Ahmedabad Institute of Technology

CE & IT Department

Compiler Design(2120701)

Laboratory Manual

Year: 2020

Prepared By: - Prof. Neha Prajapati

CE Department

Vision:

To produce technically sound and ethically responsible Computer Engineers to the society by providing Quality Education.

CE Department Mission:

- To provide healthy Learning Environment based on current and future Industrial demands.
- To promote curricular, co-curricular and extra-curricular activities for overall personality development of the students.
- To groom technically powerful and ethically dominant engineers having real life problem solving capabilities.
- To provide platform for Effective Teaching Learning.

IT Department

Vision:

To provide quality education and assistance to the students through innovative teaching learning methodology for shaping young mind technically sound and ethically strong.

IT Department Mission:

- To serve society by producing technically and ethically sound engineers.
- To generate groomed and efficient problem solvers as per Industrial needs by adopting innovative teaching learning methods.
- To emphasis on overall development of the students through various curricular, co-curricular and extra-curricular activities.

INDEX

Sr. No.	Experiment	Page No.		Date	Marks	Signature
		From	To			
1.	**WRITE A PROGRAM TO GENERATE TOKENS FOR GIVEN LEXEME**					
2.	**WRITE A C PROGRAM TO FIND WHETHER THE STRING IS PARSING OR NOT.**					
3.	**WRITE A PROGRAM TO IMPLEMENT SIMPLE LEXICAL ANALYZER USING C LANGUAGE.**					
4.	**WRITE A PROGRAM TO GENERATE SYNTAX TREE.**					
5.	**WRITE A PROGRAM TO CONSTRUCT NFA FOR THE GIVEN REGULAR EXPRESSION**					
6.	**WRITE A PROGRAM TO CONSTRUCT DFA FOR THE GIVEN REGULAR EXPRESSION**					
7.	**WRITE A PROGRAM TO IMPLEMENT SYMBOL TABLE USING C LANGUAGE.**					
8.	**WRITE A PROGRAM TO FIND FIRST & FOLLOW FROM A GRAMMAR.**					
9.	**WRITE A PROGRAM TO IMPLEMENT CONSTRUCTION OF OPERATOR PRECEDENCE PARSE TABLE**					
10.	**WRITE A C PROGRAM TO IMPLEMENT SIMPLE LR PARSING ALGORITHM .**					

EXPERIMENT NO: 1 DATE: / /

TITLE: WRITE A PROGRAM TO GENERATE TOKENS FOR GIVEN LEXEME.

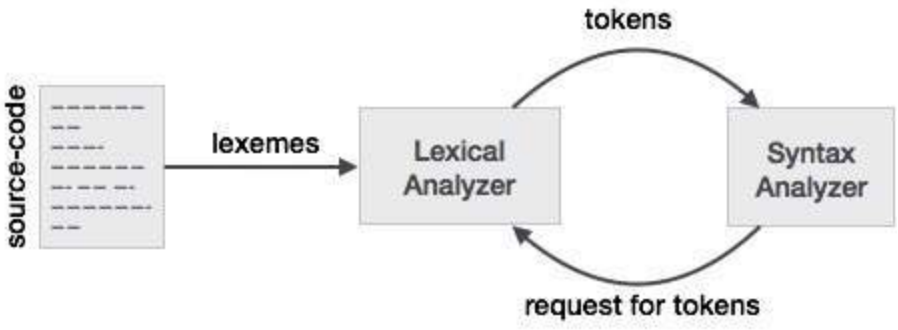
OBJECTIVES: On completion of this experiment student will able to…

- Know the concept of Token and Lexeme

THEORY:

Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors 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.



