

**NITTE MAHALINGA ADYANTHAYA MEMORIAL INSTITUTE OF
TECHNOLOGY**

An Autonomous College Affiliated to VTU Belgaum

Nitte -574110, Udupi District



PROJECT REPORT

ON

COMPILER DESIGN

SUBMITTED BY:

JASMINE GLANI MATHIAS

4NM17CS070

6 SEM, 'B' SEC

Department of CSE

NMAMIT, Nitte

JOSHNI PRINCIA SALDANHA

4NM17CS072

6 SEM, 'B' SEC

Department of CSE

NMAMIT, Nitte

SUBMITTED TO:

Mrs. Minu P Abraham

Department of Computer Science and Engineering

NMAM INSTITUTE OF TECHNOLOGY
(An autonomous Institute affiliated to VTU, Belgaum)
Nitte-574110, Karkala, Udupi District

Department of Computer Science and Engineering

CERTIFICATE

Certified that the project work carried out by Jasmine Glani Mathias, USN 4NM17CS070 and Joshni Princia Saldanha, USN 4NM17CS072 , bonafide students of NMAM Institute of Technology, Nitte in fulfillment for the compiler design lab in Computer Science and Engineering during the academic year 2019-2020.

Signature of lecturer

Date:

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 I acknowledge all those whose guidance and encouragement served as a beacon light and crowned my efforts with success.

I would like to thank our principal **Prof. Niranjan N. Chiplunkar** firstly, for providing us with this unique opportunity to do the project in the 8th semester of electronics communication and engineering.

I would like to thank my college administration for providing a conducive environment and also suitable facilities for this project. I would like to thank our HOD **Prof.Uday Kumar Reddy** for showing me the path and providing the inspiration required for taking the project to its completion. It is my great pleasure to thank my guide **Mrs. Minu P. Abraham** for her continuous encouragement, guidance and support throughout this project.

I thank all the staff members of the department of CSE for providing resources for the completion of the project.

I thank my parents and friends for their consistent support throughout this project.

Finally I thank all those who have contributed directly or indirectly in making this project a Grand Success.

JASMINEGLANI MATHIAS (4NM17CS070)
JOSHNI PRINCIA SALDANHA (4NM17CS072)

CONTENTS

- ☐ **INTRODUCTION**
- ☐ **LEXICAL ANALYSIS**
- ☐ **SYNTAX ANALYSIS**
- ☐ **CONTEXT FREE GRAMMAR**
- ☐ **PARSE TREE**
- ☐ **TYPES OF PARSING**
- ☐ **PARSING TABLE**
- ☐ **PARSING**
- ☐ **OUTPUT SCREENSHOTS**
- ☐ **OUTPUT FOR SYNTAX ERROR CONDITION**
- ☐ **CONCLUSION**
- ☐ **BIBLIOGRAPHY**

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.

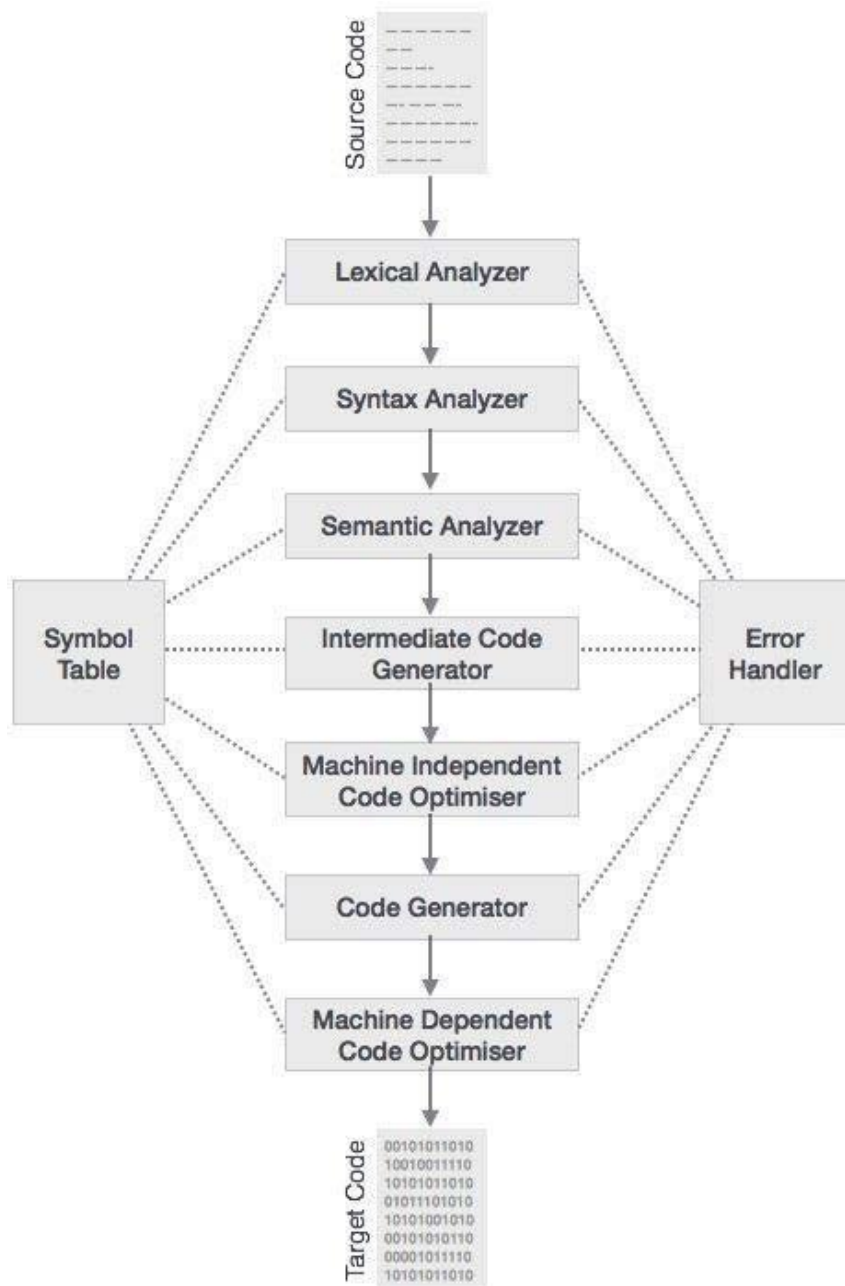
This is where the compiler comes in. A compiler translates (or compiles) a program written in a high-level programming language that is suitable for human programmers into the low-level machine language that is required by computers. During this process, the compiler will also attempt to spot and report 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 hand-written 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.



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 analyzer 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 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 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. 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 c/c++ compiler for the following pseudocode

```
int main()
begin
int n,i,sum=0;
for(i=1;i<=n;++i)
begin
    expr=expr+expr;
end
end
```

Ex: The lexical program for above problem statement

```
#include <stdbool.h>
```

```
#include <stdio.h>
```

```
#include <string.h>
```

```
#include <stdlib.h>
```

```
bool isDelimiter(char ch)
```

```
{
```

```
    if (ch == ' ' || ch == '+' || ch == '-' || ch == '*' ||
```

```
        ch == '/' || ch == ',' || ch == ';' || ch == '>' ||
```

```
        ch == '<' || ch == '=' || ch == '(' || ch == ')'
```

```
        ch == '[' || ch == ']' || ch == '{' || ch == '}')
```

```
        return (true);
```

```

        return (false);
    }

bool isOperator(char ch)
{
    if (ch == '+' || ch == '-' || ch == '*' ||
        ch == '/' || ch == '>' || ch == '<' ||
        ch == '=')
        return (true);
    return (false);
}

// Returns 'true' if the string is a VALID IDENTIFIER.
bool validIdentifier(char* str)
{
    if (str[0] == '0' || str[0] == '1' || str[0] == '2' ||
        str[0] == '3' || str[0] == '4' || str[0] == '5' ||
        str[0] == '6' || str[0] == '7' || str[0] == '8' ||
        str[0] == '9' || isDelimiter(str[0]) == true)
        return (false);
    return (true);
}

// Returns 'true' if the string is a KEYWORD.
bool isKeyword(char* str)
{
    if (!strcmp(str, "if") || !strcmp(str, "else") ||
        !strcmp(str, "while") || !strcmp(str, "do") ||

```

```

!strcmp(str, "break") || !strcmp(str, "for")||

!strcmp(str, "continue") || !strcmp(str, "int")

|| !strcmp(str, "double") || !strcmp(str, "float")

|| !strcmp(str, "return") || !strcmp(str, "char")

|| !strcmp(str, "case") || !strcmp(str, "char")

|| !strcmp(str, "sizeof") || !strcmp(str, "long")

|| !strcmp(str, "short") || !strcmp(str, "typedef")

|| !strcmp(str, "switch") || !strcmp(str, "unsigned")

|| !strcmp(str, "void") || !strcmp(str, "static")

|| !strcmp(str, "struct") || !strcmp(str, "goto"))

return (true);

return (false);

}

```

// Returns 'true' if the string is an INTEGER.

```

bool isInteger(char* str)

{

    int i, len = strlen(str);

    if (len == 0)

        return (false);

    for (i = 0; i < len; i++) {

        if (str[i] != '0' && str[i] != '1' && str[i] != '2'

            && str[i] != '3' && str[i] != '4' && str[i] != '5'

            && str[i] != '6' && str[i] != '7' && str[i] != '8'

            && str[i] != '9' || (str[i] == '-' && i > 0))

            return (false);

    }
}

