Chapter - 4

Introduction to Compiler

- A compiler is a translator that converts the high-level language into the machine language.
- High-level language is written by a developer and machine language can be understood by the processor.
- Compiler is used to show errors to the programmer.
- The main purpose of compiler is to change the code written in one language without changing the meaning of the program.
- When you execute a program which is written in HLL programming language then it executes into two parts.
- In the first part, the source program compiled and translated into the object program (low level language).
- In the second part, object program translated into the target program through the assembler.

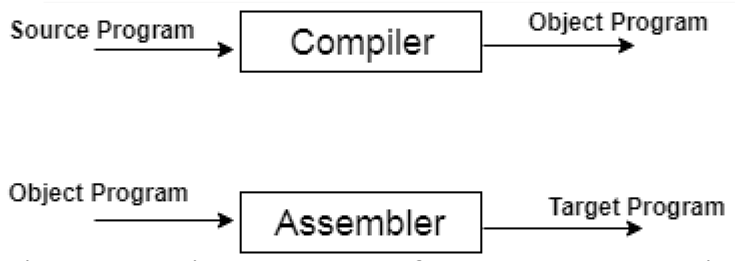
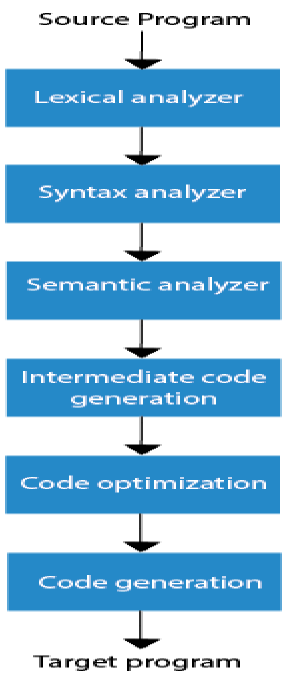


Fig: Execution process of source program in Compiler

Phases of Compiler:

The compilation process contains the sequence of various phases. Each phase takes source program in one representation and produces output in another representation. Each phase takes input from its previous stage.

There are the various phases of compiler:



Lexical Analysis:
Lexical analyzer phase is the first phase of compilation process. It takes source code as input. It reads the source program one character at a time and converts it into meaningful lexemes. Lexical analyzer represents these lexemes in the form of tokens.

Syntax Analysis
Syntax analysis is the second phase of compilation process. It takes tokens as input and generates a parse tree as output. In syntax analysis phase, the parser checks that the expression made by the tokens is syntactically correct or not.

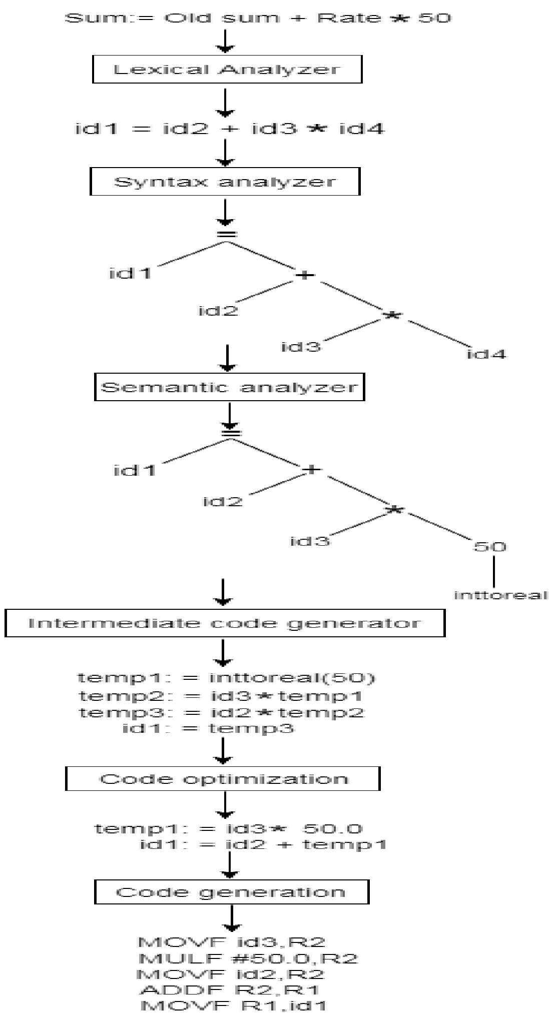
Semantic Analysis
Semantic analysis is the third phase of compilation process. It checks whether the parse tree follows the rules of language. Semantic analyzer keeps track of identifiers, their types and expressions. The output of semantic analysis phase is the annotated tree syntax.

Intermediate Code Generation
In the intermediate code generation, compiler generates the source code into the intermediate code. Intermediate code is generated between the high-level language and the machine language. The intermediate code should be generated in such a way that you can easily translate it into the target machine code.

Code Optimization
Code optimization is an optional phase. It is used to improve the intermediate code so that the output of the program could run faster and take less space. It removes the unnecessary lines of the code and arranges the sequence of statements in order to speed up the program execution.

Code Generation
Code generation is the final stage of the compilation process. It takes the optimized intermediate code as input and maps it to the target machine language. Code generator translates the intermediate code into the machine code of the specified computer.

Example:

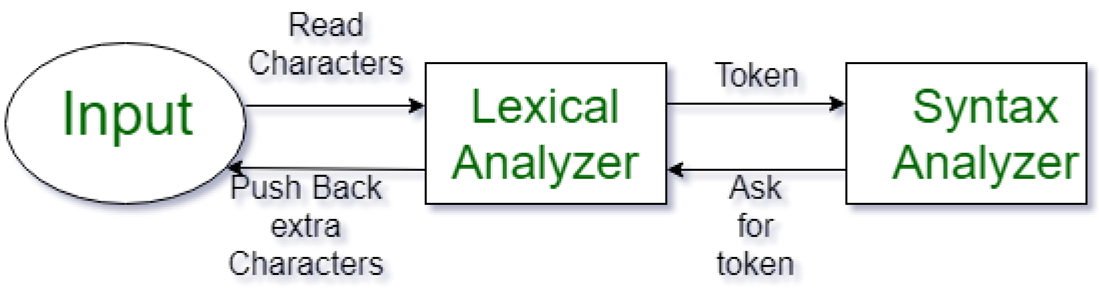


Lexical Analysis:

Lexical Analysis is the first phase of a compiler that takes the input as a source code written in a high-level language. The purpose of lexical analysis is that it aims to read the input code and break it down into meaningful elements called **tokens**. Those tokens are turned into building blocks for other phases of compilation.

What is Lexical Analysis?

Lexical analysis is the process of breaking down the source code of the program into smaller parts, called **tokens**, such that a computer can easily understand. These tokens can be individual words or symbols in a sentence, such as keywords, variable names, numbers, and punctuation. It is also known as a **scanner**. Lexical Analysis can be implemented with the Deterministic Finite Automata. The output generated from Lexical Analysis are a sequence of tokens sent to the parser for syntax analysis.



What is a Token?

A lexical token is a sequence of characters that can be treated as a unit in the grammar of the programming languages.

Example of tokens

- Type token (id, number, real, . . .)
- Punctuation tokens (IF, void, return, . . .)
- Alphabetic tokens (keywords)

Keywords; Examples - for, while, if etc.
Identifier; Examples - Variable name, function name, etc.
Operators; Examples '+', '++', '-' etc.
Separators; Examples ',' ';' etc

Example of Non-Tokens

- Comments, preprocessor directive, macros, blanks, tabs, newline, etc.

What is a Lexeme?

The sequence of characters matched by a pattern to form the corresponding token or a sequence of input characters that comprises a single token is called a lexeme. eg- “float”, “abs_zero_Kelvin”, “=”, “-”, “273”, “;” .

How Lexical Analyzer Works?

- **Input preprocessing:** This stage involves cleaning up the input text and preparing it for lexical analysis. This may include removing comments, whitespace, and other non-essential characters from the input text.
- **Tokenization:** This is the process of breaking the input text into a sequence of tokens. This is usually done by matching the characters in the input text against a set of patterns or regular expressions that define the different types of tokens.
- **Token classification:** In this stage, the lexer determines the type of each token. For example, in a programming language, the lexer might classify keywords, identifiers, operators, and punctuation symbols as separate token types.
- **Token validation:** In this stage, the lexer checks that each token is valid according to the rules of the programming language. For example, it might check that a variable name is a valid identifier, or that an operator has the correct syntax.