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

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'  ';'  '}'
```

Specifications of Tokens

Let us understand how the language theory undertakes the following terms:

Alphabets

Any finite set of symbols {0,1} is a set of binary alphabets, {0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F} is a set of Hexadecimal alphabets, {a-z, A-Z} is a set of English language alphabets.

Strings

Any finite sequence of alphabets is called a string. Length of the string is the total number of occurrence of alphabets, e.g., the length of the string tutorialspoint is 14 and is denoted by | tutorialspoint| = 14. A string having no alphabets, i.e. a string of zero length is known as an empty string and is denoted by ε (epsilon).

Special Symbols

A typical high-level language contains the following symbols:-

Arithmetic Symbols	Addition(+), Subtraction(-), Modulo(%), Multiplication(*), Division(/)
Punctuation	Comma(,), Semicolon(;), Dot(.), Arrow(->)
Assignment	=
Special Assignment	+=, /=, *=, -=
Comparison	==, !=, <, <=, >, >=
Preprocessor	#
Location Specifier	&
Logical	&, &&, |, ||, !
Shift Operator	>>, >>>, <<, <<<

Language

A language is considered as a finite set of strings over some finite set of alphabets. Computer languages are considered as finite sets, and mathematically set operations can be performed on them. Finite languages can be described by means of regular expressions.

EVALUATION:

Involvement (4)	Understanding / Problem solving (3)	Timely Completion (3)	**Total** **(10)**

Signature with date: _____

EXPERIMENT NO: 2 **DATE: / /**

TITLE: WRITE A C PROGRAM TO FIND WHETHER THE STRING IS PARSING OR NOT

OBJECTIVES: On completion of this experiment student will able to…

i. How to parse a string.

THEORY:

We have learnt how a parser constructs parse trees in the syntax analysis phase. The plain parse-tree constructed in that phase is generally of no use for a compiler, as it does not carry any information of how to evaluate the tree. The productions of context-free grammar, which makes the rules of the language, do not accommodate how to interpret them.

For example

E → E + T

The above CFG production has no semantic rule associated with it, and it cannot help in making any sense of the production.

Semantics

Semantics of a language provide meaning to its constructs, like tokens and syntax structure. Semantics help interpret symbols, their types, and their relations with each other. Semantic analysis judges whether the syntax structure constructed in the source program derives any meaning or not.

CFG + semantic rules = Syntax Directed Definitions

For example:

int a = “value”;

should not issue an error in lexical and syntax analysis phase, as it is lexically and structurally correct, but it should generate a semantic error as the type of the assignment differs. These rules are set by the grammar of the language and evaluated in semantic analysis. The following tasks should be performed in semantic analysis:

- Scope resolution
- Type checking
- Array-bound checking

Semantic Errors

We have mentioned some of the semantics errors that the semantic analyzer is expected to recognize:

- Type mismatch
- Undeclared variable
- Reserved identifier misuse.
- Multiple declaration of variable in a scope.
- Accessing an out of scope variable.
- Actual and formal parameter mismatch.

Attribute Grammar

Attribute grammar is a special form of context-free grammar where some additional information (attributes) are appended to one or more of its non-terminals in order to provide context-sensitive information. Each attribute has well-defined domain of values, such as integer, float, character, string, and expressions.

Attribute grammar is a medium to provide semantics to the context-free grammar and it can help specify the syntax and semantics of a programming language. Attribute grammar (when viewed as a parse-tree) can pass values or information among the nodes of a tree.