```

```

        return (true);
    }

// Returns 'true' if the string is a REAL NUMBER.

bool isRealNumber(char* str)
{
    int i, len = strlen(str);

    bool hasDecimal = false;

    if (len == 0)

        return (false);

    for (i = 0; i < len; i++) {

        if (str[i] != '0' && str[i] != '1' && str[i] != '2'

            && str[i] != '3' && str[i] != '4' && str[i] != '5'

            && str[i] != '6' && str[i] != '7' && str[i] != '8'

            && str[i] != '9' && str[i] != '.' ||

            (str[i] == '-' && i > 0))

            return (false);

        if (str[i] == '.')

            hasDecimal = true;

    }

    return (hasDecimal);
}

char* subString(char* str, int left, int right)
{
    int i;

    char* subStr = (char*)malloc(

        sizeof(char) * (right - left + 2));

```

```

    for (i = left; i <= right; i++)

        subStr[i - left] = str[i];

    subStr[right - left + 1] = '\0';

    return (subStr);

}

void parse(char* str)

{

    int left = 0, right = 0;

    int len = strlen(str);

    while (right <= len && left <= right) {

        if (isDelimiter(str[right]) == false)

            right++;

        if (isDelimiter(str[right]) == true && left ==

right) {

            if (isOperator(str[right]) == true)

                printf("%c' IS AN OPERATOR\n",

str[right]);

            right++;

            left = right;

        } else if (isDelimiter(str[right]) == true && left

!= right

            || (right == len && left != right)) {

            char* subStr = subString(str, left, right - 1);

            if (isKeyword(subStr) == true)

                printf("%s' IS A KEYWORD\n", subStr);

```

```

        else if (isInteger(subStr) == true)

            printf("%s' IS AN INTEGER\n", subStr);


        else if (isRealNumber(subStr) == true)

            printf("%s' IS A REAL NUMBER\n",
subStr);


        else if (validIdentifier(subStr) == true

            && isDelimiter(str[right - 1]) == false)

            printf("%s' IS A VALID IDENTIFIER\n",
subStr);


        else if (validIdentifier(subStr) == false

            && isDelimiter(str[right - 1]) == false)

            printf("%s' IS NOT A VALID
IDENTIFIER\n", subStr);

        left = right;

    }

}

return;

}


// DRIVER FUNCTION

int main()

{

    // maximum length of string is 100 here

    char str[100] = "int main() begin int n,i,sum=0;

```

```
for(i=1;i<=n;++i) begin expr=expr+expr; end End";
```

```
parse(str); // calling the parse function
```

```
return (0);
```

```
}
```

Explanation for the above code

In the code above, a string i.e. our problem statement has been taken and initialized to the str array.

In the next line of code, there is a parse function which takes the str array as the parameter and does the lexical analysis.

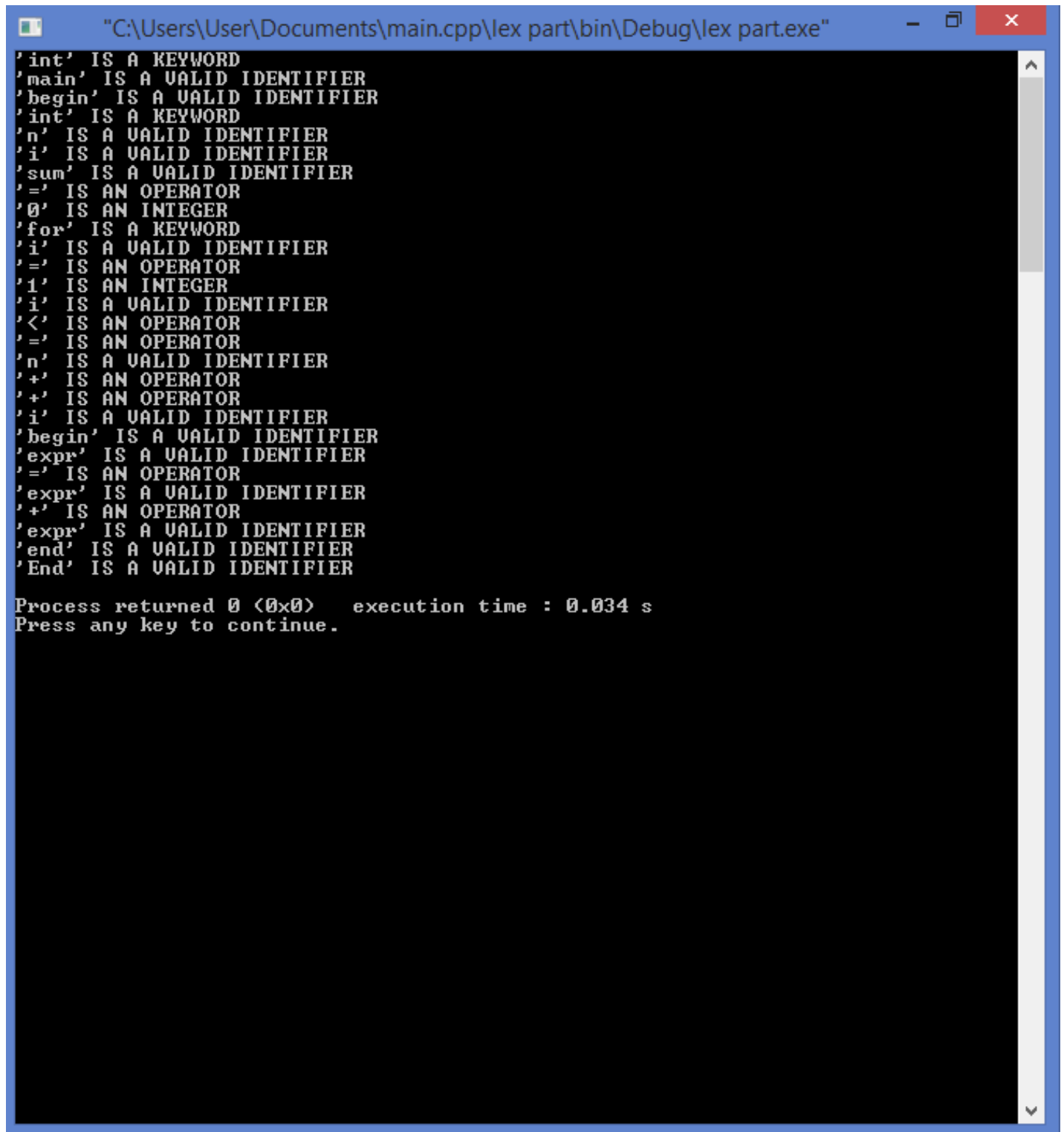
Every time the token is compared with each word in the array and if it matches any of keyword, then it is displayed as a keyword.

If it matches with any of the operators, then it is an

operator. If it matches with numbers, then it is a number.

Else it is an identifier.

Output :



```
"C:\Users\User\Documents\main.cpp\lex part\bin\Debug\lex part.exe"
'int' IS A KEYWORD
'main' IS A VALID IDENTIFIER
'begin' IS A VALID IDENTIFIER
'int' IS A KEYWORD
'n' IS A VALID IDENTIFIER
'i' IS A VALID IDENTIFIER
'sum' IS A VALID IDENTIFIER
'=' IS AN OPERATOR
'0' IS AN INTEGER
'for' IS A KEYWORD
'i' IS A VALID IDENTIFIER
'=' IS AN OPERATOR
'1' IS AN INTEGER
'i' IS A VALID IDENTIFIER
'<' IS AN OPERATOR
'=' IS AN OPERATOR
'n' IS A VALID IDENTIFIER
'+' IS AN OPERATOR
'+' IS AN OPERATOR
'i' IS A VALID IDENTIFIER
'begin' IS A VALID IDENTIFIER
'expr' IS A VALID IDENTIFIER
'=' IS AN OPERATOR
'expr' IS A VALID IDENTIFIER
'+' IS AN OPERATOR
'expr' IS A VALID IDENTIFIER
'end' IS A VALID IDENTIFIER
'End' IS A VALID IDENTIFIER

Process returned 0 (0x0)   execution time : 0.034 s
Press any key to continue.
```


SYNTAX ANALYSIS

Syntax analysis or parsing is the second phase of a compiler.

Role of the Parser

- In the compiler model, the parser obtains a string of tokens from the lexical analyzer, 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.

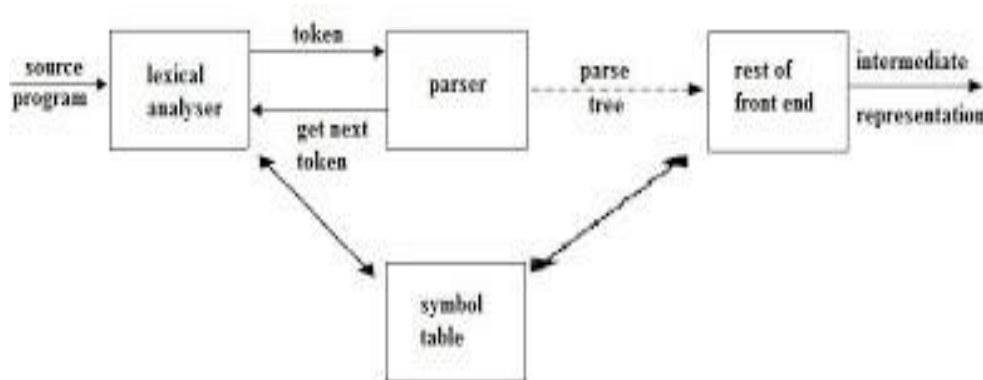


Fig 2.1 Position of parser in compiler model

- There are three general types of parsers for grammar.

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 the cases, 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.

