DESIGN AND DEVELOPMENT OF LEXER AND PARSER FOR VALID CODE SNIPPET

MINI PROJECT REPORT SUBMITTED BY

Xylene Vinitha Dsouza
4NM17CS211
VI Semester, D Section

Swetha S
4NM17CS199
VI Semester, D section

UNDER THE GUIDANCE OF

Mr. Sannidhan M.S.
Assistant Professor Gd II
Department of Computer Science and Engineering

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE AWARD OF THE DEGREE OF

Bachelor of Engineering in Computer Science & Engineering

from

Visvesvaraya Technological University, Belagavi

N.M.A.M. INSTITUTE OF TECHNOLOGY

(An Autonomous Institution under VTU, Belgaum)
AICTE approved, (ISO 9001:2015 Certified), Accredited with 'A' Grade by NAAC
NITTE -574 110, Udupi District, KARNATAKA.

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

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

June 2020

NITTE EDUCATION TRUST | N.M.A.M. INSTITUTE OF TECHNOLOGY
(An Autonomous Institution affiliated to Visvesvaraya Technological University, Belagavi)
Nitte – 574 110, Karnataka, India

(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

CERTIFICATE

"Design and development of lexer and parser for valid code snippet" is a bonafide work carried out by Xylene Vinitha Dsouza (4NM17CS211) and Swetha S (4NM17CS199) in partial fulfilment of the requirements for the award of Bachelor of Engineering Degree in Computer Science and Engineering prescribed by Visvesvaraya Technological University, Belagavi 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 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 express our deep sense of gratitude and indebtedness to our guide Mr. Sannidhan M.S., Assistant Professor GD II, Department of Computer Science and Engineering, for her inspiring guidance, constant encouragement, support and suggestions for improvement during the course of our project.
We thank all the teaching and non-teaching staff members of the Computer Science and Engineering Department and our parents and friends for their honest opinions and suggestions throughout the course of our project.
Finally, we thank all those who have supported us directly or indirectly throughout the project and making it a grand success.

Xylene Vinitha Dsouza
(4NM17CS211)

Swetha S
(4NM17CS199)

ABSTRACT

Computers are balanced mixture of software and hardware. Hardware is just a piece of mechanical device and its functions are being controlled by a compatible software .Hardware understands instructions in the form of electronic charge, which is counterpart of binary language in software programming. Binary language has only two characters ‘0’ and ‘1’ which forms the low level language. Compiler is a software which converts a program written in high level language(Source Language) to low level language(Object/Target/Machine Language).

