# Parser for the C Language


National Institute of Technology Karnataka Surathkal

**Date:** 16/09/2020

**Submitted To:** Dr. P. Santhi Thilagam

**Submitted By:**
1. Pranav Vigneshwar Kumar 181CO239
2. Ankush Chandrashekar 181CO206
3. Akshat Nambiar 181CO204
4. Mohammed Rushad 181CO232

# Abstract

A compiler is a special program that processes statements written in a particular programming language (high-level language) and turns them into machine language (low-level language) that a computer's processors use. Apart from this, the compiler is also responsible for detecting and reporting any errors in the source program during the translation process.

The file used for writing code in a specific language contains what are called the source statements. The programmer then runs the appropriate language compiler, specifying the name of the file that contains the source statements. When executing, the compiler first parses all of the language statements syntactically one after the other and then, in one or more successive stages, builds the output code, making sure that statements that refer to other statements are referred to correctly in the final code.

This report specifies the details related to the second stage of the compiler, the parsing stage. We have developed a parser for the C programming language using the lex and yacc tools. The parser makes use of the tokens outputted by the lexer developed in the previous stage to parse the C input file. The lexical analyzer can detect only lexical errors like unmatched comments etc. but cannot detect syntactical errors like missing semi-colon etc. These syntactical errors are identified by the parser i.e. the syntax analysis phase is done by the parser.

# Contents Page No



## List of Figures and Tables:

# Introduction

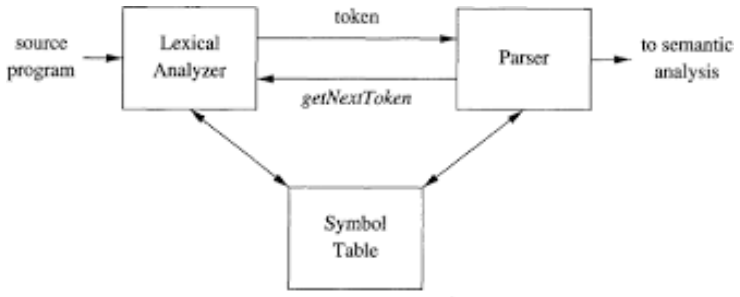
## Syntax Analysis

Syntax analysis is the second phase of the compiler design process which follows the lexical analysis phase. The parser takes as input the stream of tokens we get from the lexical analysis stage. It analyses the syntactical structure of the given input i.e. it verifies that a string of token names can be generated by the grammar of the source language. It checks if the given input is in the correct syntax of the programming language in which the input which has been written with the help of a **Parse Tree or Syntax Tree.**

The Parse Tree is developed with the help of pre-defined grammar of the language. The syntax analyzer also checks whether a given program fulfills the rules implied by a context-free grammar. If it satisfies, the parser then creates the parse tree of that source program.

In the case of an invalid grammar, we expect the parser to report the appropriate syntax errors in a neat and intelligible manner. The errors identified and reported by the parser include:

- Unbalanced Parenthesis
- Missing Semi-colons
- Errors in Structure
- Missing Operators
- Misspelt Keywords



# Parsing Techniques

Parsing techniques, in general, can be divided into two different groups.

- Top Down Parsing
- Bottom Up Parsing

**Top Down Parsing** can further be divided into **predictive parsing** and **recursive descent parsing**. **Predictive parse** can predict which production should be used to replace the specific input string. The predictive parser uses look-ahead point, which points towards next input symbols. Backtracking is not an issue with this parsing technique. It is known as LL(1) Parser. The **recursive descent parsing** technique recursively parses the input to make a parse tree. It consists of several small functions, one for each nonterminal in the grammar.

In the **Bottom Up Parsing** technique, the construction of the parse tree starts with the leaf nodes, and then it processes towards its root. It is also called as shift-reduce parsing. This type of parsing is created with the help of using some software tools.

# Yacc Script

Yacc stands for Yet Another Compiler-Compiler. Yacc is essentially a parser generator. Yacc provides a general tool for imposing structure on the input to a computer program. The Yacc user prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. A function is then generated by Yacc to control the input process. This function is called the parser which calls the lexical analyzer to get a stream of tokens from the input.

Based on the input structure rules, called grammar rules, the tokens are organized. When one of these rules has been recognized, then user code supplied for this rule, an action, is invoked. Actions have the ability to return values and make use of the values of other actions.

Yacc is written in portable C. The class of specifications accepted is a very general one, LALR(1) grammars with disambiguating rules.

The structure of our yacc script is divided into three sections, separated by lines that contain only two percent signs, as follows:

***DECLARATIONS***

***%%***

***RULES***

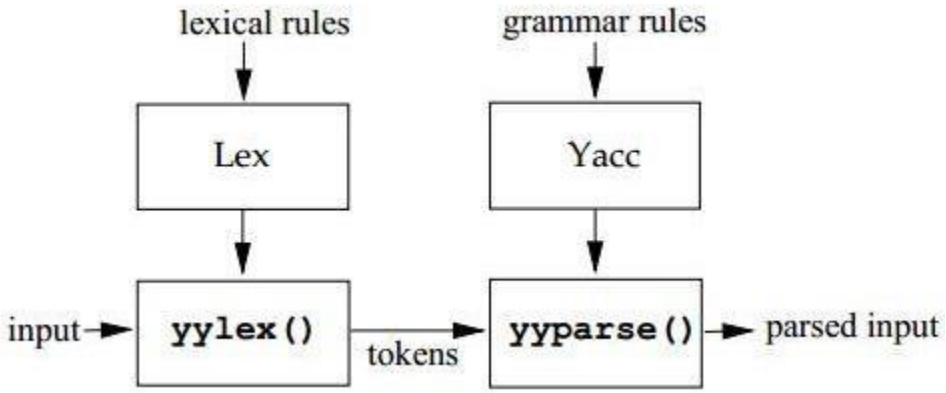
***%%***

***AUXILIARY FUNCTIONS***

The **Declarations Section** defines macros and imports header files written in C. It is also possible to write any C code here, which will be copied directly into the generated source file. We also define all parameters related to the parser here, specifications like using leftmost derivations or rightmost derivations, precedence, left and right associativity are declared here, data types and tokens which will be used by the lexical analyzer are also declared at this stage.

The **Rules Section** contains the entire grammar which is used for deciding if the input text is legally correct according to the specifications of the language. Yacc uses these rules for reducing the token stream received from the lexical analysis stage. All rules are linked to each other from the start state.

Yacc generates C code for the rules specified in the Rules section and places this code into a single function called yyparse(). The **Auxiliary Functions Section** contains C statements and functions that are copied directly to the generated source file. These statements usually contain code called by the different rules. This section essentially allows the programmer to add to the generated source code.



# C Program

The parser takes C source files as input for parsing. The input file is specified in the auxiliary functions section of the yacc script.
The workflow for testing the parser is as follows:

1. Compile the yacc script using the yacc tool
   *$ yacc -d parser.y*
2. Compile the flex script using the flex tool
   *$ lex lexer.l*
3. The first two steps generate lex.yy.c, y.tab.c, and y.tab.h. The header file is included in lexer.l file. Then, lex.yy.c and y.tab.c are compiled together.
   *$ gcc lex.yy.c y.tab.c*
4. Run the generated executable file
   *$ ./a.out*

# First and Follow Sets

## First Sets

First( begin_parse ) : { ε, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT }

First( declarations ) : {ε, INT, CHAR, FLOAT, DOUBLE, LONG,SHORT, UNSIGNED, SIGNED, VOID, STRUCT}

First( declaration ) : {INT, CHAR, FLOAT, DOUBLE, LONG, SHORT,UNSIGNED, SIGNED, VOID, STRUCT}

First( structure_dec ) : {STRUCT}

First( structure_content ) : {ε, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT}

First( variable_dec ) : { INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT }

First( structure_initialize ) : { STRUCT }

First( variables ) : { identifier }

First( multiple_variables ) : { , , ε}

First( function_dec ) : { INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID}

First( function_datatype ) : { INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID}

First( function_parameters ) : { ), ε, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID}

First( parameters ) : { ε, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID }

