Compiler for hypothetical language

A Mini Project Report Submitted by

Prathvi
(4NM17CS134)

Pavana S Salvankar
(4NM17CS123)

UNDER THE GUIDANCE OF

Mr. Ranjan Kumar H S

Designation

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

NITTE
EDUCATION TRUST

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

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

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CERTIFICATE

Compiler for hypothetical language

is a bonafide work carried out by

Prathvi(4NM17CS134) Pavana S Salvankar(4NM17CS123)

in partial fulfilment of the requirements for the award of
Bachelor of Engineering Degree in Computer Science and Engineering
prescribed by Visvesvaraya Technological University,
Belgaum during the year 2018-2019.

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.

Student names and USN

Prathvi(4NM17CS134)
Pavana S Salvankar(4NM17CS123)

ABSTRACT

This report contains the details of how one can develop the simple compiler for given language using Lex (Lexical Analyzer Generator) and YACC (Yet Another Compiler-Compiler). Lex tool helps write programs whose control flow is directed by instances of regular expressions in the input stream. Lex tool source is the table of regular expressions and corresponding program fragments. The table is translated to a program which reads an input stream, copying it to an output stream and partitioning the input into strings which match the given expressions. On the other hand, YACC tool receives input of the user grammar. Starting from this grammar it generates the C source code for the parser. YACC invokes Lex to scan the source code and uses the tokens returned by Lex to build a syntax tree. With the help of YACC and Lex tool one can write their own compiler.

CONTENTS

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 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 and 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 COMPILERS

Typically, a compiler includes several functional parts. For example, a conventional compiler may include a lexical analyser that looks at the source program and identifies successive "tokens" in the source program. A conventional compiler also includes a parser or syntactical analyser, which takes as an input a grammar defining the language being compiled and a series of actions associated with the grammar.

The syntactical analyser builds a "parse tree" for the statements in the source program in accordance with the grammar productions and actions.

For each statement in the input source program, the syntactical analyser generates a parse tree of the source input in a recursive, "bottom-up" manner in accordance with relevant productions and actions in the grammar. Generation of the parse tree allows the syntactical analyser to determine whether the parts of the source program comply with the grammar. If not, the syntactical analyser generates an error.

CLASSIFICATION OF COMPILER PHASES

There are two major parts of a compiler phases: Analysis and Synthesis. In analysis phase, an intermediate representation is created from the given source program that contains:

- Lexical Analyser
- Syntax Analyser
- Semantic Analyser

In synthesis phase, the equivalent target program is created from this intermediate representation. This contains:

- Intermediate code Generator
- Code Optimisation
- Code Generation

LEXICAL ANALYZER

Lexical analyzer takes the source program as an input and produces a string of tokens or lexemes. Lexical Analyzer reads the source program character by character and returns the tokens of the source program. The process of generation and returning the tokens is called lexical analysis. Representation of lexemes in the form of tokens as:
<token-name, attribute-value>

SYNTAX ANALYSER

A Syntax Analyzer creates the syntactic structure (generally a parse tree) of the given program. In other words, a Syntax Analyzer takes output of lexical analyser (list of tokens) and produces a parse tree. A syntax analyser is also called as a parser. The parser checks if the expression made by the tokens is syntactically correct.

SEMANTIC ANALYSER

Semantic analyser takes the output of syntax analyser. Semantic analyser checks a source program for semantic consistency with the language definition. It also gathers type information for use in intermediate-code generation.

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.

CODE OPTIMISER

The code optimizer takes the code produced by the intermediate code generator. The code optimizer reduces the code (if the code is not already optimized) without changing the meaning of the code. The optimization of code is in terms of time and space.

CODE GENERATION

This produces the target language in a specific architecture. The target program is normally is an object file containing the machine codes. Memory locations are selected for each of the variables used by the program.

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 analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code. If the lexical analyzer finds a token invalid, it generates an error. 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. The main purpose of lexical analysis is to make life easier for the subsequent syntax analysis phase.

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.

