**DESIGN AND DEVELOPMENT OF LEXICAL AND PARSER**

**PHASE FOR VALID IF STATEMENTS**

A Mini Project Report Submitted by

NIVEDITHA G POOJARY PRAJNA V

4NM17CS117 4NM17CS128

UNDER THE GUIDANCE OF

**Mr. Ranjan Kumar H S**

Assistant Professor

Department of Computer Science and Engineering

in partial fulfilment of the requirements for the award of the Degree of

Bachelor of Engineering in

Computer Science & Engineering

from

Visvesvaraya Technological University, Belgaum



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

N.M.A.M. INSTITUTE OF TECHNOLOGY

(An Autonomous Institution under VTU, Belgaum) (AICTE approved, NBA Accredited, ISO 9001:2008 Certified) NITTE -574 110, Udupi District, KARNATAKA

**April 2020**



(ISO 9001:2015 Certified), Accredited with ‘A’ Grade by NAAC

: 08258 - 281039 – 281263, Fax: 08258 – 281265

**Department of Computer Science and Engineering**

B.E. CSE Program Accredited by NBA, New Delhi from 1-7-2018 to 30-6-2021

**DEPARTMENT OF COMPUTER SCIENCE ANDENGINEERING**

# CERTIFICATE

“**DESIGN AND DEVELOPMENT OF LEXICAL AND PARSER PHASE FOR VALID IF STATEMENTS”**

is a bonafide work carried out by

NIVEDITHA G POOJARY PRAJNA V

(4NM17CS117 ) (4NM17CS128)

in partial fulfilment of the requirements for the award of Bachelor of Engineering Degree in Computer Science and Engineeringprescribed by Visvesvaraya Technological University, Belgaum during the year 2019-2020.

It is certified that all corrections/suggestions indicated for Internal

Assessment have been incorporated in the report.

The Mini project report has been approved as it satisfies the academic requirements in respect of the project work prescribed for the Bachelor of

Engineering Degree.

Signature of Guide Signature of HOD

# **ACKNOWLEDGEMENT**

We believe that our project will be complete only after we thank the people who have contributed to make this project successful.

First and foremost, our sincere thanks to our beloved principal, **Dr. Niranjan N. Chiplunkar** for giving us an opportunity to carry out our project work at our college and providing us with all the needed facilities.

We express our deep sense of gratitude and indebtedness to our guide**Mr. Ranjan Kumar H S**, Assistant Professor, Department of Computer Science and Engineering, for his inspiring guidance, constant encouragement, support and suggestions for improvement during the course of our project.

We sincerely thank **Dr. K.R. Udaya Kumar Reddy**, Head of Department of Computer Science and Engineering, Nitte Mahalinga Adyantaya Memorial Institute of Technology, Nitte.

We also thank all those who have supported us throughout the entire duration of our project.

Finally, we thank the staff members of the Department of Computer Science and Engineering and all our friends for their honest opinions and suggestions throughout the course of our project.

Niveditha G Poojary ( 4NM17CS117 )

Prajna V ( 4NM17CS128 )

**ABSTRACT**

Compiler design is ideas used in construction of programming language compilers. Program written in high level programming language and designed for humans understanding is systematically converted into low level assembly language understood by machines.

In our project, we designed have two phases of the compiler i.e. lexical analyser and parser. We wrote code for design of these two phases for the given problem statement. First ,the lexical analyser generates the tokens and the parser checks if the input is grammatically correct as per the syntax written in the programme and displays the success or failure message accordingly.

