Requirement Analysis Document

# **Project name:** Parser for Go programming

# **Team name:** Go Stop

# **Team members:** 구교준(20205006), 김두현(20205018)

# **Scope of the project**

We implemented Go programming language parser using lex and yacc. We implemented 12 major functions, which are package main, import header, main function, variable instruction, print instruction, scan instruction, for loop instruction, if instruction, switch instruction, function declaration, usage of function, calculation formula.

In addition, we implemented error handling in lex and yacc for errors that occur in the steps of tokenization and syntatic analysis, respectively. First, we used '\n' token in lex and yyerror function in yacc to print out which line is causing the error and what kind of error it is. Also, print out the appropriate data type (int8 to int64, float...) for the input number, and if a large value that could not be stored in the range was entered and the variable formation was not matched (first letter cannot be number, special character cannot be in variable) we performed lexical error handling. We implemented to ignore space, tap, and newline on the lex, and when characters other than the token we specified are entered, the unexpected character is being printed out. yacc implemented to output what syntax was found when the syntax passed through parser, and we used panic mode recovery to print out where the syntax error occurred and usage when the instruction used in wrong formation

# **Representative diagram to explain your project**

**Tools used (lex, and yacc for each functions or modules)**

**ᆞ**lex

In lex, tokens are assigned, and strings corresponding to a given string are converted into tokens, returned, and delivered to yacc. The correct definition of VAR (identifier) starts with an alphabet and uses words using alphabets and numbers as VARs, and when the first value of the word is not alphabets and special characters are used, an error is printed on the interface in lex. In the case of numbers, int, float, and complex are defined as tokens, and when a number is entered, the data type of each number is printed, and in the case of integer, it is divided into int8, int16, int32, int0. In addition, when comment is found, if input is //stmt form, ignore the characters up to \n and treated as comment, and if input is /\* stmt \*/ form, ignore the characters and print that the comment is found. Also, ugnore the space and \n \t. For an undefined token output that it is not a correct token.’

**ᆞ**yacc

Formation

start -> package stmt main\_function stmt

stmt -> import\_header

| variable\_stmt

| print\_stmt

| scan\_stmt

| for\_stmt

| if\_stmt

| switch\_stmt

| func\_declaration

| func\_usage

Our parse tree changes to start\_stmt -> package stmt main\_function stmt, and stmt can be expanded to several stmt, and functions can be used through each stmt.

- package main

package -> PACKAGEMAIN

The "package main" is treated as a token, and the package main must exist alone at the beginning.

- main\_function

main\_function -> FUNCMAIN() { stmt }

The stmt of the brackets indicates that a statement may exist inside the main function.

- import\_header

import\_header -> IMPORT STRING

| IMPORT ( strings )

It is a statement that calls for a header, and the header takes the form of a string and is used in parentheses when multiple headers come.

- variable\_stmt

variable\_stmt -> VARTOKEN vars = exprs

| VARTOKEN vardec = exprs

| VARTOKEN vardec

| vars := exprs

| vars = exprs

vardec -> vars type vardec

| vars type

As a syntax for declaring and assigning variables, vars is a list of variables, and exprs is a list of operations, variables, and values.

- print\_stmt and scan\_stmt

print\_stmt -> PRINTLN ( exprs )

| PRINT ( exprs )

| PRINTF ( STRING , exprs )

scan\_stmt -> SCANLN ( avars )

| SCAN ( avars )

| SCAN ( STRING, avars )

As the syntax in charge of input/output, operations, variables, and values can come inside the print, and in the case of scan, a list of addresses of variables is used.

- for\_stmt

for\_stmt -> FOR (dstmt; expr; expr) { stmt }

| FOR dstmt; expr; expr { stmt }

| FOR ( expr ) { stmt }

| FOR expr { stmt }

It is a syntax used for loop statements. dstmt refers to the initial statement, expr refers to conditional statements and operations, and dstmt and expr operate normally even if they become lambda.