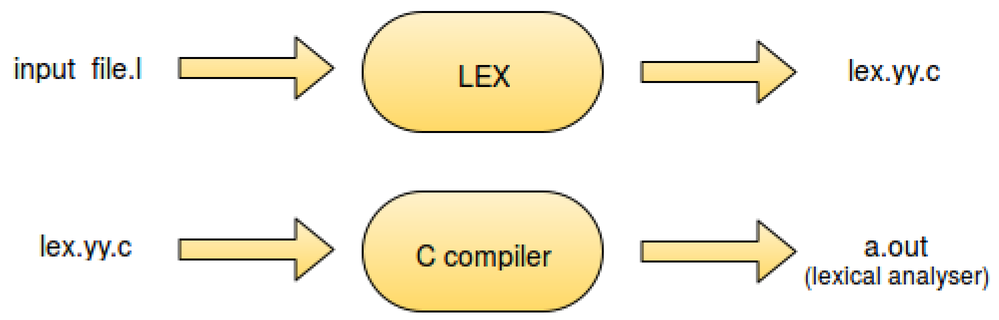
Lexeme:
A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.

We can implement lexical analyser using lex tools.

IMPLEMENTATION

LEX

LEX is a tool used to generate a lexical analyzer. Technically, LEX translates a set of regular expression specifications (given as input in inputfile.l) into a C implementation of a corresponding finite state machine (lex.yy.c). This C program, when compiled, yields an executable lexical analyzer.

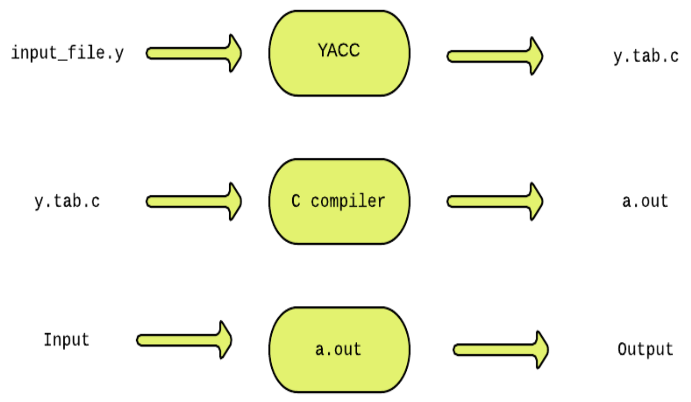


The source program is fed as the input to the lexical analyzer which produces a sequence of tokens as output. Conceptually, a lexical analyzer scans a given source program and produces an output of tokens.

Each token is specified by a token name. The token name is an abstract symbol representing the kind of lexical unit, e.g., a particular keyword, or a sequence of input characters denoting an identifier. The token names are the input symbols that the parser processes.

YACC

YACC (Yet Another Compiler Compiler)is a tool used to generate a parser. YACC translates a given Context Free Grammar (CFG) specifications (input in inputfile.y) into a C implementation (y.tab.c) of a corresponding push down automaton (i.e., a finite state machine with a stack). This C program when compiled, yields an executable parser.



The source program id fed as the input to the generated parser (a.out).The parser checks whether the program satisfies the syntax specification given in the inputfile.y file.

PARSER

A parser is a program that checks whether its input (viewed as a stream of tokens) meets a given grammar specification

YYPARSE()

The y.tab.c file contains a function yyparse() which is an implementation (in C) of a push down automaton. yyparse() is responsible for parsing the given input file.

STRUCTURE OF YACC PROGRAM

A YACC program consists of three sections: Declarations, Rules and Auxiliary functions

DECLARATION

The declarations section consists of two parts: (i) C declarations and (ii) YACC declarations . The C Declarations are delimited by **%{** and **%}**. This part consists of all the declarations required for the C code written in the *Actions* section and the *Auxiliary functions* section. The YACC declarations part comprises of declarations of tokens

RULES

In this section we will write the rules require for our program.

AUXILARY FUNCTION

The Auxiliary functions section contains the definitions of three mandatory functions main(), yylex() and yyerror(). You may wish to add your own functions (depending on the the requirement for the application) in the y.tab.c file. Such functions are written in the auxiliary functions section. The main() function must invoke yyparse() to parse the input.

