COP4020 Programming Assignment 6

1. Consider the following augmented LL(1) grammar for an expression language:

```
<expr>   -> <term> <term_tail>
<term>   -> <factor> <factor_tail>
<term_tail>   -> '+' <term> <term_tail2>
| '-' <term> <term_tail2>
| empty
<factor1> -> '(' <expr> ')' 
| '-' <factor2>
| number
<factor_tail1> -> '*' <factor> <factor_tail2>
| '/' <factor> <factor_tail2>
| empty
```

```
term_tail.subtotal := term.value;
expr.value := term_tail.value

factor1.subtotal := factor.value;
term.value := factor_tail.value

term_tail2.subtotal :=
term_tail1.subtotal+term.value;
term_tail1.value := term_tail2.value

term_tail1.subtotal :=
term_tail1.subtotal-term.value;
term_tail1.value := term_tail2.value

factor1.value := expr.value
factor1.value := -factor2.value

factor1.value := number

factor_tail1.subtotal :=
factor_tail1.subtotal*factor.value;
factor_tail1.value := factor_tail2.value

factor_tail1.subtotal :=
factor_tail1.subtotal/factor.value;
factor_tail1.value := factor_tail2.value
```

Note: the indexing (1 and 2) that is used with nonterminals, such as <factor1> and <factor2>, is only relevant to the semantic rules to identify the specific occurrences of the nonterminals in a production. For example, there is only one <factor> nonterminal and production in the grammar.

Draw the decorated parse tree for \(-2*3+1\) that shows the attributes and their values.

2. The following calculator Java program implements the attribute grammar shown above to calculate the value of an expression. To this end, the synthesized value attributes are returned as integer values from the methods that correspond to nonterminals. Inherited subtotal attributes are passed to the methods as arguments:

```java
import java.io.*;
public class Calc {
    private static StreamTokenizer tokens;
    private static int token;

    public static void main(String argv[]) throws IOException {
        InputStreamReader reader;
        if (argv.length > 0)
            reader = new InputStreamReader(new FileInputStream(argv[0]));
        else
            reader = System.in;

        /* Calc.java
         * Implements a parser and calculator for simple expressions
         * Uses java.io.StreamTokenizer and recursive descent parsing
         * Compile:
         * javac Calc.java
         * Execute:
         * java Calc
         * or:
         * java Calc <filename>
         */
        try {
            while ((token = reader.nextToken()) != StreamTokenizer.TT_EOF) {
                if (token == StreamTokenizer.TT_NUMBER) {
                    System.out.print(token); // Output the number
                } else if (factor1(token) || factor2(token) || term_tail1(token) || term_tail2(token) || expr(token) || term_tail1(value) || term_tail2(value) || factor1(value) || factor2(value) || term(value) || expr(value) {
                    System.out.print(token); // Output the symbol
                } else {
                    System.out.print(token); // Output the non-terminal
                }
            }
            System.out.println();
        }
    }
}
```
reader = new InputStreamReader(System.in);
// create the tokenizer:
tokens = new StreamTokenizer(reader);
tokens.ordinaryChar('.');
tokens.ordinaryChar('-');
tokens.ordinaryChar('/');
// advance to the first token on the input:
getToken();
// parse expression and get calculated value:
int value = expr();
// check if expression ends with ';' and print value
if (token == (int)';')
    System.out.println("Value = " + value);
else
    System.out.println("Syntax error");
}
// getToken - advance to the next token on the input
private static void getToken() throws IOException
{
    token = tokens.nextToken();
}
// expr - parse <expr> -> <term> <term_tail>
private static int expr() throws IOException
{
    int subtotal = term();
    return term_tail(subtotal);
}
// term - parse <term> -> <factor> <factor_tail>
private static int term() throws IOException
{
    int subtotal = factor();
    return factor_tail(subtotal);
}
// term_tail - parse <term_tail> -> <add_op> <term> <term_tail> | empty
private static int term_tail(int subtotal) throws IOException
{
    if (token == (int)'+')
    {
        getToken();
        int termvalue = term();
        return term_tail(subtotal + termvalue);
    }
    else if (token == (int)'-')
    {
        getToken();
        int termvalue = term();
        return term_tail(subtotal - termvalue);
    }
    else
        return subtotal;
}
// factor - parse <factor> -> '(' <expr> ')' | '-' <expr> | identifier | number
private static int factor() throws IOException
{
    if (token == (int)'(')
    {
        getToken();
        int value = expr();
        if (token == (int)')')
            getToken();
        else
            System.out.println("closing ')' expected");
    }
    else if (token == (int)'-')
    {
        getToken();
        return -factor();
    }
    else if (token == tokens.TT_WORD)
    {
        getToken();
        // ignore variable names
        return 0;
    }
    else if (token == tokens.TT_NUMBER)
    {
        getToken();
    }
return (int)tokens.nval;
} 
else 
(System.out.println("factor expected"); 
return 0; 
}