- if\_stmt

if\_stmt -> IF dstmt; expr { stmt } el

| IF expr { stmt } el

el -> ELSEIF expr { stmt } el

| ELSE { stmt }

The syntax used for conditional statements. dstmt means short variable stmt and expr means conditional statement.

- switch\_stmt

switch\_stmt -> SWITCH dstmt; expr { casestmt }

| SWITCH expr { casestmt }

casestmt -> CASE expr : stmt casestmt

| DEFAULT : stmt

It's a switch syntax. The expr when switch\_stmt is converted can exist as lambda as a comparison target, and the value at this time is defined as true and operates normally.

- func\_declaration

func\_declaration -> FUNC VAR ( vardec ) dfr

| FUNC VAR ( ) dfr

dfr -> { stmt return }

| type { stmt return expr }

| ( types ) { stmt return exprs }

| ( vardec ) { stmt return }

| ( vardec ) { stmt return exps }

Used to declare a function. Parameters can be assigned in the same format as x int, y, and z char, and the return type varies depending on the return\_type.

- func\_usage

func\_usage -> VAR.VAR ( exprs )

Syntax used in the use of functions.

- expr

expr -> expr OR expr

| expr AND expr

| expr BITOR expr

| expr BITNOT expr

| expr BITAND expr

| expr EQ expr

| expr NE expr

| expr LE expr

| expr GE expr

| expr LT expr

| expr GT expr

| expr SHIFTL expr

| expr SHIFTR expr

| expr PLUS expr

| expr MINUS expr

| expr MULTIPLY expr

| expr DIVIDE expr

| expr MOD expr

| PP expr

| MM expr

| MINUS expr

| NOT expr

| LP expr RP

| expr PP

| expr MM

| VAR

| NUM

| COMPLEX

| STRING

| BOOL

| CHAR

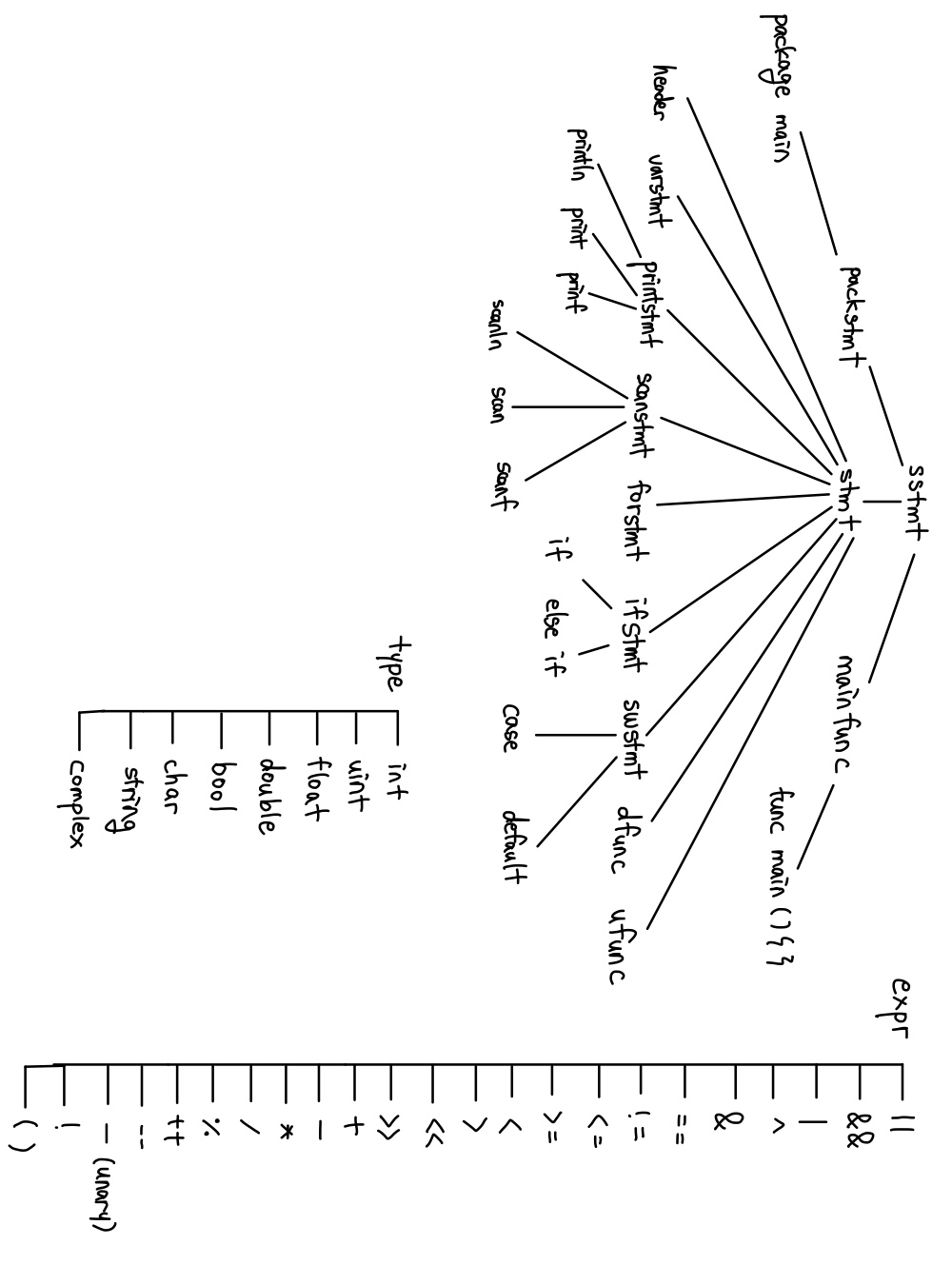
| ufunc

It is the part that is in charge of all the operations, variables, and values of the parser. Although it is not directly converted from stmt and used, it is a very important part of each function.

yacc user interface

When the syntax passes through the parser, output to the user that the corresponding syntax has been processed at the start or end of each yacc expression. If the TOKEN stream received from lex is not correct for yacc grammar, it shows where the error occurred with the line number, and output the correct usage of the grammar to the user.

