CSN 401

Compiler Design

Compiler for a Select Function Set of C Language

Submitted to:
Prof. Manish Kamboj
Dept. of Computer Science
Punjab Engineering College

Submitted by:
Aanshi Bansal 16103055
Ishita Agarwal 16103056

Table of Contents

1. Motivation

The motivation for the project is to learn how all the phases of the compiler are implemented. We are learning the theory in our compiler course and want to apply that in a practical scenario.

2. Objective

A. The primary objective is to implement all the phases of the compiler for a subset of the C programming language. By subset, it is conveyed that the basics of C language will be implemented, but the complexities like struct, union will not be implemented.
B. Showing the output of each phase of the compiler.
C. The following phases will be implemented:
    a. Lexical analysis.
    b. Syntax analysis.
    c. Semantic analysis
    d. Intermediate code generation.
    e. Target Code Generation

3. Scope

We will be implementing the following features of the C language -

It will follow the following rules -

1. **Identifier Rules**
    a. Same as in C
2. **Data Types:**

    Data types supported are:

    a. Primary Data Types (integer(int), floating point(float), character(char) and void).
    b. Derived Data Types (String and array)

3. **Expressions**
    1. Arithmetic Operators (+, -, *, /, %, ++, --)
    2. Relational Operators (== , != , > , < , >=, <=)
    3. Logical Operators (&&, ||, !)
    4. Bitwise Operators (&, | , ^ , << , >>)
4. **Statements**
    a. Declaration statement

        E.g. int a;

    b. Declaration and initialization

        E.g. int a = 5;

    c. Assignment Statement

        E.g. a = b;

    d. Conditional statement (Nesting not allowed)
        i. **Simple if (nesting not allowed)**

```
if
    ….
Else
    ….
```

        ii. **Switch Statement (nesting not allowed)**

```
Switch()
Cases
    Value 1:
Break;
.
```

.
.
Value n:
break;
Endcase

iii. Repetition Statement (nesting not allowed)

a. While loop
b. For loop (start value, end value, inc/dec)

5. Functions

a. Return type
b. Function name following identifier rules
c. () containing input parameters
d. { Function body and return statement }

4. Introduction

Compiler

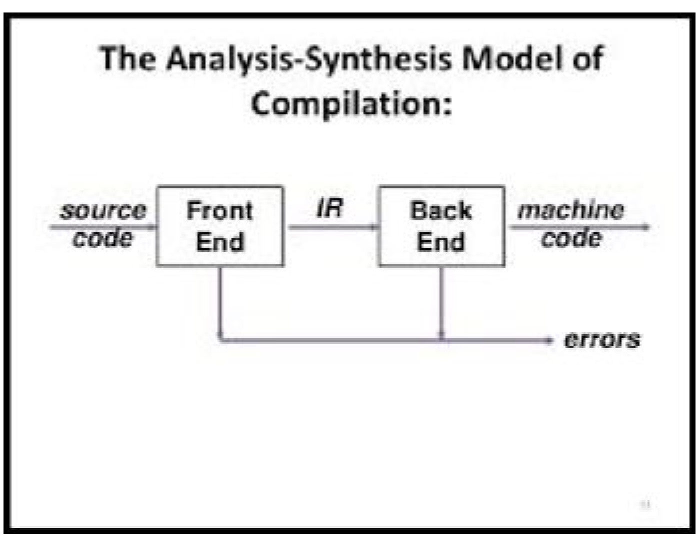
A compiler is a program that can read a program in one language - the source language - and translate it to an equivalent program in another language - the target language. An important role of the compiler is to detect any errors in the source program during the translation process.