- 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 analyzer 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 check 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:

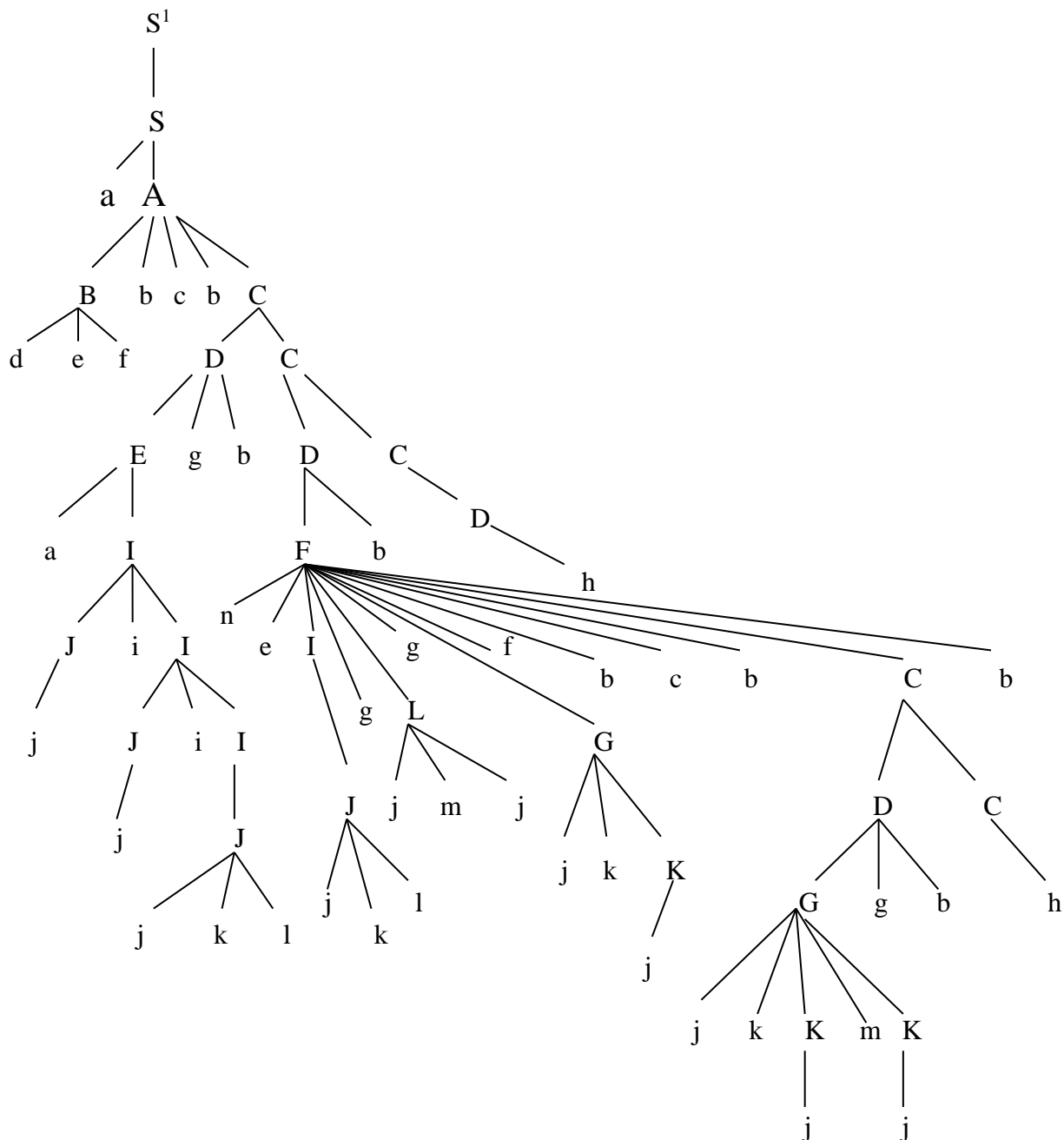
- A set of **non-terminals** (V). Non-terminals are syntactic variables that denote sets of strings. The non-terminals define sets of strings that help define the language generated by the grammar.
- A set of tokens, known as **terminal symbols** (Σ). Terminals are the basic symbols from which strings 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.

The left-most derivation is:



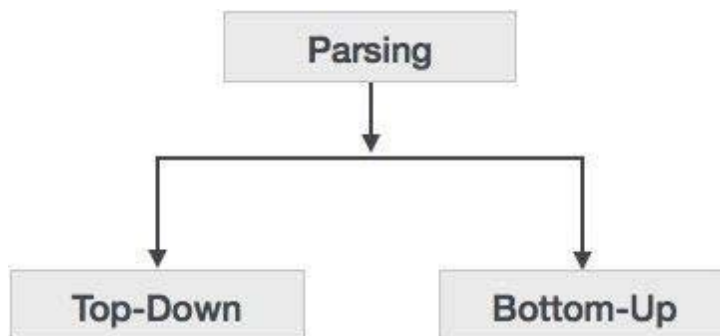
In a parse tree:

- 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 PARSING

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 BOTTOM-UP PARSER

LR PARSERS

An efficient bottom-up syntax analysis technique that can be used to parse a large class of CFG is called LR(k) parsing. The 'L' is for left-to-right scanning of the input, the 'R' for constructing a rightmost derivation in reverse, and the 'k' for the number of input symbols. When 'k' is omitted, it is assumed to be 1.

Types of LR parsing method

- SLR – Simple LR
 - Easiest to implement, least powerful.
- CLR – Canonical LR
 - Most powerful, most expensive.
- LALR – Look-Ahead LR
 - Intermediate in size and cost between the other two methods.

Ex: LR grammar for above problem statement,

S → a A
A → B b c b C
B → d e f
C → D C
C → D
D → h
D → E g b
D → F b
D → G g b
D → H b
E → a I
I → J
I → J i I
J → j
J → j k l
G → j k K m K
G → j
G → j k K

$L \rightarrow j m j$
 $F \rightarrow n e I g L g G f b c b C b$
 $H \rightarrow o e K f$
 $K \rightarrow j$
 $K \rightarrow l$

First and Follow Sets

An important part of parser table construction is to create first and follow sets. These sets can provide the actual position of any terminal in the derivation. This is done to create the parsing table where the decision of replacing $T[A, t] = \alpha$ with some production rule.

First Set

This set is created to know what terminal symbol is derived in the first position by a non-terminal. For example,

$$\alpha \rightarrow t \beta$$

That is, α derives t (terminal) in the very first position. So, $t \in \text{FIRST}(\alpha)$.

Algorithm for Calculating First Set

Look at the definition of $\text{FIRST}(\alpha)$ set:

- If α is a terminal, then $\text{FIRST}(\alpha) = \{ \alpha \}$.
- If α is a non-terminal and $\alpha \rightarrow \epsilon$ is a production, then $\text{FIRST}(\alpha) = \{ \epsilon \}$.
- If α is a non-terminal and $\alpha \rightarrow \gamma_1 \gamma_2 \gamma_3 \dots \gamma_n$ and any $\text{FIRST}(\gamma_i)$ contains t , then t is in $\text{FIRST}(\alpha)$.

First set can be seen as: $\text{FIRST}(\alpha) = \{ t \mid \alpha \rightarrow^* t \beta \} \cup \{ \epsilon \mid \alpha \rightarrow^* \epsilon \}$

Follow Set

Likewise, we calculate what terminal symbol immediately follows a non-terminal α in production rules. We do not consider what the non-terminal can generate but instead, we see what would be the next terminal symbol that follows the productions of a non-terminal.

Algorithm for Calculating Follow Set:

- If α is a start symbol, then $\text{FOLLOW}(\alpha) = \$$
- If α is a non-terminal and has a production $\alpha \rightarrow AB$, then $\text{FIRST}(B)$ is in $\text{FOLLOW}(\alpha)$ except ϵ .

- If α is a non-terminal and has a production $\alpha \rightarrow AB$, where $B \in \Sigma$, then $\text{FOLLOW}(A)$ is in $\text{FOLLOW}(\alpha)$.

Follow set can be seen as: $\text{FOLLOW}(\alpha) = \{ t \mid S \xrightarrow{*} \alpha t^* \}$

Ex : First and Follow for the above problem statement is

$\text{FIRST}(A) = \{ d, b, c, h, a, g, n, j, k, l, o \}$
 $\text{FIRST}(C) = \{ h, a, g, b, n, j, k, l, o \}$
 $\text{FIRST}(B) = \{ d \}$
 $\text{FIRST}(E) = \{ a \}$
 $\text{FIRST}(D) = \{ h, a, g, b, n, j, k, l, o \}$
 $\text{FIRST}(G) = \{ j, k, l \}$
 $\text{FIRST}(F) = \{ n \}$
 $\text{FIRST}(I) = \{ j, k, l, i \}$
 $\text{FIRST}(H) = \{ o \}$
 $\text{FIRST}(K) = \{ j, l \}$
 $\text{FIRST}(J) = \{ j, k, l \}$
 $\text{FIRST}(L) = \{ j \}$
 $\text{FIRST}(S) = \{ a \}$
 $\text{FIRST}(S') = \{ a \}$

$\text{FOLLOW}(A) = \{ \$ \}$
 $\text{FOLLOW}(C) = \{ \$, b \}$
 $\text{FOLLOW}(B) = \{ b \}$
 $\text{FOLLOW}(E) = \{ g \}$
 $\text{FOLLOW}(D) = \{ h, a, g, b, n, j, k, l, o, \$ \}$
 $\text{FOLLOW}(G) = \{ g, f \}$
 $\text{FOLLOW}(F) = \{ b \}$
 $\text{FOLLOW}(I) = \{ g \}$
 $\text{FOLLOW}(H) = \{ b \}$
 $\text{FOLLOW}(K) = \{ m, g, f \}$
 $\text{FOLLOW}(J) = \{ g, i \}$
 $\text{FOLLOW}(L) = \{ g \}$
 $\text{FOLLOW}(S) = \{ \$ \}$
 $\text{FOLLOW}(S') = \{ \$ \}$

PARSING TABLE

Building the complete table

- Need a row for every NT & a column for every T
- Need an algorithm to build the table

Filling in TABLE[X,y], $X \in NT$, $y \in T$

- entry is the rule $X ::= \beta$. if $y \in \text{FIRST}^+(\beta)$
- entry is error otherwise (can treat empty entry as implicit error)

[illegible]



PARSING

Input string: a d e f b c b a j i j i j k l g b n e j k l g j m j g j k j f b c b j k j m j g b h b b h

The screenshot shows the IntelliJ IDEA IDE with a Java project. The main editor displays a file named `try2.java` containing a Java program for LR(0) item sets and transitions. The program defines a grammar with terminals `a, b, c` and non-terminals `A, B, C`. It then defines LR(0) items and transitions. The `try2` tab shows the execution of the parser on the input `a b c b a j i j i k l g b n e j k l g j m j g j k j f b c b j k j m j g b h b b h`. The `Problems` tab shows a warning about a `terminated` exception. The `Console` tab shows the execution output, which is a table of steps, stack, input, and action.