**Diagram**



**Input**

**ᆞ**Package main and main function

In the case of package main and main function, it is a structure that must exist in Go programming language. Therefore, input is when the package main and main functions are correctly present and missing.

**ᆞ**Import header

When adding a header file, the import is followed by a header file in the form of one or more strings. At this time, if there is only one header file, it can be entered immediately after import, and if there are more than two header files, several header files can be added by dividing them into line breaks in parentheses. Therefore, we made three kinds of inputs. When the input is entered correctly, when the string does not come to the header file, and when the header file does not exist after import.

**ᆞ** Variable statement

There is a basic method of using VARTOKEN called var for variable allocation. The form var x, y int, and the order of connecting the variable and the data type takes the form of the data type after the variable name. In addition, it can be defined using = and subsequent allocation values without the existence of a data type, and the same can be done through := without VARTOKEN. Inputs such as a = 1 are also possible to enter new values in already defined variables. Therefore, we used the allocation value is not in the correct format along with this type of input, the variable name and data type are used incorrectly, and the symbol (=, :=) is not used correctly, the case where VARTOKEN and := are used simultaneously.

**ᆞ** Print and Scan instruction

In the case of fmt.Println and fmt.Print, variables, calculations, and values are sequentially located in the following brackets, and in the case of fmt.Printf, a string comes at the first of the brackets, and values corresponding to the subsequent format specifiers are continuously located.

In the case of Scan, it is similar to print, but in the case of the input value in parentheses, the address of the variable is used as the input. Therefore, in the case of input, in addition to the form in which these commands are normally used, the case where the incorrect token is contained in parentheses and the case where the parentheses are not normally closed is used as input.