Structure of a compiler

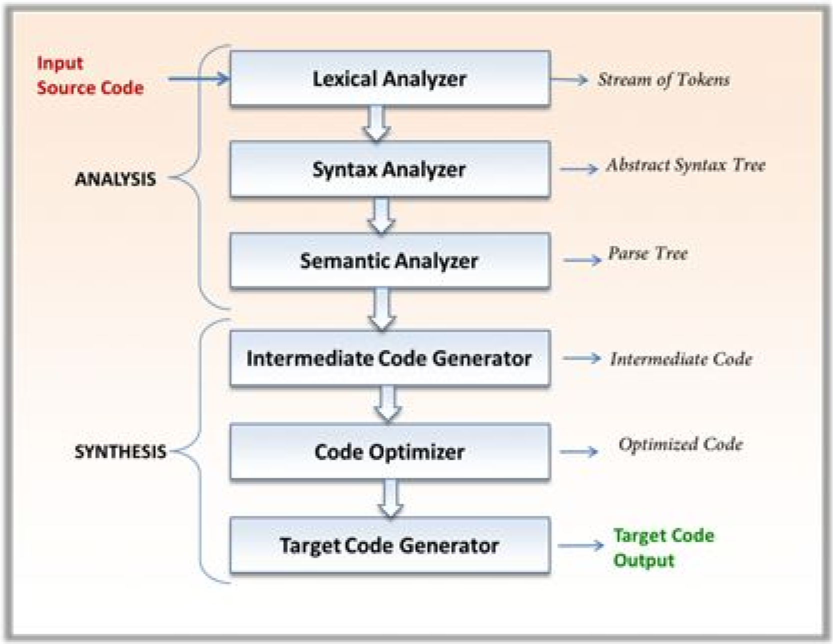
There are two parts involved in the translation of a program in the source language into a semantically equivalent target program: analysis and synthesis.

The analysis part breaks up the source program into constituent pieces and imposes a grammatical structure on them. It then uses this structure to create an intermediate representation of the source program. The analysis part also collects information about the source program and stores it in a data structure called a symbol table, which is passed along with the intermediate representation to the synthesis part.

The synthesis part constructs the desired target program from the intermediate representation and the information in the symbol table. The analysis part is often called the front end of the compiler and the synthesis part is called the back end.



Phases of compilation



The compilation process operates as a sequence of phases each of which transforms one representation of the source program to another.

- The first phase of a compiler is called **lexical analysis or scanning.** The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexemes For each lexeme, the lexical analyzer produces as output a token that it passes on to the subsequent phase, syntax analysis.
- The second phase of the compiler is **syntax analysis or parsing.** The parser uses the tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream.
- The third phase is the **semantic analysis**. The semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition. An important part of semantic analysis is type checking, where the compiler checks that each operator has matching operands.
- After syntax and semantic analysis of the source program, many compilers generate an explicit low-level or machine-like **intermediate code representation**.
- The machine-independent **code-optimization** phase attempts to improve the intermediate code so that better target code will result.

- The last phase is the **code generation**. The code generator takes as input an intermediate representation of the source program and maps it into the target language.

Lexical Analysis

As the first phase of a compiler, the main task of the lexical analyzer is to read the input characters of the source program, group them into lexemes, and produce as output a sequence of tokens for each lexeme in the source program.

The lexical analyzer maintains a data structure called as the symbol table. When the lexical analyzer discovers a lexeme constituting an identifier, it enters that lexeme into the symbol table.

The lexical analyzer performs certain other tasks besides identification of lexemes. One such task is stripping out comments and whitespace. Another task is correlating error messages generated by the compiler with the source program.

Syntax Analysis

In computer science, syntax analysis is the process of checking that the code is syntactically correct. The purpose of syntax analysis or parsing is to check that we have a valid sequence of tokens. Tokens are valid sequences of symbols, keywords, identifiers etc. The parser needs to be able to handle the infinite number of possible valid programs that may be presented to it.

The usual way to define the language is to specify a grammar. A grammar is a set of rules (or productions) that specifies the syntax of the language (i.e. what is a valid sentence in the language). There can be more than one grammar for a given language. The parser analyzes the source code (token stream) against the production rules to detect any errors in the code. The output of this phase is a parse tree.

Semantic Analysis

Semantic analysis is the task of ensuring that the declarations and statements of a program are Semantically correct, i.e that their meaning is clear and consistent with the way in which control structures and data types are supposed to be used.

Semantic analysis can compare information in one part of a parse tree to that in another part (e.g compare reference to variable that agrees with its declaration, or that parameters to a function call match the function definition). Implementing the semantic actions is conceptually simpler in recursive descent parsing because they are simply added to the recursive procedures.