STEP	STACK	INPUT	ACTION
1	0	adebcbajijijklgbnejklgjmjgjkjfbcbjkjmjgbbbbb	s2
2	0a2	defbcbajijijklgbnejklgjmjgjkjfbcbjkjmjgbbbbb	s5
3	0a2d5	efbcbajijijklgbnejklgjmjgjkjfbcbjkjmjgbbbbb	s6
4	0a2d5e6	fbcbajijijklgbnejklgjmjgjkjfbcbjkjmjgbbbbb	s9
5	0a2d5e6f9	bcbaajijijklgbnejklgjmjgjkjfbcbjkjmjgbbbbb	r3
6	0a2B4	bcbaajijijklgbnejklgjmjgjkjfbcbjkjmjgbbbbb	s7
7	0a2B4b7	cbajijijklgbnejklgjmjgjkjfbcbjkjmjgbbbbb	s8
8	0a2B4b7c8	bajijijklgbnejklgjmjgjkjfbcbjkjmjgbbbbb	s10
9	0a2B4b7c8b10	ajijijklgbnejklgjmjgjkjfbcbjkjmjgbbbbb	s17
10	0a2B4b7c8b10a17	ijijklgbnejklgjmjgjkjfbcbjkjmjgbbbbb	s29
11	0a2B4b7c8b10a17j29	ijijklgbnejklgjmjgjkjfbcbjkjmjgbbbbb	r19
12	0a2B4b7c8b10a17j28	ijijklgbnejklgjmjgjkjfbcbjkjmjgbbbbb	s33
13	0a2B4b7c8b10a17j28i33	ijijklgbnejklgjmjgjkjfbcbjkjmjgbbbbb	s29
14	0a2B4b7c8b10a17j28i33j29	ijijklgbnejklgjmjgjkjfbcbjkjmjgbbbbb	r19
15	0a2B4b7c8b10a17j28i33j28	ijijklgbnejklgjmjgjkjfbcbjkjmjgbbbbb	s33
16	0a2B4b7c8b10a17j28i33j28i33j	klgbnejklgjmjgjkjfbcbjkjmjgbbbbb	s29
17	0a2B4b7c8b10a17j28i33j28i33j29j	klgbnejklgjmjgjkjfbcbjkjmjgbbbbb	s34
18	0a2B4b7c8b10a17j28i33j28i33j29k34	lgbnejklgjmjgjkjfbcbjkjmjgbbbbb	s48
19	0a2B4b7c8b10a17j28i33j28i33j29k34l48j	gbnejklgjmjgjkjfbcbjkjmjgbbbbb	r20
20	0a2B4b7c8b10a17j28i33j28i33j28j28	gbnejklgjmjgjkjfbcbjkjmjgbbbbb	r14
21	0a2B4b7c8b10a17j28i33j28i33j45j	gbnejklgjmjgjkjfbcbjkjmjgbbbbb	r15
22	0a2B4b7c8b10a17j28i33i45j	gbnejklgjmjgjkjfbcbjkjmjgbbbbb	r15
23	0a2B4b7c8b10a17i127	gbnejklgjmjgjkjfbcbjkjmjgbbbbb	r4
24	0a2B4b7c8b10E13	gbnejklgjmjgjkjfbcbjkjmjgbbbbb	s31
25	0a2B4b7c8b10F13j23j	gbnejklgjmjgjkjfbcbjkjmjgbbbbb	εd3

```
try2
grammars = open("/home/ubuntu/Desktop/grammar.txt")

<terminated> /home/ubuntu/Desktop/syntax/try2/try2.py
26 0a2B4b7c8b10E13g31b43 nejklgmjgjkjfbcbjkjmjgbbhbs r6
27 0a2B4b7c8b10D12 nejklgmjgjkjfbcbjkjmjgbbhbs s20
28 0a2B4b7c8b10D12n20 ejklgmjgjkjfbcbjkjmjgbbhbs s22
29 0a2B4b7c8b10D12n20e22 jklgmjgjkjfbcbjkjmjgbbhbs s29
30 0a2B4b7c8b10D12n20e22j29 klgmjgjkjfbcbjkjmjgbbhbs s34
31 0a2B4b7c8b10D12n20e22j29k34 lgjmjgjkjfbcbjkjmjgbbhbs s48
32 0a2B4b7c8b10D12n20e22j29k34l48 gjmjgjkjfbcbjkjmjgbbhbs r20
33 0a2B4b7c8b10D12n20e22j28 gjmjgjkjfbcbjkjmjgbbhbs r14
34 0a2B4b7c8b10D12n20e22I35 gjmjgjkjfbcbjkjmjgbbhbs s47
35 0a2B4b7c8b10D12n20e22I35g47 jmjgjkjfbcbjkjmjgbbhbs s51
36 0a2B4b7c8b10D12n20e22I35g47j51 mjgjkjfbcbjkjmjgbbhbs s52
37 0a2B4b7c8b10D12n20e22I35g47j51m52 gjgjkjfbcbjkjmjgbbhbs s55
38 0a2B4b7c8b10D12n20e22I35g47j51m52j55 gjgjkjfbcbjkjmjgbbhbs r21
39 0a2B4b7c8b10D12n20e22I35g47L50 gjgjkjfbcbjkjmjgbbhbs s53
40 0a2B4b7c8b10D12n20e22I35g47L50g53 kjfbcbjkjmjgbbhbs s19
41 0a2B4b7c8b10D12n20e22I35g47L50g53j19 kjfbcbjkjmjgbbhbs s24
42 0a2B4b7c8b10D12n20e22I35g47L50g53j19k24 jfbcbjkjmjgbbhbs s40
43 0a2B4b7c8b10D12n20e22I35g47L50g53j19k24j40 fbcbjkjmjgbbhbs r17
44 0a2B4b7c8b10D12n20e22I35g47L50g53j19k24K39 fbcbjkjmjgbbhbs r12
45 0a2B4b7c8b10D12n20e22I35g47L50g53G54 fbcbjkjmjgbbhbs s56
46 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56 bcbjkjmjgbbhbs s57
47 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56b57 cbjkjmjgbbhbs s58
48 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56b57c58 bjkjmjgbbhbs s59
49 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56b57c58b59 kjmjbhbs s19
50 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56b57c58b59j19 kjmjbhbs s24
51 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56b57c58b59j19k24 jmjbhbs s40
52 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56b57c58b59j19k24j40 mjgbbhbs r17
53 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56b57c58b59j19k24K39 mjgbbhbs s46
54 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56b57c58b59j19k24K39m46 jgbbhbs s37
55 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56b57c58b59j19k24K39m46j37 gbbhbs r17
56 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56b57c58b59j19k24K39m46K49 qbbhbs r10
```

```
try2
grammars = open("/home/ubuntu/Desktop/grammar.txt")

<terminated> /home/ubuntu/Desktop/syntax/try2/try2.py
47 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56b57 cbjkjmjgbbhbs s58
48 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56b57c58 bjkjmjgbbhbs s59
49 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56b57c58b59 kjmjbhbs s19
50 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56b57c58b59j19 kjmjbhbs s24
51 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56b57c58b59j19k24 jmjbhbs s40
52 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56b57c58b59j19k24j40 mjgbbhbs r17
53 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56b57c58b59j19k24K39 mjgbbhbs s46
54 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56b57c58b59j19k24K39m46 jgbbhbs s37
55 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56b57c58b59j19k24K39m46j37 gbbhbs r17
56 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56b57c58b59j19k24K39m46K49 gbbhbs r10
57 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56b57c58b59G15 gbbhbs s25
58 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56b57c58b59G15g25 bbbhbs s42
59 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56b57c58b59G15g25b42 hbbhbs r8
60 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56b57c58b59D12 hbbhbs s18
61 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56b57c58b59D12h18 bbbhbs r5
62 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56b57c58b59D12D12 bbbhbs r2
63 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56b57c58b59D12C32 bbbhbs r1
64 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56b57c58b59C60 bbbhbs s61
65 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56b57c58b59C60b61 bbbhbs r13
66 0a2B4b7c8b10D12F14 bbs s26
67 0a2B4b7c8b10D12F14b26 bs r7
68 0a2B4b7c8b10D12D12 bs s18
69 0a2B4b7c8b10D12D12h18 $ r5
70 0a2B4b7c8b10D12D12D12 $ r2
71 0a2B4b7c8b10D12D12C32 $ r1
72 0a2B4b7c8b10D12C32 $ r1
73 0a2B4b7c8b10C11 $ r0
74 0a2A3 $ r22
75 0S1 $ ACCEPTED
```

