NITTE EDUCATION TRUST | NMAM INSTITUTE OF TECHNOLOGY

REPORT ON

CD Mini Project
Carried out on

DESIGN AND IMPLEMENTATION OF LALR PARSER ON GIVEN HYPOTHETICAL LANGUAGE

Submitted to

NMAM INSTITUTE OF TECHNOLOGY, NITTE
(An Autonomous Institution under VTU, Belagavi)

In partial fulfillment of the requirements for the award of the

Degree of Bachelor of Engineering
in
Computer Science and Engineering

by

Avaneesh Sharma (4NM20CS038)
Avinash Monteiro (4NM20CS040)
Bhuvanapalli Ameeth Manoj (4NM21CS044)

Submitted to,

Dr. Raju K

Assosciate Professor
Department of Computer Science and Engineering

ACKNOWLEDGEMENT

The satisfaction and euphoria that accompany the successful completion of any task would be incomplete without the mention of people who made it possible because “Success is the abstract of hard work and perseverance, but steadfast of all is encouraging guidance.” So, We acknowledge all those whose guidance and encouragement served as a beacon light and crowned my efforts with success.

We would like to thank our principal Prof. Niranjan N. Chiplunkar firstly, for providing us with this unique opportunity to do the mini project in the 6th semester of Computer Science and engineering.

We would like to thank my college administration for providing a conductive environment and suitable facilities for this mini project. We would like to thank our HOD Dr. Jyothi Shetty for showing me the path and providing the inspiration required for taking the project to its completion. It was our great pleasure to thank our mentor Dr Raju K for his continuous encouragement, guidance, and support throughout this project.

Finally, thanks to staff members of the department of CSE, my parents and friends for their honest opinions and suggestions throughout the course of our mini project.

Avanesh Sharma (4NM21CS038)
Avinash Monteiro (4NM21CS040)
Bhuvanapalli Ameeth Manoj (4NM21CS044)

TABLE OF CONTENTS

ABSTRACT

The purpose of this project is to design lexical analyzer and syntax analyzer for Context Free Grammar. The two stages are an integral part of the Analysis phase of a compilation process which involves identifying the tokens of the given program and using these tokens to identify if each of them is syntactically proper based on given production rules. The main program takes up two parts namely the source program which we need to process and the grammar rules to parse the program. The objective of the project is to generate the parsed sequence which can be further given for the later stages of the compiler.

We have designed a compiler for the given hypothetical program. We generate the parse table which has entries for each terminal and non- terminal identified in them. The final stage is the parsing which is done by using the standard LALR parsing steps.

INTRODUCTION

COMPILER

A compiler is a software that takes a program written in a high-level language and translates it into an equivalent program in a target language. Most specifically a compiler takes a computer program and translates it into an object program. Some other tools associated with the compiler are responsible for making an object program into executable form.

Source program – It is normally written in a high-level programming language. It contains a set of rules, symbols and special words used to construct a computer program. Target program – It is normally the equivalent program in machine code. It contains the binary representation of the instructions that the hardware of the computer can perform.

Error Message – A message issued by the compiler due to detection of syntax errors in the source program. Compilation is a large process. It is often broken into stages. Many phases of the compiler try to optimize by translating one form into a better (more efficient) form. Most of compiling is about "pattern matching" languages and tools that support pattern matching, are very useful. An efficient compiler must preserve semantics of the source program and it should create an efficient version of the target language.

PHASES OF COMPILER

The process of compilation is divided into several phases, each of which transforms the source program from one representation to another.