- **Output generation**: In this final stage, the lexer generates the output of the lexical analysis process, which is typically a list of tokens. This list of tokens can then be passed to the next stage of compilation or interpretation.
  - The lexical analyzer identifies the error with the help of the automation machine and the grammar of the given language on which it is based like C, C++, and gives row number and column number of the error.
  - Suppose we pass a statement through lexical analyzer: **a = b + c;**
  - It will generate token sequence like this: **id=id+id**;
  - Where each id refers to it's variable in the symbol table referencing all details For example, consider the program

  -
    ```
    int main()
    {
      // 2 variables
      int a, b;
      a = 10;
     return 0;
    }
    ```
  - All the valid tokens are:

  -
    ```
    'int'  'main'  '('  ')'  '{'  'int'  'a' ','  'b'  ';'
     'a'  '='  '10'  ';' 'return'  '0'  ';'  '}'
    ```

Exercise 1: Count number of tokens:

```
int main()
{
  int a = 10, b = 20;
  printf("sum is:%d",a+b);
  return 0;
}
```

Answer: Total number of token: 27

Input buffering:

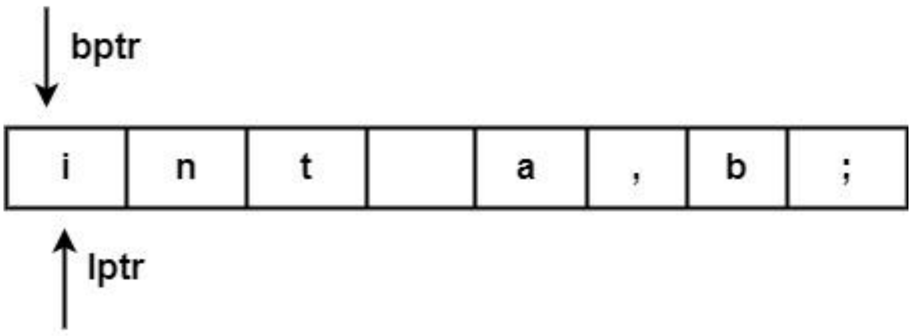
<u>Lexical Analysis</u> has to access **<u>secondary memory</u>** each time to identify tokens. It is time-consuming and costly. So, the input strings are stored into a buffer and then scanned by Lexical Analysis.

Lexical Analysis scans input string from left to right one character at a time to identify tokens. It uses two pointers to scan tokens −

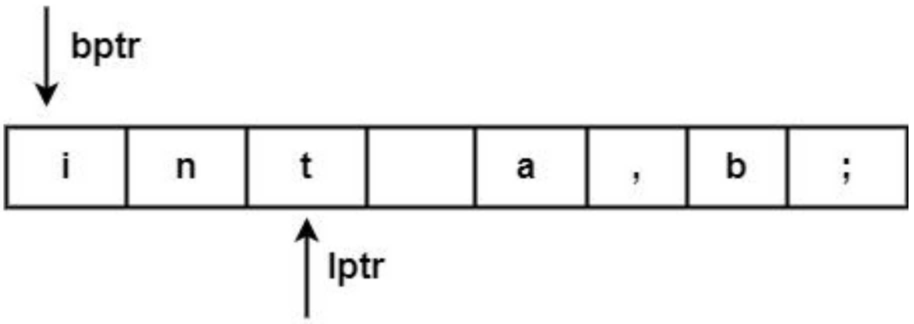
- **Begin Pointer (bptr)** − It points to the beginning of the string to be read.
- **Look Ahead Pointer (lptr)** − It moves ahead to search for the end of the token.

Example – For statement int a, b;

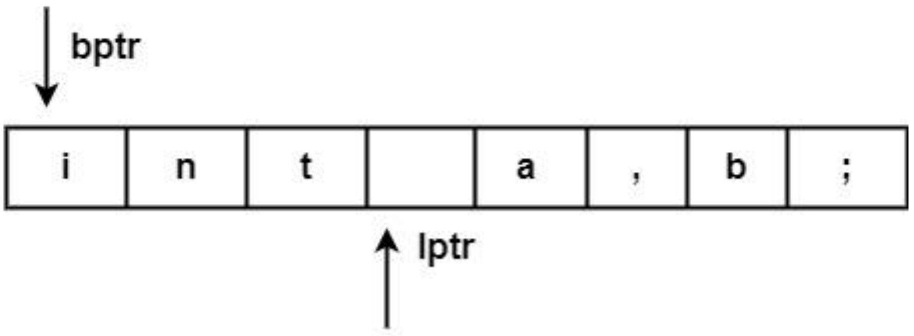
- Both pointers start at the beginning of the string, which is stored in the buffer.



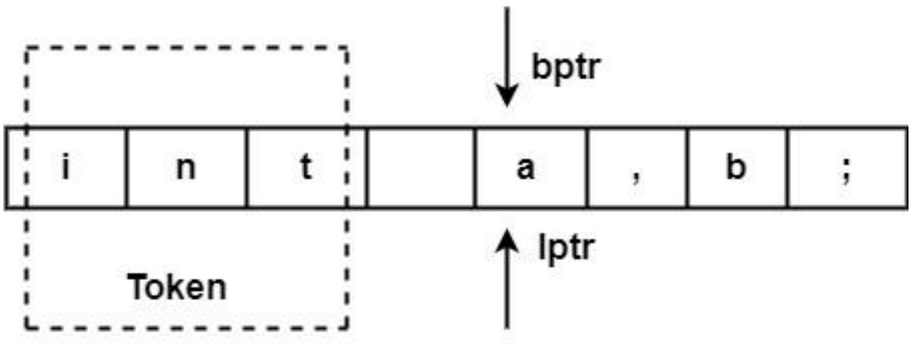
- Look Ahead Pointer scans buffer until the token is found.



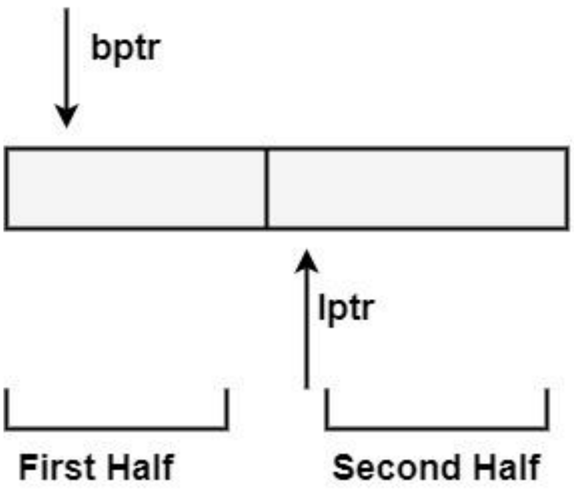
- The character ("blank space") beyond the token ("int") have to be examined before the token ("int") will be determined.



- After processing token ("int") both pointers will set to the next token ('a'), & this process will be repeated for the whole program.



A buffer can be divided into two halves. If the look Ahead pointer moves towards halfway in First Half, the second half is filled with new characters to be read. If the look Ahead pointer moves towards the right end of the buffer of the second half, the first half will be filled with new characters, and it goes on.



Input Buffering

Sentinels − Sentinels are used to making a check, each time when the forward pointer is converted, a check is completed to provide that one half of the buffer has not converted off. If it is completed, then the other half should be reloaded.

Buffer Pairs − A specialized buffering technique can decrease the amount of overhead, which is needed to process an input character in transferring characters. It includes two buffers, each includes N-character size which is reloaded alternatively.

There are two pointers such as the lexeme Begin and forward are supported. Lexeme Begin points to the starting of the current lexeme which is discovered. Forward scans ahead before a match for a pattern are discovered. Before a lexeme is initiated, lexeme begin is set to the character directly after the lexeme which is only constructed, and forward is set to the character at its right end.

Preliminary Scanning – Certain processes are best performed as characters are moved from the source file to the buffer. For example, it can delete comments. Languages like **FORTRAN** which ignores blank can delete them from the character stream. It can also collapse strings of several blanks into one blank. Pre-processing the character stream being subjected to lexical analysis saves the trouble of moving the look ahead pointer back and forth over a string of blanks.

Specification of Tokens

Specification of tokens depends on the pattern of the lexeme. Here we will be using regular expressions to specify the different types of patterns that can actually form tokens.

Although the regular expressions are inefficient in specifying all the patterns forming tokens. Yet it reveals almost all types of pattern that forms a token.

There are 3 specifications of tokens:

1. String
2. Language
3. Regular Expression

1. String

- An **alphabet** or character class is a finite set of symbols.
- A **string** over an alphabet is a finite sequence of symbols drawn from that alphabet.
- In language theory, the term "word" is often used as synonyms for "string".
- The length of a string s, usually written |s|, is the number of occurrences of symbols in s. For example, "banana" is a string of length six.
- The empty string, denoted **ε**, is the string of length zero.

Operations on String:

1. Prefix of String

The prefix of the string is the preceding symbols present in the string and the string(s) itself.

For Example: s = abcd

The prefix of the string abcd: ∈, a, ab, abc, abcd

2. Suffix of String

Suffix of the string is the ending symbols of the string and the string(s) itself.

For Example: s = abcd

Suffix of the string abcd: ∈, d, cd, bcd, abcd

3. Proper Prefix of String

The proper prefix of the string includes all the prefixes of the string excluding ∈ and the string(s) itself.

Proper Prefix of the string abcd: a, ab, abc

4. Proper Suffix of String
The proper suffix of the string includes all the suffixes excluding ∈ and the string(s) itself.

Proper Suffix of the string abcd: d, cd, bcd

5. Substring of String

You may also like...

- Three Address Code in Compiler Design
- Regular Expression in Compiler Design
- Input Buffering in Compiler Design

The substring of a string s is obtained by deleting any prefix or suffix from the string.

Substring of the string abcd: ∈, abcd, bcd, abc, …

6. Proper Substring of String
The proper substring of a string s includes all the substrings of s excluding ∈ and the string(s) itself.

Proper Substring of the string abcd: bcd, abc, cd, ab…

7. Subsequence of String
The subsequence of the string is obtained by eliminating zero or more (not necessarily consecutive) symbols from the string.

A subsequence of the string abcd: abd, bcd, bd, …

8. Concatenation of String
If s and t are two strings, then st denotes concatenation.

s = abc t = def

Concatenation of string s and t i.e. st = abcdef
2. Language
A **language** is any countable set of strings over some fixed alphabet.