First( all_parameter_identifiers ) : { ) }

First( multiple_parameters ) : { , , ε }

First( parameter_identifier ) : { identifier }

First( extended_parameter ) : { [, ε }

First( identifier_name ) : { identifier}

First( extended_identifier ) : { =, [, ε }

First( array_identifier ) : { [, ε }

First( array_dims ) : { integer_constant, ] }

First( initilization ) : { ε, = }

First( string_initilization ) : { = }

First( array_initialization) : { = }

First( array_values ): { integer_constant }

First( multiple_array_values ) : { , , ε }

First( datatype ) : { INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID }

First( unsigned_grammar ) : { INT, LONG, SHORT, ε }

First( signed_grammar ) : { INT, LONG, SHORT, ε }

First( long_grammar ) : { INT, ε }

First( short_grammar ) : { INT, ε }

First( statement ) : { ;, identifier, NOT, (, integer_constant, string_constant, float_constant, character_constant, IF, RETURN, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, {, BREAK, WHILE,FOR, DO }

First( multiple_statement ) : { { }

First( statements ) : { ε, ;, identifier, NOT, (, integer_constant, string_constant, float_constant, character_constant, IF, RETURN, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, {, BREAK, WHILE,FOR, DO }

First( expression_statment ) : { ;, identifier, NOT, (, integer_constant, string_constant, float_constant, character_constant }

First( conditional_statements ) : { IF }

First( extended_conditional_statements ) : { ELSE, ε }

First( iterative_statements ) : { WHILE, FOR, DO }

First( for_initialization ) : { ;, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, identifier, NOT, (, integer_constant, string_constant, float_constant, character_constant }

First( return_statement ) : { RETURN }

First( return_suffix ) : { ;, identifier, NOT, (, integer_constant, string_constant, float_constant, character_constant }

First( break_statement ) : { BREAK }

First( expression ) : { identifier, NOT, (, integer_constant, string_constant, float_constant, character_constant }

First( expressions ) : { =, ADD_EQUAL, SUBTRACT_EQUAL, MULTIPLY_EQUAL, DIVIDE_EQUAL, MOD_EQUAL, INCREMENT, DECREMENT }

First( simple_expression ) : { NOT, (, identifier, integer_constant, string_constant, float_constant, character_constant }

First( simple_expression_breakup ) : { OR_OR, ε }

First( and_expression ) : { NOT, (, identifier, integer_constant, string_constant, float_constant, character_constant }

First( and_expression_breakup ) : { AND_AND, ε }

First( unary_relation_expression ) : { NOT, (, identifier, integer_constant, string_constant, float_constant, character_constant }

First( regular_expression ) : { (, identifier, integer_constant, string_constant, float_constant, character_constant }

First( regular_expression_breakup ) : { ε, GREAT_EQUAL, LESS_EQUAL, GREAT, LESS, EQUAL, NOT_EQUAL }

First( relational_operators ) : { GREAT_EQUAL, LESS_EQUAL, GREAT, LESS, EQUAL, NOT_EQUAL }

First( sum_expression ) : { (, identifier, integer_constant, string_constant, float_constant, character_constant }

First( sum_expression' ) : { ε, +, - }

First( sum_operators ) : { +, - }

First( term ) : { (, identifier, integer_constant, string_constant, float_constant, character_constant }

First( term' ) : { ε, *, /, % }

First( multiply_operators ) : { *, /, % }

First( factor ) : { (, identifier, integer_constant, string_constant, float_constant, character_constant }

First( iden ) : { identifier }

First( iden’ ) : { ε, [, . }

First( extended_iden ) : { [, . }

First( func ) : { (, identifier, integer_constant, string_constant, float_constant, character_constant }

First( func_call ):{ identifier }

First( arguments ):{ ε, identifier, NOT, (, integer_constant,string_constant, float_constant, character_constant }

First( arguments_list ):{ identifier, NOT, (, integer_constant,string_constant, float_constant, character_constant }

First( extended_arguments ):{ ,, ε }

First( constant ):{ integer_constant, string_constant, float_constant, character_constant }

# Follow Sets

Follow( begin_parse ) : { $ }

Follow( declarations ) : { $ }

Follow( declaration ) : { INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, $}

Follow( structure_dec ) : { INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, $}

Follow( structure_content ) : { }}

Follow( variable_dec ) : { INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, NOT, (, identifier, integer_constant, string_constant, float_constant, character_constant, WHILE, ELSE, ;, IF, RETURN, {,BREAK, FOR, DO, }, $}

Follow( structure_initialize ) : { INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, NOT, (, identifier, integer_constant, string_constant, float_constant, character_constant, WHILE, ELSE, ;, IF, RETURN, {,BREAK, FOR, DO, }, $ }

Follow( variables ) : { ;, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, NOT, (, identifier, integer_constant, string_constant, float_constant, character_constant, WHILE, ELSE, IF, RETURN, {,BREAK, FOR, DO, }, $ }

Follow( multiple_variables ) : { ;, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, NOT, (, identifier, integer_constant, string_constant, float_constant, character_constant, WHILE, ELSE, IF, RETURN, {,BREAK, FOR, DO, }, $}

Follow( function_dec ) : { INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, $}

Follow( function_datatype ) : { ), INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID}

Follow( function_parameters ):{ INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, $}

Follow( parameters ) : { ) }

Follow( all_parameter_identifiers ) : { ) }

Follow( multiple_parameters ) : { )}

Follow( parameter_identifier ) : { ,, ) }

Follow( extended_parameter ): { ,, ) }

Follow( identifier_name ) : { ,, ;, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, NOT, (, identifier, integer_constant, string_constant, float_constant, character_constant, WHILE, ELSE, IF, RETURN, {,BREAK, FOR, DO, }, $ }

Follow( extended_identifier ) : { ,, ;, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, NOT, (, identifier, integer_constant, string_constant, float_constant, character_constant, WHILE, ELSE, IF, RETURN, {,BREAK, FOR, DO, }, $}

Follow( array_identifier ) : { ,, ;, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, NOT, (, identifier, integer_constant, string_constant, float_constant, character_constant, WHILE, ELSE, IF, RETURN, {,BREAK, FOR, DO, }, $ }

Follow( array_dims ) : { ,, ;, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, NOT, (, identifier, integer_constant, string_constant, float_constant, character_constant, WHILE, ELSE, IF, RETURN, {,BREAK, FOR, DO, }, $ }

Follow( initilization ) : { ,, ;, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, NOT, (, identifier, integer_constant, string_constant, float_constant, character_constant, WHILE, ELSE, IF, RETURN, {,BREAK, FOR, DO, }, $}

Follow( string_initilization ) : { ,, ;, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, NOT, (, identifier, integer_constant,

string_constant, float_constant, character_constant, WHILE, ELSE, IF, RETURN, {,BREAK, FOR, DO, }, $ }

Follow( array_initialization) : { ,, ;, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, NOT, (, identifier, integer_constant, string_constant, float_constant, character_constant, WHILE, ELSE, IF, RETURN, {,BREAK, FOR, DO, }, $ }

Follow( array_values ) : { } }

Follow( multiple_array_values ) : { } }

Follow( datatype ) : { identifier }

Follow( unsigned_grammar ) : { identifier }

Follow( signed_grammar ) : { identifier }

Follow( long_grammar ) : { identifier }

Follow( short_grammar ) : { identifier }

Follow( statement ) : { WHILE, ELSE, ;, identifier, NOT, (, integer_constant, string_constant, float_constant, character_constant, IF, RETURN, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, {,BREAK, FOR, DO, }, $ }

Follow( multiple_statement ) : { WHILE, ELSE, ;, identifier, NOT, (, integer_constant, string_constant, float_constant, character_constant, IF, RETURN, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, {,BREAK, FOR, DO, }, $ }

Follow( statements ) : { } }

Follow( expression_statment ) : { WHILE, ELSE, ;, identifier, NOT, (, integer_constant, string_constant, float_constant, character_constant, IF, RETURN, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, {,BREAK, FOR, DO, }, $ }

Follow( conditional_statements ) : { WHILE, ELSE, ;, identifier, NOT, (, integer_constant, string_constant, float_constant, character_constant, IF,

RETURN, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, {,BREAK, FOR, DO, }, $ }