Ex: Parser code for the above problem statement is

```
grammars = open("/home/ubuntu/Desktop/grammar.txt")
G = {}
C = {}
start = ""
terminals = []
nonterminals = []
symbols = []
error=0

def parse_grammar():
    global G, start, terminals, nonterminals, symbols
    for line in grammars:
        line = " ".join(line.split())
        if line == '\n':
            break
        head = line[:line.index("->")].strip()
        prods = [l.strip().split(' ') for l in " ".join(line[line.index("->") + 2:]).split('/')]
        if not start:
            start = head + ""
            G[start] = [[head]]
            nonterminals.append(start)
        if head not in G:
            G[head] = []
        if head not in nonterminals:
            nonterminals.append(head)
        for prod in prods:
            G[head].append(prod)
        for char in prod:
            if not char.isupper() and char not in terminals:
                terminals.append(char)
            elif char.isupper() and char not in nonterminals:
                nonterminals.append(char)
            G[char] = [] #non terminals don't produce other symbols
        symbols = nonterminals+terminals
    first_seen = []

def FIRST(X):
    global first_seen
    first = []
    first_seen.append(X)
    if X in terminals: # CASE 1
        first.append(X)
    elif X in nonterminals:
        for prods in G[X]: # CASE 2
            if prods[0] in terminals and prods[0] not in first:
                first.append(prods[0])
            else: # CASE 3
                for nonterm in prods:
                    if nonterm not in first_seen:
                        for terms in FIRST(nonterm):
                            if terms not in first:
                                first.append(terms)
        first_seen.remove(X)
    return first
    follow_seen = []

def FOLLOW(A):
    global follow_seen
    follow = []
```



```

follow_seen.append(A)
if A == start: # CASE 1
    follow.append('$')
for heads in G.keys():
    for prods in G[heads]:
        follow_head = False
        if A in prods:
            next_symbol_pos = prods.index(A) + 1
            if next_symbol_pos < len(prods): # CASE 2
                for terms in FIRST(prods[next_symbol_pos]):
                    if terms not in follow:
                        follow.append(terms)
            else: # CASE 3
                follow_head = True
        if follow_head and heads not in follow_seen:
            for terms in FOLLOW(heads):
                if terms not in follow:
                    follow.append(terms)
follow_seen.remove(A)
return follow

def closure(I):
    J = I
    while True:
        item_len = len(J) + sum(len(v) for v in J.itervalues())
        for heads in J.keys():
            for prods in J[heads]:
                dot_pos = prods.index('.') #checks if final item or not
                if dot_pos + 1 < len(prods):
                    prod_after_dot = prods[dot_pos + 1]
                    if prod_after_dot in nonterminals:
                        for prod in G[prod_after_dot]:
                            item = ["."] + prod
                            if prod_after_dot not in J.keys():
                                J[prod_after_dot] = [item]
                            elif item not in J[prod_after_dot]:
                                J[prod_after_dot].append(item)
        if item_len == len(J) + sum(len(v) for v in J.itervalues()):
            return J

def GOTO(I, X):
    goto = {}
    for heads in I.keys():
        for prods in I[heads]:
            for i in range(len(prods) - 1):
                if "." == prods[i] and X == prods[i + 1]:
                    temp_prods = prods[:]
                    temp_prods[i], temp_prods[i + 1] = temp_prods[i + 1], temp_prods[i]
                    prod_closure = closure({heads: [temp_prods]})
                    for keys in prod_closure:
                        if keys not in goto.keys():
                            goto[keys] = prod_closure[keys]
                        elif prod_closure[keys] not in goto[keys]:
                            goto[keys].append(prod)
    return goto

def items():
    global C
    i = 1
    C = {'I0': closure({start: [['.'] + G[start][0]]})}

```

```

while True:
    item_len = len(C) + sum(len(v) for v in C.itervalues())
    for I in C.keys():
        for X in symbols:
            if GOTO(C[I], X) and GOTO(C[I], X) not in C.values():
                C['I' + str(i)] = GOTO(C[I], X)
                i += 1
    if item_len == len(C) + sum(len(v) for v in C.itervalues()):
        return

def ACTION(i, a):
    global error
    for heads in C['I' + str(i)]:
        for prods in C['I' + str(i)][heads]:
            for j in range(len(prods) - 1):
                if prods[j] == '.' and prods[j + 1] == a:
                    for k in range(len(C)):
                        if GOTO(C['I' + str(i)], a) == C['I' + str(k)]:
                            if a in terminals:
                                if "r" in parse_table[i][terminals.index(a)]:
                                    if error!=1:
                                        print "ERROR: Shift-Reduce Conflict at State " + str(i) + ", Symbol \" +
str(terminals.index(a))+"\"
                                        error=1
                                    if "s"+str(k) not in parse_table[i][terminals.index(a)]:
                                        parse_table[i][terminals.index(a)] = parse_table[i][terminals.index(a)]+ "/"s" +
                                        str(k)
                                    return parse_table[i][terminals.index(a)]
                                else:
                                    parse_table[i][terminals.index(a)] = "s" + str(k)
                                else:
                                    parse_table[i][len(terminals) + nonterminals.index(a)] = str(k)
                                return "s" + str(k)
    for heads in C['I' + str(i)]:
        if heads != start:
            for prods in C['I' + str(i)][heads]:
                if prods[-1] == '.': #final item
                    k = 0
                for head in G.keys():
                    for Gprods in G[head]:
                        if head == heads and (Gprods == prods[:-1] ) and (a in terminals or a == '$'):
                            for terms in FOLLOW(heads):
                                if terms == '$':
                                    index = len(terminals)
                                else:
                                    index = terminals.index(terms)
                                if "s" in parse_table[i][index]:
                                    if error!=1:
                                        print "ERROR: Shift-Reduce Conflict at State " + str(i) + ", Symbol \" +
str(terms)+"\"
                                        error=1
                                    if "r"+str(k) not in parse_table[i][index]:
                                        parse_table[i][index] = parse_table[i][index]+ "/"r" + str(k)
                                    return parse_table[i][index]
                                elif parse_table[i][index] and parse_table[i][index] != "r" + str(k):
                                    if error!=1:
                                        print "ERROR: Reduce-Reduce Conflict at State " + str(i) + ", Symbol \" +
str(terms)+"\"
                                        error=1
                                    if "r"+str(k) not in parse_table[i][index]:
                                        parse_table[i][index] = parse_table[i][index]+ "/"r" + str(k)

```

```

        return parse_table[i][index]
    else:
        parse_table[i][index] = "r" + str(k)
    return "r" + str(k)
    k += 1
if start in C['I' + str(i)] and G[start][0] + ['.'] in C['I' + str(i)][start]:
    parse_table[i][len(terminals)] = "acc"
    return "acc"
return ""

def print_info():
    print "GRAMMAR:"
    for head in G.keys():
        if head == start:
            continue
        print "{:>{width}} ->".format(head, width=len(max(G.keys(), key=len))),
        num_prods = 0
        for prods in G[head]:
            if num_prods > 0:
                print "/",
            for prod in prods:
                print prod,
            num_prods += 1
        print
    print "\nAUGMENTED GRAMMAR:"
    i = 0
    for head in G.keys():
        for prods in G[head]:
            print "{:>{width}}:".format(str(i), width=len(str(sum(len(v) for v in G.itervalues()) - 1))),
            print "{:>{width}} ->".format(head, width=len(max(G.keys(), key=len))),
            for prod in prods:
                print prod,
            print
            i += 1
    print "\nTERMINALS  :", terminals
    print "NONTERMINALS:", nonterminals
    print "SYMBOLS    :", symbols
    print "\nFIRST:"
    for head in G:
        print "{:>{width}} =".format(head, width=len(max(G.keys(), key=len))),
        print "{",
        num_terms = 0
        for terms in FIRST(head):
            if num_terms > 0:
                print ", ",
            print terms,
            num_terms += 1
        print "]"
    print "\nFOLLOW:"
    for head in G:
        print "{:>{width}} =".format(head, width=len(max(G.keys(), key=len))),
        print "{",
        num_terms = 0
        for terms in FOLLOW(head):
            if num_terms > 0:
                print ", ",
            print terms,
            num_terms += 1
        print "]"
    print "\nITEMS:"

```

```

for i in range(len(C)):
    print 'I' + str(i) + ':'
    for keys in C['I' + str(i)]:
        for prods in C['I' + str(i)][keys]:
            print "{:>{width}} ->".format(keys, width=len(max(G.keys(), key=len))),
            for prod in prods:
                print prod,
            print
    print

for i in range(len(parse_table)):    #len gives number of states
    for j in symbols:
        ACTION(i, j)

print "PARSING TABLE:"
print "+" + "-----+" * (len(terminals) + len(nonterminals) + 1)
print "{: ^8}".format('STATE'),
for terms in terminals:
    print "{: ^7}".format(terms),
print "{: ^7}".format("$"),
for nonterms in nonterminals:
    if nonterms == start:
        continue
    print "{: ^7}".format(nonterms),
print "\n+" + "-----+" * (len(terminals) + len(nonterminals) + 1)
for i in range(len(parse_table)):
    print "{: ^8}".format(i),
    for j in range(len(parse_table[i]) - 1):
        print "{: ^7}".format(parse_table[i][j]),
    print
print "+" + "-----+" * (len(terminals) + len(nonterminals) + 1)

def process_input():
    get_input = raw_input("\nEnter Input: ")
    to_parse = " ".join((get_input + " $").split()).split(" ")
    pointer = 0
    stack = ['0']

    print "\n+-----+-----+-----+-----+"
    print "{: ^8}{: ^28}{: ^28}{: ^28}".format("STEP", "STACK", "INPUT", "ACTION")
    print "+-----+-----+-----+-----+"

    step = 1
    while True:
        curr_symbol = to_parse[pointer]
        top_stack = int(stack[-1])
        stack_content = ""
        input_content = ""

        print "{: ^8}".format(step),
        for i in stack:
            stack_content += i
        print "{:27}".format(stack_content),
        i = pointer
        while i < len(to_parse):
            input_content += to_parse[i]
            i += 1
        print "{:>26} | ".format(input_content),
        step += 1
        get_action = ACTION(top_stack, curr_symbol)