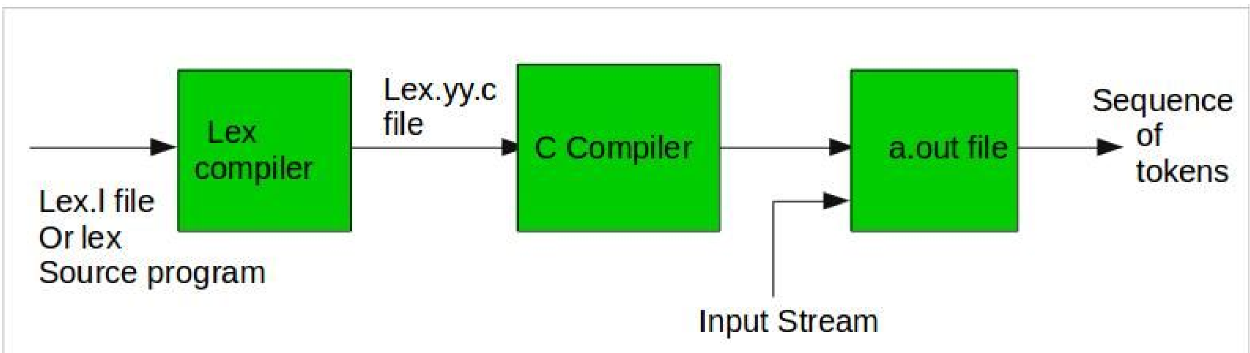
Some of the functions of Semantic analysis are that it maintains and updates the symbol table, check source programs for semantic errors and warnings like type mismatch, global and local scope of a variable, re-definition of variables, usage of undeclared variables.

5. Lexical Analyzer Generator

We will use **FLEX (fast lexical analyzer generator)** tool/computer program for generating lexical analyzers.

The function **yylex()** is automatically generated by the flex when it is provided with a **.l file** and this yylex() function is expected by parser to call to retrieve tokens from current/this token stream.

The function yylex() is the main flex function which runs the Rule Section and extension (.l) is the extension used to save the programs.



Step 1: An input file describes the lexical analyzer to be generated named lex.l is written in lex language. The lex compiler transforms lex.l to C program, in a file that is always named lex.yy.c.

Step 2: The C compiler compiles lex.yy.c file into an executable file called a.out.

Step 3: The output file a.out takes a stream of input characters and produces a stream of tokens.

Program Structure

In the input file, there are 3 sections:

1. Definition Section: The definition section contains the declaration of variables, regular definitions, manifest constants. In the definition section, text is enclosed in **"%{ %}"** brackets. Anything written in this brackets is copied directly to the file **lex.yy.c**

2. Rules Section: The rules section contains a series of rules in the form: *pattern action* and pattern must be unintended and action begin on the same line in {} brackets. The rule section is enclosed in **"%% %%"**.

3. User Code Section: This section contains C statements and additional functions. We can also compile these functions separately and load with the lexical analyzer.

Source Code for Lexical Analyser

Following is the source code for the lexical analysis part.
We have created a file called myscanner.l - this uses FLEX to generate a lexical analyser.