Follow( extended_conditional_statements ) : { WHILE, ELSE, ;, identifier, NOT, (, integer_constant, string_constant, float_constant, character_constant, IF, RETURN, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, {,BREAK, FOR, DO, }, $ }

Follow( iterative_statements ) : { WHILE, ELSE, ;, identifier, NOT, (, integer_constant, string_constant, float_constant, character_constant, IF, RETURN, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, {,BREAK, FOR, DO, }, $ }

Follow( for_initialization ) : { NOT, (, identifier, integer_constant, string_constant, float_constant, character_constant }

Follow( return_statement ) : { WHILE, ELSE, ;, identifier, NOT, (, integer_constant, string_constant, float_constant, character_constant, IF, RETURN, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, {,BREAK, FOR, DO, }, $ }

Follow( return_suffix ):{ WHILE, ELSE, ;, identifier, NOT, (, integer_constant, string_constant, float_constant, character_constant, IF, RETURN, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, {,BREAK, FOR, DO, }, $ }

Follow( break_statement ) : { WHILE, ELSE, ;, identifier, NOT, (, integer_constant, string_constant, float_constant, character_constant, IF, RETURN, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, {,BREAK, FOR, DO, }, $ }

Follow( expression ) : { ,, ), ], ;, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, NOT, (, identifier, integer_constant, string_constant, float_constant, character_constant, WHILE, ELSE, IF, RETURN, {,BREAK, FOR, DO, }, $ }

Follow( expressions ) : { ,, ), ], ;, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, NOT, (, identifier, integer_constant,

string_constant, float_constant, character_constant, WHILE, ELSE, IF, RETURN, {,BREAK, FOR, DO, }, $ }

Follow( simple_expression ) : { ), ;, ,, ], INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, NOT, (, identifier, integer_constant, string_constant, float_constant, character_constant, WHILE, ELSE, IF, RETURN, {,BREAK, FOR, DO, }, $ }

Follow( simple_expression_breakup ) : { ), ;, ,, ], INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, NOT, (, identifier, integer_constant, string_constant, float_constant, character_constant, WHILE, ELSE, IF, RETURN, {,BREAK, FOR, DO, }, $ }

Follow( and_expression ) : { OR_OR, ), ;, ,, ], INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, NOT, (, identifier, integer_constant, string_constant, float_constant, character_constant, WHILE, ELSE, IF,RETURN, {, BREAK, FOR, DO, }, $ }

Follow( and_expression_breakup ) : { OR_OR, ), ;, ,, ], INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, NOT, (, identifier, integer_constant, string_constant, float_constant, character_constant, WHILE, ELSE, IF,RETURN, {, BREAK, FOR, DO, }, $ }

Follow( unary_relation_expression ) : { AND_AND, OR_OR, ), ;, ,, ], INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, NOT, (, identifier, integer_constant, string_constant, float_constant, character_constant, WHILE,ELSE, IF, RETURN, {, BREAK, FOR, DO, }, $ }

Follow( regular_expression ) : { AND_AND, OR_OR, ), ;, ,, ], INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, NOT, (, identifier, integer_constant, string_constant, float_constant, character_constant, WHILE,ELSE, IF, RETURN, {, BREAK, FOR, DO, }, $ }

Follow( regular_expression_breakup ) : { AND_AND, OR_OR, ), ;, ,, ], INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, NOT, (, identifier, integer_constant, string_constant, float_constant, character_constant, WHILE,ELSE, IF, RETURN, {, BREAK, FOR, DO, }, $ }

Follow( relational_operators ) : { (, identifier, integer_constant, string_constant, float_constant, character_constant }

Follow( sum_expression ) : { GREAT_EQUAL, LESS_EQUAL, GREAT, LESS, EQUAL, NOT_EQUAL, AND_AND, OR_OR, ), ;, ,, ], INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, NOT, (, identifier,integer_constant, string_constant, float_constant, character_constant, WHILE, ELSE, IF, RETURN, {, BREAK, FOR, DO, }, $ }

Follow( sum_expression' ) : { GREAT_EQUAL, LESS_EQUAL, GREAT, LESS, EQUAL, NOT_EQUAL, AND_AND, OR_OR, ), ;, ,, ], INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, NOT, (, identifier,integer_constant, string_constant, float_constant, character_constant, WHILE, ELSE, IF, RETURN, {, BREAK, FOR, DO, }, $ }

Follow( sum_operators ) : { (, identifier, integer_constant, string_constant, float_constant, character_constant }

Follow( term ) : { +, -, GREAT_EQUAL, LESS_EQUAL, GREAT, LESS, EQUAL, NOT_EQUAL, AND_AND, OR_OR, ), ;, ,, ], INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, NOT, (, identifier,integer_constant, string_constant, float_constant, character_constant, WHILE, ELSE, IF, RETURN, {, BREAK, FOR, DO, }, $ }

Follow( term' ) : { +, -, GREAT_EQUAL, LESS_EQUAL, GREAT, LESS, EQUAL, NOT_EQUAL, AND_AND, OR_OR, ), ;, ,, ], INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, NOT, (, identifier,integer_constant, string_constant, float_constant, character_constant, WHILE, ELSE, IF, RETURN, {, BREAK, FOR, DO, }, $ }

Follow( multiply_operators ) : { (, identifier, integer_constant, string_constant, float_constant, character_constant}

Follow( factor ) : { *, /, %, +, -, GREAT_EQUAL, LESS_EQUAL, GREAT, LESS, EQUAL, NOT_EQUAL, AND_AND, OR_OR, ), ;, ,, ], INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, NOT, (,identifier, integer_constant, string_constant, float_constant, character_constant, WHILE, ELSE, IF, RETURN, {, BREAK, FOR, DO, }, $ }

Follow( iden ) : { =, ADD_EQUAL, SUBTRACT_EQUAL, MULTIPLY_EQUAL, DIVIDE_EQUAL, MOD_EQUAL, INCREMENT, DECREMENT, *, /, %, +, -, GREAT_EQUAL, LESS_EQUAL, GREAT, LESS, EQUAL, NOT_EQUAL,AND_AND, OR_OR, ), ;, ,, ], INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, NOT, (, identifier, integer_constant, string_constant, float_constant, character_constant, WHILE,ELSE, IF, RETURN, {, BREAK, FOR, DO, }, $ }

Follow( iden' ) : { =, ADD_EQUAL, SUBTRACT_EQUAL, MULTIPLY_EQUAL, DIVIDE_EQUAL, MOD_EQUAL, INCREMENT, DECREMENT, *, /, %, +, -, GREAT_EQUAL, LESS_EQUAL, GREAT, LESS, EQUAL, NOT_EQUAL,AND_AND, OR_OR, ), ;, ,, ], INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, NOT, (, identifier, integer_constant, string_constant, float_constant, character_constant, WHILE,ELSE, IF, RETURN, {, BREAK, FOR, DO, }, $ }

Follow( extended_iden ) : { [, ., =, ADD_EQUAL, SUBTRACT_EQUAL, MULTIPLY_EQUAL, DIVIDE_EQUAL, MOD_EQUAL, INCREMENT, DECREMENT, *, /, %, +, -, GREAT_EQUAL, LESS_EQUAL, GREAT, LESS, EQUAL,NOT_EQUAL, AND_AND, OR_OR, ), ;, ,, ], INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, NOT, (, identifier, integer_constant, string_constant, float_constant,character_constant, WHILE, ELSE, IF, RETURN, {, BREAK, FOR, DO, }, $ }

Follow( func ) : { *, /, %, +, -, GREAT_EQUAL, LESS_EQUAL, GREAT, LESS, EQUAL, NOT_EQUAL, AND_AND, OR_OR, ), ;, ,, ], INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, NOT, (,identifier, integer_constant, string_constant, float_constant, character_constant, WHILE, ELSE, IF, RETURN, {, BREAK, FOR, DO, }, $ }

Follow( func_call ) : { *, /, %, +, -, GREAT_EQUAL, LESS_EQUAL, GREAT, LESS, EQUAL, NOT_EQUAL, AND_AND, OR_OR, ), ;, ,, ], INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, NOT, (,identifier, integer_constant, string_constant, float_constant, character_constant, WHILE, ELSE, IF, RETURN, {, BREAK, FOR, DO, }, $ }