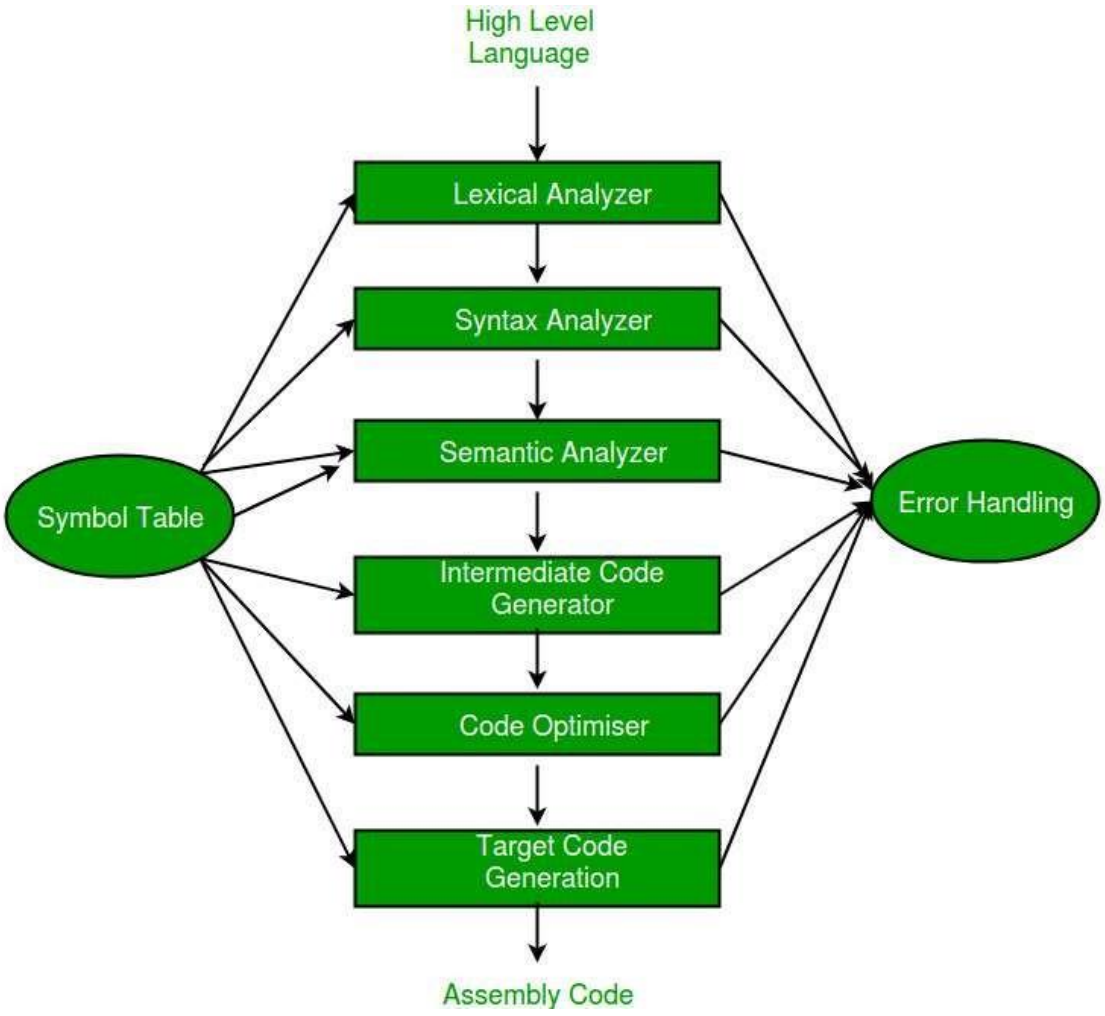

Figure 1: Phases of Compiler

1. Lexical Analysis: This is the first phase of a compiler, also known as scanning. The lexical analyzer reads the stream of characters that make up the source program and groups them into meaningful sequences called lexemes. For each lexeme, the lexical analyzer produces a token in the form of (token-name, attribute-value).

2. Syntax Analysis: This is the second phase of the compiler, also known as parsing. The parser uses the first components of the tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream.

3. 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. It also gathers type information and saves it in either the syntax tree or the symbol table for subsequent use during intermediate-code generation.

4. Intermediate Code Generation: During the translation of a source program into target code, a compiler may construct one or more intermediate representations. Syntax trees are a form of intermediate representation commonly used during syntax and semantic analysis.

5. Code Optimization: The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result. Usually, “better” means faster, but other objectives may be desired, such as shorter code or target code that consumes less power.

6. Code Generation: The code generator takes the intermediate representation of the source program as input and maps it into the target language. If the target language is machine code, registers or memory locations are selected for each of the variables used by the program. Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task.