**ᆞ** For instruction

In the case of for statement, it takes the form of for (a:=1; a<2; 1++) { stmt }. Even if there is no parentheses, and even if the post, middle, and end statements of the for statement are not present, it operates same. In addition, the for statement is used as a while of other languages, and in this case, it takes the form of for a<10 { stmt }, and if there is no conditional statement, such as for { stmt }, it becomes an infinite loop and operates normally. Therefore, with these types of inputs the part inside the parentheses is not norma, and the stmt after the for statement is not the correct form is added and used as input.

**ᆞ** If instruction

In the case of if statement, it takes the form of if (a <10) { stmt } and operates the same regardless of whether it is in parentheses. It is also possible for short variable statements to come with semicolons before parentheses. After that, else if statements and else statements may come, and in else if statements, conditional statement exist the same as if, several else statements can be located continuously. In the case of an else statement, it can exist only at the end. It is also possible that else if and else statement do not exist, respectively. Therefore, we use the inputs that the semicolon does not come after a short variable statement, the comparative syntax is incorrect or does not exist, the stmt inside the if statement is not in the correct form, and the else appears in a non-last place.

**ᆞ** Switch instruction

In the case of switch instruction, it takes the form of switch a { casestmt }, and similar to if, short variable statements can come with semicolons. In the case of an incoming value after the switch, variables, values, etc. may come, and if nothing exists, it means true. Afterwards, case statements and default statements may come in the case statements, case value: stmt may come in the form of case statements. Several case statements may come in succession, and default may be in the form of default statements and may only be located at the end of case statements. It is also possible that the case and the default statement do not exist, respectively. Therefore, we use the inputs that the semicolon does not come after the short variable statement, the stmt inside casestmt is not in the correct form, a non-normal value comes in the position of the value, a default comes in the non-last position.

**ᆞ** Function instruction

In the case of function declaration, it has the form of func name (parameters) (return\_type) { stmt return\_stmt}. In the case of parameters, it has the form of x int, y float64, similar to the variable statement. After that, return\_stmt is determined according to return\_type, and return\_type takes the form of a continuous data type or a form like a variable statement. At this time, if one data type is used as return\_type, parentheses do not need to be used. If there is no return\_type, return\_stmt does not have to exist, and if return\_type indicates multiple datatypes, return\_stmt must specify a value after return, and if return value is specified in return\_type, such as x int, it is also possible to use only return.

In the case of function usage, it is used in a form like print statement. Therefore, we use the inputs that the parameters form is not correct along with the input satisfying these conditions, the statement of the function is not correct, and the relationship between return\_type and return\_stmt is not correct, return\_stmt is not in the correct form.

**ᆞ** Calculation formula

In the case of expression, it is indirectly used in the input of the above examples.

**Output (proper error message)**

While syntax passes through a parser, we print what instruction the syntax is, and if a number is entered, prints what type of data the corresponding number has. In addition, if comments are entered, prints the location of that comments. And if characters are not the correct form of TOKEN, print “unexpected error” message. If an error occurs, prints the line number that error occurred, where the problem occurred in which syntax, and what is the correct usage.

# Limitations of your project

Go language uses ; and \n as line separators. However, we blank \n in our lex and yacc. Since line separator was used with lambda and ;, error handling and yacc grammar were produced with a lot of ambiguity, and there were cases where it was necessary to perform additional parts that were not supposed to be executed in the original Go programming language.

When creating an example sentence where an error occurs in the part of the example, there was a case where ; had to be used for the same reason as above. In addition, in the process of creating an sample input in which errors occur continuously, there were cases where errors were incompletely processed when errors occurred consecutively, because we did not performed errorok()

Since we did not consider semantic analysis, many cases of simplifying grammar occurred. As a result, there was a case where the grammar that caused errors worked normally in the parser we created. ex) var a, b int = 1, 2, 3

# Sample input and output

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명

<Normal code example input/output>

텍스트이(가) 표시된 사진

자동 생성된 설명

<Error code example input>

테이블이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

<Error code example output>

We will show additional code example in project demo time.