MINI-GO LANGUAGE GRAMMAR

El lenguaje Mini-GO es un subconjunto del lenguaje GOLang que conserva mucha de su sintaxis original aunque limitado no solo en contenido sintáctico si no también semántico. En este sentido, los ejemplos de código del lenguaje original que sean aplicables a la sintaxis aquí presentada, deberían ser compatibles, lo que allanaría el trabajo de escribir códigos fuente sintácticamente correctos para este lenguaje.

REGLA PRINCIPAL: "root"

root : 'package' IDENTIFIER ';' topDeclarationList

topDeclarationList : (variableDecl | typeDecl | funcDecl)*

variableDecl : 'var' (singleVarDecl ';' | '(' innerVarDecls ')' ';' | '(' ')' ';')

innerVarDecls : singleVarDecl ';' (singleVarDecl ';')*

singleVarDecl : identifierList declType '=' expressionList
| identifierList '=' expressionList
| singleVarDeclNoExps

singleVarDeclNoExps : identifierList declType

typeDecl : 'type' (singleTypeDecl ';' | '(' innerTypeDecls ')' ';' | '(' ')' ';')

innerTypeDecls : singleTypeDecl ';' (singleTypeDecl ';')*

singleTypeDecl : IDENTIFIER declType

funcDecl : funcFrontDecl block ';'

funcFrontDecl : 'func' IDENTIFIER '(' (funcArgDecls | epsilon) ')' (declType | epsilon)

funcArgDecls : singleVarDeclNoExps (',' singleVarDeclNoExps)*

declType : '(' declType ')'
| IDENTIFIER
| sliceDeclType
| arrayDeclType
| structDeclType

sliceDeclType : '[' ']' declType

arrayDeclType : '[' INTLITERAL ']' declType

structDeclType : 'struct' '{' (structMemDecls | epsilon) '}'

```
structMemDecls      : singleVarDeclNoExps ';' (singleVarDeclNoExps ';')*
identifierList      : IDENTIFIER (',' IDENTIFIER)*
expression          : primaryExpression
                    | expression operator expression
                    | operator expression

expressionList      : expression (',' expression)*
primaryExpression   : operand
                    | primaryExpression (selector | index | arguments )
                    | appendExpression
                    | lengthExpression
                    | capExpression
operand             : literal
                    | IDENTIFIER
                    | '(' expression ')'
literal             : INTLITERAL
                    | FLOATLITERAL
                    | RUNELITERAL
                    | RAWSTRINGLITERAL
                    | INTERPRETEDSTRINGLITERAL
index               : '[' expression ']'
arguments           : '(' expressionList | epsilon ')'
selector          : '.' IDENTIFIER
appendExpression    : 'append' '(' expression ',' expression ')'
lengthExpression    : 'len' '(' expression ')'
capExpression       : 'cap' '(' expression ')'
statementList       : statement*
block               : '{' statementList '}'
statement           : 'print' '(' expressionList | epsilon ')' ';'
```

```
                    | 'println' '(' expressionList | epsilon ')' ';'
                    | 'return' (expresión | epsilon) ';'
                    | 'break' ';'
                    | 'continue' ';'
                    | simpleStatement ';'
                    | block ';'
                    | switch ';'
                    | ifStatement ';'
                    | loop ';'
                    | typeDecl
                    | variableDecl
simpleStatement     : epsilon
                    | expression ('++' | '--' | epsilon)
                    | assignmentStatement
                    | expressionList ':=' expressionList
assignmentStatement : expressionList operator expressionList
                    |expression operator  expression

ifStatement         : 'if' expression block
                    | 'if' expression block 'else' ifStatement
                    | 'if' expression block 'else' block
                    | 'if' simpleStatement ';' expression block
                    | 'if' simpleStatement ';' expression block 'else' ifStatement
                    | 'if' simpleStatement ';' expression block 'else' block
loop                : 'for' block
                    | 'for' expression block
                    | 'for' simpleStatement ';' expression ';' simpleStatement block
                    | 'for' simpleStatement ';' ';' simpleStatement block
switch              : 'switch' simpleStatement ';' expression '{' expressionCaseClauseList '}'
```

| 'switch' expression '{' expressionCaseClauseList '}'

| 'switch' simpleStatement ';' '{' expressionCaseClauseList '}'

| 'switch' '{' expressionCaseClauseList '}'

expressionCaseClauseList : epsilon

| expressionCaseClause expressionCaseClauseList

expressionCaseClause : expressionSwitchCase ':' statementList

expressionSwitchCase : 'case' expressionList

| 'default'

operator : '*'

| '/' | '%' | '<<' | '>>' | '&' |'&^' |'+' |'-' |'|' |'^' |'==' |'!=' |'<' |'<=' |'>='
|'&&' |'||' | '!' |'=' |'+=' |'&=' |'-=' |'|=' |'*=' |'^=' |'<<=' |'>>=' |'&^=' |'%=' |'/='

INTLITERAL : Digit+ ;

FLOATLITERAL : Digit+ '.' Digit* ([eE] [+\-]? Digit+)? ;

RUNELITERAL : '\'' (UnicodeValue | EscapedChar) '\'' ;

RAWSTRINGLITERAL : '`' .*? '`' ; // Se permite cualquier cosa entre las comillas invertidas

INTERPRETEDSTRINGLITERAL : '"' .*? '"' ; // Se permite cualquier cosa entre las comillas dobles

fragment DIGIT : '0'..'9' ;

fragment UnicodeValue : '\\' 'u' HexDigit HexDigit HexDigit HexDigit ;

fragment HexDigit : [0-9a-fA-F] ;

fragment EscapedChar : '\\' [abfnrtv'"\\"];

ACLARACIONES

- La gramática en su mayoría está representada en EBNF y sin embargo pueden existir recursiones estilo BNF y posibilidades de factorización.
- Los tokens están representados entre comillas '*token*'
- El epsilon es un metasímbolo que denota que el elemento puede ser vacío. Este símbolo en cada generador de parser tiene su forma particular de ser expresado (revisar para ANTLR4)
- Los identificadores, números enteros y números flotantes (IDENTIFIER, INTLITERAL Y FLOATLITERAL) siguen el estándar de GoLang.
- Las literales carácter igualmente siguen el estándar de GoLang y están denotadas por RUNELITERAL
- Las cadenas de caracteres de igual forma siguen el estándar pero en dos formatos específicos que en GoLang se conocen como “raw string” (RAWSTRINGLITERAL) y “intepreted string” (INTERPRETEDSTRINGLITERAL)