Each phase of the compiler plays a crucial role in transforming the high-level source code into low-level machine code. The phases are designed to sequentially analyze, transform, and optimize the code to ensure efficient execution.

DESIGN

1.Lexical Analyzer

Lexical analysis is the process of converting a sequence of characters from a source program which is taken as input to a sequence of tokens. A lexical analyzer/lexer/scanner is a program which performs lexical analysis. The roles of lexical analyzer are.

- Reading of input characters from the source program.
- Correlation of error messages with the source program.
- Striping white spaces and comments from the source program
- Expansion macros if found in the source program.
- Entering identified tokens to symbol table.

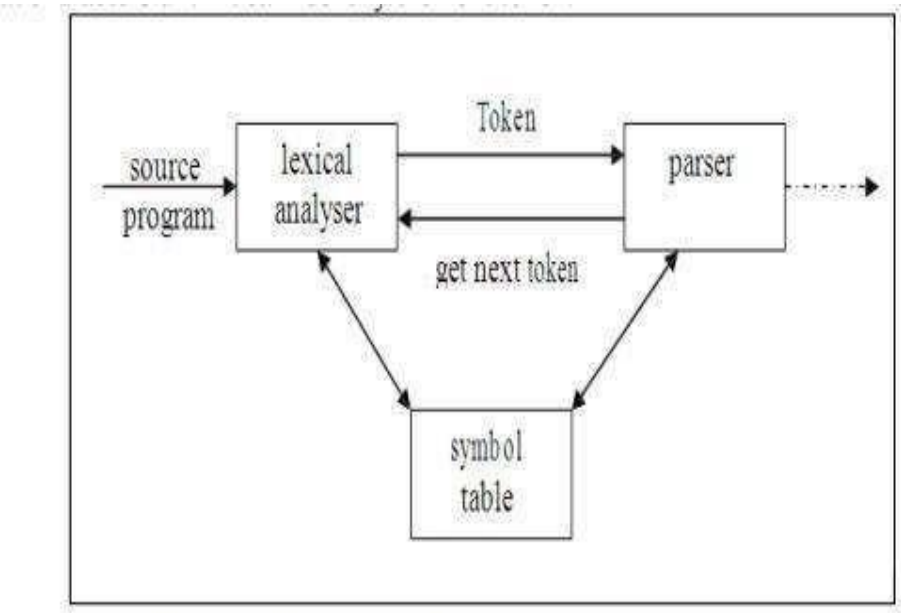

Figure 2: Lexical Analyzer

Problem Statement:

```
BEGIN
        PRINT “HELLO”
        INTEGER A, B, C
        REAL D, E
        STRING X, Y
            A := 2
            B := 4
            C := 6
            D := -3.65E-8
            E := 4.567
        X := “text1”
        Y := “hello there”

        FOR I:=1 TO 5
                PRINT “Strings are [X] and [Y]”
END
```

2.Syntax Analyzer

In our compiler model, the parser obtains a string of tokens from the lexical analyzer, as shown in the figure below, and verifies that the string of token names can be generated by the grammar for the source language. We expect the parser to report any syntax errors in an intelligible fashion and to recover from commonly occurring errors to continue processing the remainder of the program. Conceptually, for well-formed programs, the parser constructs a parse tree and passes it to the rest of the compiler for further processing.

Parser:

Parser is that phase of the compiler which takes token string as input and with the help of existing grammar, converts it into the corresponding parse tree. Parser is also known as Syntax Analyzer. The parser obtains a string of tokens from the lexical analyzer and verifies that the string can be the grammar for the source language. It detects and reports any syntax errors and produces a parse tree from which intermediate code can be generated.

LALR (Look-Ahead LR) parsing is a bottom-up parsing method in compiler design. It uses a set of LR (1) items to build a parsing table, which is then used to parse the input string. LALR parsers are more powerful than LL (1) parsers and can handle a larger class of grammar.