Follow( arguments ) : { ) }

Follow( arguments_list ) : { ) }

Follow( extended_arguments ) : { ) }

Follow( constant ) : { *, /, %, +, -, GREAT_EQUAL, LESS_EQUAL, GREAT, LESS, EQUAL, NOT_EQUAL, AND_AND, OR_OR, ), ;, ,, ], INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, VOID, STRUCT, NOT, (,identifier, integer_constant, string_constant, float_constant, character_constant, WHILE, ELSE, IF, RETURN, {, BREAK, FOR, DO, }, $ }

# Design of Programs
## Updated Lexical Analyzer Code

```
%{
    #include <stdio.h>
    #include <string.h>
    #include "y.tab.h"

    struct ConstantTable{
        char constant_name[100];
        char constant_type[100];
        int exist;
    }CT[1000];

    struct SymbolTable{
        char symbol_name[100];
        char symbol_type[100];
        char array_dimensions[100];
        char class[100];
        char value[100];
        char parameters[100];
        int line_number;
        int exist;
    }ST[1000];

    unsigned long hash(unsigned char *str)
    {
        unsigned long hash = 5381;
        int c;

        while (c = *str++)
            hash = ((hash << 5) + hash) + c;

        return hash;
    }

    int search_ConstantTable(char* str){
        unsigned long temp_val = hash(str);
        int val = temp_val%1000;

        if(CT[val].exist == 0){
            return 0;
        }

        else if(strcmp(CT[val].constant_name, str) == 0)
        {
            return 1;
        }
        else
        {
```

```
            for(int i = val+1 ; i!=val ; i = (i+1)%1000)
            {
                if(strcmp(CT[i].constant_name,str)==0)
                {
                    return 1;
                }
            }
            return 0;
        }
    }


    int search_SymbolTable(char* str){
        unsigned long temp_val = hash(str);
        int val = temp_val%1000;

        if(ST[val].exist == 0){
            return 0;
        }

        else if(strcmp(ST[val].symbol_name, str) == 0)
        {
            return 1;
        }
        else
        {
            for(int i = val+1 ; i!=val ; i = (i+1)%1000)
            {
                if(strcmp(ST[i].symbol_name,str)==0)
                {
                    return 1;
                }
            }
            return 0;
        }
    }


    void insert_ConstantTable(char* name, char* type){
        int index = 0;
         if(search_ConstantTable(name)){
            return;
        }
        else{
            unsigned long temp_val = hash(name);
            int val = temp_val%1000;
            if(CT[val].exist == 0){
```

```
            strcpy(CT[val].constant_name, name);
            strcpy(CT[val].constant_type, type);
            CT[val].exist = 1;
            return;
        }

        for(int i = val+1; i != val; i = (i+1)%1000){
            if(CT[i].exist == 0){
                index = i;
                break;
            }
        }
        strcpy(CT[index].constant_name, name);
        strcpy(CT[index].constant_type, type);
        CT[index].exist = 1;
    }
}

void insert_SymbolTable(char* name, char* class){
    int index = 0;
    //printf("BBBB");
     if(search_SymbolTable(name)){
        //printf("AAAAAA");
        return;
    }
    else{
        unsigned long temp_val = hash(name);
        int val = temp_val%1000;
        if(ST[val].exist == 0){
            strcpy(ST[val].symbol_name, name);
            strcpy(ST[val].class, class);
            ST[val].line_number = yylineno;
            ST[val].exist = 1;
            return;
        }

        for(int i = val+1; i != val; i = (i+1)%1000){
            if(ST[i].exist == 0){
                index = i;
                break;
            }
        }
        strcpy(ST[index].symbol_name, name);
        strcpy(ST[val].class, class);
        ST[index].exist = 1;
    }
}
```

```
void insert_SymbolTable_type(char *str1, char *str2)
{
    for(int i = 0 ; i < 1000 ; i++)
    {
        if(strcmp(ST[i].symbol_name,str1)==0)
        {
            strcpy(ST[i].symbol_type,str2);
        }
    }
}

void insert_SymbolTable_value(char *str1, char *str2)
{
    for(int i = 0 ; i < 1000 ; i++)
    {
        if(strcmp(ST[i].symbol_name,str1)==0)
        {
            strcpy(ST[i].value,str2);
        }
    }
}

void insert_SymbolTable_arraydim(char *str1, char *dim)
{
    for(int i = 0 ; i < 1000 ; i++)
    {
        if(strcmp(ST[i].symbol_name,str1)==0)
        {
            strcpy(ST[i].array_dimensions,dim);
        }
    }
}

void insert_SymbolTable_funcparam(char *str1, char *param)
{
    for(int i = 0 ; i < 1000 ; i++)
    {
        if(strcmp(ST[i].symbol_name,str1)==0)
        {
            strcat(ST[i].parameters," ");
            strcat(ST[i].parameters,param);
        }
    }
}
```

```
    void insert_SymbolTable_line(char *str1, int line)
    {
        for(int i = 0 ; i < 1000 ; i++)
        {
            if(strcmp(ST[i].symbol_name,str1)==0)
            {
                ST[i].line_number = line;
            }
        }
    }

    void printConstantTable(){
        printf("%20s | %20s\n", "CONSTANT","TYPE");
        for(int i = 0; i < 1000; ++i){
            if(CT[i].exist == 0)
                continue;

            printf("%20s | %20s\n", CT[i].constant_name, CT[i].constant_type);
        }
    }

    void printSymbolTable(){
        printf("%10s | %18s | %10s | %10s | %10s | %10s\n","SYMBOL", "CLASS", "TYPE","VALUE","DIMENSIONS","PARAMETERS","LINE NO");
        for(int i = 0; i < 1000; ++i){
            if(ST[i].exist == 0)
                continue;
            printf("%10s | %18s | %10s | %10s | %10s | %10s | %d\n", ST[i].symbol_name, ST[i].class, ST[i].symbol_type, ST[i].value,ST[i].
        }
    }
    char current_identifier[20];
    char current_type[20];
    char current_value[20];
    char current_function[20];
    char previous_operator[20];
    int flag;

%}

num                 [0-9]
alpha               [a-zA-Z]
alphanum            {alpha}|{num}
escape_sequences    0|a|b|f|n|r|t|v|"\\"|"\""|"\'"
ws                  [ \t\r\f\v]+
%x MLCOMMENT
DE "define"
IN "include"
```

```
    int nested_count = 0;
    int check_nested = 0;

\n  {yylineno++;}
"#include"[ ]*"<"{alpha}({alphanum})*".h>"                                    { }
"#define"[ ]+(_|{alpha})({alphanum})*[ ]*(.)+                                 { }
"//".*                                                                        { }

"/*"                                        { BEGIN MLCOMMENT; }
<MLCOMMENT>"/*"                             { ++nested_count;
                                              check_nested = 1;
                                            }
<MLCOMMENT>"*"+"/"                          { if (nested_count) --nested_count;