```



```

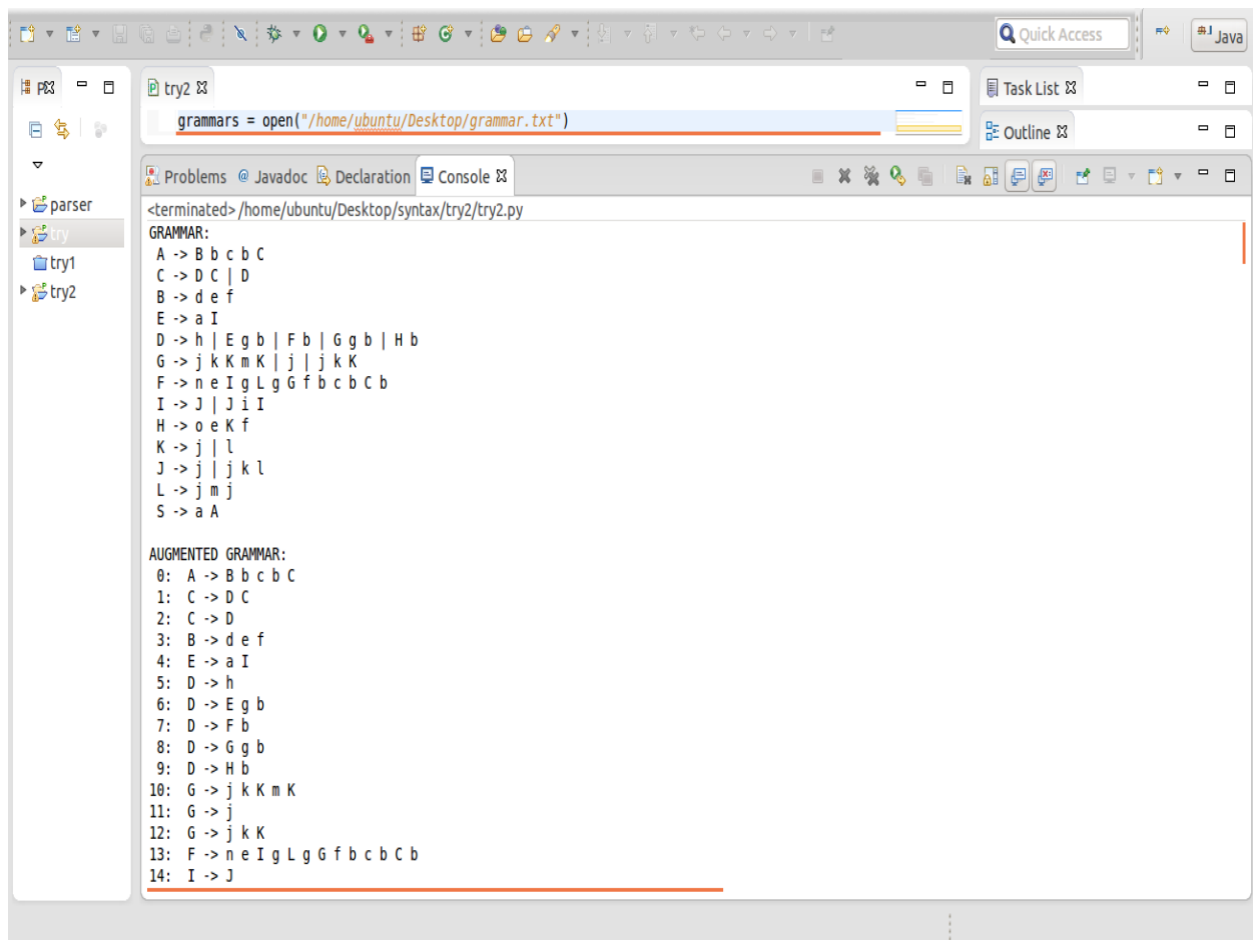
if "/" in get_action:
    print "{:^26}".format(get_action+" So conflict")
    break
if "s" in get_action:
    print "{:^26}".format(get_action)
    stack.append(curr_symbol)
    stack.append(get_action[1:])
    pointer += 1
elif "r" in get_action:
    print "{:^26}".format(get_action)
    i = 0
    for head in G.keys():
        for prods in G[head]:
            if i == int(get_action[1:]):
                for j in range(2 * len(prods)):
                    stack.pop()
                    state = stack[-1]
                    stack.append(head)
                    stack.append(parse_table[int(state)][len(terminals) + nonterminals.index(head)])
                i += 1
    elif get_action == "acc":
        print "{:^26}".format("ACCEPTED")
        break
    else:
        print "ERROR: Unrecognized symbol", curr_symbol, "/"
        break
print "+-----+-----+-----+-----+"

def main():
    parse_grammar()
    items()
    global parse_table
    parse_table = [[ "" for c in range(len(terminals) + len(nonterminals) + 1)] for r in range(len(C))]
    print_info()
    process_input()
if __name__ == '__main__':
    main()

```

We used SLR parsing which is a bottom up parsing. File grammar.txt is taken as input. It contains grammar of our problem statement. Then the program produces augmented grammar. LR(0) items are produced. Goto operation is performed. Parsing table is constructed using functions action and goto for the grammar. Input string is given for parsing. The parser then executes the program until as accept or error action is encountered.

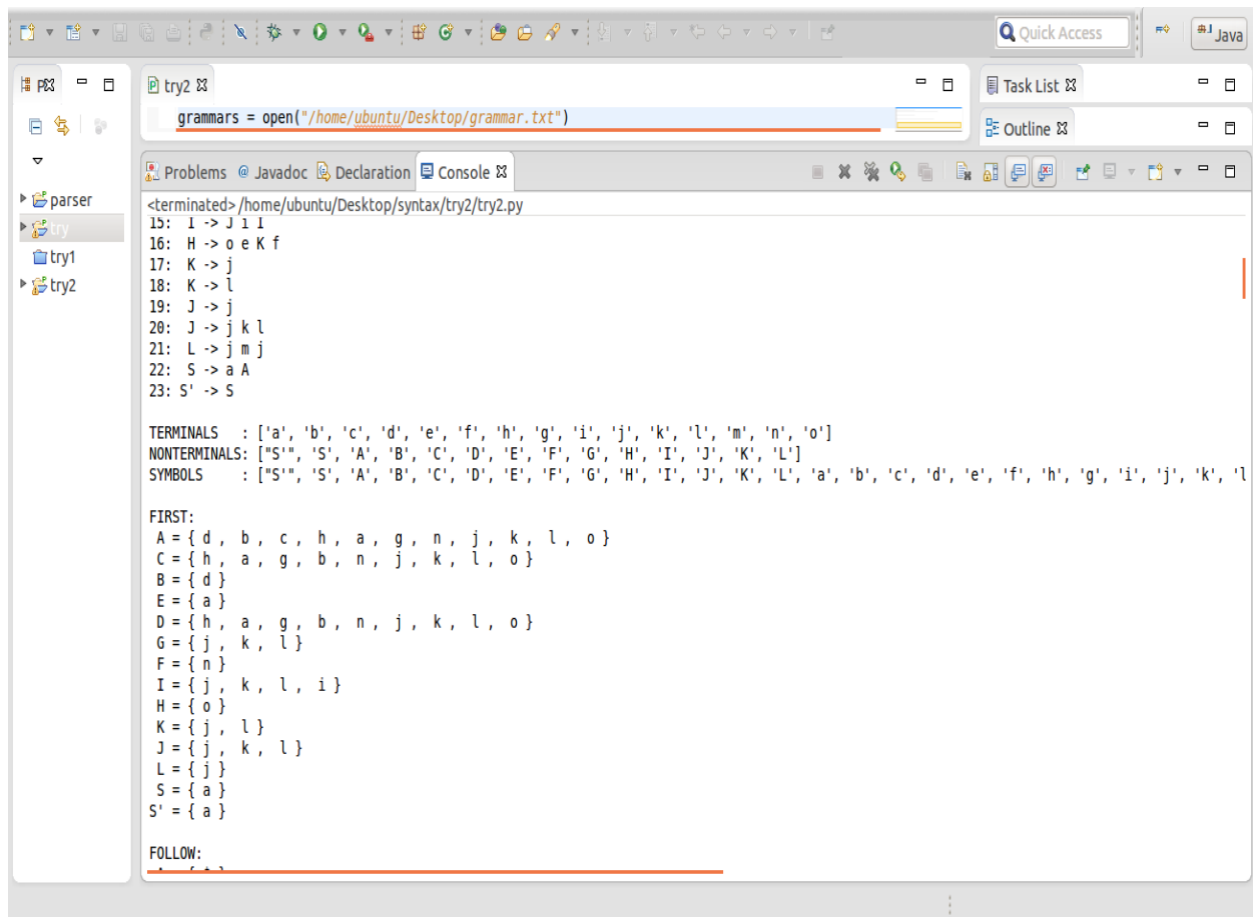
OUTPUT SCREENSHOTS:



```
grammars = open("/home/ubuntu/Desktop/grammar.txt")

<terminated> /home/ubuntu/Desktop/syntax/try2/try2.py
GRAMMAR:
A -> B b c b C
C -> D C | D
B -> d e f
E -> a I
D -> h | E g b | F b | G g b | H b
G -> j k K m K | j | j k K
F -> n e I g L g G f b c b C b
I -> J | J i I
H -> o e K f
K -> j | l
J -> j | j k l
L -> j m j
S -> a A

AUGMENTED GRAMMAR:
0: A -> B b c b C
1: C -> D C
2: C -> D
3: B -> d e f
4: E -> a I
5: D -> h
6: D -> E g b
7: D -> F b
8: D -> G g b
9: D -> H b
10: G -> j k K m K
11: G -> j
12: G -> j k K
13: F -> n e I g L g G f b c b C b
14: I -> J
```



```
grammars = open("/home/ubuntu/Desktop/grammar.txt")

<terminated> /home/ubuntu/Desktop/syntax/try2/try2.py
15: I -> J i I
16: H -> o e K f
17: K -> j
18: K -> l
19: J -> j
20: J -> j k l
21: L -> j m j
22: S -> a A
23: S' -> S

TERMINALS : ['a', 'b', 'c', 'd', 'e', 'f', 'h', 'g', 'i', 'j', 'k', 'l', 'm', 'n', 'o']
NONTERMINALS: ["S'", 'S', 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L']
SYMBOLS : ["S'", 'S', 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'a', 'b', 'c', 'd', 'e', 'f', 'h', 'g', 'i', 'j', 'k', 'l']

FIRST:
A = {d, b, c, h, a, g, n, j, k, l, o}
C = {h, a, g, b, n, j, k, l, o}
B = {d}
E = {a}
D = {h, a, g, b, n, j, k, l, o}
G = {j, k, l}
F = {n}
I = {j, k, l, i}
H = {o}
K = {j, l}
J = {j, k, l}
L = {j}
S = {a}
S' = {a}

FOLLOW:
```

```
grammars = open("/home/ubuntu/Desktop/grammar.txt")

<terminated> /home/ubuntu/Desktop/syntax/try2/try2.py
A -> $
C = { $, b }
B = { b }
E = { g }
D = { h, a, g, b, n, j, k, l, o, $ }
G = { g, f }
F = { b }
I = { g }
H = { b }
K = { m, g, f }
J = { g, i }
L = { g }
S = { $ }
S' = { $ }

ITEMS:
I0:
S -> . a A
S' -> . S

I1:
S' -> S .

I2:
A -> . B b c b C
S -> a . A
B -> . d e f

I3:
S -> a A .

I4:
```

```
grammars = open("/home/ubuntu/Desktop/grammar.txt")

<terminated> /home/ubuntu/Desktop/syntax/try2/try2.py
A -> B . b c b C

I5:
B -> d . e f

I6:
B -> d e . f

I7:
A -> B b . c b C

I8:
A -> B b c . b C

I9:
B -> d e f .

I10:
A -> B b c b . C
C -> . D C
C -> . D
E -> . a I
D -> . h
D -> . E g b
D -> . F b
D -> . G g b
D -> . H b
G -> . j k K m K
G -> . j k K
G -> . j k K
F -> . n e I a l a G f b c b C b
```

