Alphabet:

---------- LEXIC ----------

a. Upper (A-Z) and lower case letters (a-z) of the English alphabet

b. Underline character '_';
c. Decimal digits (0-9);

Lexic:

a. Special symbols, representing:

- operators: + - * ** / % = < <= == != >= >
- separators: { } : ; space
- reserved words: if else while for

b. identifiers < 256 chars

- a sequence of letters and digits, such that the first character is a letter; the rule is:

identifier ::= letter | letter{letter|digit}

letter ::= 'A' | 'B' | . ..| 'Z'
nzDigit ::= '1' | '2' |...| '9'
digit ::= '0' | nzDigit

c.constants

constant ::= number | constString | boolean

---

number ::= int {'.' ['0'] unsignedInt}
int ::= '0' | ({'+'|'-'} nzInt)
unsignedInt ::= '0' | nzInt
nzInt ::= nzDigit {digit} {nzInt}

constString ::= "" string "" | """ string """
string ::= char {string}
char ::= letter | digit

boolean :: trueVal | falseVal
trueVal ::= 'True'
falseVal ::= 'False'

---------- TOKEN ----------

+
-
*
**

/
%
=
<
<=
==
!=
>=
>
{
}
(
)
;
space
newline
"
'
for
if
else
while

---------- SYNTAX ----------

program ::= stmtWrapper {program}

stmtWrapper ::= simpleStmt ';' | structStmt
| '{' (simpleStmt ';' | structStmt) '}'
simpleStmt ::= outStmt | assignStmt

outStmt ::= 'out(' {IDENTIFIER | CONSTANTS} ')' (* output, can also output nothing *)
assignStmt ::= IDENTIFIER '=' expression

expression ::= {(expression | value) OPERATOR} (expression | value)
| '(' {(expression | value) OPERATOR} (expression | value) ')'

value ::= (ioValue | arrayValue | airthmeticValue)
ioValue ::= 'in()' (* input *)
arrayValue ::= '[' {value} ']'

arithmeticExpression ::= {(arithmeticExpression | value) arithmeticRelation} (arithmeticExpression | value)
| '(' {(arithmeticExpression | value) arithmeticRelation} (arithmeticExpression | value) ')'
arithmeticValue ::= (castValue | IDENTIFIER | CONSTANTS)
arithmeticRelation ::= '+' | '-' | '*' | '**' | '/' | '%'

castValue ::= 'Number(' IDENTIFIER | CONSTSTRING ')'

structStmt ::= ifStmt | whileStmt | forStmt
ifStmt ::= 'if{' condition '}' stmtWrapper {'else if(' condition ')' stmtWrapper} ['else(' condition ')' stmtWrapper]
whileStmt ::= 'while{' condition '}' stmtWrapper
forStmt ::= 'for(' assignStmt ';' condition ';' assignStmt ')' stmtWrapper

condition ::= expression {(compRelation | boolRelation) expression}

compRelation ::= '<' | '<=' | '==' | '!=' | '>=' | '>'
boolRelation ::= '&&' | '||'