**TABLE OF CONTENTS**

|  |  |
| --- | --- |
| Contents | Page No. |
| 1. Introduction | 6 |
| 1.1 The phases of compiler | 6 |
| 2. Project Description | 9 |
| 2.1 Lexical Analysis | 9 |
| 2.2 Syntax Analysis | 10 |
| 3.Objective of the Project | 11 |
| 4.Implementation | 12 |
| 5.Result | 14 |
| 6.Conclusion | 16 |
| 7.References | 16 |

**INTRODUCTION**

Programming languages are notations for describing computations to people and to machines. The world as we know it depends on programming language, because all the software running on all the computers was written in some programming language. But, before a program can be run , it first must be translated into a form in which it can be executed by a computer. The software systems that do this translation are called compilers. So, we can tell that a compiler is a software which converts a program written in high level language(source language) to low level language(target/object/machine language).

An important role of the compiler is to report any errors in the source program that it detects during the translation process. Compilers are large, complex, carefully engineered objects. While many issues in compiler design are amendable to multiple solutions and interpretations,

there are two fundamental principles-

1. The compiler must preserve the meaning of the program being compiled.
2. The compiler must improve the input program in some discernible way.

**Uses of compiler technology:**

* **Code generation:** To translate a program in a high-level language to machine code for particular processor
* **Optimization:** Improve program performance for a given target machine
* **Interpreters:** “On-the-fly” translation of code, ex: Java, Perl, Postscript
* **Debugging aids:** Purify for debugging memory access errors

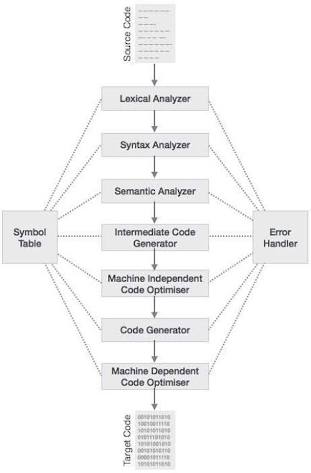
**THE PHASES OF COMPILER**

The compilation process is a sequence of various phases. Each phase takes input from its previous stage, has its own representation of source program, and feeds its output to the next phase of the compiler.

**There are 6 phases in a compiler:**

1. Lexical Analysis
2. Syntax Analysis
3. Semantic Analysis
4. Intermediate Code Generation
5. Code Optimization
6. Code Generation

Each of this phase helps in converting the high-level language to the machine code. All these phases convert the source code by dividing into tokens, creating parse trees, and optimizing the source code by different phases.



**Phases Of Compiler**

**Lexical Analysis**

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 of the form

<token-name, attribute-value>

that it passes on to the subsequent phase, syntax analysis. In the token, the first component token-name is an abstract symbol that is used during syntax analysis, and the second component attribute-value points to an entry in the symbol table for this token. Information from the symbol-table entry is needed for semantic analysis and code generation.

**Syntax Analysis**

The second phase of the compiler is syntax analysis or 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. A typical representation is a syntax tree in which each interior node represents an operation and the children of the node represent the arguments of the operation.

**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, forsubsequent use during intermediate-code generation.

An important part of semantic analysis is type checking, where the compiler checks that each operator has matching operands. For example, many programming language definitions require an array index to be an integer; the compiler must report an error if a floating-point number is used to index an array.

The language specification may permit some type conversions called coercions. For example, a binary arithmetic operator may be applied to either a pair of integers or to a pair of floating-point numbers. If the operator is applied to a floating-point number and an integer, the compiler may convert or coerce the integer into a floating-point number.

**Intermediate Code Generation**

In the process of translating a source program into target code, a compiler may construct one or more intermediate representations, which can have a variety of forms. Syntax trees are a form of intermediate representation; they are commonly used during syntax and semantic analysis.

After syntax and semantic analysis of the source program, many compilers generate an explicit low-level or machine-like intermediate representation, which we can think of as a program for an abstract machine. This intermediate representation should have two important properties: it should be easy to produce and it should be easy to translate into the target machine.

**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.

**Code Generation**

The code generator takes as input an intermediate representation of the source program 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. A crucial aspect of code generation is the judicious assignment of registers to hold variables.

