COMP 520: Compilers
Sample Solutions – Written Assignment 3

(a) Show a stratified LL(1) grammar for arithmetic expressions over the three terminals {*num*, +, -} so that "−" can be used both as a binary subtraction operator and as a unary arithmetic negation operator. The grammar should yield a concrete syntax tree reflecting that unary negation is right-associative and has higher precedence than addition and subtraction. Addition and subtraction have the same precedence and are left-associative. For example, the expression `2 – 3 + -4 - - -5` should have a concrete syntax tree that reflects the ordering `(((2-3)+(-4))-(-(-5)))`.

Soln (a): A stratified grammar reflecting the desired precedence and associativity

S ::= E $	augment the grammar for parseability
E ::= E (**+** | **-**) U | U	**+**,**-** binary opns: left associative, lower precedence
U ::= **-** U | T	**-** unary opn: right associative, higher precedence
T ::= *num*	leaves of the expression trees – numbers, highest precedence

Place the grammar into LL(1) form by removing the left recursion in the second rule.

S ::= E $	(1)
E ::= U ((**+** | **-**) U)*	(2)
U ::= **-** U | T	(3)
T ::= *num*	(4)

Rules (2) and (3) are the only rules with choices. Rule (3) trivially meets the LL(1) condition. In rule (2) the Kleene star choice point can be resolved with a single terminal lookahead. Since

Predict((**+** | **-**) U) = {**+**,**-**}

is disjoint from

Predict(ε) = Follow(E) = {$}

So our grammar satisfies the LL(1) condition and is suitable for recursive descent parsing.

(b) Describe how you would modify the simpleAST example on the class website to build *Expr* ASTs using *BinExpr*, *UnaryExpr* and *NumExpr* nodes? Describe your extensions to the scanner, parser, AbstractSyntaxTrees, and visitors. It's not necessary to write out complete code, but if you are interested you can extend the simpleAST example to try it out.

Soln (b): (refer to simpleAST example (#2 on web page) to understand the additions)

To scan the "**-**" terminal we need to:

- Add a MINUS token to the TokenKind enum in the scanner.
- Extend the scanner to return a MINUS token when encountered.

To build an AST with unary expressions, we need to add to `AbstractSyntaxTrees` package:

```
public class UnaryExpr extends Expr {
   public Token oper;
   public Expr right;
}
```

Also add a constructor for this class.

To traverse ASTs we need to add to `interface Visitor<Inh,Syn>` the method

```
public Syn visitUnaryExpr(UnaryExpr expr, Inh arg);
```

and add a corresponding `visit` method in the `UnaryExpr` class.

To build the AST we need to modify the `Parser` class

- Extend the `parseE()` method to use whichever token (+ or -) is encountered in constructing a `BinExpr`
- Add a `parseU()` method

```
Expr parseU() {
   if (currentToken.kind == TokenKind.MINUS) {
      Token negation = currentToken;
      acceptIt();
      return new UnaryExpr(negation, parseU());
   }
   else
      return parseT();
}
```

To evaluate the AST

- Extend the `DisplayAST` visitor in the `TraverseAST` package to include.

```
public String visitUnaryExpr(UnaryExpr e, Object arg){
   return "(" + "-" + e.right.visit(this, null) + ")";
}
```

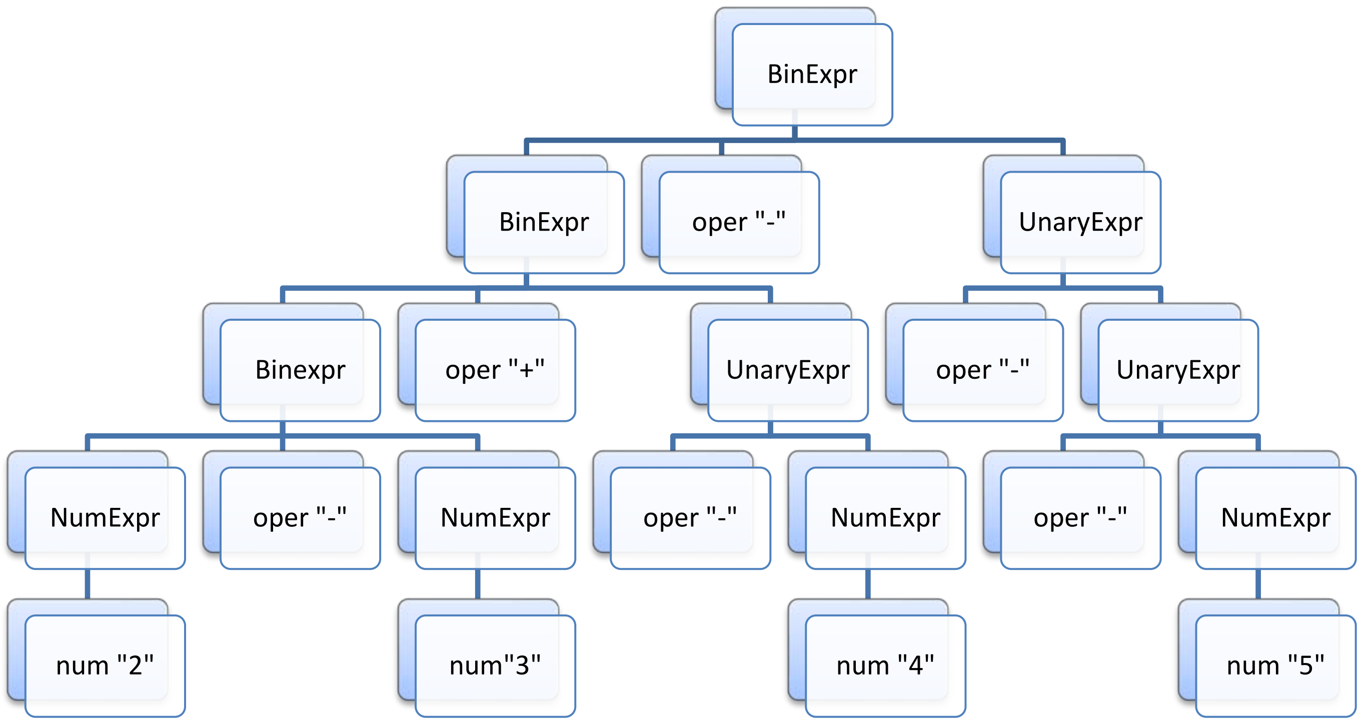
  (display of"-"in a `BinaryExpr` is automatically correct as it uses the operator spelling)
- Extend the `Eval` visitor in the `TraverseAST` package to include unary negation

```
public Integer visitUnaryExpr(UnaryExpr e, Object arg){
    // evaluate right subtree
    Integer rval = e.right.visit(this, null);
    return –rval;
}
```

- Also add the subtraction case in evaluation of a `BinaryExpr`

(c) Draw the AST for `2 – 3 + -4 - - -5` using *BinExpr*, *UnaryExpr* and *NumExpr* nodes and show the spelling of the tokens in the AST.

Soln (c). The AST is shown below. It corresponds to `(((2 - 3) + (-4)) - (-(-5)))`


BinExpr
BinExpr
oper "-"
UnaryExpr
Binexpr
oper "+"
UnaryExpr
oper "-"
UnaryExpr
NumExpr
oper "-"
NumExpr
oper "-"
NumExpr
oper "-"
NumExpr
num "2"
num"3"
num "4"
num "5"