STEPS INVOLVED TO WRITE LEX AND YACC PROGRAM

Step1: A YACC source program has three parts as follows:

Declarations %% translation rules %% supporting C routines

Step2: Declarations Section: This section contains entries that:

i. Include standard I/O header file.

ii. Define global variables.

iii. Define the list rule as the place to start processing.

iv. Define the tokens used by the parser.

v. Define the operators and their precedence.

Step3: Rules Section: The rules section defines the rules that parse the input stream. Each rule of a grammar production and the associated semantic action.

Step4: Programs Section: The programs section contains the following subroutines. Because these subroutines are included in this file, it is not necessary to use the YACC library when processing this file.

Step5: Main- The required main program that calls the yyparse subroutine to start the program.

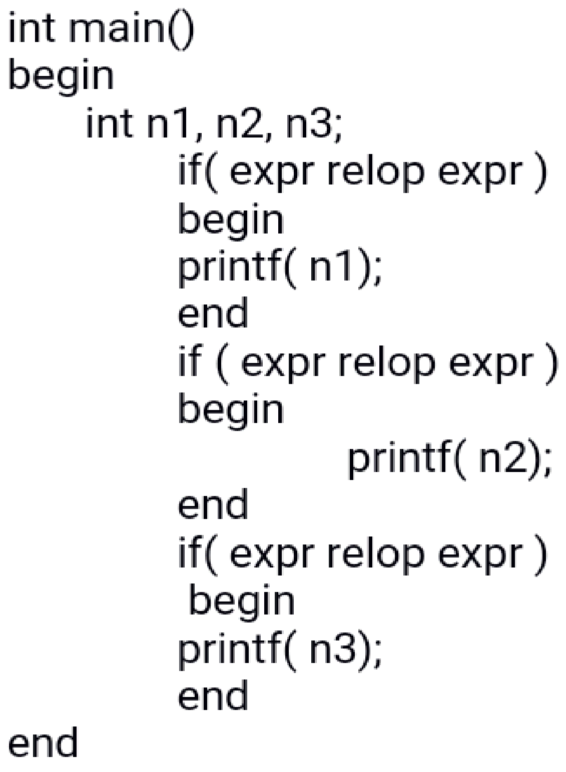
Step6: yyerror(s) -This error-handling subroutine only prints a syntax error message.

Step7: yywrap -The wrap-up subroutine that returns a value of 1 when the end of input occurs. The project.lex file contains include statements for standard input and output, as programmar file information if we use the -d flag with the yacc command. The y.tab.h file contains definitions for the tokens that the parser program uses.

Step8: project.lex contains the rules to generate these tokens from the input stream.