LANGUAGE PROCESSING SYSTEM

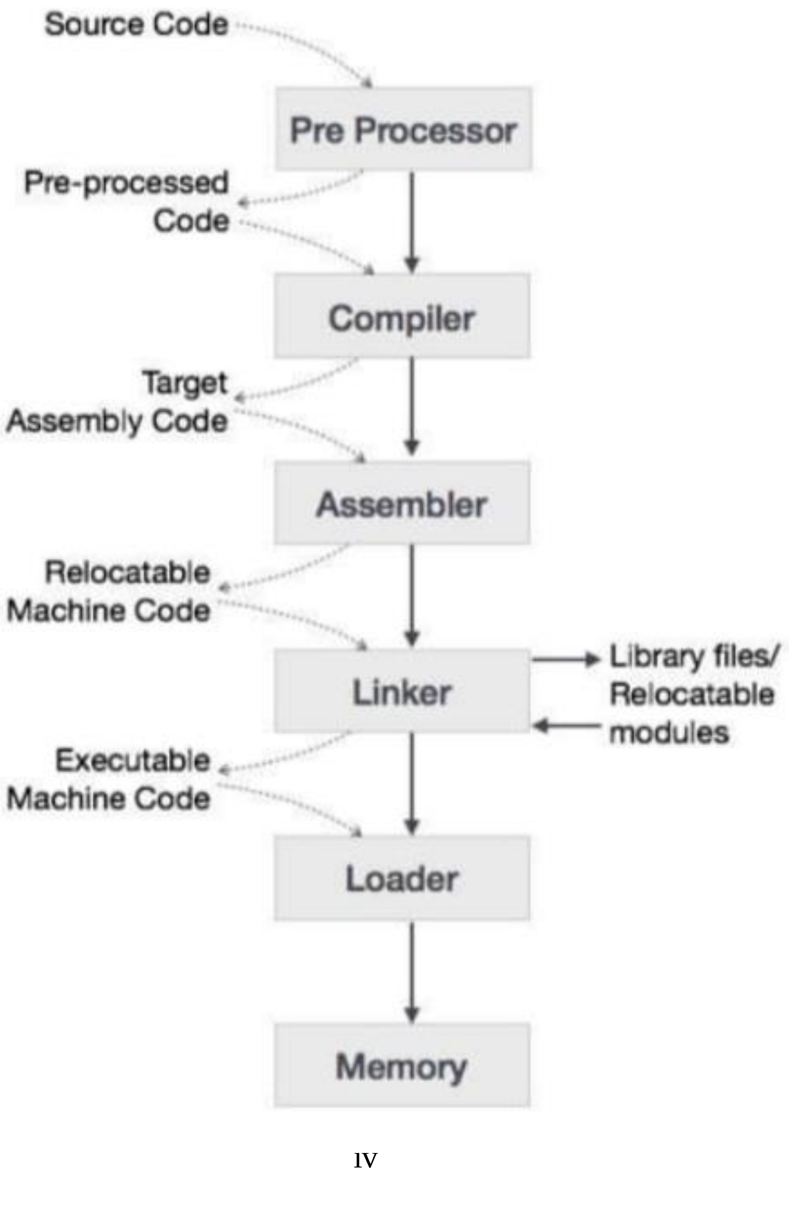


TABLE OF CONTENTS

INTRODUCTION

What is a compiler?

In order to reduce the complexity of designing and building computers, nearly all of these are made to execute relatively simple commands (but do so very quickly). A program for a computer must be built by combining these very simple commands into a program in what is called machine language. Since this is a tedious and error prone process most programming is, instead, done using a high-level programming language. This language can be very different from the machine language that the computer can execute, so some means of bridging the gap is required. This is where the compiler comes in. A compiler translates (or compiles) a program written in a high-level programming language that is suitable for human programmers into the low-level machine language that is required by computers. During this process, the compiler will also attempt to spot and report obvious programmer mistakes. Using a high-level language for programming has a large impact on how fast programs can be developed.

The main reasons for this are:

- Compared to machine language, the notation used by programming languages is closer to the way humans think about problems.
- The compiler can spot some obvious programming mistakes.
- Programs written in a high-level language tend to be shorter than equivalent programs written in machine language.
- Another advantage of using a high-level level language is that the same program can be compiled to many different machine languages and, hence, be brought to run on many different machines.
- On the other hand, programs that are written in a high-level language and automatically translated to machine language may run somewhat slower than programs that are hand-coded in machine language. Hence, some time critical programs are still written partly in machine language.
- A good compiler will, however, be able to get very close to the speed of handwritten machine code when translating well-structured programs.

THE PHASES OF A 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. Let us understand the phases of a compiler.

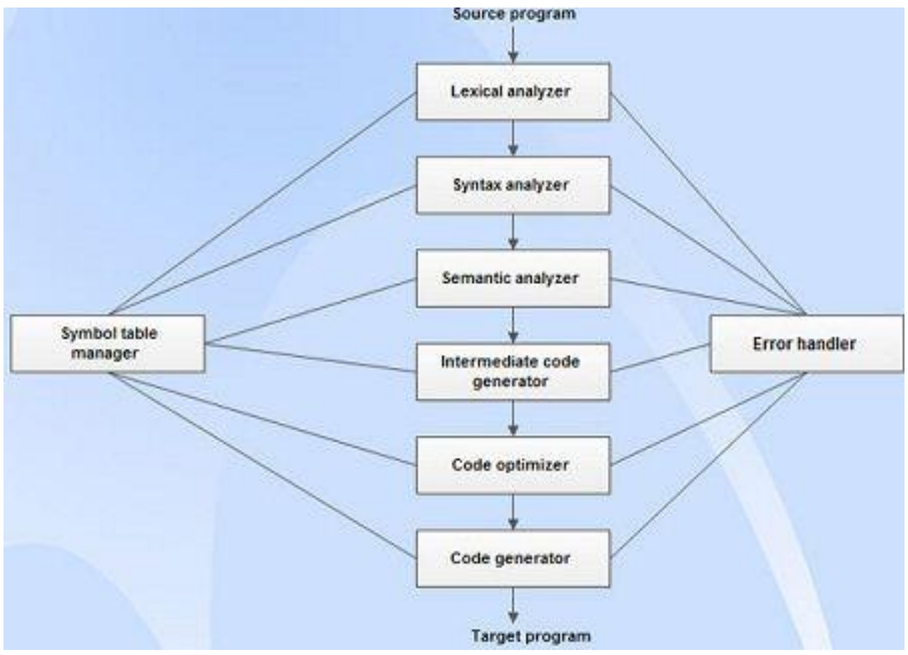


Fig: Phases of Compiler

Lexical Analysis

The first phase of scanner works as a text scanner. This phase scans the source code as a stream of characters and converts it into meaningful lexemes. Lexical analyser represents these lexemes in the form of tokens as:

<token-name, attribute-value>

Syntax Analysis

The next phase is called the syntax analysis or **parsing**. It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree). In this 3 phase, token arrangements are checked against the source code grammar, i.e., the parser checks if the expression made by the tokens is syntactically correct.

Semantic Analysis

Semantic analysis checks whether the parse tree constructed follows the rules of language. For example, assignment of values is between compatible data types, and adding string to an integer. Also, the semantic analyzer keeps track of identifiers, their types and expressions; whether identifiers are declared before use or not, etc. The semantic analyzer produces an annotated syntax tree as an output.

Intermediate Code Generation

After semantic analysis, the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in between the high-level language and the machine language. This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code.

