# Parser assignment

## Description

Write the parser, the second phase of your compiler. The parser should use your previous scanner (or the one that we will provide) to read all tokens of an input program and check whether the program is grammatically correct. The specific requirements are:

- As in the scanner project, your parser should ignore (but copy) "meta-statements".
- The grammer definition is here. Regardless the choice of your implementation, you must convert the grammar into an LL(1) grammar and submit the modified grammar in writing.
- When submitting your grammar, put down the number of production rules and the number of distinct non-terminals used in your grammar. For example, A -> aA | epsilon should be counted as two production rules and one non-terminal.
- For each input program, your parser should report "pass" or "error" as the result of grammar analysis. It suffices to include only one of these two words in the output (stdout or stderr) of your program.
- Count the number of global and local variables (through <data decls>), functions (through <func list> but not function declarations), and statements (through <statements>) in correct programs. Output the result in the following format: variable *_num_variables* function *_num_functions* statement *_num_statements*. For example, for a correct program with 4 variables, 2 functions and 6 statements, your compiler should output at least the following:
    - Pass variable 4 function 2 statement 6

## Test programs

As in the Scanner project, a set of test programs can be downloaded from here.

## Syntax of the input language *

<program> --> <data decls> <func list>
<func list> --> **empty** | <func> <func list>
<func> --> <func decl> **semicolon** | <func decl> **left_brace** <data decls> <statements> **right_brace**
<func decl> --> <type name> **ID left_parenthesis** <parameter list> **right_parenthesis**
<type name> --> **int** | **void** | **binary** | **decimal**
<parameter list> --> **empty** | **void** | <non-empty list>
<non-empty list> --> <type name> **ID** | <non-empty list> **comma** <type name> **ID**
<data decls> --> **empty** | <type name> <id list> **semicolon** <data decls>
<id list> --> <id> | <id list> **comma** <id>
<id> --> **ID** | **ID left_bracket** <expression> **right_bracket**

<block statements> --> **left_brace** <statements> **right_brace**
<statements> --> **empty** | <statement> <statements>
<statement> --> <assignment> | <func call> | <if statement> | <while statement> | <return statement> | <break statement> | <continue statement> | **read left_parenthesis ID right_parenthesis semicolon** | **write left_parenthesis** <expression> **right_parenthesis semicolon** | **print left_parenthesis STRING right_parenthesis semicolon**
<assignment> --> <id> **equal_sign** <expression> **semicolon**
<func call> --> **ID left_parenthesis** <expr list> **right_parenthesis semicolon**
<expr list> --> **empty** | <non-empty expr list>
<non-empty expr list> --> <expression> | <non-empty expr list> **comma** <expression>

<if statement> --> **if left_parenthesis** <condition expression> **right_parenthesis** <block statements>
<condition expression> --> <condition> | <condition> <condition op> <condition>
<condition op> --> **double_and_sign** | **double_or_sign**
<condition> --> <expression> <comparison op> <expression>
<comparison op> --> **==** | **!=** | **>** | **>=** | **<** | **<=**

<while statement> --> **while left_parenthesis** <condition expression> **right_parenthesis** <block statements>
<return statement> --> **return** <expression> **semicolon** | **return semicolon**
<break statement> ---> **break semicolon**
**<**continue statement> ---> **continue semicolon**

<expression> --> <term> | <expression> <addop> <term>
<addop> --> **plus_sign** | **minus_sign**
<term> --> <factor> | <term> <mulop> <factor>
<mulop> --> **star_sign** | **forward_slash**
<factor> --> **ID** | **ID left_bracket** <expression> **right_bracket** | **ID left_parenthesis** <expr list> **right_parenthesis** | **NUMBER** | **minus_sign NUMBER** | **left_parenthesis** <expression> **right_parenthesis**