Homework: ArithLang

Questions:

1. (4 pt) Write two Arithlang programs that compute 24, containing more than three operators.

   Sol

   - (* 2 (+ (- 5 3) (* 5 2)))
   - (+ 4 (* (+ 1 2) (- 3 1))

2. (4 pt) Get familiar with the syntax of Arithlang: write the derivations for the following programs in ArithLang

   (a) leftmost derivation: (* (+ 2 (- 5 8)) (* 2 1))

   (b) rightmost derivation: (/ 34 (* 4 2) (+ 34 67))

   Sol

   - leftmost derivation: (* (+ 2 (- 5 8)) (* 2 1))

```
program
-> exp
-> mulexp
-> ( ∗ exp exp )
-> ( ∗ addexp exp )
-> ( ∗ ( + exp exp ) exp )
-> ( ∗ ( + 2 exp ) exp )
-> ( ∗ ( + 2 subexp ) exp )
-> ( ∗ ( + 2 ( − exp exp ) ) exp )
-> ( ∗ ( + 2 ( − 5 exp ) ) exp )
-> ( ∗ ( + 2 ( − 5 8 ) ) exp )
-> ( ∗ ( + 2 ( − 5 8 ) ) multexp )
-> ( ∗ ( + 2 ( − 5 8 ) ) ( ∗ exp exp ) )
-> ( ∗ ( + 2 ( − 5 8 ) ) ( ∗ 2 exp ) )
-> ( ∗ ( + 2 ( − 5 8 ) ) ( ∗ 2 1 ) )
```

   - rightmost derivation: (/ 34 (* 4 2) (+ 34 67))

```
program
-> exp
-> divexp
-> (/ exp exp exp)
-> (/ exp exp addexp)
-> (/ exp exp (+ exp exp))
-> (/ exp exp (+ exp 67))
-> (/ exp exp (+ 34 67))
-> (/ exp mulexp (+ 34 67))
-> (/ exp (∗ exp exp) (+ 34 67))
-> (/ exp (∗ exp 2) (+ 34 67))
-> (/ exp (∗ 4 2) (+ 34 67))
-> (/ 34 (∗ 4 2) (+ 34 67))
```

3. (6 pt) Get familiar with the ArithLang source code: the Evaluator and Formatter classes used the visitor patterns, explain what are the purposes of the two classes and how the visitor patterns are implemented.

   Sol The purpose of the Evaluator class is to compute a value of the arithmetic expression from an AST. The purpose of the Formatter class is to generate a string of the arithmetic expression from the AST. To implement the visitor pattern, the class must provide the type of the value it will generate. For the Evaluator, the value is the class Value and for the Formatter, the final value is a string. The implementation must override visit methods for each type of node (or subclass) of the AST node types (e.g. NumExp, AddExp). For each sub-expression in each expression, the visitor must call the method accept, so these sub-expressions are evaluated.

4. (10 pt) Extend ArithLang to support the greatest common factor operation, (gcf num1 num2), e.g.,
   (gcf 30 60) = 30
   (gcf (gcf 45 60) 90) = 15
   (gcf 45 60 90) = 15
   (gcf 10) error
   (gcf -10 20) error

   Sol Found in hw2code-sol.zip

   Files to modify

   (2 pt) grammar file

```
exp returns [Exp ast]:
  | gc=gcfexp { $ast = $sq.ast; }

gcfexp returns [GcfExp ast] :
     '(' Gcf e=exp ')'
     { $ast = new GcfExp($e.ast); }
     ;

Gcf : 'gcf' ;
```

   (2 pt)

```
public interface AST {
  public static class GcfExp extends CompoundArithExp {
           public GcfExp(List<Exp> args) {
              super(args);
           }
           public Object accept(Visitor visitor) {
              return visitor.visit(this);
           }
  }
}
```

   (5 pt, 2 pt for checking of negative value, zero or float)

```
public class Evaluator implements Visitor<Value> {
  @Override
```

```
    @Override
    public Value visit(GcfExp e) {
            List<Exp> operands = e.all();

            List<Integer> values = new ArrayList<>();
            for(int i = 0; i < operands.size(); i++) {
                NumVal rVal = (NumVal) operands.get(i).accept(this);
                if (rVal.v() <= 0 || rVal.v() % 1 > 0) {
                  return new DynamicError("Negative operator for gcf is not permitted. " + ts.visit(e))
                      ;
              }
              values.add((int) rVal.v());
            }

            int result = values.get(0);
            for(int i = 1; i < values.size(); i++) {
              result = gcf(result, values.get(i));
            }

            return new NumVal(result);
    }

    private int gcf(int a, int b) {
            if (b == 0) return a;
            return gcf(b,a % b);
    }
  }
```

(1 pt)

```
  public class Printer {
    public static class Formatter implements AST.Visitor<String> {
      public String visit(GcfExp e) {
              String result = "(gcf ";
              for(AST.Exp exp : e.all())
                      result += " " + exp.accept(this);
              result += ")";
              return result;
      }
    }
  }
```

5. (26 pt) Implement an interpreter for AbstractLang described as follows. You can modify from the ArithLang code.
   (a) This language contains only three terminals, 0, p, n, u
   (b) p represents positive numbers, n represents negative numbers and u represents unknown values
   (c) There are three operators *, -, + that can be applied on the terminals, their syntactic rules are similar to *, - and + in ArithLang, except that each operator only can take two operands

(d) The semantics of AbstractLang are defined by the following rules (the first column in the table represents the first operand and the first row in the table represents the second operand of the operation):

+	**0**	**p**	**n**	**u**
0	0	p	n	u
p	p	p	u	u
n	n	u	n	u
u	u	u	u	u

-	**0**	**p**	**n**	**u**
0	0	n	p	u
p	p	u	p	u
n	n	n	u	u
u	u	u	u	u

*****	**0**	**p**	**n**	**u**
0	0	0	0	0
p	0	p	n	u
n	0	n	p	u
u	0	u	u	u

Sol Found in hw2code-5-sol.zip. Files to modify:

- (6 pt) grammar file (check AbstractLang.g)
- (4 pt) AST file (AST.java)
- (3 pt) printer file (Printer.java)
- (4 pt) value file (Value.java). Need to create a val type for each type of value: PositiveVal, NegativeVal, ZeroVal and UnknownVal (1 pt each)
- (9 pt) evaluator (Evaluator.java). The implementation of each operation (add, sub and mult) is worth 3 pt