Go Language Grammar

newline = “\n”.

unicode_letter= “a” ... ”z”|”A”...”Z”.

unicode_char= /* all characters except newline*/

letter = unicode_letter | "_" .

decimal_digit = "0" ... "9" .

identifier = letter { letter | unicode_digit } . **Except KeyWords**

keywords = break | default | func | case | struct | else | package | switch | const | if | range | type | continue | for | import | return | var

binary_op = "||" | "&&" | rel_op | add_op | mul_op .

rel_op = "==" | "!=" | "<" | "<=" | ">" | ">=" .

add_op = "+" | "-" | "|" .

mul_op = "*" | "/" | "%" | "<<" | ">>" | "&" .

unary_op = "+" | "-" | "!" | "*" | "&" | "<-" .

int_lit = ("1" ... "9") { decimal_digit } |”0”.

unicode_value= unicode_char | escaped_char .

escaped_char = `\` ("a" | "b" | "f" | "n" | "r" | "t" | "v" | `\` | "'" | `"`) .

string_lit = raw_string_lit | interpreted_string_lit .

raw_string_lit = "`" { unicode_char | newline } "`" .

interpreted_string_lit = `"` { unicode_value } `"` .

Type = TypeName | TypeLit | "(" Type ")" .

TypeName = identifier .

TypeLit = ArrayType | StructType | FunctionType

ArrayType = "[" ArrayLength "]" ElementType .

ArrayLength = Expression .

ElementType = Type .

SliceType = "[" "]" ElementType .

StructType = "struct" "{" { FieldDecl ";" } "}" .

FieldDecl = (IdentifierList Type | AnonymousField) [Tag] .

AnonymousField = ["*"] TypeName .

Tag = string_lit .

FunctionType = "func" Signature .

```
Signature       = Parameters [ Result ] .
Result          = Parameters | Type .
Parameters      = "(" [ ParameterList [ "," ] ] ")" .
ParameterList = ParameterDecl { "," ParameterDecl } .
ParameterDecl = [ IdentifierList ] [ "..." ] Type .
MethodSpec      = MethodName Signature | InterfaceTypeName .
MethodName = identifier .
Block           = "{" StatementList "}" .
StatementList = { Statement ";" } .
Declaration     = ConstDecl | TypeDecl | VarDecl .
TopLevelDecl    = Declaration | FunctionDecl | MethodDecl .
ConstDecl       = "const" ( ConstSpec | "(" { ConstSpec ";" } ")" ) .
ConstSpec       = IdentifierList [ [ Type ] "=" ExpressionList ] .
IdentifierList  = identifier { "," identifier } .
ExpressionList = Expression { "," Expression } .
Expression = UnaryExpr | Expression binary_op Expression .
UnaryExpr  = PrimaryExpr | unary_op UnaryExpr
TypeDecl        = "type" ( TypeSpec | "(" { TypeSpec ";" } ")" ) .
TypeSpec        = identifier Type .
VarDecl         = "var" ( VarSpec | "(" { VarSpec ";" } ")" ) .
VarSpec         = IdentifierList ( Type [ "=" ExpressionList ] | "=" ExpressionList ) .
ShortVarDecl    = IdentifierList ":=" ExpressionList .
FunctionDecl    = "func" FunctionName ( Function | Signature ) .
FunctionName= identifier .
Function        = Signature FunctionBody .
FunctionBody  = Block .
MethodDecl      = "func" Receiver MethodName ( Function | Signature ) .
Receiver        = Parameters .
Operand         = Literal | OperandName | MethodExpr | "(" Expression ")" .
Literal         = BasicLit | FunctionLit .
BasicLit        = int_lit | string_lit .
OperandName = identifier | QualifiedIdent.
```

```
QualifiedIdent = PackageName "." identifier .
FunctionLit     = "func" Function .
PrimaryExpr     = Operand | PrimaryExpr Selector | PrimaryExpr Index | PrimaryExpr Slice
                 |  PrimaryExpr Arguments .
Selector        = "." identifier .
Index           = "[" Expression "]" .
Slice           = "[" [ Expression ] ":" [ Expression ] "]" | "[" [ Expression ] ":" Expression ":" Expression "]" .
Arguments       = "(" [ ( ExpressionList | Type [ "," ExpressionList ] ) [ "..." ] [ "," ] ] ")" .
MethodExpr      = ReceiverType "." MethodName .
ReceiverType    = TypeName | "(" "*" TypeName ")" | "(" ReceiverType ")" .
Statement       = Declaration | SimpleStmt | ReturnStmt | BreakStmt  | Block | IfStmt |
                  SwitchStmt | ForStmt .
SimpleStmt      = ExpressionStmt | IncDecStmt | Assignment | ShortVarDecl .
ExpressionStmt          = Expression .
IncDecStmt      = Expression ( "++" | "--" ) .
Assignment      = ExpressionList assign_op ExpressionList .
assign_op       = [ add_op | mul_op ] "=" .
IfStmt          = "if" [ SimpleStmt ";" ] Expression Block [ "else" ( IfStmt | Block ) ] .
SwitchStmt      = ExprSwitchStmt .
ExprSwitchStmt = "switch" [ SimpleStmt ";" ] [ Expression ] "{" { ExprCaseClause } "}" .
ExprCaseClause = ExprSwitchCase ":" StatementList .
ExprSwitchCase = "case" ExpressionList | "default" .
ForStmt         = "for" [ Condition | ForClause ] Block .
Condition       = Expression .
ForClause       = [ InitStmt ] ";" [ Condition ] ";" [ PostStmt ] .
InitStmt        = SimpleStmt .
PostStmt        = SimpleStmt .
ReturnStmt      = "return" [ ExpressionList ] .
BreakStmt       = "break" .
SourceFile     = PackageClause ";" { ImportDecl ";" } { TopLevelDecl ";" } .
PackageClause= "package" PackageName .
PackageName = identifier .
```

```
ImportDecl     = "import" ( ImportSpec | "(" { ImportSpec ";" } ")" ) .
ImportSpec     = [ "." | PackageName ] ImportPath .
ImportPath     = string_lit .
```

Read carefully:

1- This grammar is a fragment of the Go language specifications that you can find here:

https://golang.org/ref/spec#Selectors

2- According to the Go language specifications, these are the following conventions used for stating the grammar:

- Anything written between “ ” is a literal.
- [] denoates 0 or 1.
- {} denoates 0 or more .
- () is used for grouping one or more expression together.

3- All underlined grammar rules are also part of the language literals.