Operation on Language
As we have learnt language is a set of strings that are constructed over some fixed alphabets. Now the operation that can be performed on languages are:

1. Union
Union is the most common set operation. Consider the two languages L and M. Then the union of these two languages is denoted by:

L ∪ M = { s | s is in L or s is in M}

That means the string s from the union of two languages can either be from language L or from language M.

If L = {a, b} and M = {c, d} Then L ∪ M = {a, b, c, d}

2. Concatenation
Concatenation links the string from one language to the string of another language in a series in all possible ways. The concatenation of two different languages is denoted by:

L · M = {st | s is in L and t is in M} If L = {a, b} and M = {c, d}

Then L · M = {ac, ad, bc, bd}

3. Kleene Closure
Kleene closure of a language L provides you with a set of strings. This set of strings is obtained by concatenating L zero or more time. The Kleene closure of the language L is denoted by:

If L = {a, b} then L* = {∈, a, b, aa, bb, aaa, bbb, …}

4. Positive Closure
The positive closure on a language L provides a set of strings. This set of strings is obtained by concatenating 'L' one or more times. It is denoted by:

It is similar to the Kleene closure. Except for the term L^0, i.e. L^+ excludes ∈ until it is in L itself.

If L = {a, b} then L+ = {a, b, aa, bb, aaa, bbb, …}

So, these are the four operations that can be performed on the languages in the lexical analysis phase.

3. Regular Expression
A regular expression is a sequence of symbols used to specify lexeme patterns. A regular expression is helpful in describing the languages that can be built using operators such as union, concatenation, and closure over the symbols.

A regular expression 'r' that denotes a language L(r) is built recursively over the smaller regular expression using the rules given below.

The following rules define the regular expression over some alphabet Σ and the languages denoted by these regular expressions.

1. ∈ is a regular expression that denotes a language L(∈). The language L(∈) has a set of strings {∈} which means that this language has a single empty string.
2. If there is a symbol 'a' in Σ then 'a' is a regular expression that denotes a language L(a). The language L(a) = {a} i.e. the language has

only one string of length one and the string holds 'a' in the first position.

3. Consider the two regular expressions r and s then:

- r|s denotes the language L(r) ∪ L(s).
- (r) (s) denotes the language L(r) · L(s).
- (r)* denotes the language (L(r))*.
- (r)+ denotes the language L(r).

Regular Definitions:

Giving names to regular expressions is referred to as a Regular definition. If Σ is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form.

$d_1 \rightarrow r_1$

$d_2 \rightarrow r_2$

………

$d_n \rightarrow r_n$

- Each d_i is a distinct name.
- Each r_i is a regular expression over the alphabet Σ U $\{d_1, d_2, . . . , d_{i-1}\}$.

Example: Identifiers is the set of strings of letters and string beginning with a letter. Regular definition for this set:

letter → A | B | …. | Z | a | b | …. | z | digit → 0 | 1 | …. | 9

id → letter (letter | digit) *.

Recognition of Tokens

- Tokens obtained during lexical analysis are **recognized by Finite Automata**.
- Finite Automata (FA) is a simple idealized machine that can be used to recognize patterns within input taken from a character set or alphabet (denoted as C). The primary task of an FA is to accept or reject an input based on whether the defined pattern occurs within the input.
- There are two notations for representing Finite Automata. They are:

1. **Transition Table**
2. **Transition Diagram**

1. Transition Table
It is a tabular representation that lists all possible transitions for each state and input symbol combination.

EXAMPLE
Assume the following grammar fragment to generate a specific language

where the terminals **if**, **then**, **else**, **relop**, **id** and **num** generates sets of strings given by following regular definitions.

You may also like...

- Type Checking in Compiler Design
- Input Buffering in Compiler Design
- Code Optimization in Compiler Design - BtechVibes

- where letter and digits are defined as - (**letter → [A-Z a-z]** & **digit → [0-9]**)
- For this language, the lexical analyzer will recognize the keywords **if**, **then**, and **else**, as well as lexemes that match the patterns for **relop**, **id**, and **number**.
- To simplify matters, we make the common assumption that keywords are also reserved words: that is they cannot be used as identifiers.
- The num represents the unsigned integer and real numbers of Pascal.
- In addition, we assume lexemes are separated by white space, consisting of nonnull sequences of blanks, tabs, and newlines.
- Our lexical analyzer will strip out white space. It will do so by comparing a string against the regular definition **ws**, below.

- If a match for **ws** is found, the lexical analyzer does not return a token to the parser.
- It is the following token that gets returned to the parser.

2. Transition Diagram
It is a directed labeled graph consisting of nodes and edges. Nodes represent states, while edges represent state transitions.

Components of Transition Diagram

1. One state is labelled the **Start State.** It is the initial state of transition diagram where control resides when we begin to recognize a token.
2. Position is a transition diagram are drawn as circles and are called states.
3. The states are connected by **Arrows** called edges. Labels on edges are indicating the input characters

4. Zero or more **final** states or **Accepting** states are represented by double circle in which the tokens has been found.
5. **Example:**

- Where state "1" is initial state and state 3 is final state.

Here is the transition diagram of Finite Automata that recognizes the lexemes matching the token **relop.**

Here is the Finite Automata Transition Diagram for the Identifiers and Keywords.

Here is the Finite Automata Transition Diagram for recognizing white spaces.

Note:

These Finite Automata can be constructed using either the transition diagram or the transition table representation. Both transition diagrams and transition tables serve the same purpose of defining and representing the behavior of an FA. They provide different visual and structural representations, allowing designers to choose the format that best suits their preferences or requirements.

A LANGUAGE FOR SPECIFYING LEXICAL ANALYZER
There is a wide range of tools for constructing lexical analyzers.
Lex
YACC

Lex is a computer program that generates lexical analyzers. Lex is commonly used with the yacc parser generator.
Creating a lexical analyzer

• First, a specification of a lexical analyzer is prepared by creating a program lex.l in the Lex language. Then, lex.l is run through the Lex compiler to produce a C program lex.yy.c.

• Finally, lex.yy.c is run through the C compiler to produce an object progra m a.out, which is the lexical analyzer that transforms an input stream into a sequence of tokens.

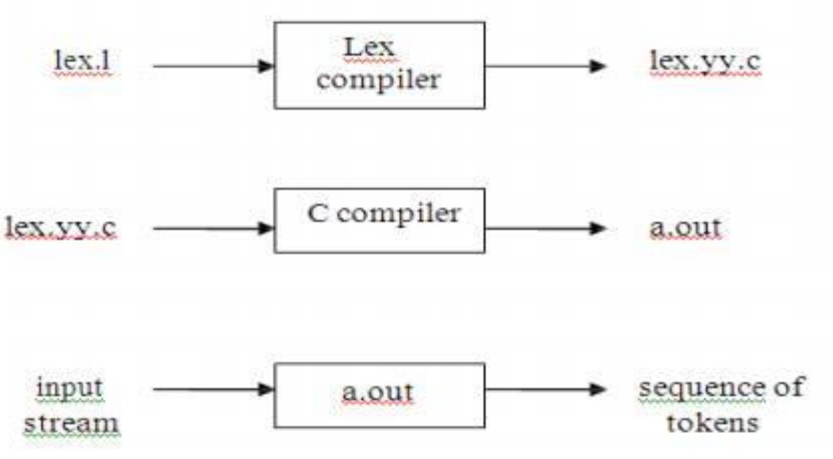


Fig1.11 Creating a lexical analyzer with lex

Creating a lexical analyzer with lex

Lex Specification

A Lex program consists of three parts:

{ definitions }
%%
{ rules }
%%
{ user subroutines }

Definitions include declarations of variables, constants, and regular definitions

Ø **Rules** are statements of the form

p_1 {action$_1$}
p_2 {action$_2$}
…
p_n {action$_n$}

where p_i is regular expression and action$_i$ describes what action the lexical analyzer should take
when pattern p_i matches a lexeme. Actions are written in C code.

¬ **User subroutines** are auxiliary procedures needed by the actions. These can be compiled separately and loaded with the lexical analyzer.

YACC- YET ANOTHER COMPILER-COMPILER

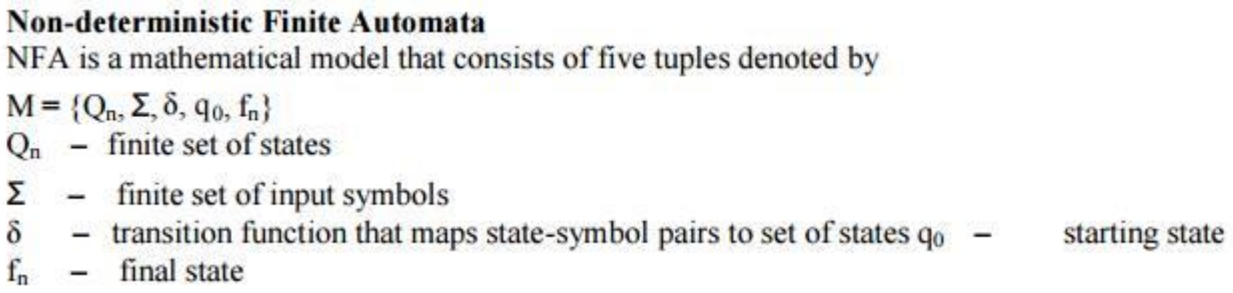
Yacc provides a general tool for describing the input to a computer program. The Yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized.

Yacc turns such a specification into a subroutine that handles the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine.