                                              else{ if(check_nested){
                                                        check_nested = 0;
                                                        BEGIN INITIAL;
                                                        }
                                                    else{
                                                        BEGIN INITIAL;
                                                    }
                                              }
                                            }
<MLCOMMENT>"*"+                             ;
<MLCOMMENT>[^/*\n]+                         ;
<MLCOMMENT>[/]                              ;
<MLCOMMENT>\n                               ;
<MLCOMMENT><<EOF>>                          { printf("Line No. %d ERROR: MULTI LINE COMMENT NOT CLOSED\n", yylineno); return 0;}

"["                                         {return *yytext;}
"]"                                         {return *yytext;}
"("                                         {return *yytext;}
")"                                         {return *yytext;}
"{"                                         {return *yytext;}
"}"                                         {return *yytext;}
","                                         {return *yytext;}
";"                                         {return *yytext;}



"char"          { strcpy(current_type,yytext); insert_SymbolTable(yytext, "Keyword"); return CHAR;}
"double"        { strcpy(current_type,yytext); insert_SymbolTable(yytext, "Keyword"); return DOUBLE;}
"else"          { insert_SymbolTable_line(yytext, yylineno); insert_SymbolTable(yytext, "Keyword"); return ELSE;}
"float"         { strcpy(current_type,yytext); insert_SymbolTable(yytext, "Keyword");return FLOAT;}
"while"         { insert_SymbolTable(yytext, "Keyword"); return WHILE;}
"do"            { insert_SymbolTable(yytext, "Keyword"); return DO;}
"for"           { insert_SymbolTable(yytext, "Keyword"); return FOR;}
```

```
"if"            { insert_SymbolTable(yytext, "Keyword"); return IF;}
"int"           { strcpy(current_type,yytext); insert_SymbolTable(yytext, "Keyword");return INT;}
"long"          { strcpy(current_type,yytext); insert_SymbolTable(yytext, "Keyword");  return LONG;}
"return"        { insert_SymbolTable(yytext, "Keyword");  return RETURN;}
"short"         { strcpy(current_type,yytext); insert_SymbolTable(yytext, "Keyword");  return SHORT;}
"signed"        { strcpy(current_type,yytext); insert_SymbolTable(yytext, "Keyword");  return SIGNED;}
"sizeof"        { insert_SymbolTable(yytext, "Keyword");  return SIZEOF;}
"struct"        { strcpy(current_type,yytext); insert_SymbolTable(yytext, "Keyword");  return STRUCT;}
"unsigned"      { insert_SymbolTable(yytext, "Keyword");  return UNSIGNED;}
"void"          { strcpy(current_type,yytext); insert_SymbolTable(yytext, "Keyword");  return VOID;}
"break"         { insert_SymbolTable(yytext, "Keyword");  return BREAK;}
"continue"      { insert_SymbolTable(yytext, "Keyword");  return CONTINUE;}
"goto"          { insert_SymbolTable(yytext, "Keyword");  return GOTO;}
"switch"        { insert_SymbolTable(yytext, "Keyword");  return SWITCH;}
"case"          { insert_SymbolTable(yytext, "Keyword");  return CASE;}
"default"       { insert_SymbolTable(yytext, "Keyword");  return DEFAULT;}

("\"")[^\n\"]*("\"")                        {strcpy(current_value,yytext); insert_ConstantTable(yytext,"String Constant"); return string_constant;}
("\"")[^\n\"]*                              { printf("Line No. %d ERROR: UNCLOSED STRING - %s\n", yylineno, yytext); return 0;}
("\'")(("\"({escape_sequences}))|.)("\'")   {strcpy(current_value,yytext); insert_ConstantTable(yytext,"Character Constant"); return character_constant;}
("\'")((("\")[^0abfnrtv\\\"\'][^\n\']*))|[^\n\''][^\n\'']+)("\'") {printf("Line No. %d ERROR: NOT A CHARACTER - %s\n", yylineno, yytext); return 0; }
{num}+(\.{num}+)?e{num}+                {strcpy(current_value,yytext); insert_ConstantTable(yytext, "Floating Constant"); return float_constant;}
{num}+\.{num}+                          {strcpy(current_value,yytext); insert_ConstantTable(yytext, "Floating Constant"); return float_constant;}
{num}+                                  {strcpy(current_value,yytext); insert_ConstantTable(yytext, "Number Constant"); return integer_constant;}
(_|{alpha})({alpha}|{alpha}|_)*         {strcpy(current_identifier,yytext);insert_SymbolTable(yytext,"Identifier");  return identifier;}
(_|{alpha})({alpha}|{alpha}|_)*/\[      {strcpy(current_identifier,yytext);insert_SymbolTable(yytext,"Array Identifier");  return identifier;}
{ws}                                    ;

"+"                                     {return *yytext;}
"-"                                     {return *yytext;}
"*"                                     {return *yytext;}
"/"                                     {return *yytext;}
"="                                     {return *yytext;}
"%"                                     {return *yytext;}
"&"                                     {return *yytext; }
"^"                                     {return *yytext; }
"++"                                    {return INCREMENT;}
"--"                                    {return DECREMENT;}
"!"                                     {return NOT;}
"+="                                    {return ADD_EQUAL;}
"-="                                    {return SUBTRACT_EQUAL;}
"*="                                    {return MULTIPLY_EQUAL;}
"/="                                    {return DIVIDE_EQUAL;}
"%="                                    {return MOD_EQUAL;}
"&&"                                    {return AND_AND;}

"||"                                        {return OR_OR;}
">"                                         {return GREAT;}
"<"                                         {return LESS;}
">="                                        {return GREAT_EQUAL;}
"<="                                        {return LESS_EQUAL;}
"=="                                        {return EQUAL;}
"!="                                        {return NOT_EQUAL;}
.                                           { flag = 1;
                                              if(yytext[0] == '#')
                                                   printf("Line No. %d PREPROCESSOR ERROR - %s\n", yylineno, yytext);
                                              else
                                                   printf("Line No. %d ERROR ILLEGAL CHARACTER - %s\n", yylineno, yytext);
                                             return 0;}

%%
```