Code Optimization

The next phase does code optimization of the intermediate code. Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory).

Code Generation

In this phase, the code generator takes the optimized representation of the intermediate code and maps it to the target machine language. The code generator translates the intermediate code into a sequence of (generally) re-locatable machine code. Sequence of instructions of machine code performs the task as the intermediate code would do.

SYMBOL TABLE

It is a data-structure maintained throughout all the phases of a compiler. All the identifiers' names along with their types are stored here. The symbol table makes it easier for the compiler to quickly search the identifier record and retrieve it. The symbol table is also used for scope management.

LEXICAL ANALYSIS

Lexical analysis is the first phase of a compiler. It takes the modified source code from language pre-processors that are written in the form of sentences. The lexical analyser breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code. If the lexical analyser finds a token invalid, it generates an error. The lexical analyser works closely with the syntax analyser. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyser when it demands. The main purpose of lexical analysis is to make life easier for the subsequent syntax analysis phase. In theory, the work that is done during lexical analysis can be made an integral part of syntax analysis, and in simple systems this is indeed often done. However, there are reasons for keeping the phases separate:

• Efficiency: A lexer may do the simple parts of the work faster than the more general parser can. Furthermore, the size of a system that is split in two may be smaller than a combined system. This may seem paradoxical but, as we shall see, there is a non- linear factor involved which may make a separated system smaller than a combined system.

• Modularity: The syntactical description of the language need not be cluttered with small lexical details such as white-space and comments.

• Tradition: Languages are often designed with separate lexical and syntactical phases in mind, and the standard documents of such languages typically separate lexical and syntactical elements of the languages.

Token:

Token is a sequence of characters that can be treated as a single logical entity.
Typical tokens are,

1) Identifiers
2) keywords
3) operators
4) special symbols
5)constants

Pattern:

A set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a pattern associated with the token.

Lexeme:

A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.

Problem statement: Design a compiler for the following pseudocode

```
X:integer;
Procedure foo ( b : integer )
b:=13;
if x=12 and b=13 then
printf("by copy");
elseif x=13 and b=13 then
printf("by address");
else
printf("a mystery");
end if;
end foo  gedit
```

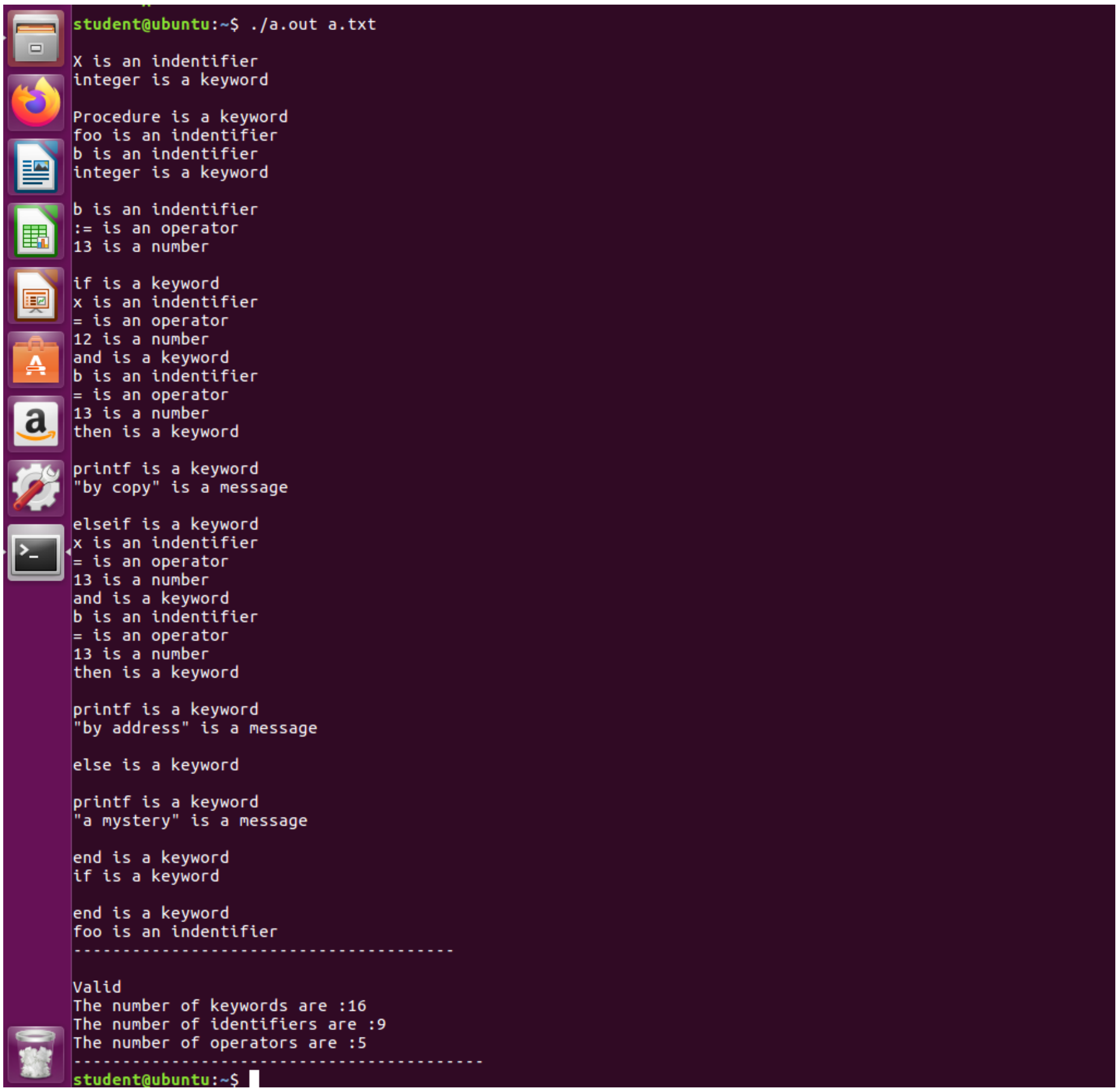
LEXICAL PROGRAM FOR THE ABOVE PROBLEM STATEMENT

```
%{
  #include<stdio.h>
  #include "y.tab.h"
  int k=0,i=0,o=0;
%}
%%
"(" {return OP1;}
")" {return OP2;}
"if" {printf("\n%s is a keyword",yytext);k++;return IF;}
"else" {printf("\n%s is a keyword",yytext);k++;return ELSE;}
"elseif" {printf("\n%s is a keyword",yytext);k++;return ELSEIF;}
"then" {printf("\n%s is a keyword",yytext);k++;return THEN;}
"printf" {printf("\n%s is a keyword",yytext);k++;return PRINTF;}
"and" {printf("\n%s is a keyword",yytext);k++;return AND;}
"integer" {printf("\n%s is a keyword",yytext);k++;return INTEGER;}
"Procedure" {printf("\n%s is a keyword",yytext);k++;return PROC;}
"end" {printf("\n%s is a keyword",yytext);k++;return END;}
["][a-zA-Z0-9_\- ]*["] {printf("\n%s is a message",yytext);return MSG;}
[a-zA-Z][a-zA-Z0-9_]*  {printf("\n%s  is  an  identifier",yytext);i++;return
ID;}
[0-9]* {printf("\n%s is a number",yytext);return NUM;}
":=" {printf("\n%s is an operator",yytext);o++;return EQQ;}
"=" {printf("\n%s is an operator",yytext);o++;return EQ;}
":" {return C;}
";" {return SC;}
. {;}
%%
int yywrap()
{
  return 1;
}
```

