Gramática:

```
grammar Language;

program: (declaration NL?)*;

declaration: varDeclaration | statement | slicesDeclaration | funcDeclaration |
structDeclaration | matrixDeclaration;

varDeclaration: 'var' ID TYPE ('=' expr)?
    | ID ':=' expr
    | ID (TYPE | ID);

funcDeclaration: 'func' ID '(' params? ')' TYPE? '{' declaration* '}'
   | 'func' '(' ID ID ')' ID '(' params? ')' TYPE? '{' declaration* '}';

params: ID TYPE (',' ID TYPE)*;

slicesDeclaration: ID ':=' '[]' TYPE '{' exprList? '}'
    | 'var' ID '[]' TYPE ;

matrixDeclaration: ID ':=' '[][]' TYPE '{' matrixRows '}'
    | 'var' ID '[][]' TYPE;

matrixRows: '{' exprList? '}' (',' '{' exprList? '}')* ','?;

structDeclaration: 'type' ID  'struct' '{' structBody* '}';

structBody: varDeclaration;

statement: expr                                          # ExprStmt
    | 'fmt.Println' '(' exprList ')'                     # PrintStmt
    | '{' declaration* '}'                               # BlockStmt
    | 'if' expr  statement ('else' statement)?           # IfStmt
    | 'switch' expr  '{' caseClauseStmt* '}'             # SwitchStmt
    | 'for' ID ',' ID ':=' 'range' ID statement          # ForRangeStmt
    | 'for' expr statement                               # ForStmt
    | 'for' forInit ';' expr? ';' expr? statement        # ForDeclStmt
    | 'break'                                            # BreakStmt
    | 'continue'                                         # ContinueStmt
    | 'return' expr?                                     # ReturnStmt;
```

```
forInit: varDeclaration | expr ';';

caseClauseStmt: 'case' expr ':' declaration*        # CaseClause
    | 'default' ':' declaration*                    # DefaultClause;
exprList: expr (',' expr)*;

expr:
    '(' expr ')'                                   # Parens
    | expr call+                                    # Callee
    | ID '[' expr ']'                               # Index
    | ID '[' expr']' '[' expr ']'                   # MatrixIndex
    | '[]' TYPE '{' exprList? '}'                   # Slice
    | 'slices' '.' 'Index' '('ID ',' expr ')'       # indexMethod
    | 'strings' '.' 'Join' '(' ID ',' expr ')'      # joinMethod
    | 'len' '(' expr ')'                            # lenMethod
    | 'append' '(' ID ',' expr ')'                  # appendMethod
    | 'strconv' '.' 'Atoi' '(' expr ')'             # atoiMethod
    | 'strconv' '.' 'ParseFloat' '(' expr ')'       # parseFloatMethod
    | 'reflect' '.' 'TypeOf' '(' ID ')'             # typeOfMethod
    | BOOL                                          # Bool
    | FLOAT                                         # Float
    | STRING                                        # String
    | INT                                           # Number
    | RUNE                                          # Rune
    | NIL                                           # Nil
    | '!' expr                                      # Not
    | '-' expr                                      # Negate
    | expr '%' expr                                 # Mod
    | expr op = ('*' | '/') expr                    # MulDiv
    | expr op = ('+' | '-') expr                    # AddSub
    | expr op = ('>' | '<' | '>=' | '<=') expr      # Relational
    | expr op = ('==' | '!=') expr                  # Equality
    | expr op = ('&&' | '||') expr                  # Logical
    | expr '=' expr                                 # Assign
    | ID '{' fields '}'                             # New
    | ID                                            # Identifier
    | ID '+=' expr                                  # AddAssign
    | ID '-=' expr                                  # SubAssign
    | ID '++'                                       # Inc
    | ID '--'                                       # Dec;
```

```
call: '(' args? ')' #FuncCall | '.' ID #Get;
args: expr (',' expr)*;

fields: fieldInit (',' fieldInit)* ','?;

fieldInit: ID ':' expr;

INT: [0-9]+;
BOOL: 'true' | 'false';
FLOAT: [0-9]+ '.' [0-9]+;
STRING: '"' (ESCAPE | ~["\\\r\n])* '"';
RUNE: '\'' (ESCAPE | ~['\\]) '\'';
NIL: 'nil';

fragment ESCAPE: '\\' (["\\/bfnrt] | 'u' HEX HEX HEX HEX);
fragment HEX: [0-9a-fA-F];

TYPE: 'int' | 'float64' | 'bool' | 'string' | 'rune';
ID: [a-zA-Z_][a-zA-Z_0-9]*;

WS: [ \t]+ -> skip;
NL: [\r\n]+ -> skip;

COMMENT: '//' ~[\r\n]* -> skip;
MULTILINE_COMMENT: '/*' .*? '*/' -> skip;
```