S' -> S $ S -> A | B A -> α X β

B -> γ Y δ X -> a X -> b

Y -> c Y -> €

In the given LALR grammar, the production S -> A | B signifies that the nonterminal S can derive either A or B. The conditions you provided for LL (1) parsing, such as "For no terminal t could A and B derive strings beginning with t" and "At most one of A and B can derive the empty string," are implicit in the structure of the grammar itself. LALR parsing tables consider these conditions during construction to ensure that the parser makes the correct decisions when faced with different possible derivations.

IMPLEMENTATION

Lex Program:

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

%%
"BEGIN" { return BEGIN_; }
"PRINT" { return PRINT; }
"INTEGER" { return INT; }
"STRING" { return STRING; }
"REAL" {return REAL_NUM;}
"TO" { return TO; }
"END" { return END_;}
"FOR" {return FOR;}
[a-zA-Z_][a-zA-Z0-9_]* {return ID;}
-?[0-9]+(\.[0-9]+)? { return NUM; }
-?[0-9]*\.?[0-9]+([eE][+-]?[0-9]+)?  {return REAL_DIG;}
\"([^"]*)\"  {return LIT;}
\r\n {return NL;}
"    " {;}
[ \t]+ {;}
"," { return CM; }
":=" {return EQ; }
. { return yytext[0]; }
%%
```

Figure 3: Lex Program

Yacc Program:

```
%{
    #include<stdio.h>
    #include<stdlib.h>
    extern int yydebug=1;
    extern FILE *yyin;
%}
%token ID NUM NL BEGIN_ PRINT END_ INT FLOAT REAL_NUM STRING CM EQ FOR TO LIT REAL_DIG
%%
S: BEGIN_ NL statement_list END_ { printf("\nValid language\n"); exit(0); };
statement_list: statement NL | statement_list statement NL;
statement: declaration | assignment | loop |  PRINT LIT;
declaration: type ID_list;
type: INT | FLOAT | STRING | REAL_NUM;
ID_list: ID | ID_list CM ID;
assignment: ID EQ expression;
expression: ID | NUM | REAL_DIG | LIT;
loop: FOR ID EQ NUM TO NUM NL statement;
%%
int yyerror(char *msg){
    printf("Invalid input");
    return 1;
}
int yywrap(){
return 1;
}
int main(int argc,char **argv)
{
    yyin=fopen(argv[1],"r");
    yyparse();
    return 0;
}
```

Figure 4: Yacc Program

RESULTS

Commands

yacc -t -g --report=all -d main.y

lex main.l gcc *.c -ll

```
-> $$ = nterm statement_list ()
Stack now 0 1 3
Entering state 12
Reading a token: Next token is token FOR ()
Shifting token FOR ()
Entering state 11
Reading a token: Next token is token ID ()
Shifting token ID ()
Entering state 20
Reading a token: Next token is token EQ ()
Shifting token EQ ()
Entering state 31
Reading a token: Next token is token NUM ()
Shifting token NUM ()
Entering state 34
Reading a token: Next token is token TO ()
Shifting token TO ()
Entering state 36
Reading a token: Next token is token NUM ()
Shifting token NUM ()
Entering state 37
Reading a token: Next token is token NL ()
Shifting token NL ()
Entering state 38
Reading a token: Next token is token PRINT ()
Shifting token PRINT ()
Entering state 6
Reading a token: Next token is token LIT ()
Shifting token LIT ()
Entering state 19
Reducing stack by rule 7 (line 12):
```

Figure 6: String parsing Validation

```
   $5 = token TO ()
   $6 = token NUM ()
   $7 = token NL ()
   $8 = nterm statement ()
-> $$ = nterm loop ()
Stack now 0 1 3 12
Entering state 17
Reducing stack by rule 6 (line 12):
   $1 = nterm loop ()
-> $$ = nterm statement ()
Stack now 0 1 3 12
Entering state 22
Reading a token: Next token is token NL ()
Shifting token NL ()
Entering state 32
Reducing stack by rule 3 (line 11):
   $1 = nterm statement_list ()
   $2 = nterm statement ()
   $3 = token NL ()
-> $$ = nterm statement_list ()
Stack now 0 1 3
Entering state 12
Reading a token: Next token is token END_ ()
Shifting token END_ ()
Entering state 21
Reducing stack by rule 1 (line 10):
   $1 = token BEGIN_ ()
   $2 = token NL ()
   $3 = nterm statement_list ()
   $4 = token END_ ()