Finite Automata

Finite Automata is one of the mathematical models that consist of a number of states and edges. It is a transition diagram that recognizes a regular expression or grammar.
There are tow types of Finite Automata :

· Non-deterministic Finite Automata (NFA)
· Deterministic Finite Automata (DFA)

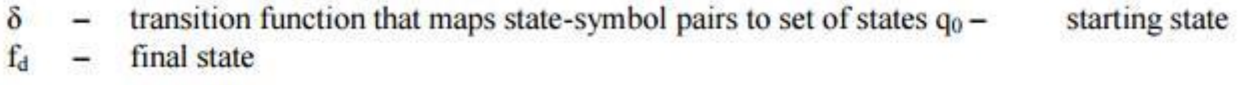


Non-deterministic Finite Automata
NFA is a mathematical model that consists of five tuples denoted by
$M = \{Q_n, \Sigma, \delta, q_0, f_n\}$
Q_n – finite set of states
Σ – finite set of input symbols
δ – transition function that maps state-symbol pairs to set of states q_0 – starting state
f_n – final state

Deterministic Finite Automata
DFA is a special case of a NFA in which i) no state has an ε-transition.

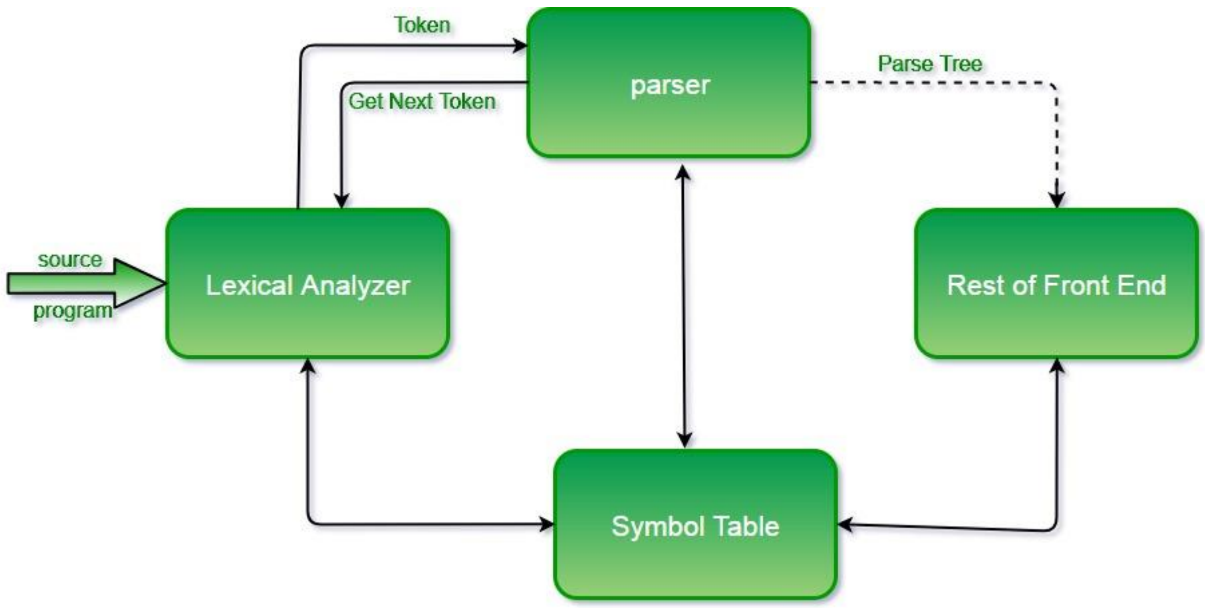
ii) there is at most one transition from each state on any input.

DFA has five tuples denoted by
$M = \{Q_d, \Sigma, \delta, q_0, f_d\}$
Q_d – finite set of states
Σ – finite set of input symbols



δ – transition function that maps state-symbol pairs to set of states q_0 – starting state
f_d – final state

What is the Role of Parser?
In the syntactic checking step, a compiler checks whether or not tokens grouped by the lexical classifier are classified according to the syntactic rules of the language This is done by the classifier. The parser receives a string of tokens from the lexical parser and checks that the string must be a native language. It detects and reports any syntax errors and generates a parse tree from which intermediate code can be generated.



Parser

Types of Parsing

The parsing is divided into two types, which are as follows:

- Top-down Parsing
- Bottom-up Parsing

Top-Down Parsing

Top-down parsing attempts to build the parse tree from the root node to the leaf node. The top-down parser will start from the start symbol and proceed to the string. It follows the leftmost derivation. In leftmost derivation, the leftmost non-terminal in each sentential is always chosen.

- Recursive parsing or predictive parsing are other names for top-down parsing.
- A parse tree is built for an input string using bottom-up parsing.
- When parsing is done top-down, the input symbol is first transformed into the start symbol.

The top-down parsing is further categorized as follows:

1. With Backtracking

- Brute Force Technique

2. Without Backtracking

- Recursive Descent Parsing
- Predictive Parsing or Non-Recursive Parsing or LL(1) Parsing or Table Driver Parsing

Bottom-Up Parsing

Bottom-up parsing builds the parse tree from the leaf node to the root node. The bottom-up parsing will reduce the input string to the start symbol. It traces the rightmost derivation of the string in reverse. Bottom-up parsers are also known as shift-reduce parsers.

1. Shift-reduce parsing is another name for bottom-up parsing.
2. A parse tree is built for an input string using bottom-up parsing.

3. When parsing from the bottom up, the process begins with the input symbol and builds the parse tree up to the start symbol by reversing the rightmost string derivations.

Generally, bottom-up parsing is categorized into the following types:

1. LR parsing/Shift Reduce Parsing: Shift reduce Parsing is a process of parsing a string to obtain the start symbol of the grammar.

- LR(0)
- SLR(1)
- LALR
- CLR

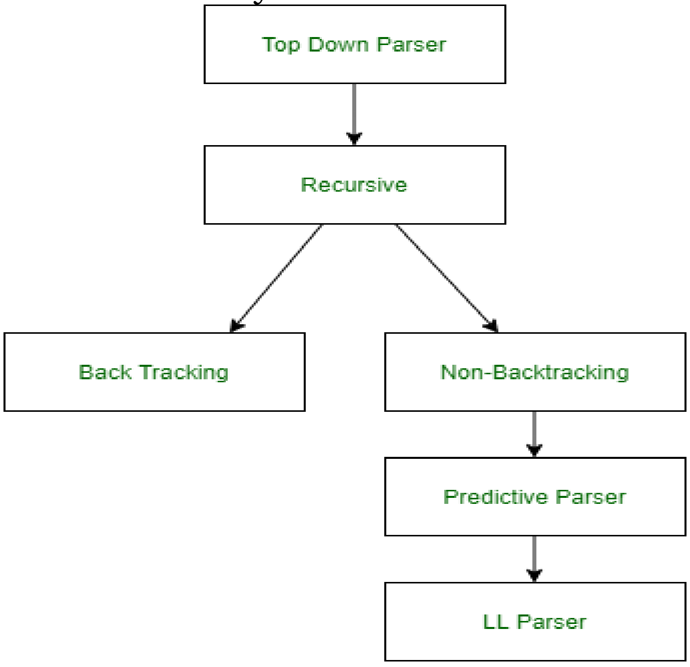
2. Operator Precedence Parsing: The grammar defined using operator grammar is known as operator precedence parsing. In operator precedence parsing there should be no null production and two non-terminals should not be adjacent to each other.

Conclusion

Parsing is an important step in understanding the structure of the source code and following the rules of the language. Top-down and bottom-up parsers provide different ways to create a parse tree, each with its own advantages. While top-down parsers work by extending the grammar from the starting symbol, bottom-up parsers reduce the input to the starting symbol. Both have ways of handling grammar well and spotting errors in the source code.

Predictive PARSING

In this, we will cover the overview of Predictive Parser and mainly focus on the role of Predictive Parser. And will also cover the algorithm for the implementation of the Predictive parser algorithm and finally will discuss an example by implementing the algorithm for precedence parsing. Let’s discuss it one by one.



Predictive Parser :
A predictive parser is a recursive descent parser with no backtracking or backup. It is a top-down parser that does not require backtracking. At each step, the choice of the rule to be expanded is made upon the next terminal symbol.
Consider

```
A -> A1 | A2 | ... | An
```

If the non-terminal is to be further expanded to 'A', the rule is selected based on the current input symbol 'a' only.
Predictive Parser Algorithm :

1. Make a transition diagram(DFA/NFA) for every rule of grammar.
2. Optimize the DFA by reducing the number of states, yielding the final transition diagram.
3. Simulate the string on the transition diagram to parse a string.
4. If the transition diagram reaches an accept state after the input is consumed, it is parsed.

Consider the following grammar –

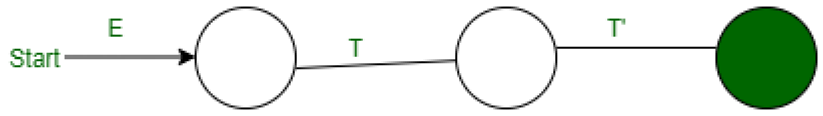
```
E->E+T|T
T->T*F|F
F->(E)|id
```

After removing left recursion, left factoring

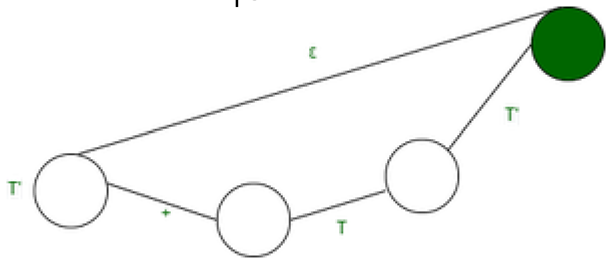
```
E->TT'
T'->+TT'|ε
T->FT''
T''->*FT''|ε
F->(E)|id
```

STEP 1:
Make a transition diagram(DFA/NFA) for every rule of grammar.