**Figure 1**

# Parser Code

```
%{
    void yyerror(char* s);
    int yylex();
    #include "stdio.h"
    #include "stdlib.h"
    #include "ctype.h"
    #include "string.h"
    void insert_type();
    void insert_value();
    void insert_dimensions();
    void insert_parameters();
    extern int flag=0;
    int insert_flag = 0;

    extern char current_identifier[20];
    extern char current_type[20];
    extern char current_value[20];
    extern char current_function[20];
    extern char previous_operator[20];

%}

%nonassoc IF
%token INT CHAR FLOAT DOUBLE LONG SHORT SIGNED UNSIGNED STRUCT
%token RETURN MAIN
%token VOID
%token WHILE FOR DO
%token BREAK CONTINUE GOTO
%token ENDIF
%token SWITCH CASE DEFAULT
%expect 2

%token identifier
%token integer_constant string_constant float_constant character_constant

%nonassoc ELSE

%right MOD_EQUAL
%right MULTIPLY_EQUAL DIVIDE_EQUAL
%right ADD_EQUAL SUBTRACT_EQUAL
%right '='

%left OR_OR
%left AND_AND
%left '^'
%left EQUAL NOT_EQUAL
%left LESS_EQUAL LESS GREAT_EQUAL GREAT
%left '+' '-'
%left '*' '/' '%'
```

```
%right SIZEOF
%right NOT
%left INCREMENT DECREMENT


%start begin_parse

%%
begin_parse
            : declarations;

declarations
            : declaration declarations
            |
            ;

declaration
            : variable_dec
            | function_dec
            | structure_dec;

structure_dec
            : STRUCT identifier { insert_type(); } '{' structure_content  '}' ';';

structure_content : variable_dec structure_content | ;

variable_dec
            : datatype variables ';'
            | structure_initialize;

structure_initialize
            : STRUCT identifier variables;

variables
            : identifier_name multiple_variables;

multiple_variables
            : ',' variables
            | ;

identifier_name
            : identifier { insert_type(); } extended_identifier;

extended_identifier : array_identifier | '='{strcpy(previous_operator,"=");} expression ;

array_identifier
            : '[' array_dims
            | ;
```

```
array_dims
            : integer_constant {insert_dimensions();} ']' initilization
            | ']' string_initilization;

initilization
            : string_initilization
            | array_initialization
            | ;

string_initilization
            : '='{strcpy(previous_operator,"=");} string_constant { insert_value(); };

array_initialization
            : '='{strcpy(previous_operator,"=");} '{' array_values '}';

array_values
            : integer_constant multiple_array_values;

multiple_array_values
            : ',' array_values
            | ;


datatype
            : INT | CHAR | FLOAT | DOUBLE
            | LONG long_grammar
            | SHORT short_grammar
            | UNSIGNED unsigned_grammar
            | SIGNED signed_grammar
            | VOID ;

unsigned_grammar
            : INT | LONG long_grammar | SHORT short_grammar | ;

signed_grammar
            : INT | LONG long_grammar | SHORT short_grammar | ;

long_grammar
            : INT | ;

short_grammar
            : INT | ;

function_dec
            : function_datatype function_parameters;

function_datatype
            : datatype identifier '('  {strcpy(current_function,current_identifier); insert_type();};
```

```
function_parameters
            : parameters ')' statement;

parameters
            : datatype all_parameter_identifiers | ;

all_parameter_identifiers
            : parameter_identifier multiple_parameters;

multiple_parameters
            : ',' parameters
            | ;

parameter_identifier
            : identifier { insert_parameters(); insert_type(); } extended_parameter;

extended_parameter
            : '[' ']'
            | ;

statement
            : expression_statment | multiple_statement
            | conditional_statements | iterative_statements
            | return_statement | break_statement
            | variable_dec;

multiple_statement
            : '{' statments '}' ;

statments
            : statement statments
            | ;

expression_statment
            : expression ';'
            | ';' ;

conditional_statements
            : IF '(' simple_expression ')' statement extended_conditional_statements;

extended_conditional_statements
            : ELSE statement
            | ;

iterative_statements
            : WHILE '(' simple_expression ')' statement
            | FOR '(' for_initialization simple_expression ';' expression ')'
            | DO statement WHILE '(' simple_expression ')' ';';
```

```
for_initialization
            : variable_dec
            | expression ';'
            | ';' ;

return_statement
            : RETURN return_suffix;

return_suffix
            : ';'
            | expression ';' ;

break_statement
            : BREAK ';' ;

expression
            : iden expressions
            | simple_expression ;

expressions
            : '='{strcpy(previous_operator,"=");} expression
            | ADD_EQUAL{strcpy(previous_operator,"+=");} expression
            | SUBTRACT_EQUAL{strcpy(previous_operator,"-=");} expression
            | MULTIPLY_EQUAL{strcpy(previous_operator,"*=");} expression
            | DIVIDE_EQUAL{strcpy(previous_operator,"/=");} expression
            | MOD_EQUAL{strcpy(previous_operator,"%=");} expression
            | INCREMENT
            | DECREMENT ;

simple_expression
            : and_expression simple_expression_breakup;

simple_expression_breakup
            : OR_OR and_expression simple_expression_breakup | ;

and_expression
            : unary_relation_expression and_expression_breakup;

and_expression_breakup
            : AND_AND unary_relation_expression and_expression_breakup
            | ;

unary_relation_expression
            : NOT unary_relation_expression
            | regular_expression ;

regular_expression
            : sum_expression regular_expression_breakup;
```

```
regular_expression_breakup
            : relational_operators sum_expression
            | ;

relational_operators
            : GREAT_EQUAL{strcpy(previous_operator,">=");}
            | LESS_EQUAL{strcpy(previous_operator,"<=");}
            | GREAT{strcpy(previous_operator,">");}
            | LESS{strcpy(previous_operator,"<");}
            | EQUAL{strcpy(previous_operator,"==");}
            | NOT_EQUAL{strcpy(previous_operator,"!=");} ;

sum_expression
            : sum_expression sum_operators term
            | term ;

sum_operators
            : '+'
            | '-' ;

term
            : term multiply_operators factor
            | factor ;

multiply_operators
            : '*' | '/' | '%' ;

factor
            : func | iden ;

iden
            : identifier
            | iden extended_iden;

extended_iden
            : '[' expression ']'
            | '.' identifier;

func
            : '('{strcpy(previous_operator,"(");} expression ')'
            | func_call | constant;

func_call
            : identifier '('{strcpy(previous_operator,"(");} arguments ')';

arguments
            : arguments_list | ;
```

```
extended_arguments
            : ',' expression extended_arguments
            | ;

constant
            : integer_constant  { insert_value(); }
            | string_constant   { insert_value(); }
            | float_constant    { insert_value(); }
            | character_constant{ insert_value(); };

%%

extern FILE *yyin;
extern int yylineno;
extern char *yytext;
void insert_SymbolTable_type(char *,char *);
void insert_SymbolTable_value(char *, char *);
void insert_ConstantTable(char *, char *);
void insert_SymbolTable_arraydim(char *, char *);
void insert_SymbolTable_funcparam(char *, char *);
void printSymbolTable();
void printConstantTable();


int main()
{
    yyin = fopen("test15.c", "r");
    yyparse();

    if(flag == 0)
    {
        printf("VALID PARSE\n");
        printf("%30s SYMBOL TABLE \n", " ");
        printf("%30s %s\n", " ", "------------");
        printSymbolTable();

        printf("\n\n%30s CONSTANT TABLE \n", " ");
        printf("%30s %s\n", " ", "--------------");
        printConstantTable();
    }
}

void yyerror(char *s)
{
    printf("Line No. : %d %s %s\n",yylineno, s, yytext);
    flag=1;
    printf("INVALID PARSE\n");
}
```

```
void insert_type()
{
    insert_SymbolTable_type(current_identifier,current_type);
}

void insert_value()
{
    if(strcmp(previous_operator, "=") == 0)
    {   insert_SymbolTable_value(current_identifier,current_value);
    }
}

void insert_dimensions()
{
    insert_SymbolTable_arraydim(current_identifier, current_value);
}

void insert_parameters()
{
    insert_SymbolTable_funcparam(current_function, current_identifier);
}

int yywrap()
{
    return 1;
}
```

**Figure 2**

# Explanation

The lex code is used to detect tokens and generate a stream of tokens from the input C source code. In the first phase of the project, we only stored the different symbols and constants their respective tables and printed out the different tokens with their corresponding line numbers. For this stage, we need to return the tokens identified by the lexer to the parser so that the parser is able to use it for further computation. In addition to the functions used in the previous stage, we added functions to help the parser insert the type, value, function parameter, and array dimensions into the symbol table.

## Definition Section

In the definition section of the yacc program, we include all the required header files, function definitions and other variables. All the tokens which are returned by the lexical analyzer are also listed in the order of their precedence in this section. Operators are also declared here according to their associativity and precedence. This helps ensure that the grammar given to the parser is unambiguous.

## Rules Section

In this section, grammar rules for the C Programming Language is written. The grammar rules are written in such a way that there is no left recursion and the grammar is also deterministic. Non deterministic grammar is converted by applying left factoring. The grammar productions does the syntax analysis of the source code. When the complete statement with proper syntax is matched by the parser, the parser recognizes that it is a valid parse and prints the symbol and constant table. If the statement is not matched, the parser recognizes that there is an error and outputs the error along with the line number.

## C Code Section

The yyparse() function was called to run the program on the given input file. After that, both the symbol table and the constant table were printed in order to show the result.

## Functionality

This section describes how the code identifies various constructs of the C language. All the rules are described using context-free grammar. If the input sentence can be produced using the grammar, the sentence is accepted. Otherwise, the parser displays an error.

1. **Start Symbol**

In our implementation of the grammar for the C language, we use *begin_parse* as the start variable. This is done with the help of *%start begin_parse*. In case the start symbol is not declared explicitly, Yacc automatically assumes the first non terminal on the left side as the start symbol.

2. **Declarations**

A C program is essentially made up of a bunch of declarations. Any code is made up of a function, variable or structure declarations.

*declarations -> declaration declarations |*
*declaration -> variable_dec | function_dec | structure_dec;*

3. **Variable Declaration**

*variable_dec -> datatype variables ';'*
*| structure_initialize*
*variables -> identifier_name multiple_variables*
*multiple_variables -> ',' variables |*

*identifier_name -> identifier extended_identifier*
*extended_identifier -> array_identifier | '=' expression*
*array_identifier -> '[' array_dims |*
*array_dims -> integer_constant ']' initilization*
*| ']' string_initilization;*

The above rules are being used to construct variable declaration. The *multiple_variables* rule helps us to declare multiple identifiers in a single statement and also all statements should end with a semi-colon. These rules also allow variables to be initialized. These rules also allow declaration and initialization of array variables.

**4. Function Declarations**

*function_dec ->function_datatype function_parameters*
*function_datatype -> datatype identifier '('*
*function_parameters -> parameters ')' statement*
*parameters -> datatype all_parameter_identifiers |*
*all_parameter_identifiers -> parameter_identifier multiple_parameters*
*multiple_parameters -> ',' parameters |*

The above rules are used to define as well as declare functions. *function_dec* is used for declarations. It produces all cases of valid declarations with return type, name and parameter list. To generate all valid cases of parameter list a non-terminal *parameters* is used. To generate more than one parameter *all_parameter_identifiers* and *multiple_parameters* work together.

**5. Structure Declaration and Initialization**

*structure_dec -> STRUCT identifier '{' structure_content '}' ';'*
*structure_content -> variable_dec structure_content |*
*structure_initialize -> STRUCT identifier variables;*

The above production rules are used to identify structure declarations and initializations.