Explanation for the above code

In the code above, variables k, o and i are the variables declared for counting number of keywords, operators and identifiers respectively.
When a file containing the problem statement is given as input to the program, the rules in lex recognizes identifiers, keywords, numbers and operators etc and respective tokens are generated to pass on to yacc program and variables declared above are incremented as an Action for a rule.

<u>Output:</u>



```
student@ubuntu:~$ ./a.out a.txt

X is an indentifier
integer is a keyword

Procedure is a keyword
foo is an indentifier
b is an indentifier
integer is a keyword

b is an indentifier
:= is an operator
13 is a number

if is a keyword
x is an indentifier
= is an operator
12 is a number
and is a keyword
b is an indentifier
= is an operator
13 is a number
then is a keyword

printf is a keyword
"by copy" is a message

elseif is a keyword
x is an indentifier
= is an operator
13 is a number
and is a keyword
b is an indentifier
= is an operator
13 is a number
then is a keyword

printf is a keyword
"by address" is a message

else is a keyword

printf is a keyword
"a mystery" is a message

end is a keyword
if is a keyword

end is a keyword
foo is an indentifier
---------------------------------------

Valid
The number of keywords are :16
The number of identifiers are :9
The number of operators are :5
------------------------------------------
student@ubuntu:~$
```

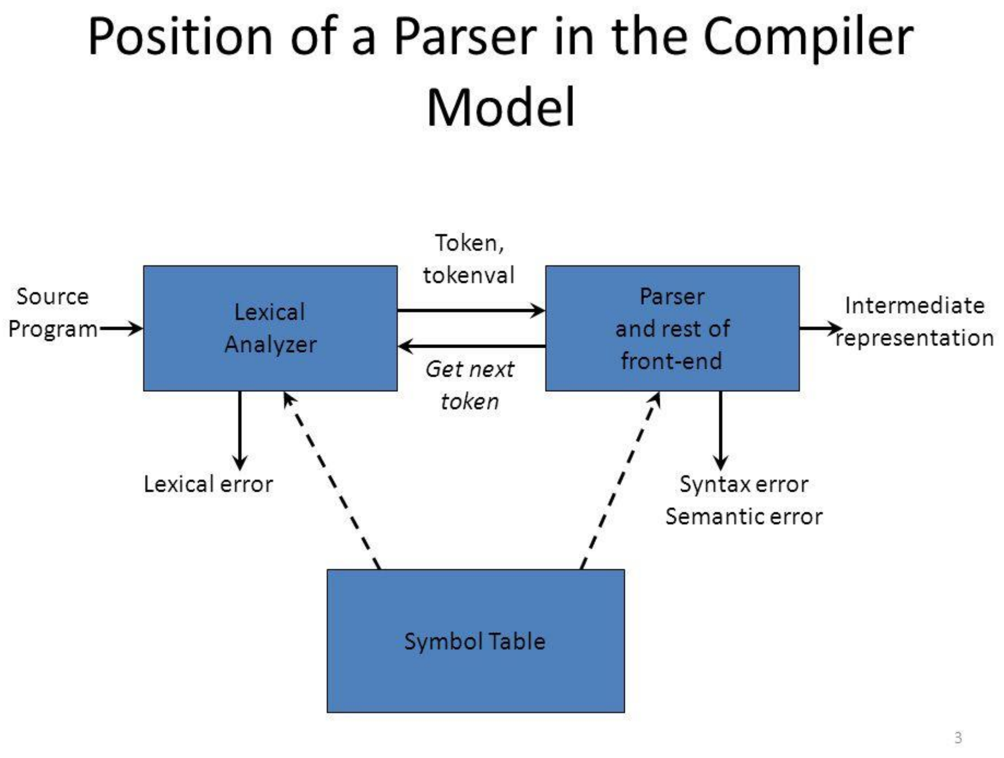
SYNTAX ANALYSIS

Syntax analysis or parsing is the second phase of a compiler. In this chapter, we shall learn the basic concepts used in the construction of a parser.

Role of the Parser

In the compiler model, the parser obtains a string of tokens from the lexical analyser, and verifies that the string can be generated by the grammar for the source language.

The parser returns any syntax error for the source language.



There are three general types parsers for grammars.

- Universal parsing methods such as the Cocke-Younger-Kasami algorithm and Earley's algorithm can parse any grammar. These methods are too inefficient to use in production compilers.
- The methods commonly used in compilers are classified as either top-down parsing or bottom-up parsing.
- Top-down parsers build parse trees from the top (root) to the bottom (leaves).
- Bottom-up parsers build parse trees from the leaves and work up to the root.

In both case input to the parser is scanned from left to right, one symbol at a time.
The output of the parser is some representation of the parse tree for the stream of tokens.

There are number of tasks that might be conducted during parsing. Such as;