**Symbol Table**

An essential function of a compiler is to record the variable names used in the source program and collect information about various attributes of each name. These attributes may provide information about the storage allocated for a name, its type, its scope (where in the program its value may be used), and in the case of procedure names, such things as the number and types of its arguments, the method of passing each argument (for example, by value or by reference), and the type returned.

The symbol table is a data structure containing a record for each variable name, with fields for the attributes of the name. The data structure should be designed to allow the compiler to find the record for each name quickly and to store or retrieve data from that record quickly.

**PROJECT DESCRIPTION**

**LEXICAL ANALYSIS**

When the input string (source code or a program in some language) is givrn to a compiler,the compiler processes t in several phases,starting from lexical analysis(scans the input and divides it into tokens) to get target code generation.

Lexical analyzer converts stream of input characters into a stream of tokens.It reads the input source code character by character,recognizes the lexemes and outputs a sequence of tokens describing the lexemes.

**LEXEME**

A lexeme is a sequence of characters in the source program that matces the pattern for a token and is identified by the lexical analyzers as an instance of that token.

**TOKEN**

A lexical token or simply token is a string with an assigned and thus identified meaning.It is structured as a pair consisting of a token name and an optional token value.The token name is a category of lexical unit.Common token names are

* Identifier : names the programmer chooses;
* keyword : names already in the programming language;
* separator ( also kmown as punctuators ): punctuation characters and paired-delimiters;
* operator : symbols that operate on arguments and produce results;
* literal : numeric, logical, textual, reference literals;
* comment : line, block.

The different tokens that our lexical analyzer identifies are as follows:

* KEYWORDS :int, if, end, begin, printf
* OPERATORS : >, <, ==, >=, <=, !=
* IDENTIFIERS : main , n1, n2, n3 etc
* SEPARATORS : ‘(’ , ’)’ , ’;’ , ‘,’

**PATTERN**

A pattern is a description which is used by the token. In the case of a keyword which uses a token, the pattern is a sequence of characters.

**ADVANTAGES OF LEXICAL ANALYSIS**

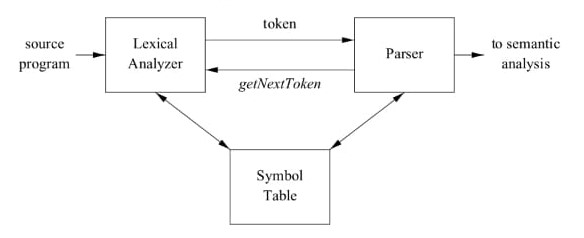
* Lexical analyzer method is used by programs like compilers which can use the parsed data from a programmer’s code to create a compiled binary executable code
* A separate lexical analyzer helps you to construct a specialized and potentially more efficient processor for the task
* It is used by web browsers to format and display a web page with the help of parsed data from Javascript, HTML, CSS

**SYNTAX ANALYSIS**

Syntax Analysis or Parsing is the second phase, i.e. after lexical analysis. It checks the syntactical structure of the given input, i.e. whether the given input is in the correct syntax (of the language in which the input has been written ) or not. It does so by building a data structure , called a Parse tree or Syntax tree.

The parse tree is constructed by using the pre-defined Grammar of the language and the input string. If the given input string can be produced with the help of the syntax tree (in the derivation process),the input string is found to be in the correct syntax. If not, error is reported by the syntax analyzer.

**INTERACTIONS BETWEEN THE LEXICAL ANALYZER AND THE PARSER**



The lexical analyzer works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer

when it demands.Commonly, the interaction is implemented by having the parser call the lexical analyzer. The call, suggested by the getNextToken command, causes the lexical analyzer to read chaaracters from its input until the next lexeme and produce for it the next token, which it returns to the parser.

**PARSING TECHNIQUES**

* **Top-Down Parsing**

In the top-down parsing, construction of the parse tree starts at the root and then proceeds towards the leaves.