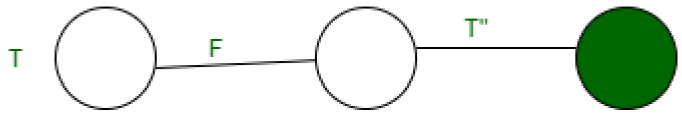
- E->TT'



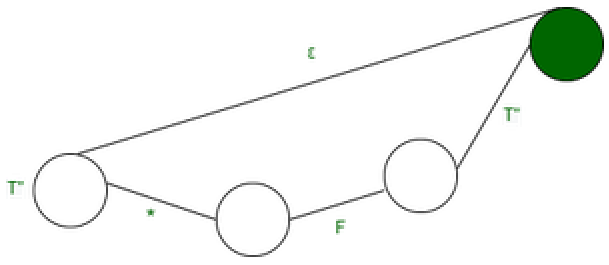
- T'->+TT'|ε



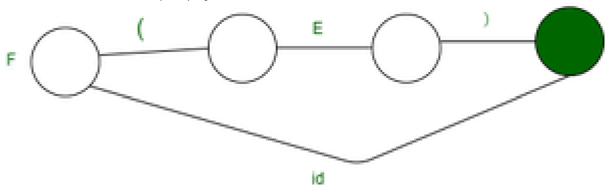
- T->FT"



- T"->*FT"|ε

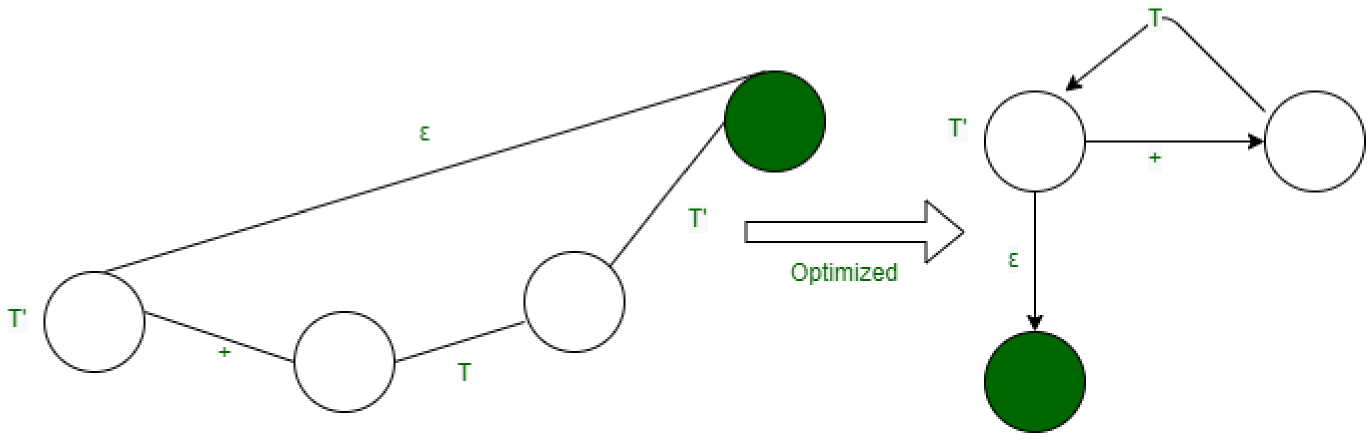


- F->(E)|id

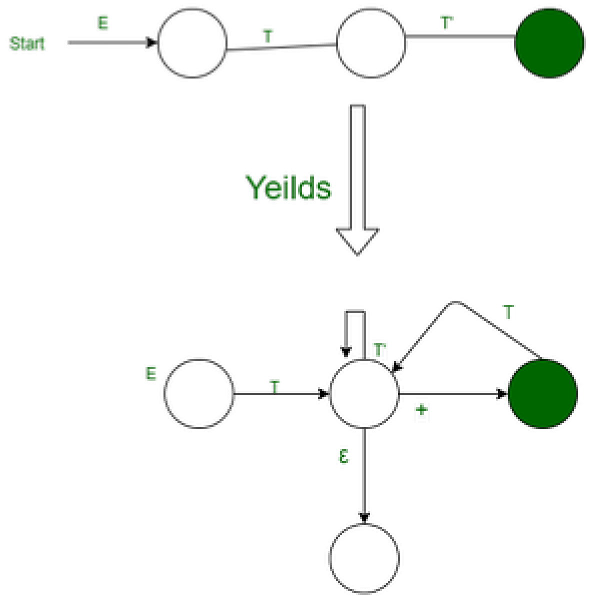


STEP 2:
Optimize the DFA by decreases the number of states, yielding the final transition diagram.

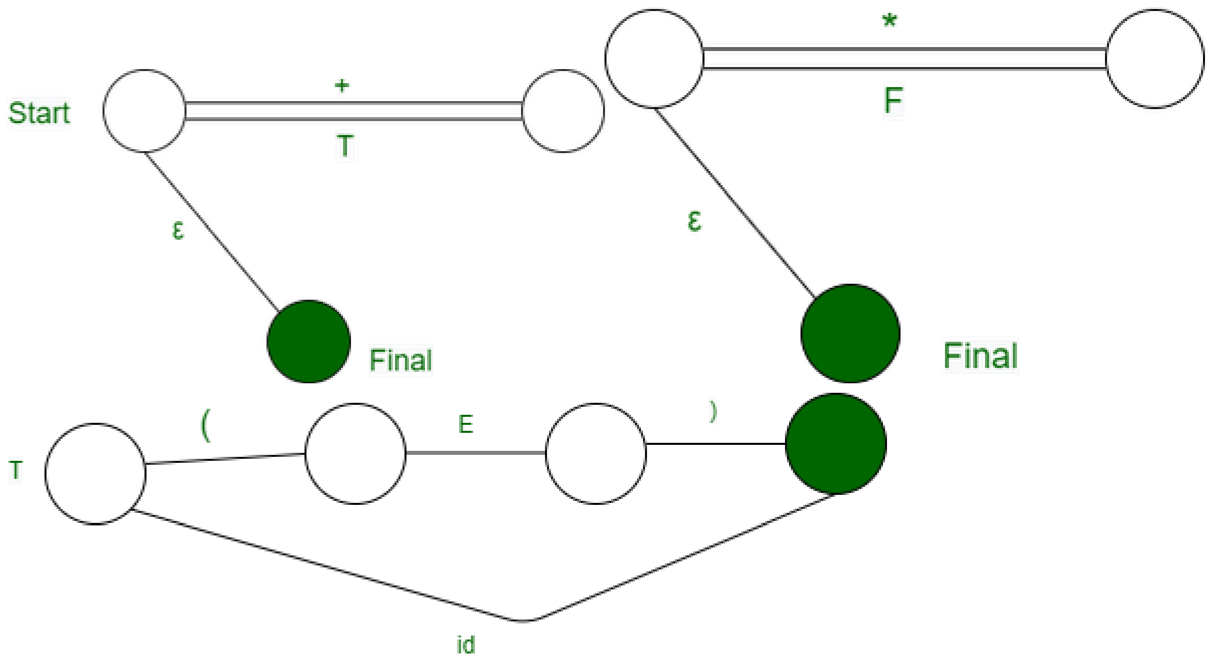
- T'->+TE'|ε



It can be optimized ahead by combining it with DFA for E'->TE'



Accordingly, we optimize the other structures to produce the following DFA



STEP 3:
Simulation on the input string.
Steps involved in the simulation procedure are:

1. Start from the starting state.
2. If a terminal arrives consume it, move to the next state.
3. If a non-terminal arrive go to the state of the DFA of the non-terminal and return on reached up to the final state.
4. Return to actual DFA and Keep doing parsing.
5. If one completes reading the input string completely, you reach a final state, and the string is successfully parsed.

First and Follow

<u>Why FIRST?</u>
If the compiler would have come to know in advance, that what is the "first character of the string produced when a production rule is applied", and comparing it to the current character or token in the input string it sees, it can wisely take decision on which production rule to apply. Let's take the same grammar from the previous article:

```
S -> cAd
A -> bc|a
And the input string is "cad".
```

Thus, in the example above, if it knew that after reading character 'c' in the input string and applying S->cAd, next character in the input string is 'a', then it would have ignored the production rule A->bc (because 'b' is the first character of the string produced by this production rule, not 'a'), and directly use the production rule A->a (
<u>because 'a' is the first character of the string produced by this production rule, and is same as the current character of the input string which is also 'a'</u>

). Hence it is validated that if the compiler/parser knows about
first character of the string that can be obtained by applying a production rule,
then it can wisely apply the correct production rule to get the correct syntax tree for the given input string.
Grasping concepts like *FIRST* and *FOLLOW* is essential for students delving into compiler design, especially for GATE CS aspirants. For a comprehensive understanding of these and other critical topics, explore the GATE CS Self-Paced Course. It provides in-depth explanations and practice resources to help you ace GATE and deepen your knowledge of compiler design.

Why FOLLOW?