// factor_tail - parse <factor_tail> -> <mult_op> <factor> <factor_tail> | empty
private static int factor_tail(int subtotal) throws IOException
{ if (token == (int)'*')
{ getToken();
int factorvalue = factor();
return factor_tail(subtotal * factorvalue);
}
else if (token == (int)'/')
{ getToken();
int factorvalue = factor();
return factor_tail(subtotal / factorvalue);
}
else 
return subtotal;
}

Download this example calculator Java program from:
http://www.cs.fsu.edu/~engelen/courses/COP4020/Calc.java

Explain why the input 1/2; to this program produces the value 0. What are the relevant parts of the program involved in computing this result?

3. We extend the attribute grammar with two new productions and two new attributes for all nonterminals:

- the **in** inherited attribute is a symbol table with identifier-value bindings that defines the bindings of identifiers in the scope (context) in which (part of) the expression is evaluated,
- the **out** synthesized attribute is a symbol table with identifier-value bindings that holds the **in** bindings plus the new bindings introduced by (part of) the expression as explained below.

The two new productions with corresponding semantic rules are:

```java
<expr1> -> 'let' identifier '=' <expr2> expr2.in := expr1.in;
            expr1.value := expr2.value
            expr1.out := expr2.out.put(identifier=expr2.value)
| <term> <term_tail>
            term.in := expr1.in;
            term_tail.in := term.out;
            term_tail.subtotal := term.value;
            expr1.value := term_tail.value;
            expr1.out := term_tail.out

<factor1> -> '(' <expr> ')'
            expr.in := factor1.in;
            factor1.value := expr.value
            factor1.out := expr.out
| '-' <factor2>
            factor2.in := factor1.in;
            factor1.value := -factor2.value;
            factor1.out := factor2.out
| identifier
            factor1.value := factor1.in.get(identifier)
            factor1.out := factor1.in
| number
            factor1.value := number;
            factor1.out := factor1.in
```

3
The first production introduces an assignment construct as an expression, similar to the C/C++ assignment which can also be used within an expression. For example,

```
(let x = 3) + x;
Value = 6
```

The semantic rule `expr2.in := expr1.in` copies the symbol table of the context in which `expr1` is evaluated to the context of `expr2`. The evaluation of `expr2` may change the symbol table and the table is copied to `expr1` with the semantic rule `expr1.out := expr2.out`.

For this part of the assignment, you have to change the semantic rules of all other productions in the grammar to include assignments for the `in` and `out` attributes to pass the symbol table. Write down the grammar with these new semantic rules.

4. Implement the two new productions and semantic rules in an updated `Calc.java` program.

To implement a symbol table with identifier-value bindings, you can use the Java `java.util.Hashtable` class as follows:

```java
import java.util.*;
...
public class Calc {
    ...
    public static void main(String argv[]) throws IOException {
        ...
        Hashtable exprin = new Hashtable();
        Hashtable exprout = new Hashtable();
        ...
        int value = expr(exprin, exprout);
        ...
    }
    private static int expr(Hashtable exprin, Hashtable exprout) throws IOException {
        if (token == tokens.TT_WORD && tokens.sval.equals("let")) {
            getToken(); // advance to identifier
            String id = tokens.sval;
            getToken(); // advance to '='
            getToken(); // advance to <expr>
            int value = expr(exprin, exprout);
            exprout.put(id, new Integer(value));
            return val;
        } else {
            Hashtable termout;
            int subtotal = term(exprin, termout);
            return term_tail(subtotal, termout, exprout);
        }
    }
    private static int factor(Hashtable factorin, Hashtable factorout) throws IOException {
        ...
        else if (token == tokens.TT_WORD) {
            String id = tokens.sval;
            getToken();
            factorout.putAll(factorin);
            return ((Integer)factorin.get(id)).intValue();
        }
        ...
    }
}
```

The `put` method puts a key and value in the hashtable, where the value must be a class instance so an `Integer` instance is created. The `get` method returns the
value of a key. The `intValue` method of `Integer` class returns an `int`. The `containsKey` method checks if a key is present in the hashtable. The `putAll` method takes a `Hashtable` and copies its contents into the `Hashtable` object.

Test your new `Calc.java` application. For example:

```
let x = 1;
Value = 1

(let x = 1) + x;
Value = 2

(let a = 2) + 3 * a;
Value = 8

1 + (let a = (let b = 1) + b) + a;
Value = 5
```