* **Bottom-Up Parsing**

In the bottom-up parsing technique, the construction of the parse tree starts with the leaves, and then it processes towards it’s root.

**NEED OF SYNTAX ANALYZER**

* Check if the code is valid grammatically
* The syntactical analyzer helps you to apply rules to the code
* Helps you to make sure that each opening brace has a corresponding closing brace
* Each declaration has a type and that type must exist

**OBJECTIVE OF THE PROJECT**

**AIM OF THE PROJECT**

To design a compiler ( Lexical and Parser phase ) for the given hypothetical language.

**GOALS**

* To create tokens from the given input stream
* To check whether the given input is in the correct syntax ( of the language in which the input has been written ) or not
* Incase the given input is syntactically incorrect, display the line number where the error has occurred

**SCOPE OF THE PROJECT**

Lexical analyzer converts the input program into character stream of valid words of language, knows as tokens.

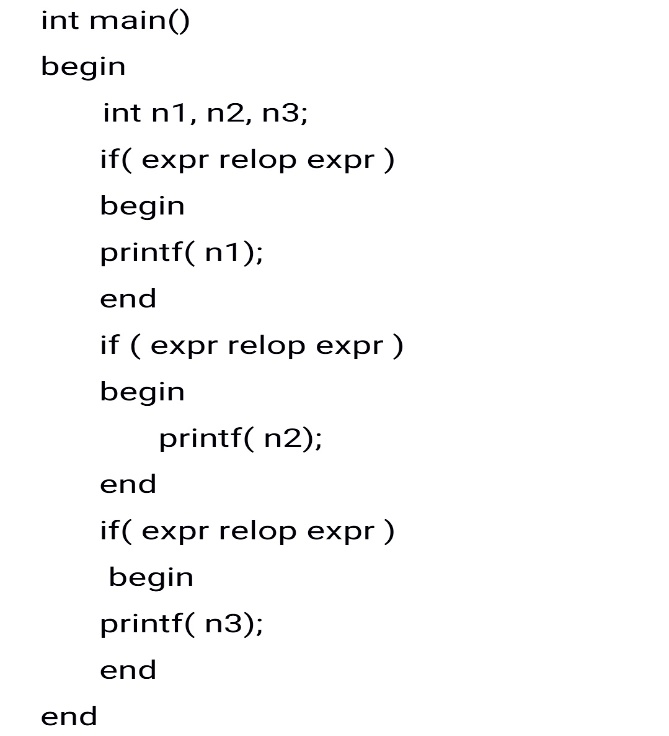
The parser looks into the sequence of these tokens and identifies the language construct occuring in the input program. The parser and the lexical analyer work hand in hand; in the sense that whenever the parser needs further tokens to proceed , it requests the lexical analyzer. The lexical analyzer in turn scans the remaining input stream and returns the next token occuring there.

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. The parser accomplishes two tasks, i.e., parsing the code, looking for errors and generating a parse tree as the output of the phase.