The parser faces one more problem. Let us consider below grammar to understand this problem.

```
A -> aBb
B -> c | ε
And suppose the input string is "ab" to parse.
```

As the first character in the input is a, the parser applies the rule A->aBb.

```
     A
   / |  \
 a   B   b
```

Now the parser checks for the second character of the input string which is b, and the Non-Terminal to derive is B, but the parser can't get any string derivable from B that contains b as first character. But the Grammar does contain a production rule B -> ε, if that is applied then B will vanish, and the parser gets the input "ab", as shown below. But the parser can apply it only when it knows that the character that follows B in the production rule is same as the current character in the input. In RHS of A -> aBb, b follows Non-Terminal B, i.e. FOLLOW(B) = {b}, and the current input character read is also b. Hence the parser applies this rule. And it is able to get the string "ab" from the given grammar.

```
      A                A
   /  |  \           /   \
 a    B    b   =>   a     b
      |
      ε
```

So FOLLOW can make a Non-terminal vanish out if needed to generate the string from the parse tree.   The conclusions is, we need to find FIRST and FOLLOW sets for a given grammar so that the parser can properly apply the needed rule at the correct position. In the next article, we will discuss formal definitions of FIRST and FOLLOW, and some easy rules to compute these sets

LR Parser:

LR parsers is an efficient bottom-up syntax analysis technique that can be used to parse large classes of context-free grammar is called LR(k) parsing.
L stands for left-to-right scanning

R stands for rightmost derivation in reverse
k is several input symbols. when k is omitted k is assumed to be 1.

Advantages of LR parsing

- LR parsers handle context-free grammars. These grammars describe the structure of programming languages-how statements, expressions, and other language constructs fit together.
- LR parsers ensure that your code adheres to these rules.
- It is able to detect syntactic errors
- It is an efficient non-backtracking shift shift-reducing parsing method.

Types of LR Parsing Methods

- SLR parser
- LALR parser
- Canonical LR parser

SLR Parser:

- LR parser is also called as SLR parser
- it is weakest of the three methods but easier to implement
- a grammar for which SLR parser can be constructed is called SLR grammar

Steps for constructing the SLR parsing table

1. Writing augmented grammar
2. LR(0) collection of items to be found
3. Find FOLLOW of LHS of production
4. Defining 2 functions:goto[list of terminals] and action[list of non-terminals] in the parsing table

EXAMPLE – Construct LR parsing table for the given context-free **grammar**

S–>AA
A–>aA|b

Solution:

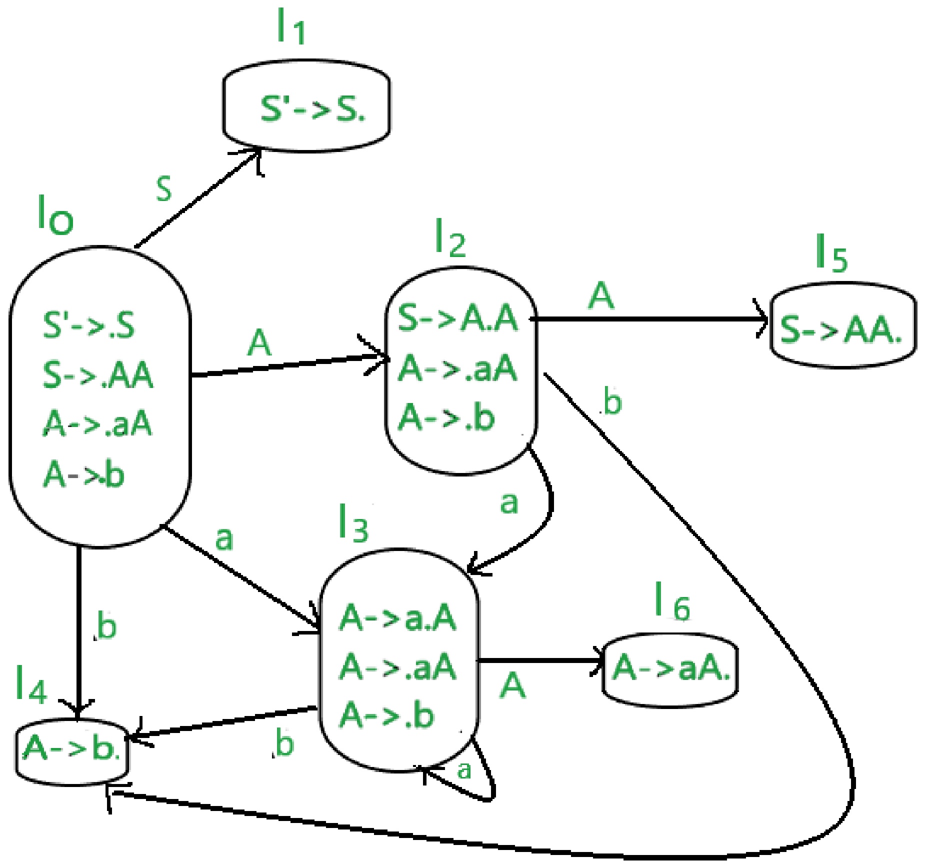
STEP1: Find augmented grammar

The augmented grammar of the given grammar is:-

S'–>.S [0th production]
S–>.AA [1st production]
A–>.aA [2nd production]
A–>.b [3rd production]

STEP2: Find LR(0) collection of items

Below is the figure showing the LR(0) collection of items. We will understand everything one by one.



The terminals of this grammar are {a,b}.
The non-terminals of this grammar are {S,A}
RULE –
If any non-terminal has ‘ . ‘ preceding it, we have to write all its production and add ‘ . ‘ preceding each of its production.
RULE –
from each state to the next state, the ‘ . ‘ shifts to one place to the right.

- In the figure, I0 consists of augmented grammar.
- Io goes to I1 when ‘ . ‘ of 0th production is shifted towards the right of S(S’->S.). this state is the accepted state. S is seen by the compiler.
- Io goes to I2 when ‘ . ‘ of 1st production is shifted towards right (S->A.A) . A is seen by the compiler
- I0 goes to I3 when ‘ . ‘ of the 2nd production is shifted towards right (A->a.A) . a is seen by the compiler.
- I0 goes to I4 when ‘ . ‘ of the 3rd production is shifted towards right (A->b.) . b is seen by the compiler.
- I2 goes to I5 when ‘ . ‘ of 1st production is shifted towards right (S->AA.) . A is seen by the compiler
- I2 goes to I4 when ‘ . ‘ of 3rd production is shifted towards right (A->b.) . b is seen by the compiler.
- I2 goes to I3 when ‘ . ‘ of the 2nd production is shifted towards right (A->a.A) . a is seen by the compiler.

- I3 goes to I4 when ‘ . ‘ of the 3rd production is shifted towards right (A->b.) . b is seen by the compiler.
- I3 goes to I6 when ‘ . ‘ of 2nd production is shifted towards the right (A->aA.) . A is seen by the compiler
- I3 goes to I3 when ‘ . ‘ of the 2nd production is shifted towards right (A->a.A) . a is seen by the compiler.

STEP3: Find FOLLOW of LHS of production
FOLLOW(S)=$
FOLLOW(A)=a,b,$
To find FOLLOW of non-terminals, please read follow set in syntax analysis.
STEP 4: Defining 2 functions:goto[list of non-terminals] and action[list of terminals] in the parsing table. Below is the SLR parsing table.

	ACTION			GOTO	
	a	b	$	A	S
0	S3	S4		2	1
1			accept		
2	S3	S4		5	
3	S3	S4		6	
4	R3	R3	R3		
5			R1		
6	R2	R2	R2		

- $ is by default a nonterminal that takes accepting state.
- 0,1,2,3,4,5,6 denotes I0,I1,I2,I3,I4,I5,I6
- I0 gives A in I2, so 2 is added to the A column and 0 rows.
- I0 gives S in I1,so 1 is added to the S column and 1 row.
- similarly 5 is written in A column and 2 row, 6 is written in A column and 3 row.
- I0 gives a in I3 .so S3(shift 3) is added to a column and 0 row.
- I0 gives b in I4 .so S4(shift 4) is added to the b column and 0 row.
- Similarly, S3(shift 3) is added on a column and 2,3 row ,S4(shift 4) is added on b column and 2,3 rows.
- I4 is reduced state as ‘ . ‘ is at the end. I4 is the 3rd production of grammar(A–>.b).LHS of this production is A. FOLLOW(A)=a,b,$. write r3(reduced 3) in the columns of a,b,$ and 4th row
- I5 is reduced state as ‘ . ‘ is at the end. I5 is the 1st production of grammar(S–>.AA). LHS of this production is S. FOLLOW(S)=$. write r1(reduced 1) in the column of $ and 5th row
- I6 is a reduced state as ‘ . ‘ is at the end. I6 is the 2nd production of grammar(A–>.aA). The LHS of this production is A. FOLLOW(A)=a,b,$. write r2(reduced 2) in the columns of a,b,$ and 6th row

APPLICATIONS GALORE:

- Compiler
- Data Validation

- Natural Language Processing(NLP)
- Protocol Parsing

CLR Parser :
The CLR parser stands for canonical LR parser.It is a more powerful LR parser.It makes use of lookahead symbols. This method uses a large set of items called LR(1) items.The main difference between LR(0) and LR(1) items is that, in LR(1) items, it is possible to carry more information in a state, which will rule out useless reduction states.This extra information is incorporated into the state by the lookahead symbol. The general syntax becomes [A->∝.B, a]
where A->∝.B is the production and a is a terminal or right end marker $
LR(1) items=LR(0) items + look ahead
How to add lookahead with the production?
CASE 1 –

```
A->∝.BC, a
```

Suppose this is the 0th production.Now, since ‘ . ‘ precedes B,so we have to write B’s productions as well.

```
B->.D [1st production]
```

Suppose this is B’s production. The look ahead of this production is given as we look at previous productions ie 0th production. Whatever is after B, we find FIRST(of that value) , that is the lookahead of 1st production.So,here in 0th production, after B, C is there. assume FIRST(C)=d, then 1st production become

```
B->.D, d
```

CASE 2 –
Now if the 0th production was like this,

```
A->∝.B, a
```

Here, we can see there’s nothing after B. So the lookahead of 0th production will be the lookahead of 1st production. ie-

```
B->.D, a
```

CASE 3 –
Assume a production A->a|b

```
A->a,$ [0th production]

A->b,$ [1st production]
```

Here, the 1st production is a part of the previous production, so the lookahead will be the same as that of its previous production.
These are the 2 rules of look ahead.