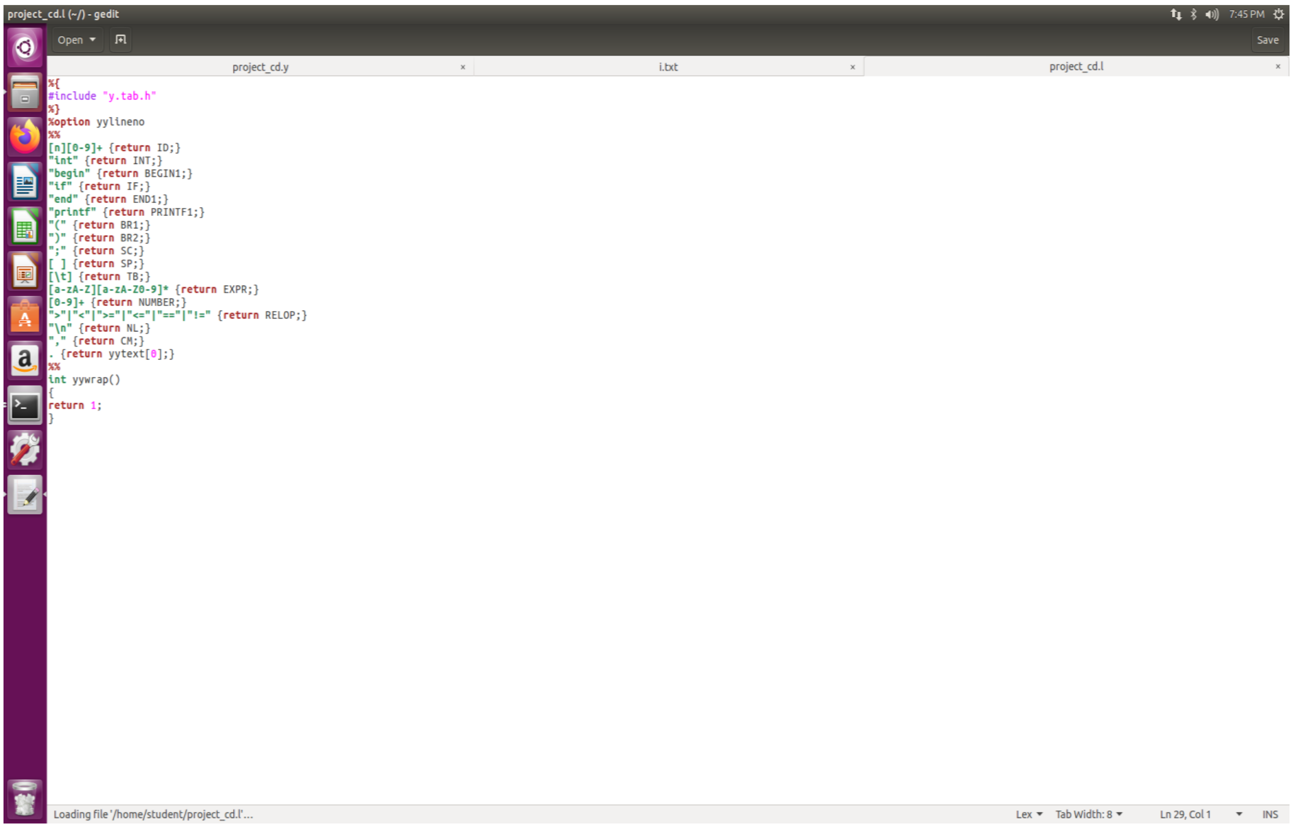
PROGRAM CODE

question



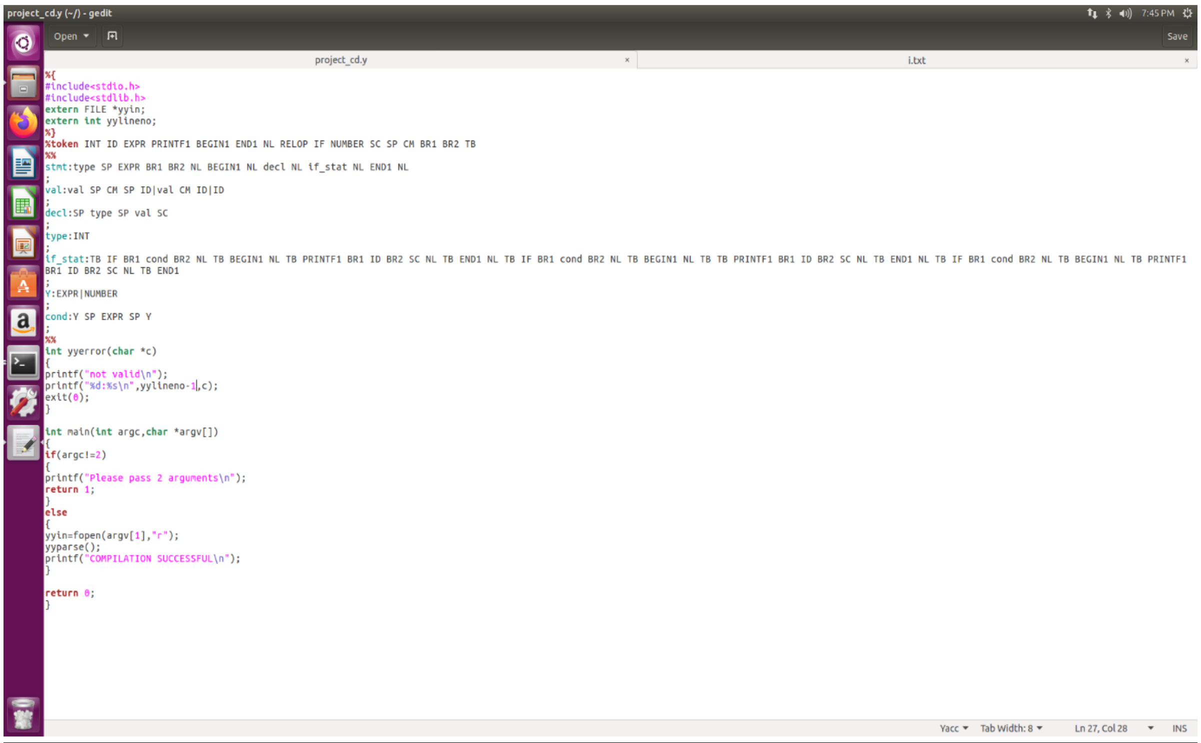
```
int main()
begin
      int n1, n2, n3;
              if( expr relop expr )
              begin
              printf( n1);
              end
              if ( expr relop expr )
              begin
                            printf( n2);
              end
              if( expr relop expr )
               begin
              printf( n3);
              end
end
```

