**PPL Exercise 3: Parser**

In this exercise, we will focus on the non – terminal node (parser rule). Parser rule represents the order of tokens of the programming languages. There are two ways to write a parser rule:

- BNF way (recursive way): this representation is easy for us when doing Assignment 2: AST Generation because we only use recursion for rules and for AST method (discuss later). General syntax for BNF way is:

S: aSb | c

- EBNF way (RegEx way): we will apply the operator *, +, ? in the parser rule. It is easy to write EBNF way but maybe difficult for Assignment 2. Bc carefull when choosing this way when defining parser rule

Follow the MT232's specification to finish these exercise (You can define supported lexer/parser rule to complete these exercise):

1. Complete the `program` rule which represents a MT232's program structure. Recall that MT232's program includes numerous declarations (variable and function) (choose appropriate way to implement this parser rules)

2. When defining the parser rules `program`, you defined two new rules for variable declarations and function declarations. Based on the instructions in MT232's specification, complete these two rules

3. Implement parser rule for statements in section 7 of MT232's specification

4. Implement parser rule for expressions in section 6 of MT232's specification. Note that these expressions must follow the table of precedence and associativity in subsection 6.9 (hint: using right recursion technique)

Optional:

5. A MT22's variable declaration starts with a comma-separated list of identifiers, then a colon, a type (which is one of these type: integer, string, boolean, float, auto). It can optionally follow by an equal sign and then, a comma-separated list of expressions (the number of expressions must be equal to the number of identifiers). The declaration ends with a semi colon. Using this assumption and the following supported rules, implement the `mt22_var_decl` rule to complete the exercise.

INT: 'int';

FLOAT: ‘float’;

STRING: ‘string’;

BOOLEAN: ‘boolean’;

AUTO: ‘auto’;

COMMA: ‘,’;

COLON: ‘:’;

SEMI: ‘;’

ID: [A-Za-z][A-Za-z0-9]*;

INT_LIT: [0] | [1-9][0-9]*;

expr: INT_LIT;

6. Implement the `expr` rule which accepts these operator: and, or, +, -, *, /, %, >, >=, <, <=, ==, != follows the following table (precedence is from the highest to the lowest):

| Meaning | Operators | Arity | Position | Associativity |
|---|---|---|---|---|
| Unary sign | +, - | Unary | Prefix | Right |
| Multiplication | *, /, % | Binary | Infix | Left |
| Addition | +, - | Binary | Infix | Left |
| Logical OR | or | Binary | Infix | Left |
| Logical AND | and | Binary | Infix | Left |
| Relational | >,>=,<,<=,==,!= | Binary | Infix | None |

The expressions in the bracket (‘()’) has a highest precedence.