Myscanner.l

```
%{
#include "myscanner.h"
%}

%%
"//"[^\n]*                      ;
\/\*(.|\n)*?\*\/                ;
[ \t\n]                         ;
"{"                             return OPEN_BRACE;
"}"                             return CLOSE_BRACE;
"("                             return OPEN_BRACKET;
")"                             return CLOSE_BRACKET;
";"                             return SEMI_COLON;
[+-]?[1-9][0-9]*                return INT_CONST;
\"(\\.|[^"\\])*\"               return STRING_CONST;
\'.\'                           return CHAR_CONST;
[+-]?([0-9]*[.])?[0-9]+         return FLOAT_CONST;
"return"                        return RETURN;
"int"                           return INT_DTYPE;
"char"                          return CHAR_DTYPE;
"float"                         return FLOAT_DTYPE;
"void"                          return VOID_DTYPE;
"#include"                      return INCLUDE;
"if"                            return IF;
"else"                          return ELSE;
"for"                           return FOR;
"while"                         return WHILE;
"switch"                        return SWITCH;
"case"                          return CASE;
"break"                         return BREAK;
"continue"                      return CONTINUE;
```

```
"%d"|"%f"|"%u"|"%s"          return TYPE_SPEC;
"<="                         return LESSER_EQUAL;
">="                         return GREATER_EQUAL;
"=="                         return EQEQ;
"!="                         return NEQ;
"||"                         return LOR;
"&&"                         return LAND;
"="                          return ASSIGNMENT;
"++"                         return INCR;
"--"                         return DECR;
"+"                          return ADD;
"-"                          return SUB;
"*"                          return MUL;
"/"                          return DIV;
"%"                          return MOD;
"<"                          return LESSER;
">"                          return GREATER;
","                          return COMMA;
"["                          return OPEN_SQUARE;
"]"                          return CLOSED_SQUARE;
"<"[a-z.]+">"                return HEADER_NAME;
[a-zA-Z][_a-zA-Z0-9]*        return IDENTIFIER;
"\""                         return DOUBLEQUOTES;
.                            printf("unexpected character\n");

%%

int yywrap()
{
	return 1;
}

int main()
{
	int scan;
	yyin = fopen("test1.c", "r");
	printf("\n\n");
	scan = yylex();
	while(scan){
```

```
printf("Token: %s\t\t ", yytext);

switch(scan){
        case 1: printf("IDENTIFIER"); break;
        case 2: printf("INT_CONST"); break;
        case 3: printf("INT_DTYPE"); break;
        case 4: printf("INCLUDE"); break;
        case 5: printf("RETURN"); break;
        case 6: printf("HEADER_NAME"); break;
        case 7: printf("OPEN_BRACE"); break;
        case 8: printf("CLOSE_BRACE"); break;
        case 9: printf("OPEN_BRACKET"); break;
        case 10: printf("CLOSE_BRACKET"); break;
        case 11: printf("SEMI_COLON"); break;
        case 12: printf("ASSIGNMENT"); break;
        case 13: printf("ADD"); break;
        case 14: printf("SUB"); break;
        case 15: printf("MUL"); break;
        case 16: printf("DIV"); break;
        case 17: printf("IF"); break;
        case 18: printf("ELSE"); break;
        case 19: printf("FOR"); break;
        case 20: printf("WHILE"); break;
        case 21: printf("CHAR_DTYPE"); break;
        case 22: printf("FLOAT_DTYPE"); break;
        case 23: printf("MOD"); break;
        case 24: printf("INCR"); break;
        case 25: printf("DECR"); break;
        case 26: printf("LESSER"); break;
        case 27: printf("GREATER"); break;
        case 28: printf("LESSER_EQUAL"); break;
        case 29: printf("GREATER_EQUAL"); break;
        case 30: printf("EQEQ"); break;
        case 31: printf("NEQ"); break;
        case 32: printf("LOR"); break;
        case 33: printf("LAND"); break;
        case 34: printf("COMMA"); break;
        case 35: printf("SWITCH"); break;
        case 36: printf("CASE"); break;
```

```
                    case 37: printf("BREAK"); break;
                    case 38: printf("CONTINUE"); break;
                    case 39: printf("FLOAT_CONST"); break;
                    case 40: printf("CHAR_CONST"); break;
                    case 41: printf("VOID_DTYPE"); break;
                    case 42: printf("STRING_CONST"); break;
                    case 43: printf("OPEN_SQUARE"); break;
                    case 44: printf("CLOSED_SQUARE"); break;
                    case 45: printf("TYPE_SPEC"); break;
                    case 46: printf("DOUBLEQUOTES"); break;

                    default: printf("unexpected");

            }



            printf("\n");
            scan = yylex();
        }
}
```

Myscanner.h

This is a header file which has all the definitions of tokens which are to be tokenised using lexical analysis.

```
#define IDENTIFIER 1  //any variable name, function name, array name
#define INT_CONST 2  // integer literals only
#define INT_DTYPE 3 //int
#define INCLUDE 4
#define RETURN 5
#define HEADER_NAME 6
#define OPEN_BRACE 7
#define CLOSE_BRACE 8
#define OPEN_BRACKET 9
#define CLOSE_BRACKET 10
#define SEMI_COLON 11
#define ASSIGNMENT 12
#define ADD 13
```

```
#define SUB 14
#define MUL 15
#define DIV 16
#define IF 17
#define ELSE 18
#define FOR 19
#define WHILE 20
#define CHAR_DTYPE 21
#define FLOAT_DTYPE 22
#define MOD 23 //%
#define INCR 24 //++
#define DECR 25 //--
#define LESSER 26
#define GREATER 27
#define LESSER_EQUAL 28
#define GREATER_EQUAL 29
#define EQEQ 30 // ==
#define NEQ 31
#define LOR 32 // LOGICAL OR -> ||
#define LAND 33
#define COMMA 34
#define SWITCH 35
#define CASE 36
#define BREAK 37
#define CONTINUE 38
#define FLOAT_CONST 39
#define CHAR_CONST 40
#define VOID_DTYPE 41
#define STRING_CONST 42
#define OPEN_SQUARE 43
#define CLOSED_SQUARE 44
#define TYPE_SPEC 45
#define DOUBLEQUOTES 46
```