Steps for constructing CLR parsing table :
1. Writing augmented grammar

2. LR(1) collection of items to be found
3. Defining 2 functions: goto[list of terminals] and action[list of non-terminals] in the CLR parsing table

EXAMPLE

Construct a CLR parsing table for the given context-free grammar

```
S-->AA

A-->aA|b
```

Solution :

STEP 1 – Find augmented grammar

The augmented grammar of the given grammar is:-

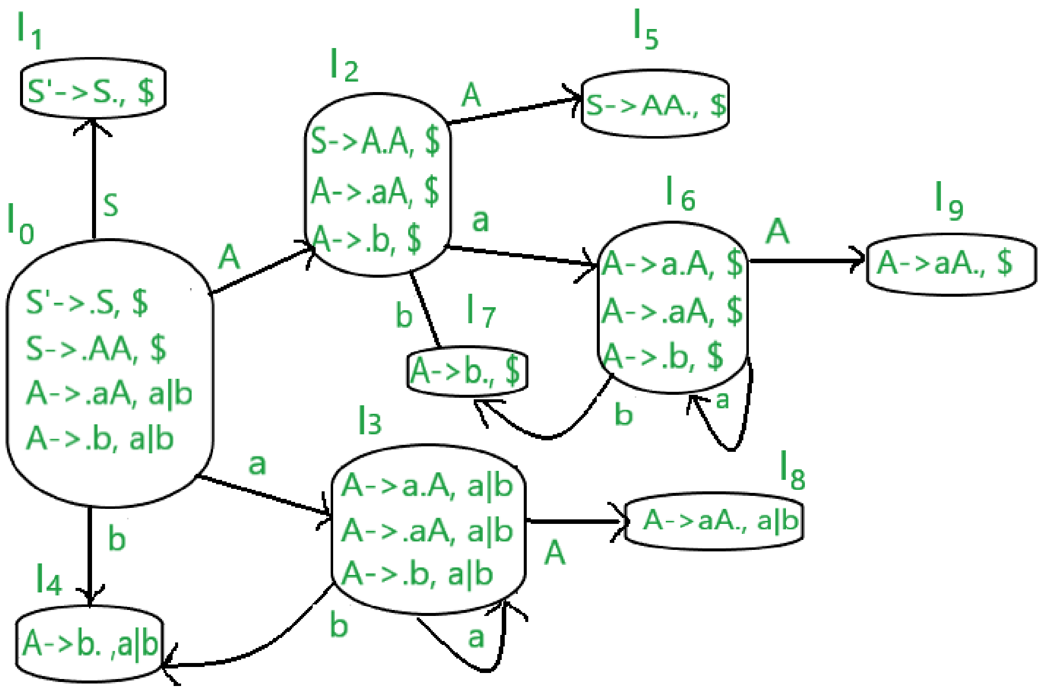
```
S'-->.S ,$   [0th production]

S-->.AA ,$ [1st production]

A-->.aA ,a|b [2nd production]

A-->.b ,a|b [3rd production]
```

Let’s apply the rule of lookahead to the above productions

- The initial look ahead is always $
- Now, the 1st production came into existence because of ‘ . ‘ Before ‘S’ in 0th production.There is nothing after ‘S’, so the lookahead of 0th production will be the lookahead of 1st production. ie: S–>.AA ,$
- Now, the 2nd production came into existence because of ‘ . ‘ Before ‘A’ in the 1st production.After ‘A’, there’s ‘A’. So, FIRST(A) is a,b Therefore,the look ahead for the 2nd production becomes a|b.
- Now, the 3rd production is a part of the 2nd production.So, the look ahead will be the same.

STEP 2 – Find LR(1) collection of items

Below is the figure showing the LR(1) collection of items. We will understand everything one by one.



The terminals of this grammar are {a,b}
The non-terminals of this grammar are {S,A}

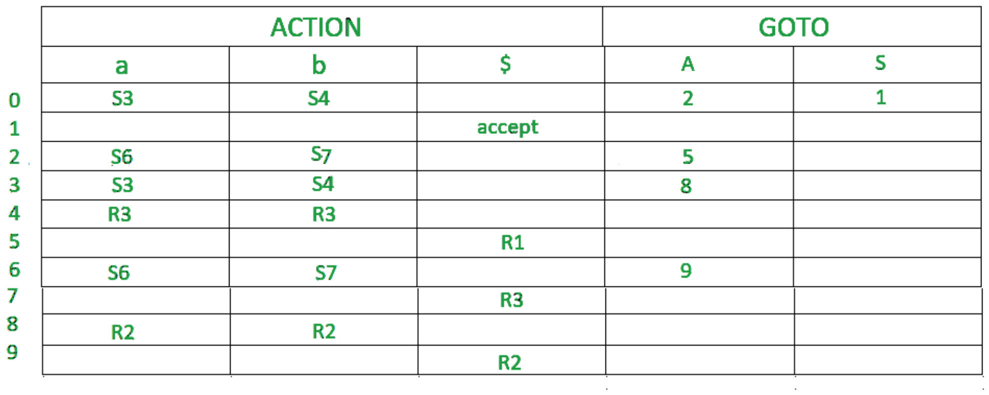
RULE-

1. If any non-terminal has ' . ' preceding it, we have to write all its production and add ' . ' preceding each of its production.
2. from each state to the next state, the ' . ' shifts to one place to the right.
3. All the rules of lookahead apply here.

- In the figure, I0 consists of augmented grammar.
- Io goes to I1 when ' . ' of 0th production is shifted towards the right of S(S'->S.). This state is the accept state . S is seen by the compiler. Since I1 is a part of the 0th production, the lookahead is the same ie $
- Io goes to I2 when ' . ' of 1st production is shifted towards right (S->A.A) . A is seen by the compiler. Since I2 is a part of the 1st production, the lookahead is the same i.e. $.
- I0 goes to I3 when ' . ' of the 2nd production is shifted towards right (A->a.A) . a is seen by the compiler. Since I3 is a part of the 2nd production, the lookahead is the same ie a|b.
- I0 goes to I4 when ' . ' of the 3rd production is shifted towards right (A->b.) . b is seen by the compiler. Since I4 is a part of the 3rd production, the lookahead is the same i.e. a | b.
- I2 goes to I5 when ' . ' of 1st production is shifted towards right (S->AA.) . A is seen by the compiler. Since I5 is a part of the 1st production, the lookahead is the same i.e. $.

- I2 goes to I6 when ‘ . ‘ of 2nd production is shifted towards the right (A->a.A) . A is seen by the compiler. Since I6 is a part of the 2nd production, the lookahead is the same i.e. $.
- I2 goes to I7 when ‘ . ‘ of 3rd production is shifted towards right (A->b.) . A is seen by the compiler. Since I6 is a part of the 3rd production, the lookahead is the same i.e. $.
- I3 goes to I3 when ‘ . ‘ of the 2nd production is shifted towards right (A->a.A) . a is seen by the compiler. Since I3 is a part of the 2nd production, the lookahead is the same i.e. a|b.
- I3 goes to I8 when ‘ . ‘ of 2nd production is shifted towards the right (A->aA.) . A is seen by the compiler. Since I8 is a part of the 2nd production, the lookahead is the same i.e. a|b.
- I6 goes to I9 when ‘ . ‘ of 2nd production is shifted towards the right (A->aA.) . A is seen by the compiler. Since I9 is a part of the 2nd production, the lookahead is the same i.e. $.
- I6 goes to I6 when ‘ . ‘ of the 2nd production is shifted towards right (A->a.A) . a is seen by the compiler. Since I6 is a part of the 2nd production, the lookahead is the same i.e. $.
- I6 goes to I7 when ‘ . ‘ of the 3rd production is shifted towards right (A->b.) . b is seen by the compiler. Since I6 is a part of the 3rd production, the lookahead is the same ie $.

STEP 3- defining 2 functions:goto[list of terminals] and action[list of non-terminals] in the parsing table.Below is the CLR parsing table



	ACTION			GOTO	
	a	b	$	A	S
0	S3	S4		2	1
1			accept		
2	S6	S7		5	
3	S3	S4		8	
4	R3	R3			
5			R1		
6	S6	S7		9	
7			R3		
8	R2	R2			
9			R2		

- $ is by default a non terminal which takes accepting state.
- 0,1,2,3,4,5,6,7,8,9 denotes I0,I1,I2,I3,I4,I5,I6,I7,I8,I9
- I0 gives A in I2, so 2 is added to the A column and 0 row.
- I0 gives S in I1,so 1 is added to the S column and 1st row.
- similarly 5 is written in A column and 2nd row, 8 is written in A column and 3rd row, 9 is written in A column and 6th row.
- I0 gives a in I3, so S3(shift 3) is added to a column and 0 row.
- I0 gives b in I4, so S4(shift 4) is added to the b column and 0 row.
- Similarly, S6(shift 6) is added on ‘a’ column and 2,6 row ,S7(shift 7) is added on b column and 2,6 row,S3(shift 3) is added on ‘a’ column and 3 row ,S4(shift 4) is added on b column and 3 row.