- Collecting information about various tokens into the symbol table.
- Performing type checking and other kinds of semantic analysis.
- Generating intermediate code.
- Syntax Error Handling:
- Planning the error handling right from the start can both simplify the structure of a compiler and improve its response to errors.

The program can contain errors at many different levels. e.g.,

- Lexical – such as misspelling an identifier, keyword, or operator.
- Syntax – such as an arithmetic expression with unbalanced parenthesis.
- Semantic – such as an operator applied to an incompatible operand.
- Logical – such as an infinitely recursive call.

Much of the error detection and recovery in a compiler is centered on the syntax analysis phase.
One reason for this is that many errors are syntactic in nature or are exposed when the stream of tokens coming from the lexical analyser disobeys the grammatical rules defining the programming language.
Another is the precision of modern parsing methods; they can detect the presence of syntactic errors in programs very efficiently.

The error handler in a parser has simple goals:

- It should the presence of errors clearly and accurately.
- It should recover from each error quickly enough to be able to detect subsequent errors.
- It should not significantly slow down the processing of correct programs.

Error-Recovery Strategies:

There are many different general strategies that a parser can employ to recover from a syntactic error.

- Panic mode
- Phrase level
- Error production
- Global correction

CONTEXT-FREE GRAMMAR

In this section, we will first see the definition of context-free grammar and introduce terminologies used in parsing technology.

A context-free grammar has four components:

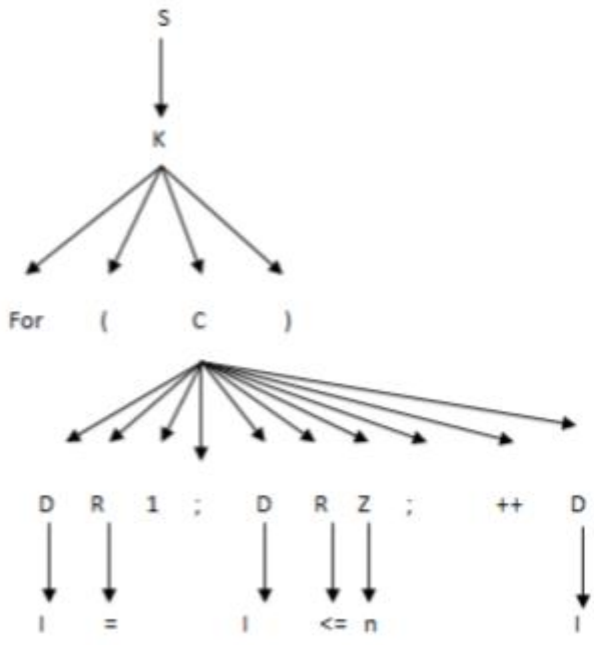
• A set of non-terminals (V). Non-terminals are syntactic variables that denote sets of strings. The non-terminals define sets of strings that help define the language generated by the grammar.
• A set of tokens, known as terminal symbols (Σ). Terminals are the basic symbols from which strings are formed.
• A set of productions (P). The productions of a grammar specify the manner in which the terminals and non-terminals can be combined to form strings. Each production consists of a non-terminal called the left side of the production, an arrow, and a sequence of tokens and/or on- terminals, called the right side of the production.
• One of the non-terminals is designated as the start symbol (S); from where the production begins. The strings are derived from the start symbol by repeatedly replacing a non-terminal (initially the start symbol) by the right side of a production, for that non-terminal.