The screenshot shows an IDE with a file named `try2` open. The file contains the following code:

```
grammars = open("/home/ubuntu/Desktop/grammar.txt")
```

The IDE also shows a `Problems` panel with the following output:

```
<terminated> /home/ubuntu/Desktop/syntax/try2/try2.py
H -> . o e K f

I11:
A -> B b c b C .

I12:
C -> D . C
C -> . D C
C -> . D
C -> D .
E -> . a I
D -> . h
D -> . E g b
D -> . F b
D -> . G g b
D -> . H b
G -> . j k K m K
G -> . j
G -> . j k K
F -> . n e I g L g G f b c b C b
H -> . o e K f

I13:
D -> E . g b

I14:
D -> F . b

I15:
D -> G . g b
```

The screenshot shows the same IDE with the same file `try2` open. The file contains the same code as the previous screenshot:

```
grammars = open("/home/ubuntu/Desktop/grammar.txt")
```

The IDE also shows the same `Problems` panel output as the previous screenshot:

```
<terminated> /home/ubuntu/Desktop/syntax/try2/try2.py
I16:
D -> H . b

I17:
I -> . J
I -> . J i I
J -> . j
J -> . j k l
E -> a . I

I18:
D -> h .

I19:
G -> j . k K m K
G -> j .
G -> j . k K

I20:
F -> n . e I g L g G f b c b C b

I21:
H -> o . e K f

I22:
I -> . J
I -> . J i I
J -> . j
J -> . j k l
F -> n e . I g L g G f b c b C b
```

```
grammars = open("/home/ubuntu/Desktop/grammar.txt")

<terminated> /home/ubuntu/Desktop/syntax/try2/try2.py
I23:
H -> o e . K f
K -> . j
K -> . l

I24:
K -> . j
K -> . l
K -> . j
K -> . l
G -> j k . K m K
G -> j k . K

I25:
D -> G g . b

I26:
D -> F b .

I27:
E -> a I .

I28:
I -> J .
I -> J . i I

I29:
J -> j .
J -> j . k l

I30:
D -> H b
```

```
grammars = open("/home/ubuntu/Desktop/grammar.txt")

<terminated> /home/ubuntu/Desktop/syntax/try2/try2.py

I31:
D -> E g . b

I32:
C -> D C .

I33:
I -> J i . I
I -> . J
I -> . J i I
J -> . j
J -> . j k l

I34:
J -> j k . l

I35:
F -> n e I . g L g G f b c b c b

I36:
H -> o e K . f

I37:
K -> j .

I38:
K -> l .

I39:
G -> i k K . m K
```

```
grammars = open("/home/ubuntu/Desktop/grammar.txt")

<terminated> /home/ubuntu/Desktop/syntax/try2/try2.py
G -> j k K .

I40:
K -> j .
K -> j .

I41:
K -> l .
K -> l .

I42:
D -> G g b .

I43:
D -> E g b .

I44:
H -> o e K f .

I45:
I -> J i I .

I46:
K -> . j
K -> . l
G -> j k K m . K

I47:
L -> . j m j
F -> n e I g . L g G f b c b C b
```

```
grammars = open("/home/ubuntu/Desktop/grammar.txt")

<terminated> /home/ubuntu/Desktop/syntax/try2/try2.py
I48:
J -> j k l .

I49:
G -> j k K m K .

I50:
F -> n e I g L . g G f b c b C b

I51:
L -> j . m j

I52:
L -> j m . j

I53:
G -> . j k K m K
G -> . j
G -> . j k K
F -> n e I g L g . G f b c b C b

I54:
F -> n e I g L g G . f b c b C b

I55:
L -> j m j .

I56:
F -> n e I g L g G f . b c b C b

I57:
```


try2

grammars = open("/home/ubuntu/Desktop/grammar.txt")

Problems Javadoc Declaration Console

<terminated> /home/ubuntu/Desktop/syntax/try2/try2.py

```

32 |
33 | r1
34 |
35 |
36 |
37 | s44
38 | r17
39 | r18
40 | r12
41 | r17
42 | r18
43 | r8
44 | r6
45 | r16
46 |
47 | r15
48 |
49 | r18
50 | s53
51 |
52 |
53 |
54 | s56
55 | r21
56 | s57
57 | s58
58 | s17
59 | s61
60 | r13
61 |

```

try

try2

grammars = open("/home/ubuntu/Desktop/grammar.txt")

Problems Javadoc Declaration Console

<terminated> /home/ubuntu/Desktop/syntax/try2/try2.py

Enter Input: adefbcabajijijklgbnejklgjmjgkjfbcbjkmjgbbbh

STEP	STACK	INPUT	ACTION
1	0	adefbcabajijijklgbnejklgjmjgkjfbcbjkmjgbbbh	s2
2	0a2	defbcabajijijklgbnejklgjmjgkjfbcbjkmjgbbbh	s5
3	0a2d5	efbcabajijijklgbnejklgjmjgkjfbcbjkmjgbbbh	s6
4	0a2d5e6	fbcbabajijijklgbnejklgjmjgkjfbcbjkmjgbbbh	s9
5	0a2d5e6f9	bcabajijijklgbnejklgjmjgkjfbcbjkmjgbbbh	r3
6	0a2B4	bcabajijijklgbnejklgjmjgkjfbcbjkmjgbbbh	s7
7	0a2B4b7	cbabajijijklgbnejklgjmjgkjfbcbjkmjgbbbh	s8
8	0a2B4b7c8	bajijijklgbnejklgjmjgkjfbcbjkmjgbbbh	s10
9	0a2B4b7c8b10	ajijijklgbnejklgjmjgkjfbcbjkmjgbbbh	s17
10	0a2B4b7c8b10a17	ijijijklgbnejklgjmjgkjfbcbjkmjgbbbh	s29
11	0a2B4b7c8b10a17j29	ijijklgbnejklgjmjgkjfbcbjkmjgbbbh	r19
12	0a2B4b7c8b10a17J28	ijijklgbnejklgjmjgkjfbcbjkmjgbbbh	s33
13	0a2B4b7c8b10a17J28133	ijijklgbnejklgjmjgkjfbcbjkmjgbbbh	s29
14	0a2B4b7c8b10a17J28133j29	ijijklgbnejklgjmjgkjfbcbjkmjgbbbh	r19
15	0a2B4b7c8b10a17J28133J28	ijijklgbnejklgjmjgkjfbcbjkmjgbbbh	s33
16	0a2B4b7c8b10a17J28133J28133	jklgbnejklgjmjgkjfbcbjkmjgbbbh	s29
17	0a2B4b7c8b10a17J28133J28133j29	klgbnejklgjmjgkjfbcbjkmjgbbbh	s34
18	0a2B4b7c8b10a17J28133J28133j29k34	lgbnejklgjmjgkjfbcbjkmjgbbbh	s48
19	0a2B4b7c8b10a17J28133J28133j29k34l48	gbnejklgjmjgkjfbcbjkmjgbbbh	r20
20	0a2B4b7c8b10a17J28133J28133j29k34l48	gbnejklgjmjgkjfbcbjkmjgbbbh	r14
21	0a2B4b7c8b10a17J28133J28133j29k34l48	gbnejklgjmjgkjfbcbjkmjgbbbh	r15
22	0a2B4b7c8b10a17J28133j29k34l48	gbnejklgjmjgkjfbcbjkmjgbbbh	r15
23	0a2B4b7c8b10a17J28133j29k34l48	gbnejklgjmjgkjfbcbjkmjgbbbh	r4
24	0a2B4b7c8b10a17J28133j29k34l48	gbnejklgjmjgkjfbcbjkmjgbbbh	s31
25	0a2B4b7c8b10a17J28133j29k34l48	gbnejklgjmjgkjfbcbjkmjgbbbh	c43

