**Subject:** Interpreter implementation for a three-valued logic language

**Problem:** In this assignment, you are expected to implement an interpreter for a custom three-valued logic language. The language is called *ETU TVL Language* and its grammar description in **EBNF** is given below.

**The language**

<ETU TVL Language> $\rightarrow$ PROGRAM <ID> ; <Declaration Section> <Initialization Section> <Main Section>

<Declaration Section> $\rightarrow$ DECLARATION SECTION [<Variable Name List>] ;
<Variable Name List> $\rightarrow$ <Variable Name> | <Variable Name> , <Variable Name List>
<Variable Name> $\rightarrow$ *id*

<Initialization Section> $\rightarrow$ INITIALIZATION SECTION {<Init List> ;}
<Init List> $\rightarrow$ <Variable Name> = <Logical Value>
<Logical Value> $\rightarrow$ TRUE | FALSE | UNKNOWN

<Main Section> $\rightarrow$ MAIN SECTION { <Statement> ;}
<Statement> $\rightarrow$ <Input Stmt> | <Output Stmt> | <Bool Assignment Stmt>
<Input Stmt> $\rightarrow$ INPUT '*message*' <Variable Name>
<Output Stmt> $\rightarrow$ OUTPUT '*message*' <Bool Expression>
<Bool Assignment Stmt> $\rightarrow$ <Variable Name> = <Bool Expression>
<Bool Expression> $\rightarrow$ <Bool Term> | <Bool Expression> OR <Bool Term>
<Bool Term> $\rightarrow$ <Bool Factor> | <Bool Term> AND <Bool Factor>
<Bool Factor> $\rightarrow$ <Bool Primary> | NOT <Bool Primary>
<Bool Primary> $\rightarrow$ <Logical Value> | <Variable Name> | (<Bool Expression>)

Other key information about the language is as follows.

- Tokens: The token *id* is a letter followed by any number of letters or digits, i.e. `["a"-"z","A"-"Z"] ( ["a"-"z","A"-"Z", "0"-"9"])*`. The second is the *message*, any sequence of characters other than newline, for user interaction. The other tokens have exactly one lexeme.

- Comments: The rest of any line starting with the text "- -" is comment, therefore it should be skipped. Multi-line comments are not allowed.

- Whitespace characters: All whitespace characters, *blank*, *tab* and *newline*, are neglected. The only exception is the whitespace characters within the *message*.

- Operator Precedence: Parenthesis have the highest precedence. The unary operator **NOT** has the second while the **AND** operator has the third highest precedences. Operator **OR** has the lowest precedence. Note that the operator precedences are already enforced by the grammar.

- Operator Associativity: Operators **OR** and **AND** are associative.

- **OR** operator: The usual 'OR' logic applies in case both operands are either TRUE or FALSE. In case, one of the operand is UNKNOWN and the other is TRUE, the result is TRUE. The result is UNKNOWN otherwise.

- **AND** operator: The usual 'AND' logic applies in case both operands are either TRUE or FALSE. In case, one of the operand is UNKNOWN and the other is FALSE, the result is FALSE. The result is UNKNOWN otherwise.

- **NOT** operator: The usual 'NOT' logic applies in case the operand is either TRUE or FALSE. Otherwise, the result is UNKNOWN, i.e. NOT UNKNOWN = UNKNOWN).

- Ambiguity : The grammar is unambiguous but during implementation you may come up with some problems due to, for instance, lookahead. In such problematic cases, you need to rewrite some part of the grammar to get rid of the problems.

- Variables and Types: Every variable belongs to the only type of three-valued logic type, e.g. the set {*TRUE, FALSE, UNKNOWN*}. Every variable has to be declared but need not to be initialized in initialization section. At declaration, every variable is initialized with the value *UNKNOWN*. Use of undeclared variables in initialization section or main section is an error and this encounter terminates the execution.

- Case sensitivity: The language is case-sensitive like C and Java.
- Reserved Keywords: All the lexemes (e.g. TRUE, PROGRAM, OR etc) are reserved and can not be used as identifiers.

### An example program

The code given below is a valid program in the specified language and asks user the values of two variables and computes the XOR and XNOR of them. The results are also displayed on the screen.

```
PROGRAM xorxnor;
DECLARATION SECTION
-- Declare variables
P, Q, R;

INITIALIZATION SECTION
-- Initialize the variables

Q=TRUE;
P=UNKNOWN;
R=FALSE;

MAIN SECTION

-- Compute the XOR of P and Q
INPUT 'Enter P:' P;
INPUT 'Enter Q:' Q;
R = NOT (P AND Q OR NOT P AND NOT Q);
OUTPUT 'P XOR Q is:'  R;
OUTPUT 'P XNOR Q is:'  ((P AND Q) OR (NOT P AND NOT Q));
```

## Implementation

Use *Javacc* for implementing your version of interpreter. *Javacc* is both a lexical analyzer generator and a parser generator in Java language.

To ease the development process, it is strongly advised that you first write a language recognizer and then fill the actions associated to each construct in a second round to obtain the fully functional interpreter.

## References

- *Javacc* can be freely downloaded from its homepage of "**https://javacc.org/**". *Java version 1.6* or later (both *JDK* and *JVM*) is required with the latest version (version 6) of *Javacc*. The homepage is also a good source for tutorials and examples.