Output

Tokens Produced by Lexical Analysis -

The lexical analyser generated using myscanner.l is used to tokenize various test C programs below, in order to validate the analyser.

Test 1:

```
#include<stdio.h>
void main()
{
        int x = 4;
        float y = 4.5, 9.0;
        while(x>0){
                x--;
                printf("%d", y);
        }
}
```

Output:

```
Compiler-for-Subset-of-C — -bash — 101×41
Token: #include          INCLUDE
Token: <stdio.h>                 HEADER_NAME
Token: void              VOID_DTYPE
Token: main              IDENTIFIER
Token: (                 OPEN_BRACKET
Token: )                 CLOSE_BRACKET
Token: {                 OPEN_BRACE
Token: int               INT_DTYPE
Token: x                 IDENTIFIER
Token: =                 ASSIGNMENT
Token: 4                 INT_CONST
Token: ;                 SEMI_COLON
Token: float             FLOAT_DTYPE
Token: y                 IDENTIFIER
Token: =                 ASSIGNMENT
Token: 4.5               FLOAT_CONST
Token: ,                 COMMA
Token: 9.0               FLOAT_CONST
Token: ;                 SEMI_COLON
Token: while             WHILE
Token: (                 OPEN_BRACKET
Token: x                 IDENTIFIER
Token: >                 GREATER
Token: 0                 FLOAT_CONST
Token: )                 CLOSE_BRACKET
Token: {                 OPEN_BRACE
Token: x                 IDENTIFIER
Token: --                DECR
Token: ;                 SEMI_COLON
Token: printf            IDENTIFIER
Token: (                 OPEN_BRACKET
Token: "                 DOUBLEQUOTES
Token: %d                TYPE_SPEC
Token: "                 DOUBLEQUOTES
Token: ,                 COMMA
Token: y                 IDENTIFIER
Token: )                 CLOSE_BRACKET
Token: ;                 SEMI_COLON
Token: }                 CLOSE_BRACE
Token: }                 CLOSE_BRACE
(base) Ishitas-MacBook-Pro-2:Compiler-for-Subset-of-C ishitaagarwal$
```

Test 2:

```
#include <stdio.h>

int main()
{
	int x = 54; //this is a single line comment
	/*
	Multi
	Line
	Comment
	*/
	x = x+1;
}
```

Output:

```
Compiler-for-Subset-of-C — -bash — 101×24
(base) Ishitas-MacBook-Pro-2:Compiler-for-Subset-of-C ishitaagarwal$ lex myscanner.l
(base) Ishitas-MacBook-Pro-2:Compiler-for-Subset-of-C ishitaagarwal$ gcc lex.yy.c
(base) Ishitas-MacBook-Pro-2:Compiler-for-Subset-of-C ishitaagarwal$ ./a.out


Token: #include         INCLUDE
Token: <stdio.h>                HEADER_NAME
Token: int              INT_DTYPE
Token: main             IDENTIFIER
Token: (                OPEN_BRACKET
Token: )                CLOSE_BRACKET
Token: {                OPEN_BRACE
Token: int              INT_DTYPE
Token: x                IDENTIFIER
Token: =                ASSIGNMENT
Token: 54               INT_CONST
Token: ;                SEMI_COLON
Token: x                IDENTIFIER
Token: =                ASSIGNMENT
Token: x                IDENTIFIER
Token: +1               INT_CONST
Token: ;                SEMI_COLON
Token: }                CLOSE_BRACE
(base) Ishitas-MacBook-Pro-2:Compiler-for-Subset-of-C ishitaagarwal$
```

Test 3:

```
#include<stdio.h>

int main()
{
        int val = 1;
        for(i = 0 ; i < 3 ; i++) {
                val++;
        }
}
```