**6. Statements**

*statement -> expression_statment | multiple_statement*
*| conditional_statements | iterative_statements*
*| return_statement | break_statement*
*| variable_dec*
*multiple_statement -> '{' statments '}'*
*statments -> statement statements |*

The above rules are used to generate any kind of statements in C. These

statements usually arise inside functions. Multiple statements with curly brackets are generated using the *multiple_statement* rule. This rule on its own can produce a list of statements denoted by *statements* which is made up of multiple statements or can also be empty.

**7. If-Else Statements**

*conditional_statements* -> *IF '(' simple_expression ')' statement*
*extended_conditional_statements*
*extended_conditional_statements -> ELSE statement |*

This rule is used to verify the syntax of all if-else statements. This rule also handles the dangling else problem. We use the non-terminal *simple_expression* to derive all possible inputs to if statement for evaluation. The non-terminal *statement* is used to signify all possible blocks of code which can come after an if or an else statement.

**8. Iterative Statements**

*iterative_statements -> WHILE '(' simple_expression ')' statement*
*| FOR '(' for_initialization simple_expression ';'*
*expression ')'*
*| DO statement WHILE '(' simple_expression ')' ';'*

*for_initialization* *->variable_dec*
*| expression ';'*
*| ';'*

The *iterative_statements* production rule is used for identifying all iterative programs in the C language, here we have included rules for While loops, for loops and do while loops. The rules follow the simple syntactical specifications of C. The *statement* non terminal is used for all code blocks that will follow the loop statements. The *expression* and *simple_expression* non terminals are used for identifying statements inside the loop. The *for_initialization* non-terminal handles the initialization of variables inside a for loop.

**9. Expressions**

*expression* -> *iden expression | simple_expression*
*expressions* -> *'=' expression*
*| ADD_EQUAL expression*
*| SUBTRACT_EQUAL expression*
*| MULTIPLY_EQUAL expression*
*| DIVIDE_EQUAL expression*
*| MOD_EQUAL expression*
*| INCREMENT*

```
                                | DECREMENT
simple_expression           -> and_expression simple_expression_breakup
simple_expression_breakup -> OR_OR and_expression
                                simple_expression_breakup |
and_expression -> unary_relation_expression and_expression_breakup;
and_expression_breakup    -> AND_AND unary_relation_expression
                                and_expression_breakup |
unary_relation_expression -> NOT unary_relation_expression
                                | regular_expression ;
regular_expression          -> sum_expression regular_expression_breakup;
regular_expression_breakup -> relational_operators sum_expression |
relational_operators        -> GREAT_EQUAL
                                | LESS_EQUAL
                                | GREAT
                                | LESS
                                | EQUAL
                                | NOT_EQUAL
sum_expression              -> sum_expression sum_operators term
                                | term
```

The above rules are pretty straightforward and are used to derive a large subset of the allowed expressions in the C language. Expression can be an assignment expression or a simple expression. The above grammar thus allows multi-assignment statements.

# TEST CASES (ERROR FREE)

## Test Case 1:

```c
//ERROR FREE - This test includes a declaration and a print statement
#include<stdio.h>

int main()
{
    //This is the first test program.
    int a;
    /* This is the declaration
    of an integer value */

    printf("Hello World");
    return 0;
}
```

## Output:

```
VALID PARSE

                        SYMBOL TABLE
                        ------------
  SYMBOL |             CLASS |       TYPE |      VALUE | DIMENSIONS | PARAMETERS |    LINE NO
     int |           Keyword |            |            |            |            | 4
    main |        Identifier |        int |            |            |            | 4
  printf |        Identifier |            |            |            |            | 10
       a |        Identifier |        int |            |            |            | 7
  return |           Keyword |            |            |            |            | 11


                        CONSTANT TABLE
                        --------------
        CONSTANT |                TYPE
   "Hello World" |     String Constant
               0 |     Number Constant
```

**Figure 3**

## Test case 2:

```c
//ERROR FREE - This test case includes a function
#include<stdio.h>

int multiply(int a)
{
    return 2*a;
}

int main()
{
    int a = 5;
    int b = multiply(a);
    printf("%d ", b);
}
```

## Output:

```
VALID PARSE
                        SYMBOL TABLE
                        ------------
   SYMBOL |             CLASS |       TYPE |      VALUE | DIMENSIONS | PARAMETERS |    LINE NO
 multiply |          Identifier |        int |            |            |          a | 4
      int |             Keyword |            |            |            |            | 4
     main |          Identifier |        int |            |            |            | 9
   printf |          Identifier |            |            |            |            | 13
        a |          Identifier |        int |          5 |            |            | 4
        b |          Identifier |        int |            |            |            | 12
   return |             Keyword |            |            |            |            | 6


                        CONSTANT TABLE
                        --------------
          CONSTANT |               TYPE
                 2 |      Number Constant
                 5 |      Number Constant
               "%d " |      String Constant
```

**Figure 4**

# Test case 3:

```c
//ERROR FREE - This test case includes array declarations, and conditional statement
#include<stdio.h>

int main()
{
    int A[5] = {1,2,3,4,5};
    char B[10] = "Hello";

    if(B[0] == 'H'){
        if(A[0] == '1')
            printf("Hello 1");
        else
            printf("Hello 2");
    }
    else
        printf("Not Hello");
}
```

# Output:

```
VALID PARSE
                        SYMBOL TABLE
                        ------------
 SYMBOL |             CLASS |       TYPE |      VALUE | DIMENSIONS | PARAMETERS |    LINE NO
   else |           Keyword |            |            |            |            | 15
    int |           Keyword |            |            |            |            | 4
   char |           Keyword |            |            |            |            | 7
   main |         Identifier |        int |            |            |            | 4
 printf |         Identifier |            |            |            |            | 11
     if |           Keyword |            |            |            |            | 9
      A |   Array Identifier |        int |            |          5 |            | 6
      B |   Array Identifier |       char |            |         10 |            | 7


                        CONSTANT TABLE
                        --------------
       CONSTANT |               TYPE
        "Hello" |      String Constant
            '1' |   Character Constant
  "Not Hello" |      String Constant
             10 |      Number Constant
              0 |      Number Constant
              1 |      Number Constant
              2 |      Number Constant
              3 |      Number Constant
              4 |      Number Constant
              5 |      Number Constant
            'H' |   Character Constant
      "Hello 1" |      String Constant
      "Hello 2" |      String Constant
```

**Figure 5**

## Test case 4:

```c
//ERROR FREE - This test case includes declaration of a structure
#include<stdio.h>

struct book
{
    char name[10];
    char author[10];
};

int main()
{
    int num = 3;
    printf("Hello");

}
```

## Output:

```
VALID PARSE
                        SYMBOL TABLE
                        ------------
 SYMBOL |             CLASS |       TYPE |      VALUE | DIMENSIONS | PARAMETERS |    LINE NO
    int |           Keyword |            |            |            |            | 10
   char |           Keyword |            |            |            |            | 6
   main |        Identifier |        int |            |            |            | 10
   name |   Array Identifier |       char |            |         10 |            | 6
 printf |        Identifier |            |            |            |            | 13
   book |        Identifier |     struct |            |            |            | 4
    num |        Identifier |        int |          3 |            |            | 12
 author |   Array Identifier |       char |            |         10 |            | 7
 struct |           Keyword |            |            |            |            | 4


                        CONSTANT TABLE
                        --------------
       CONSTANT |                TYPE
        "Hello" |      String Constant
             10 |      Number Constant
              3 |      Number Constant
```

**Figure 6**

## Test case 5:

```c
//ERROR FREE - This test case includes nested loops
#include<stdio.h>

int main()
{
    int num = 3;

    for(int i = 0; i<num; i++)
    {
        for(int j = 0; j < num; j++)
            printf("Hello");
    }
}
```

## Output:

```
VALID PARSE

                        SYMBOL TABLE
                        ------------
 SYMBOL |             CLASS |       TYPE |      VALUE | DIMENSIONS | PARAMETERS |    LINE NO
    int |           Keyword |            |            |            |            | 4
   main |        Identifier |        int |            |            |            | 4
 printf |        Identifier |            |            |            |            | 11
      i |        Identifier |        int |          0 |            |            | 8
      j |        Identifier |        int |          0 |            |            | 10
    num |        Identifier |        int |          3 |            |            | 6
    for |           Keyword |            |            |            |            | 8


                        CONSTANT TABLE
                        --------------
       CONSTANT |                TYPE
        "Hello" |      String Constant
              0 |      Number Constant
              3 |      Number Constant
```