Valid language
```

Figure 7: String parsing validation result

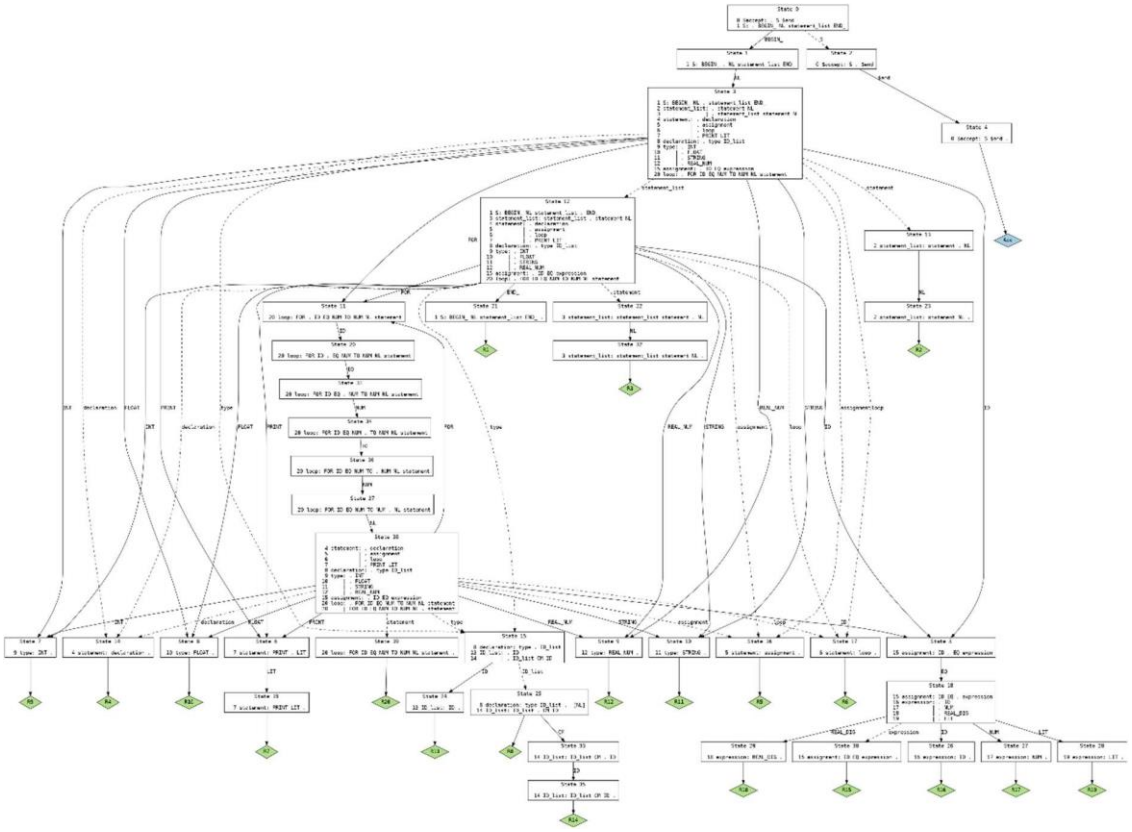

Figure 8: String Validation

CONCLUSION

The lexical analyzer processes the code file as input and generates tokens based on the input stream. In this context, we have implemented LALR parsing for the specified grammar. The grammar serves as the foundation for parsing statements. Upon successful parsing of the input string, the system outputs a success message. However, if the parsing process encounters an error, a syntax error is indicated along with the corresponding line number where the error occurred.

REFERENCES

- https://www.geeksforgeeks.org/introduction-of-compiler-design/
- https://www.tutorialspoint.com/compiler_design/compiler_design_types_of_parsing.htm
- https://byjus.com/gate/lexical-analysis/#:~:text=on%20Lexical%20Analysis-,What%20is%20Lexical%20Analysis%3F,whitespace%20in%20the%20source%20code.
- https://core.ac.uk/download/pdf/214452802.pdf