Output:

```
Compiler-for-Subset-of-C — -bash — 101×37
(base) Ishitas-MacBook-Pro-2:Compiler-for-Subset-of-C ishitaagarwal$ lex myscanner.l
(base) Ishitas-MacBook-Pro-2:Compiler-for-Subset-of-C ishitaagarwal$ gcc lex.yy.c
(base) Ishitas-MacBook-Pro-2:Compiler-for-Subset-of-C ishitaagarwal$ ./a.out


Token: #include         INCLUDE
Token: <stdio.h>                HEADER_NAME
Token: int              INT_DTYPE
Token: main             IDENTIFIER
Token: (                OPEN_BRACKET
Token: )                CLOSE_BRACKET
Token: {                OPEN_BRACE
Token: int              INT_DTYPE
Token: val              IDENTIFIER
Token: =                ASSIGNMENT
Token: 1                INT_CONST
Token: ;                SEMI_COLON
Token: for              FOR
Token: (                OPEN_BRACKET
Token: i                IDENTIFIER
Token: =                ASSIGNMENT
Token: 0                FLOAT_CONST
Token: ;                SEMI_COLON
Token: i                IDENTIFIER
Token: <                LESSER
Token: 3                INT_CONST
Token: ;                SEMI_COLON
Token: i                IDENTIFIER
Token: ++               INCR
Token: )                CLOSE_BRACKET
Token: {                OPEN_BRACE
Token: val              IDENTIFIER
Token: ++               INCR
Token: ;                SEMI_COLON
Token: }                CLOSE_BRACE
Token: }                CLOSE_BRACE
(base) Ishitas-MacBook-Pro-2:Compiler-for-Subset-of-C ishitaagarwal$
```

Test 4:

```
#include<stdio.h>

int main()
{
        printf("This is a string");
        char c = 'a';
        int arr[2] = {1,2};
}
```

Output:

```
(base) Ishitas-MacBook-Pro-2:Compiler-for-Subset-of-C ishitaagarwal$ lex myscanner.l
(base) Ishitas-MacBook-Pro-2:Compiler-for-Subset-of-C ishitaagarwal$ gcc lex.yy.c
(base) Ishitas-MacBook-Pro-2:Compiler-for-Subset-of-C ishitaagarwal$ ./a.out


Token: #include          INCLUDE
Token: <stdio.h>                 HEADER_NAME
Token: int               INT_DTYPE
Token: main              IDENTIFIER
Token: (                 OPEN_BRACKET
Token: )                 CLOSE_BRACKET
Token: {                 OPEN_BRACE
Token: printf            IDENTIFIER
Token: (                 OPEN_BRACKET
Token: "This is a string"                STRING_CONST
Token: )                 CLOSE_BRACKET
Token: ;                 SEMI_COLON
Token: char              CHAR_DTYPE
Token: c                 IDENTIFIER
Token: =                 ASSIGNMENT
Token: 'a'               CHAR_CONST
Token: ;                 SEMI_COLON
Token: int               INT_DTYPE
Token: arr               IDENTIFIER
Token: [                 OPEN_SQUARE
Token: 2                 INT_CONST
Token: ]                 CLOSED_SQUARE
Token: =                 ASSIGNMENT
Token: {                 OPEN_BRACE
Token: 1                 INT_CONST
Token: ,                 COMMA
Token: 2                 INT_CONST
Token: }                 CLOSE_BRACE
Token: ;                 SEMI_COLON
Token: }                 CLOSE_BRACE
(base) Ishitas-MacBook-Pro-2:Compiler-for-Subset-of-C ishitaagarwal$
```

6. What next?

The lexical analyzer that we created helps us to break down a C source file into tokens as per the C language specifications as mentioned in the scope. Each token (such as identifiers, keywords, special symbols, operators, etc.) has an integer value associated with it, as specified in the **MyScanner.h** file.

When we design the parser in the next phase, the parser will call upon the Flex program to give it tokens and the lexical analyzer will return to the parser the integer value associated with the tokens as and when required by the parser.

Together with the symbol, the parser will prepare a syntax tree with the help of a grammar that we provide it with. The parser can then logically group the tokens to form meaningful statements and can detect C programming constructs such as arrays, loops, and functions. The parser will also help us identify errors that could not be detected in the lexical analysis phase such as unbalanced parentheses, unterminated statements, missing operators, two operators in a row, etc.