- I4 is reduced as ‘ . ‘ is at the end. I4 is the 3rd production of grammar. So write r3(reduce 3) in lookahead columns. The lookahead of I4 are a and b, so write R3 in a and b column.
- I5 is reduced as ‘ . ‘ is at the end. I5 is the 1st production of grammar. So write r1(reduce 1) in lookahead columns. The lookahead of I5 is $ so write R1 in $ column.
- Similarly, write R2 in a,b column and 8th row, write R2 in $ column and 9th row.

LALR Parser :
LALR Parser is lookahead LR parser. It is the most powerful parser which can handle large classes of grammar. The size of CLR parsing table is quite large as compared to other parsing table. LALR reduces the size of this table.LALR works similar to CLR. The only difference is , it combines the similar states of CLR parsing table into one single state.
The general syntax becomes [A->∝.B, a]
where A->∝.B is production and a is a terminal or right end marker $
LR(1) items=LR(0) items + look ahead
How to add lookahead with the production?
CASE 1 –

```
A->∝.BC, a
```

Suppose this is the 0th production.Now, since ‘ . ‘ precedes B,so we have to write B’s productions as well.

```
B->.D [1st production]
```

Suppose this is B’s production. The look ahead of this production is given as- we look at previous production i.e. – 0th production. Whatever is after B, we find FIRST(of that value) , that is the lookahead of 1st production. So, here in 0th production, after B, C is there. Assume FIRST(C)=d, then 1st production become.

```
B->.D, d
```

CASE 2 –
Now if the 0th production was like this,

```
A->∝.B, a
```

Here,we can see there’s nothing after B. So the lookahead of 0th production will be the lookahead of 1st production. ie-

```
B->.D, a
```

CASE 3 –
Assume a production A->a|b

```
A->a,$ [0th production]

A->b,$ [1st production]
```

Here, the 1st production is a part of the previous production, so the lookahead will be the same as that of its previous production.

Steps for constructing the LALR parsing table :

1. Writing augmented grammar
2. LR(1) collection of items to be found
3. Defining 2 functions: goto[list of terminals] and action[list of non-terminals] in the LALR parsing table

EXAMPLE

Construct CLR parsing table for the given context free grammar

S-->AA

A-->aA|b

Solution:

STEP1- Find augmented grammar

The augmented grammar of the given grammar is:-

S'-->.S ,$ [0th production]

S-->.AA ,$ [1st production]

A-->.aA ,a|b [2nd production]

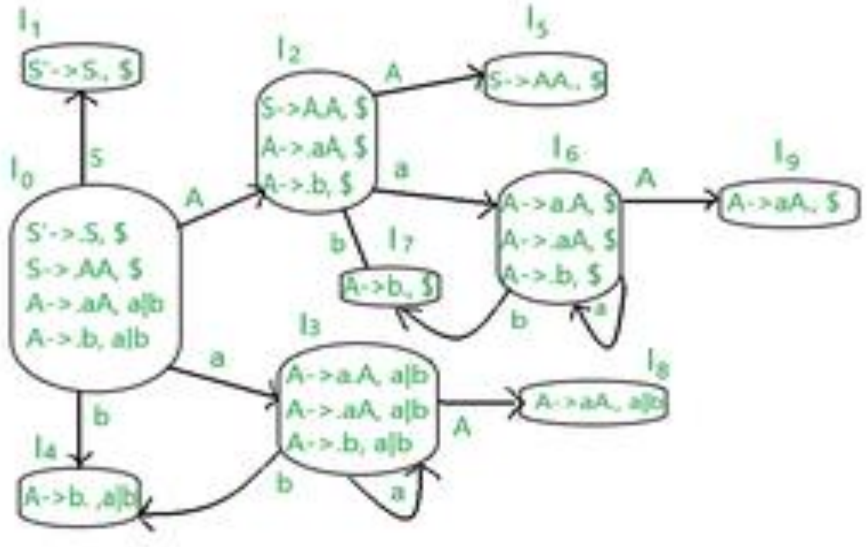
A-->.b ,a|b [3rd production]

Let’s apply the rule of lookahead to the above productions.

- The initial look ahead is always $
- Now,the 1st production came into existence because of ‘ . ‘ before ‘S’ in 0th production.There is nothing after ‘S’, so the lookahead of 0th production will be the lookahead of 1st production. i.e. : S–>.AA ,$
- Now,the 2nd production came into existence because of ‘ . ‘ before ‘A’ in the 1st production.
  After ‘A’, there’s ‘A’. So, FIRST(A) is a,b. Therefore, the lookahead of the 2nd production becomes a|b.
- Now,the 3rd production is a part of the 2nd production.So, the look ahead will be the same.

STEP2 – Find LR(0) collection of items

Below is the figure showing the LR(0) collection of items. We will understand everything one by one.



The terminals of this grammar are {a,b}
The non-terminals of this grammar are {S,A}

RULES –

1. If any non-terminal has ‘ . ‘ preceding it, we have to write all its production and add ‘ . ‘ preceding each of its production.
2. from each state to the next state, the ‘ . ‘ shifts to one place to the right.

YACC:
YACC is an LALR parser generator developed at the beginning of the 1970s by Stephen C. Johnson for the Unix operating system. It automatically generates the LALR(1) parsers from formal grammar specifications. YACC plays an important role in compiler and interpreter development since it provides a means to specify the grammar of a language and to produce parsers that either interpret or compile code written in that language.
Key Concepts and Features of YACC

- **Grammar Specification:** The input to YACC is a context-free grammar (usually in the Backus-Naur Form, BNF) that describes the syntax rules of the language it parses.
- **Parser Generation:** YACC translates the grammar into a C function that could perform an efficient parsing of input text according to such predefined rules.
- **LALR(1) Parsing:** This is a bottom-up parsing method that makes use of a single token lookahead in determining the next action of parsing.
- **Semantic Actions:** These are the grammar productions that are associated with an action; this enables the execution of code, usually in C, used in the construction of abstract syntax trees, the generation of intermediate representations, or error handling.
- **Attribute Grammars:** These grammars consist of non-terminal grammar symbols with attributes, which through semantic actions are used in the construction of parse trees or the output of code.

- **Integration with Lex:** It is often used along with Lex, a tool that generates lexical analyzers-scanners-which breaks input into tokens that are then processed by the YACC parser.

Semantic Actions and Attribute Grammars in YACC

The semantic actions associated with productions achieve the building of an intermediate representation or target code as follows:

- Every nonterminal symbol in the parser has an attribute.
- The semantic action associated with a production can access attributes of nonterminal symbols used in that production–a symbol "$n' in the semantic action, where n is an integer, designates the attribute of the nonterminal symbol in the RHS of the production and the symbol '$$' designates the attribute of the LHS nonterminal symbol of the production.
- The semantic action uses the values of these attributes for building the intermediate representation or target code.

A parser generator is a program that takes as input a specification of a syntax and produces as output a procedure for recognizing that language. Historically, they are also called compiler compilers. YACC (yet another compiler-compiler) is an LALR(1) (LookAhead, Left-to-right, Rightmost derivation producer with 1 lookahead token) parser generator. YACC was originally designed for being complemented by Lex.

Input File: YACC input file is divided into three parts.

```
/* definitions */
 ....

%%
/* rules */
....
%%

/* auxiliary routines */
....
```

Input File: Definition Part:

- The definition part includes information about the tokens used in the syntax definition:

```
%token NUMBER
%token ID
```

- Yacc automatically assigns numbers for tokens, but it can be overridden by

```
%token NUMBER 621
```

- Yacc also recognizes single characters as tokens. Therefore, assigned token numbers should no overlap ASCII codes.
- The definition part can include C code external to the definition of the parser and variable declarations, within **%{** and **%}** in the first column.
- It can also include the specification of the starting symbol in the grammar:

```
%start nonterminal
```

Input File: Rule Part:

- The rules part contains grammar definitions in a modified BNF form.

- Actions is C code in {} and can be embedded inside (Translation schemes).

Input File: Auxiliary Routines Part:

- The auxiliary routines part is only C code.
- It includes function definitions for every function needed in the rules part.
- It can also contain the main() function definition if the parser is going to be run as a program.
- The main() function must call the function yyparse().

Input File:

- If yylex() is not defined in the auxiliary routines sections, then it should be included:

```
#include "lex.yy.c"
```

- YACC input file generally finishes with:

```
.y
```

Output Files:

- The output of YACC is a file named **y.tab.c**
- If it contains the **main()** definition, it must be compiled to be executable.
- Otherwise, the code can be an external function definition for the function **int yyparse()**
- If called with the **–d** option in the command line, Yacc produces as output a header file **y.tab.h** with all its specific definition (particularly important are token definitions to be included, for example, in a Lex input file).
- If called with the **–v** option, Yacc produces as output a file **y.output** containing a textual description of the LALR(1) parsing table used by the parser. This is useful for tracking down how the parser solves conflicts.