Parse Tree

A parse tree is a graphical depiction of a derivation. It is convenient to see how strings are derived from the start symbol. The start symbol of the derivation becomes the root of the parse tree. Let us see this by an example from the last topic.

We take the left-most derivation of a + b * c

The left-most derivation is:



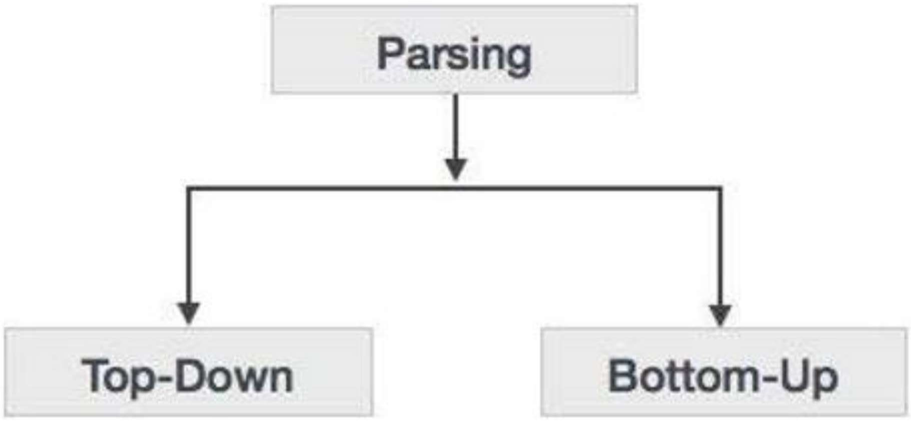
In a parse tree:

• All leaf nodes are terminals.
• All interior nodes are non-terminals.
• In-order traversal gives original input string.

A parse tree depicts associativity and precedence of operators. The deepest sub-tree is traversed first, therefore the operator in that sub-tree gets precedence over the operator which is in the parent nodes.

Types Of Parser

Syntax analyzers follow production rules defined by means of context-free grammar. The way the production rules are implemented (derivation) divides parsing into two types : top-down parsing and bottom-up parsing.



Top-down Parsing

When the parser starts constructing the parse tree from the start symbol and then tries to transform the start symbol to the input, it is called top-down parsing.

- Recursive descent parsing : It is a common form of top-down parsing. It is called recursive, as it uses recursive procedures to process the input. Recursive descent parsing suffers from backtracking.
- Backtracking : It means, if one derivation of a production fails, the syntax analyzer restarts the process using different rules of same production. This technique may process the input string more than once to determine the right production.

Bottom-up Parsing

As the name suggests, bottom-up parsing starts with the input symbols and tries to construct the parse tree up to the start symbol.

NOTE:THE TYPE OF PARSER WE HAVE USED IS TOP-DOWN PARSER

LL(1) Grammars

LL(1) GRAMMARS AND LANGUAGES. A context-free grammar G = (VT, VN, S, P) whose parsing table has no multiple entries is said to be LL(1). In the name LL(1),

• the first L stands for scanning the input from left to right
• the second L stands for producing a leftmost derivation

• and the 1 stands for using one input symbol of lookahead at each step to make parsing action decision.
A language is said to be LL(1) if it can be generated by a LL(1) grammar. It can be shown that
LL(1) grammars are
• not ambiguous and
• not left-recursive.

LL(1) grammar for above problem statement is:

stmt : ID C INTEGER SC PROC BODY END ID

BODY : ID OP1 ID C INTEGER OP2 PBODY

PBODY: ID EQQ NUM SC IF ID EQ NUM AND ID EQ NUM THEN IFBODY END IF SC

IFBODY: PRINTF OP1 MSG OP2 SC ELSEIF ID EQ NUM AND ID EQ NUM THEN ELSEIFBODY

ELSEIFBODY: PRINTF OP1 MSG OP2 SC ELSE S

S: PRINTF OP1 MSG OP2 SC

<u>Parser code for the above problem statement is :</u>

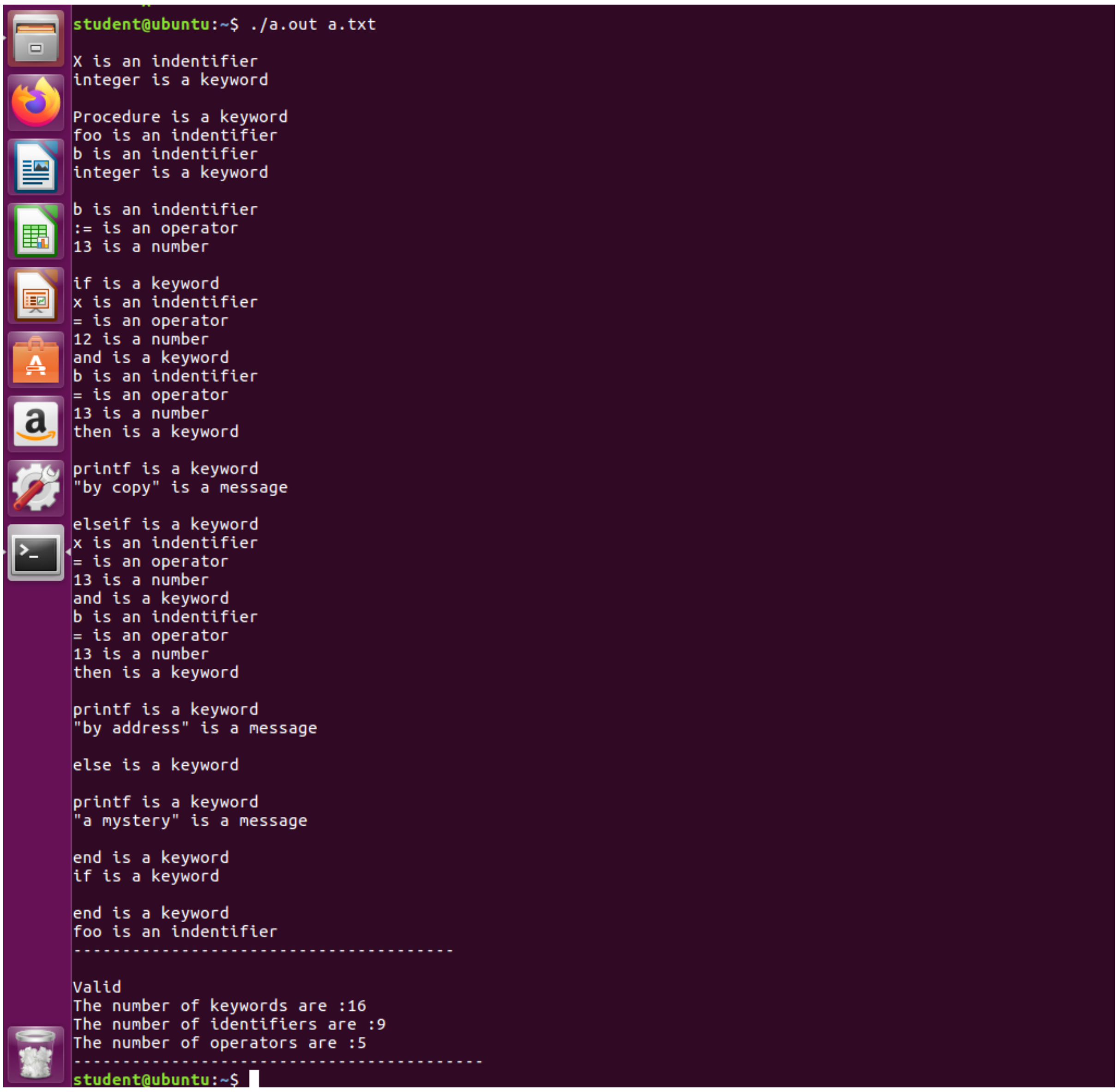
```
%{
  #include<stdio.h>
  #include<stdlib.h>
  #include<string.h>
  extern int k,o,i;
  extern int yylex();
  extern int *yyin;
  extern int *yyout;
%}
%token IF ELSE ELSEIF THEN PRINTF AND INTEGER PROC END
MSG ID NUM EQQ EQ C SC OP1 OP2
%%
stmt : ID C INTEGER SC PROC BODY END ID {
     printf("\n--------------------------------------\n");
     printf("\nValid\n");
     printf("The number of keywords are :%d\n",k);
     printf("The number of identifiers are :%d\n",i);
     printf("The number of operators are :%d\n",o);
     printf("-----------------------------------------\n");
     exit(0);}
         ;
BODY : ID OP1 ID C INTEGER OP2 PBODY
           ;
PBODY: ID EQQ NUM SC IF ID EQ NUM AND ID EQ NUM THEN
IFBODY END IF SC
            ;
IFBODY: PRINTF OP1 MSG OP2 SC ELSEIF ID EQ NUM AND ID EQ
NUM THEN ELSEIFBODY ;
ELSEIFBODY: PRINTF OP1 MSG OP2 SC ELSE S
                      ;
S: PRINTF OP1 MSG OP2 SC
  ;
%%
int yyerror()
{
   printf("\nInvalid\n");
   exit(0);
}
int main(int argc,char *argv[])
{
  yyin=fopen(argv[1],"r");
  yyparse();
 }
```

Explanation of the above code:

In the above yacc code, it takes the tokens generated by lex program and parse them.
If the input statement is as per the defined productions then the program prints valid message with the number of keywords, identifiers and operators, else it prints invalid message for invalid input.