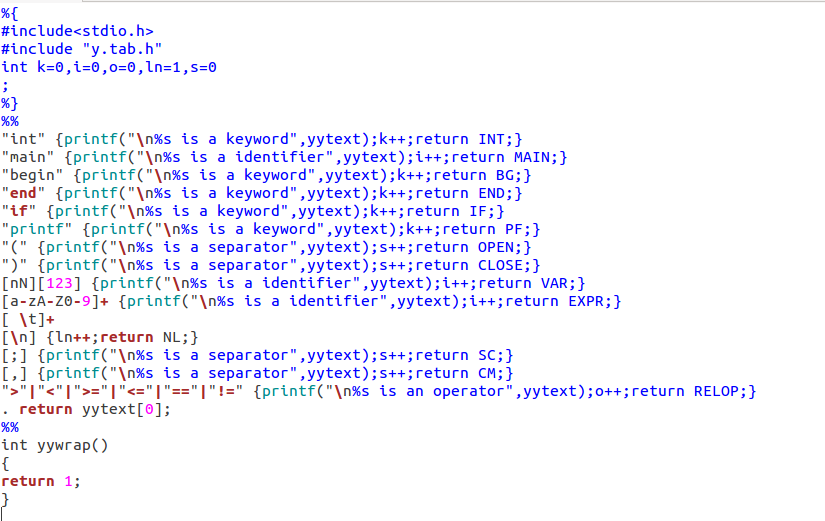
**IMPLEMENTATION**

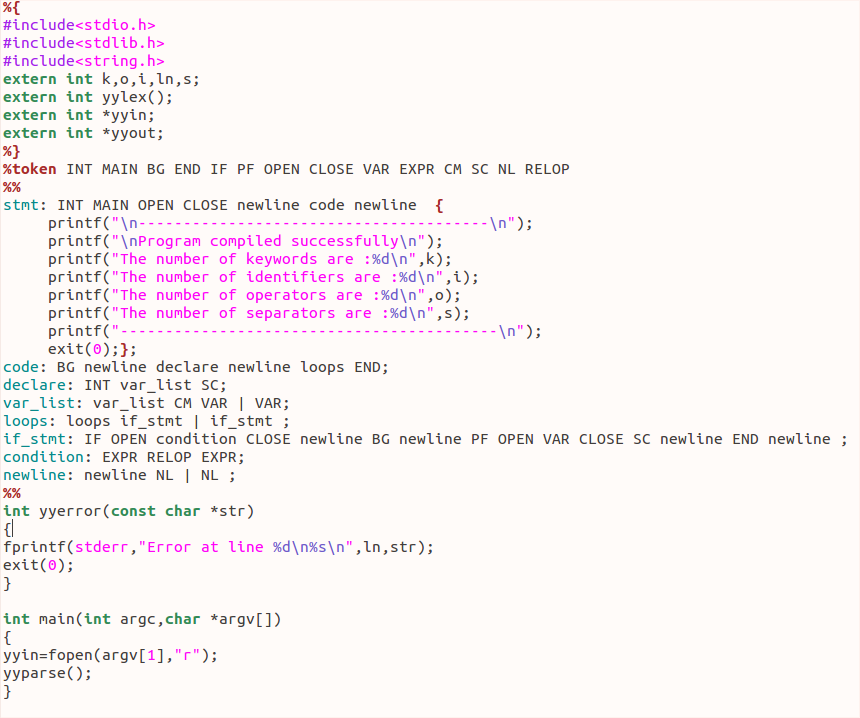
**Problem Statement**

Design a compiler(Lexical and Parser phase) for-



**Lex Program for the above Problem Statement**



**Yacc Program for the above Problem Statement**

**EXPLANATION OF THE CODE**

The lex part of the code specifies the Lex command specification file that defines the lexical analysis rules.The file contains include statements for standard input and output as well as for the y.tab.h file. Variables k, i, o, s and ln are declared for counting the number of keywords, identifiers, operators, separators and new lines respectively.When the file containing the problem statement is given as input to the program, the rules in lex recognizes keywords, identifiers, operators, separators and newlines and respective tokens are generated to pass on to yacc program.The variables declared above are incremented as an action for a rule.The y.tab.h file contains definitions for the tokens that the parser program uses.

The yacc part of the code specifies the yacc command grammar file that defines the parsing rules, and calls the yylex subroutine created by the lex command to provide input.

The file contains the following sections:

* **Declaration section**

This section contains entries that:

* Include standard I/O header file
* Declarations for any variables or constants used in other parts of the grammar file
* **Rules section**

The rules section defines the rules that parse the input stream .

