LeteExp e, Env env ) {
            List<String> names = e.names();
            List<Exp> value_exps = e.value_exps();

            Env new_env = env;
            for(int i = 0; i < names.size(); i++) {
                    Exp exp = value_exps.get(i);
                    Value value = (Value)exp.accept(this, new_env);
                    new_env = new ExtendEnv(new_env, names.get(i), value);
            }

            return (Value) e.body().accept(this, new_env);
        }
}
```

7. (20 pt) Extend the interpreter: Security is a major concern for any system. To deal with this it becomes important that deallocated memory of some program does not contain the data which can be read by malicious programs thereby causing leak of information. To avoid such information leak due to environment storage we can augment the Varlang language with a *lete* (encoded let) expression that encodes the value before storing in environment and *dec* expression that decodes it prior to using it. Extend the Varlang programming language to support these two expressions. Implement an encrypted let (lete for let encrypted), which is similar to let but it uses a key and a dec expression that is similar to VarExp. All values are stored by encrypting them with key, and read by decrypting them with key.

> (lete 42 ((x 1)) x)
43 > (lete 42 ((x 1)) (dec 42 x))
1 > (lete 10 ((y 12)) y)
22 > (lete 10 ((y 12)) (dec 10 y))
12

Sol

Grammar file (3pt):

```
  exp returns [Exp ast]:
 v=varexp { $ast = $v.ast; }
 | n=numexp { $ast = $n.ast; }
 | a=addexp { $ast = $a.ast; }
 | s=subexp { $ast = $s.ast; }
 | m=multexp { $ast = $m.ast; }
 | d=divexp { $ast = $d.ast; }
 | l=letexp { $ast = $l.ast; }
 | le=leteexp { $ast = $le.ast; }
 | de=decexp { $ast = $de.ast; }
 ;


leteexp  returns [LeteExp ast]
 locals [ArrayList<String> names = new ArrayList<String>(), ArrayList<Exp> value_exps = new ArrayList<Exp>()] :
        '(' Lete
        key=numexp
        '(' ( '(' id=Identifier e=exp ')' { $names.add($id.text); $value_exps.add($e.ast); } )+ ')'
        body=exp
        ')' { $ast = new LeteExp($names, $value_exps, $body.ast, $key.ast); }
   ;

decexp returns [DecExp ast]:
        '(' Dec
        key=numexp
        id=Identifier
        ')'     { $ast = new DecExp($key.ast, $id.text); }
   ;

 Lete : 'lete';
 Dec : 'dec';
```

Evaluator(10pt):

```
  public class Evaluator implements Visitor<Value> {
    public Value visit( LeteExp e, Env env ) {
      List< String > names = e.names( );
      List< Exp > value_exps = e.value_exps( );
      List< Value > values = new ArrayList< Value >( value_exps.size( ) );
      NumVal key = (NumVal) e.key().accept(this, env);
      for ( Exp exp : value_exps )
      values.add( ( Value ) exp.accept( this, env ) );
      Env new_env = env;
      for ( int i = 0; i < names.size( ); i++ ) {
        Value val=values.get( i );
        if(val instanceof NumVal){
          val=new NumVal(((NumVal)val).v()+key.v());
        }
        new_env = new ExtendEnv( new_env, names.get( i ), (val));
      }
      return ( Value ) e.body( ).accept( this, new_env );
    }
    public Value visit( DecExp e, Env env ) {
      Value val =env.get(e.name());
```

```
        if(val instanceof NumVal){
          NumVal key= (NumVal) e.key().accept(this, env);
          return new NumVal(((NumVal)val).v())-key.v());
        }
        return val;
      }
    }
```

AST(4pt):

```
    public interface AST {
      public static class LeteExp extends LetExp {
        NumExp _key;
        public LeteExp( List< String > names, List< Exp > value_exps, Exp body, NumExp key ) {
          super( names, value_exps, body );
          _key = key;
        }
        public NumExp key( ) {
          return _key;
        }
        public Object accept( Visitor visitor, Env env ) {
          return visitor.visit( this, env );
        }
      }
      public static class DecExp extends VarExp {
        NumExp _key;
        public DecExp( NumExp key, String varExp ) {
          super( varExp );
          _key = key;
        }
        public NumExp key() {
          return _key;
        }
        public Object accept( Visitor visitor, Env env ) {
          return visitor.visit( this, env );
        }
        public boolean isBound(String variableName) {
          return this.key().isBound(variableName);
        }
      }
      public interface Visitor <T> {
        public T visit(AST.LeteExp e, Env env); // New for the varlang
        public T visit(AST.DecExp e, Env env); // New for the varlang
      }
    }
```

Printer (3pt)

```
    public class Printer {
      public static class Formatter implements AST.Visitor<String> {
        public String visit( LeteExp e, Env env ) {
          String result = "(lete␣";
          result += e.key( );
          result += "␣(";
          List< String > names = e.names( );
          List< AST.Exp > value_exps = e.value_exps( );
          int num_decls = names.size( );
          for ( int i = 0; i < num_decls; i++ ) {
            result += "␣(";
            result += names.get( i ) + "␣";
            result += value_exps.get( i ).accept( this, env ) + ")";
          }
          result += ")␣";
          result += e.body( ).accept( this, env ) + "␣";
          return result + ")";
        }
        public String visit( DecExp e, Env env ) {
          String result = "(dec␣";
          result += e.key( ) + "␣";
          result += e.name( ) + "␣)";
          return result;
        }
      }
    }
```