Lex part



```
%{
#include "y.tab.h"
%}
%option yylineno
%%
[n][0-9]+ {return ID;}
"int" {return INT;}
"begin" {return BEGIN1;}
"if" {return IF;}
"end" {return END1;}
"printf" {return PRINTF1;}
"(" {return BR1;}
")" {return BR2;}
";" {return SC;}
[ ] {return SP;}
[\t] {return TB;}
[a-zA-Z][a-zA-Z0-9]* {return EXPR;}
[0-9]+ {return NUMBER;}
">"|"<"|">="|"<="|"=="|"!=" {return RELOP;}
"\n" {return NL;}
"," {return CM;}
. {return yytext[0];}
%%
int yywrap()
{
return 1;
}
```

Yacc part



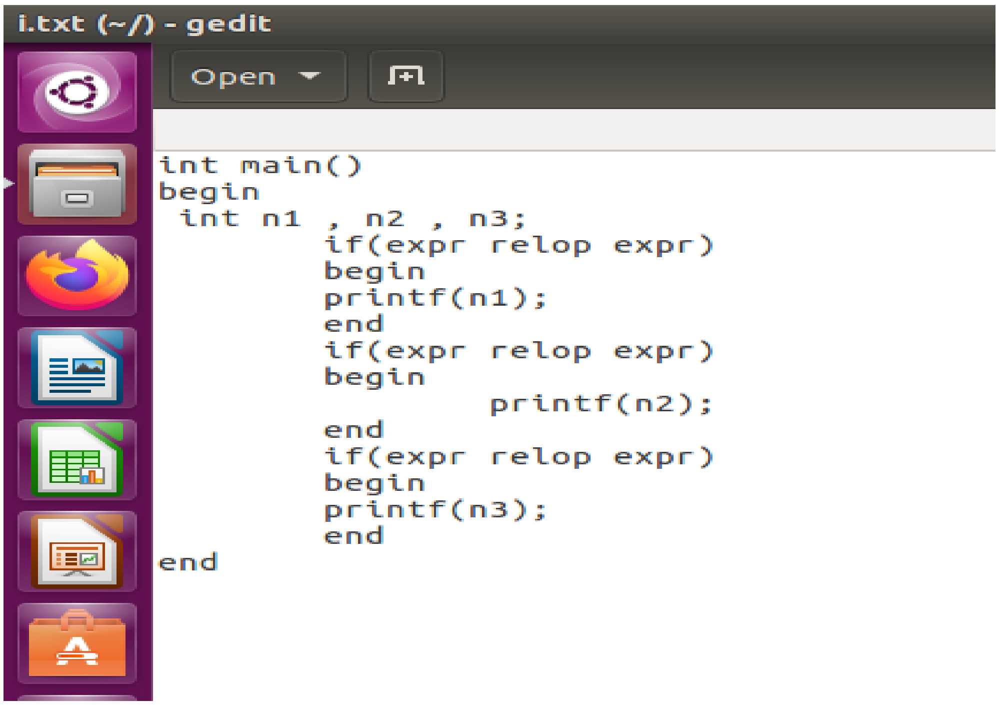
```
%{
#include<stdio.h>
#include<stdlib.h>
extern FILE *yyin;
extern int yylineno;
%}
%token INT ID EXPR PRINTF1 BEGIN1 END1 NL RELOP IF NUMBER SC SP CM BR1 BR2 TB
%%
stmt:type SP EXPR BR1 BR2 NL BEGIN1 NL decl NL if_stat NL END1 NL
;
val:val SP CM SP ID|val CM ID|ID
;
decl:SP type SP val SC
;
type:INT
;
if_stat:TB IF BR1 cond BR2 NL TB BEGIN1 NL TB PRINTF1 BR1 ID BR2 SC NL TB END1 NL TB IF BR1 cond BR2 NL TB BEGIN1 NL TB TB PRINTF1 BR1 ID BR2 SC NL TB END1 NL TB IF BR1 cond BR2 NL TB BEGIN1 NL TB PRINTF1
BR1 ID BR2 SC NL TB END1
;
Y:EXPR|NUMBER
;
cond:Y SP EXPR SP Y
;
%%
int yyerror(char *c)
{
printf("not valid\n");
printf("%d:%s\n",yylineno-1,c);
exit(0);
}

int main(int argc,char *argv[])
{
if(argc!=2)
{
printf("Please pass 2 arguments\n");
return 1;
}
else
{
yyin=fopen(argv[1],"r");
yyparse();
printf("COMPILATION SUCCESSFUL\n");
}

return 0;
}
```