try

try2

```
grammars = open("/home/ubuntu/Desktop/grammar.txt")
```

Problems Javadoc Declaration Console

<terminated> /home/ubuntu/Desktop/syntax/try2/try2.py

```
26 | 0a2B4b7c8b10E13g31b43 | nejklgjmjgkjfbcbjkmjgbbhbs | r6 |
27 | 0a2B4b7c8b10D12 | nejklgjmjgkjfbcbjkmjgbbhbs | s20 |
28 | 0a2B4b7c8b10D12n20 | ejklgmjgkjfbcbjkmjgbbhbs | s22 |
29 | 0a2B4b7c8b10D12n20e22 | jklgmjgkjfbcbjkmjgbbhbs | s29 |
30 | 0a2B4b7c8b10D12n20e22j29 | klgmjgkjfbcbjkmjgbbhbs | s34 |
31 | 0a2B4b7c8b10D12n20e22j29k34 | lgjmjgkjfbcbjkmjgbbhbs | s48 |
32 | 0a2B4b7c8b10D12n20e22j29k34l48 | gjmjmjgkjfbcbjkmjgbbhbs | r20 |
33 | 0a2B4b7c8b10D12n20e22j28 | gjmjmjgkjfbcbjkmjgbbhbs | r14 |
34 | 0a2B4b7c8b10D12n20e22I35 | gjmjmjgkjfbcbjkmjgbbhbs | s47 |
35 | 0a2B4b7c8b10D12n20e22I35g47 | jmjgkjfbcbjkmjgbbhbs | s51 |
36 | 0a2B4b7c8b10D12n20e22I35g47j51 | mjgkjfbcbjkmjgbbhbs | s52 |
37 | 0a2B4b7c8b10D12n20e22I35g47j51m52 | gjgkjfbcbjkmjgbbhbs | s55 |
38 | 0a2B4b7c8b10D12n20e22I35g47j51m52j55 | gjkjfbcbjkmjgbbhbs | r21 |
39 | 0a2B4b7c8b10D12n20e22I35g47L50 | gjkjfbcbjkmjgbbhbs | s53 |
40 | 0a2B4b7c8b10D12n20e22I35g47L50g53 | kjfbcbjkmjgbbhbs | s19 |
41 | 0a2B4b7c8b10D12n20e22I35g47L50g53j19 | kjfbcbjkmjgbbhbs | s24 |
42 | 0a2B4b7c8b10D12n20e22I35g47L50g53j19k24 | jfbcbjkmjgbbhbs | s40 |
43 | 0a2B4b7c8b10D12n20e22I35g47L50g53j19k24j40 | fbcbjkmjgbbhbs | r17 |
44 | 0a2B4b7c8b10D12n20e22I35g47L50g53j19k24K39 | fbcbjkmjgbbhbs | r12 |
45 | 0a2B4b7c8b10D12n20e22I35g47L50g53G54 | fbcbjkmjgbbhbs | s56 |
46 | 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56 | bcbjkmjgbbhbs | s57 |
47 | 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56b57 | cbjkmjgbbhbs | s58 |
48 | 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56b57c58 | bjkjmjgbbhbs | s59 |
49 | 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56b57c58b59 | kjmjmjgbbhbs | s19 |
50 | 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56b57c58b59j19 | kjmjmjgbbhbs | s24 |
51 | 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56b57c58b59j19k24 | mjgbbhbs | s40 |
52 | 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56b57c58b59j19k24j40 | mjgbbhbs | r17 |
53 | 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56b57c58b59j19k24K39 | mjgbbhbs | s46 |
54 | 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56b57c58b59j19k24K39m46 | jgbbhbs | s37 |
55 | 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56b57c58b59j19k24K39m46j37 | gbbhbs | r17 |
56 | 0a2B4b7c8b10D12n20e22I35n47L50n53G54f56b57c58b59j19k24K39m46K49 | qbbhbs | r10 |
```

try2

```
grammars = open("/home/ubuntu/Desktop/grammar.txt")
```

Problems Javadoc Declaration Console

<terminated> /home/ubuntu/Desktop/syntax/try2/try2.py

```
47 | 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56b57 | cbjkmjgbbhbs | s58 |
48 | 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56b57c58 | bjkjmjgbbhbs | s59 |
49 | 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56b57c58b59 | kjmjmjgbbhbs | s19 |
50 | 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56b57c58b59j19 | kjmjmjgbbhbs | s24 |
51 | 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56b57c58b59j19k24 | mjgbbhbs | s40 |
52 | 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56b57c58b59j19k24j40 | mjgbbhbs | r17 |
53 | 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56b57c58b59j19k24K39 | mjgbbhbs | s46 |
54 | 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56b57c58b59j19k24K39m46 | jgbbhbs | s37 |
55 | 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56b57c58b59j19k24K39m46j37 | gbbhbs | r17 |
56 | 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56b57c58b59j19k24K39m46K49 | gbbhbs | r10 |
57 | 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56b57c58b59G15 | gbbhbs | s25 |
58 | 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56b57c58b59G15g25 | bbbhbs | s42 |
59 | 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56b57c58b59G15g25b42 | hbbhbs | r8 |
60 | 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56b57c58b59D12 | hbbhbs | s18 |
61 | 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56b57c58b59D12h18 | bbhs | r5 |
62 | 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56b57c58b59D12D12 | bbhs | r2 |
63 | 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56b57c58b59D12C32 | bbhs | r1 |
64 | 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56b57c58b59C60 | bbhs | s61 |
65 | 0a2B4b7c8b10D12n20e22I35g47L50g53G54f56b57c58b59C60b61 | bhs | r13 |
66 | 0a2B4b7c8b10D12F14 | bhs | s26 |
67 | 0a2B4b7c8b10D12F14b26 | hs | r7 |
68 | 0a2B4b7c8b10D12D12 | hs | s18 |
69 | 0a2B4b7c8b10D12D12h18 | $ | r5 |
70 | 0a2B4b7c8b10D12D12D12 | $ | r2 |
71 | 0a2B4b7c8b10D12D12C32 | $ | r1 |
72 | 0a2B4b7c8b10D12C32 | $ | r1 |
73 | 0a2B4b7c8b10C11 | $ | r0 |
74 | 0a2A3 | $ | r22 |
75 | 0S1 | $ | ACCEPTED |
```

OUTPUT FOR SYNTAX ERROR CONDITION:

```
grammars = open("/home/ubuntu/Desktop/grammar.txt")
G = {}
C = {}
start = ""
terminals = []
nonterminals = []
symbols = []
error=0

def parse_grammar():
    global G, start, terminals, nonterminals, symbols
    for line in grammars:
        line = " ".join(line.split())
        if line == '\n':
            break
        head = line[line.index(">")+1:].strip()
        prods = [l.strip().split(' ') for l in line[line.index(">")+2:].split('|')]
        if not start:
```

<terminated> /home/ubuntu/Desktop/syntax/try2/try2.py
Enter Input: a d e f b c b

STEP	STACK	INPUT	ACTION
1	0	adefbcb\$	s2
2	0a2	defbcb\$	s5
3	0a2d5	efbcb\$	s6
4	0a2d5e6	fbcb\$	s9
5	0a2d5e6f9	bcbs\$	r3
6	0a2B4	bcbs\$	s7
7	0a2B4b7	cb\$	s8
8	0a2B4b7c8	b\$	s10
9	0a2B4b7c8b10	\$	ERROR: Unrecognized symbol \$

```
grammars = open("/home/ubuntu/Desktop/grammar.txt")
G = {}
C = {}
start = ""
terminals = []
nonterminals = []
symbols = []
error=0

def parse_grammar():
    global G, start, terminals, nonterminals, symbols
    for line in grammars:
        line = " ".join(line.split())
        if line == '\n':
            break
        head = line[line.index(">")+1:].strip()
        prods = [l.strip().split(' ') for l in line[line.index(">")+2:].split('|')]
        if not start:
            start = head + " "
            G[start] = [[head]]
            nonterminals.append(start)
        if head not in G:
```

<terminated> /home/ubuntu/Desktop/syntax/try2/try2.py
Enter Input: a d f b c b a j i j k l g b n e j k l g j m j g j k j f b c b j k j m j g b h b b h

STEP	STACK	INPUT	ACTION
1	0	adfbcbajijklgbnejklgjmjgjkjfbcbjkmjgbbbh\$	s2
2	0a2	dfbcbajijklgbnejklgjmjgjkjfbcbjkmjgbbbh\$	s5
3	0a2d5	fbcbajijklgbnejklgjmjgjkjfbcbjkmjgbbbh\$	ERROR: Unrecognized symbol f

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.

BIBLIOGRAPHY

- PRINCIPLES OF COMPILER DESIGN-ALFRED AHO, JEFFERY D.ULMAN