* %token-Lists the tokens which come from lex tool with their type
* **Program section**

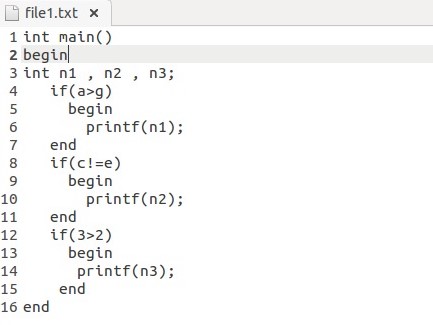
The program section contains the following subroutines:

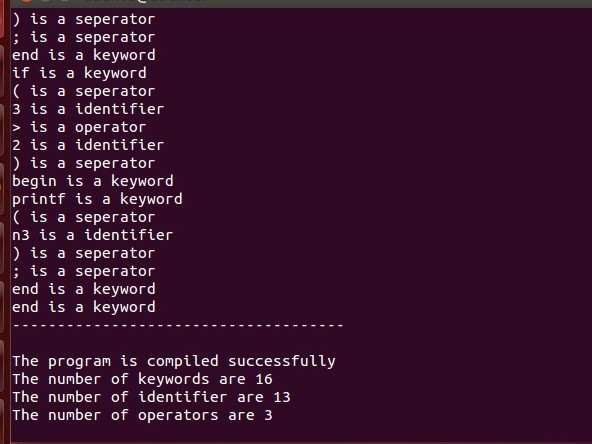
* main : The required main program that calls the yyparse subroutines to start the program
* yyerror(str) : The error-handling subroutine only prints a syntax error message.It uses the variable ln to print the line number where the error has occurred.
* File \*yyin : Input file for lex program and defaukts to stdin

If the input statement is as per the defined productions then the program prints “Program compiled successfully “ message with the number of keywords, identifiers and operators.If the given input is syntactically incorrect, the program prints “Error at line (the line number where the error has occurred)” along with the message “syntax error”.

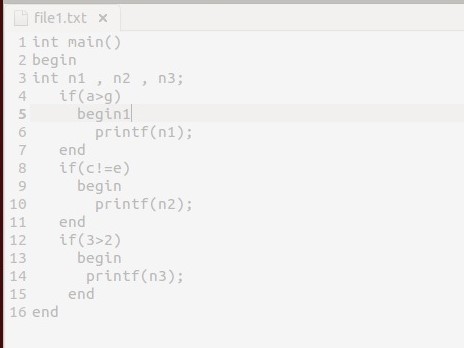
**RESULT**

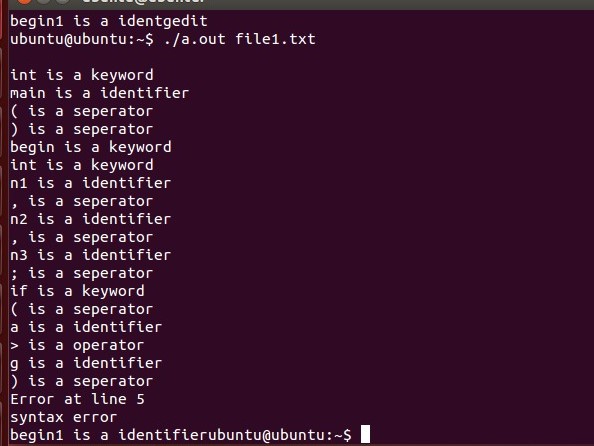
On running the above code for the given problem statement, we get the following output.



****

**Output for syntactically wrong input**

****

****

**CONCLUSION**

In lexical analysis,when we give a program statement as input,the keywords, identifiers, operators and separators are displayed. Each one of these are the tokens and then given as input to the parser.

In parser, the input is parsed as per the grammar given in the program. If the input was parsed completely then the success message is displayed in the output else a syntax error is displayed along with the line number.

**REFERENCES**

* Lexical analysis-Wikipedia

<https://en.wikipedia.org/wiki/Lexical_analysis>

* Compiler Design-Phases of Compiler-Tutorialspoint

<https://www.tutorialspoint.com/compiler_design/compiler_design_phases_of_compiler.htm>