**Figure 7**

## Test case 6:

```c
//ERROR FREE - This test case includes for and while loops
#include<stdio.h>

int main()
{
    int num = 3;

    for(int i = 0; i<num; i++)
        printf("Hello");

    while(num > 0)
    {
        printf("Hello");
        num--;
    }
}
```

## Output:

```
VALID PARSE

                        SYMBOL TABLE
                        ------------
   SYMBOL |             CLASS |       TYPE |      VALUE | DIMENSIONS | PARAMETERS |    LINE NO
      int |           Keyword |            |            |            |            | 4
     main |        Identifier |        int |            |            |            | 4
   printf |        Identifier |            |            |            |            | 9
        i |        Identifier |        int |          0 |            |            | 8
      num |        Identifier |        int |          3 |            |            | 6
    while |           Keyword |            |            |            |            | 11
      for |           Keyword |            |            |            |            | 8


                        CONSTANT TABLE
                        --------------
         CONSTANT |                TYPE
          "Hello" |      String Constant
                0 |      Number Constant
                3 |      Number Constant
```

**Figure 8**

## Test Case 7:

```c
//ERROR FREE - This test case includes escape sequences
#include<stdio.h>

int main()
{
    char es = '\a';
    printf("Hello");
    }
}
```

## Output:

```
VALID PARSE

                        SYMBOL TABLE
                        ------------
  SYMBOL |            CLASS |       TYPE |      VALUE | DIMENSIONS | PARAMETERS |    LINE NO
     int |          Keyword |            |            |            |            | 4
    char |          Keyword |            |            |            |            | 6
    main |       Identifier |        int |            |            |            | 4
  printf |       Identifier |            |            |            |            | 7
      es |       Identifier |       char |       '\a' |            |            | 6


                        CONSTANT TABLE
                        --------------
        CONSTANT |                TYPE
         "Hello" |      String Constant
            '\a' |   Character Constant
```

**Figure 9**

# Test Case 8:

```c
//ERROR FREE - This test case includes nested comments
#include<stdio.h>

int main()
{
    /*This is /* nested comment */!!*/
    /*This is a
    normal comment*/
}
```

# Output:

```
VALID PARSE
                        SYMBOL TABLE
                        ------------
   SYMBOL |            CLASS |       TYPE |      VALUE | DIMENSIONS | PARAMETERS |    LINE NO
      int |          Keyword |            |            |            |          | 4
     main |       Identifier |        int |            |            |          | 4


                        CONSTANT TABLE
                        --------------
         CONSTANT |               TYPE
```

**Figure 10**

# TEST CASES (WITH ERROR)

## Test case 9:

```c
//NOT ERROR FREE - This test case includes an error in the header file statement
#include stdio.h>

int main()
{
    printf("Hello");
    return 0;
}
```

## Output:

```
pranav@LAPTOP-P8NVVSN4:~/Mini C Compiler/Parser$ ./a.out
Line No. 2 PREPROCESSOR ERROR - #
```

**Figure 11**

## Test Case 10:

```c
//NOT ERROR FREE - This test case includes an error in which a multi line comment is not c
#include <stdio.h>

int main()
{
    /* This comment is not closed
    printf("Hello");
    return 0;
}
```

## Output:

```
pranav@LAPTOP-P8NVVSN4:~/Mini C Compiler/Parser$ ./a.out
Line No. 6 ERROR: MULTI LINE COMMENT NOT CLOSED
syntax error
INVALID PARSE
```

**Figure 12**

## Test case 11:

```c
//NOT ERROR FREE - This test case includes an error in which a string is not closed
#include <stdio.h>

int main()
{
    char str[] = "Hello"
    printf("Hello);
    return 0;
}
```

## Output:

```
pranav@LAPTOP-P8NVVSN4:~/Mini C Compiler/Parser$ ./a.out
syntax error printf
INVALID PARSE
```

**Figure 13**

## Test case 12:

```c
//NOT ERROR FREE - This test case includes an error in which the declaration is wrong
#include <stdio.h>

int main()
{
    int @ a;
    printf("Hello");
    return 0;
}
```

## Output:

```
pranav@LAPTOP-P8NVVSN4:~/Mini C Compiler/Parser$ ./a.out
Line No. 6 ERROR ILLEGAL CHARACTER - @
syntax error @
INVALID PARSE
```

**Figure 14**

## Test case 13:

```
//NOT ERROR FREE - This test case includes a declaration missing a semi colon
#include<stdio.h>

int main()
{
    int a = 9
    int b = 10;
}
```

## Output:

```
pranav@LAPTOP-P8NVVSN4:~/Mini C Compiler/Parser$ ./a.out
Line No. : 7 syntax error int
INVALID PARSE
```

**Figure 15**

## Test Case 14:

```
//NOT ERROR FREE - This test case does not have the closing brace
#include<stdio.h>

int main()
{
    int a = 9;
```

## Output:

```
pranav@LAPTOP-P8NVVSN4:~/Mini C Compiler/Parser$ ./a.out
Line No. : 7 syntax error
INVALID PARSE
```

**Figure 16**

# Test Case 15:

```c
//NOT ERROR FREE - This test case includes declaration of a structure with semi colon missing
#include<stdio.h>

struct book
{
    char name[10];
    char author[10];
}
int main()
{
    int num = 3;
    printf("Hello");

}
```

# Output:

```
pranav@LAPTOP-P8NVVSN4:~/Mini C Compiler/Parser$ ./a.out
Line No. : 9 syntax error int
INVALID PARSE
```

**Figure 17**

# Implementation

The regular expressions used to identify the different tokens of the C language are fairly straightforward. Similar lexer code to the one submitted in the previous phase was used. A few features require a significant amount of thought are mentioned below:

- **The Regex for Identifiers**
- **Support for Multi-line and Nested Comments and Error Handling for Unmatched Comments**
- **Literals**
- **Error Handling for Incomplete String**

The parser uses a number of grammar production rules to implement the C programming language grammar. The parser also takes the help from the lexer for its functioning. The lexer outputs tokens one at a time and these tokens are used by the parser. The parser then applies the corresponding production rules on the token to insert the type, value, array dimensions, function parameters etc. into the symbol table.

The lexer also shares some value with the parser such as current_identifier, current_value, current_type. These values help the parser insert the correct values into the correct index of the symbol table. The parser performs syntactical analysis and outputs if the parse in valid or not. If the parse is not valid, then it outputs the line number along with the corresponding error.

We maintain two hash tables, one for the symbol table which stores information about identifiers and other for the constant table which stores information about constants.

- Two functions have been implemented, namely search_ConstantTable() and search_SymbolTable(), which is used for checking if a string is already present in the hash tables. Two other functions have also been implemented, namely insert_ConstantTable() and insert_SymbolTable(), which is used to add a new string into the hash tables.
- The functions insert_SymbolTable_type, insert_SymbolTable_value,

insert_SymbolTable_arraydim, insert_SymbolTable_funcparam are used to insert the identifier type, identifier value, array dimensions, function parameters respectively into the symbol table.

- In the end, in main() function, after yyparse() returns, we call printConstantTable() and printSymbolTable(), which prints all the values in the constant and symbol table in a neat manner.

# Future work

The yacc script presented in this report takes care of all the rules of C language, but is not fully exhaustive in nature. Our future work would include making the script even more robust in order to handle all aspects of C language and making it more efficient.

# Results and Conclusion

The parser was able to successfully parse the tokens recognized by the flex script. The symbol and constant table are populated and the parser also generated error messages in the case of invalid parses. Thus, the parsing stage is an essential part of the compiler and is needed for the simplification of the design of the compiler. It helps improve the efficiency of the compiler while also speeding up the compilation process.

# References


- Compiler Principles, Techniques and Tool by Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jefferey D. Ullman
- https://www.geeksforgeeks.org/introduction-to-yacc/
- https://www.lysator.liu.se/c/ANSI-C-grammar-y.html