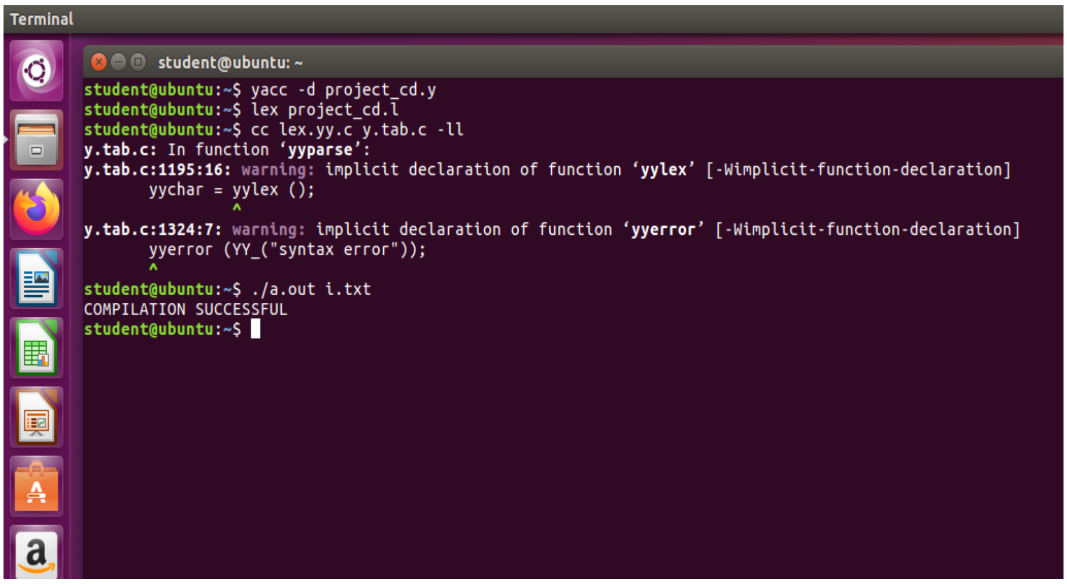
RESULT

INPUT



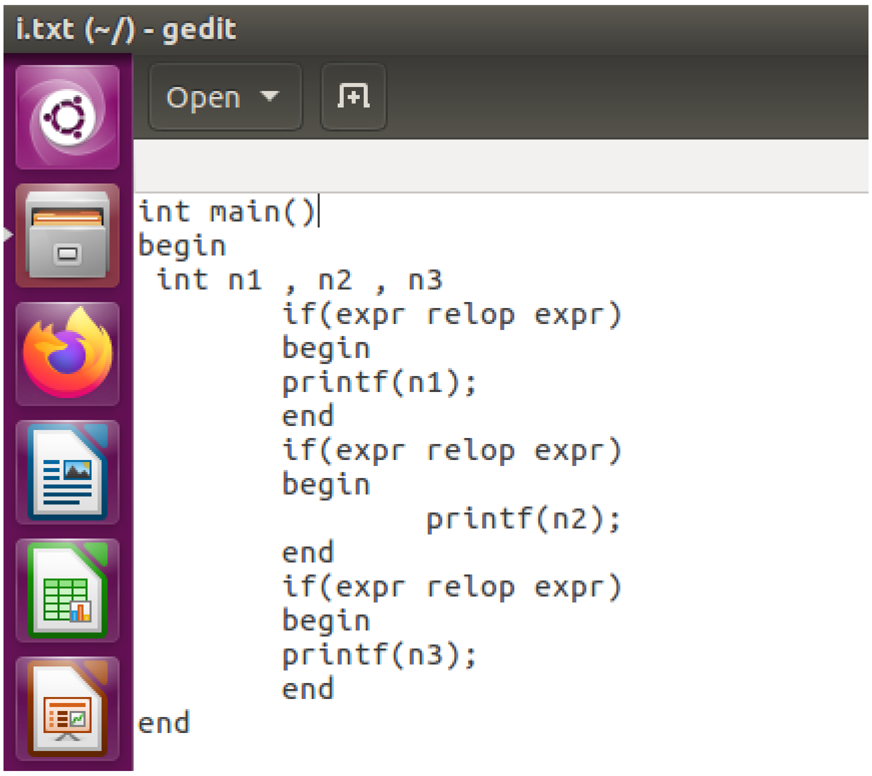
```
int main()
begin
 int n1  , n2  , n3;
        if(expr relop expr)
        begin
        printf(n1);
        end
        if(expr relop expr)
        begin
                printf(n2);
        end
        if(expr relop expr)
        begin
        printf(n3);
        end
end
```

OUTPUT



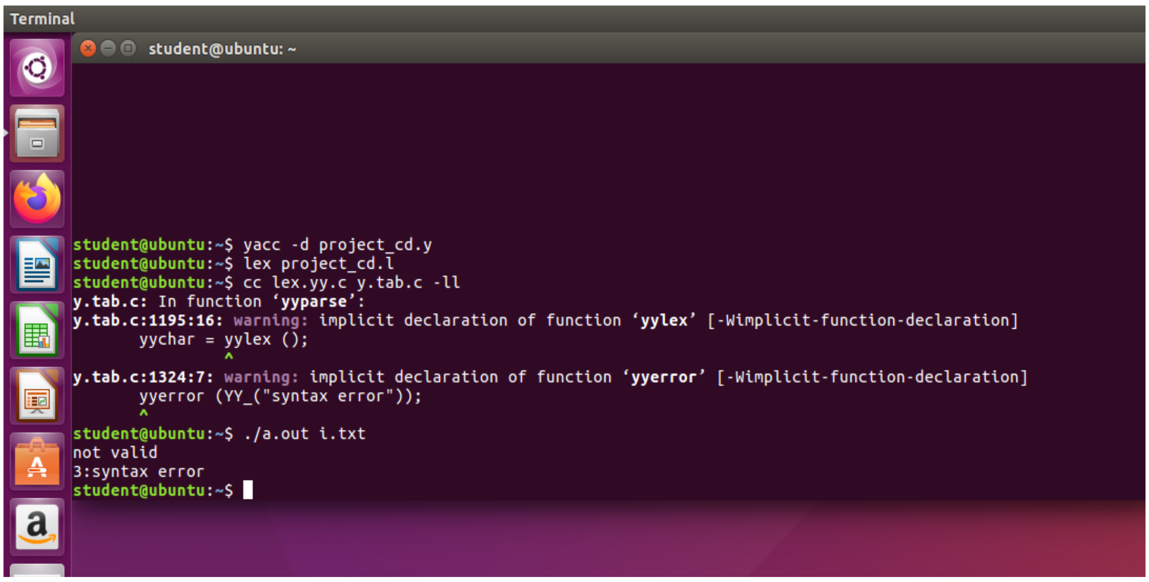
```
student@ubuntu:~$ yacc -d project_cd.y
student@ubuntu:~$ lex project_cd.l
student@ubuntu:~$ cc lex.yy.c y.tab.c -ll
y.tab.c: In function ‘yyparse’:
y.tab.c:1195:16: warning: implicit declaration of function ‘yylex’ [-Wimplicit-function-declaration]
       yychar = yylex ();
                ^
y.tab.c:1324:7: warning: implicit declaration of function ‘yyerror’ [-Wimplicit-function-declaration]
       yyerror (YY_("syntax error"));
       ^
student@ubuntu:~$ ./a.out i.txt
COMPILATION SUCCESSFUL
student@ubuntu:~$ 
```