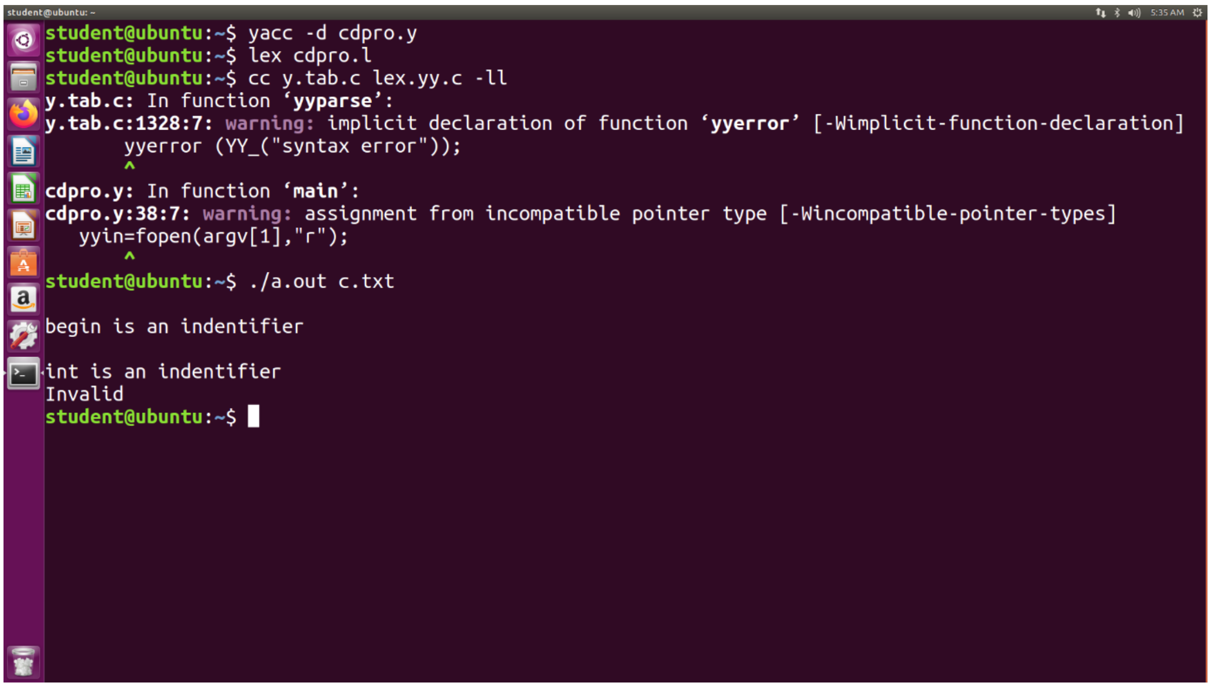
RESULTS

i)For valid input:



```
student@ubuntu:~$ ./a.out a.txt

X is an indentifier
integer is a keyword

Procedure is a keyword
foo is an indentifier
b is an indentifier
integer is a keyword

b is an indentifier
:= is an operator
13 is a number

if is a keyword
x is an indentifier
= is an operator
12 is a number
and is a keyword
b is an indentifier
= is an operator
13 is a number
then is a keyword

printf is a keyword
"by copy" is a message

elseif is a keyword
x is an indentifier
= is an operator
13 is a number
and is a keyword
b is an indentifier
= is an operator
13 is a number
then is a keyword

printf is a keyword
"by address" is a message

else is a keyword

printf is a keyword
"a mystery" is a message

end is a keyword
if is a keyword

end is a keyword
foo is an indentifier
---------------------------------------

Valid
The number of keywords are :16
The number of identifiers are :9
The number of operators are :5
------------------------------------------
student@ubuntu:~$
```

ii)For invalid input:-



```
student@ubuntu:~$ yacc -d cdpro.y
student@ubuntu:~$ lex cdpro.l
student@ubuntu:~$ cc y.tab.c lex.yy.c -ll
y.tab.c: In function ‘yyparse’:
y.tab.c:1328:7: warning: implicit declaration of function ‘yyerror’ [-Wimplicit-function-declaration]
       yyerror (YY_("syntax error"));
       ^
cdpro.y: In function ‘main’:
cdpro.y:38:7: warning: assignment from incompatible pointer type [-Wincompatible-pointer-types]
   yyin=fopen(argv[1],"r");
       ^
student@ubuntu:~$ ./a.out c.txt

begin is an indentifier

int is an indentifier
Invalid
student@ubuntu:~$
```

CONCLUSION

In lexical analysis when we give a program statement as input, the keywords, identifiers, operators and numbers are displayed. Each of these are the tokens of the statement

In top down parser, on giving an input string, the string is parsed as per the grammar and parsing table given in the program. If the string was parsed completely then the leftmost derivation of the string is displayed in the output else a syntax error is displayed.