INPUT

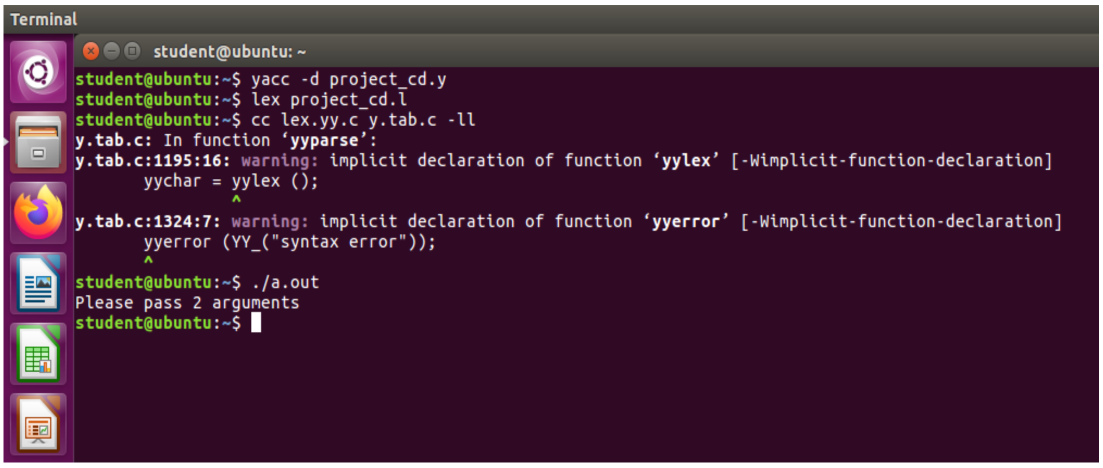


```
int main()
begin
 int n1 , n2 , n3
        if(expr relop expr)
        begin
        printf(n1);
        end
        if(expr relop expr)
        begin
                printf(n2);
        end
        if(expr relop expr)
        begin
        printf(n3);
        end
end
```

OUTPUT



```
student@ubuntu:~$ yacc -d project_cd.y
student@ubuntu:~$ lex project_cd.l
student@ubuntu:~$ cc lex.yy.c y.tab.c -ll
y.tab.c: In function ‘yyparse’:
y.tab.c:1195:16: warning: implicit declaration of function ‘yylex’ [-Wimplicit-function-declaration]
       yychar = yylex ();
                ^
y.tab.c:1324:7: warning: implicit declaration of function ‘yyerror’ [-Wimplicit-function-declaration]
       yyerror (YY_("syntax error"));
       ^
student@ubuntu:~$ ./a.out i.txt
not valid
3:syntax error
student@ubuntu:~$ 
```



```
student@ubuntu:~$ yacc -d project_cd.y
student@ubuntu:~$ lex project_cd.l
student@ubuntu:~$ cc lex.yy.c y.tab.c -ll
y.tab.c: In function ‘yyparse’:
y.tab.c:1195:16: warning: implicit declaration of function ‘yylex’ [-Wimplicit-function-declaration]
       yychar = yylex ();
                ^
y.tab.c:1324:7: warning: implicit declaration of function ‘yyerror’ [-Wimplicit-function-declaration]
       yyerror (YY_("syntax error"));
       ^
student@ubuntu:~$ ./a.out
Please pass 2 arguments
student@ubuntu:~$
```

CONCLUSION

In lexical analyses ,when we give a program statement as input ,the input is parsed .If it is parsed successfully, the success message is displayed in the output .Otherwise the error message is displayed.