Example:

E → E + T { E.value = E.value + T.value }

The right part of the CFG contains the semantic rules that specify how the grammar should be interpreted. Here, the values of non-terminals E and T are added together and the result is copied to the non-terminal E.

Semantic attributes may be assigned to their values from their domain at the time of parsing and evaluated at the time of assignment or conditions. Based on the way the attributes get their values, they can be broadly divided into two categories : synthesized attributes and inherited attributes.

Synthesized attributes

These attributes get values from the attribute values of their child nodes. To illustrate, assume the following production:

S → ABC

If S is taking values from its child nodes (A,B,C), then it is said to be a synthesized attribute, as the values of ABC are synthesized to S.

As in our previous example (E → E + T), the parent node E gets its value from its child node. Synthesized attributes never take values from their parent nodes or any sibling nodes.

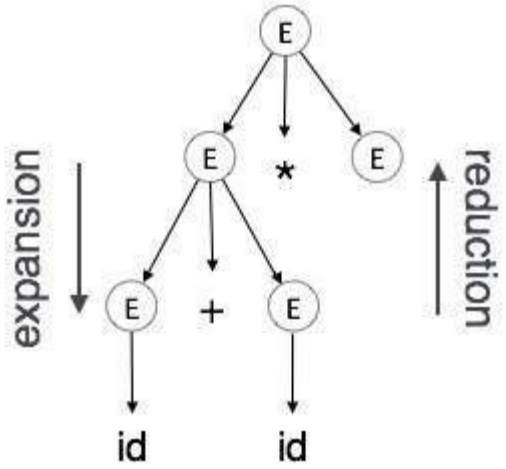
Inherited attributes

In contrast to synthesized attributes, inherited attributes can take values from parent and/or siblings. As in the following production,

S → ABC

A can get values from S, B and C. B can take values from S, A, and C. Likewise, C can take values from S, A, and B.

Expansion : When a non-terminal is expanded to terminals as per a grammatical rule



Reduction : When a terminal is reduced to its corresponding non-terminal according to grammar rules. Syntax trees are parsed top-down and left to right. Whenever reduction occurs, we apply its corresponding semantic rules (actions).

Semantic analysis uses Syntax Directed Translations to perform the above tasks.

Semantic analyzer receives AST (Abstract Syntax Tree) from its previous stage (syntax analysis).

Semantic analyzer attaches attribute information with AST, which are called Attributed AST.

Attributes are two tuple value, <attribute name, attribute value>

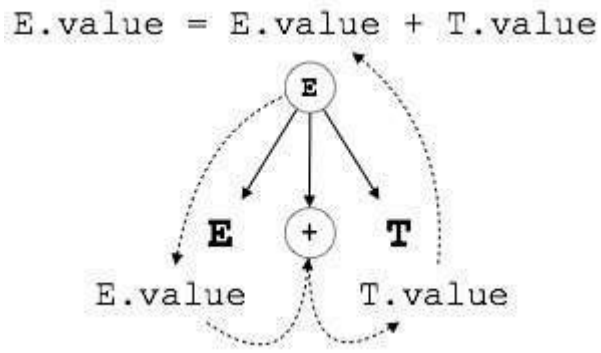
For example:

int value = 5;
<type, “integer”>
<presentvalue, “5”>

For every production, we attach a semantic rule.

S-attributed SDT

If an SDT uses only synthesized attributes, it is called as S-attributed SDT. These attributes are evaluated using S-attributed SDTs that have their semantic actions written after the production (right hand side).



As depicted above, attributes in S-attributed SDTs are evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes.

L-attributed SDT

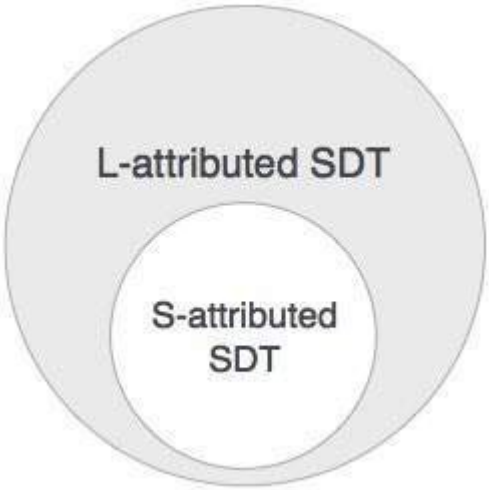
This form of SDT uses both synthesized and inherited attributes with restriction of not taking values from right siblings.

In L-attributed SDTs, a non-terminal can get values from its parent, child, and sibling nodes. As in the following production

S → ABC

S can take values from A, B, and C (synthesized). A can take values from S only. B can take values from S and A. C can get values from S, A, and B. No non-terminal can get values from the sibling to its right.

Attributes in L-attributed SDTs are evaluated by depth-first and left-to-right parsing manner.



We may conclude that if a definition is S-attributed, then it is also L-attributed as L-attributed definition encloses S-attributed definitions.

ALGORITHM:

1. Start the Program
2. Read the input from right to left
3.Then collect the information about the tokens as per the input.
4. Then perform type checking and other type of semantic analysis.
5.From that generate the intermediate representation then it will be parsed.
6. If it's not generated means then the string will not be not parsed.

EVALUATION:

Involvement (4)	Understanding / Problem solving (3)	Timely Completion (3)	**Total (10)**

Signature with date: _____

EXPERIMENT NO: 3 **DATE: / /**

TITLE: WRITE A PROGRAM TO IMPLEMENT SIMPLE LEXICAL ANALYZER USING C LANGUAGE.

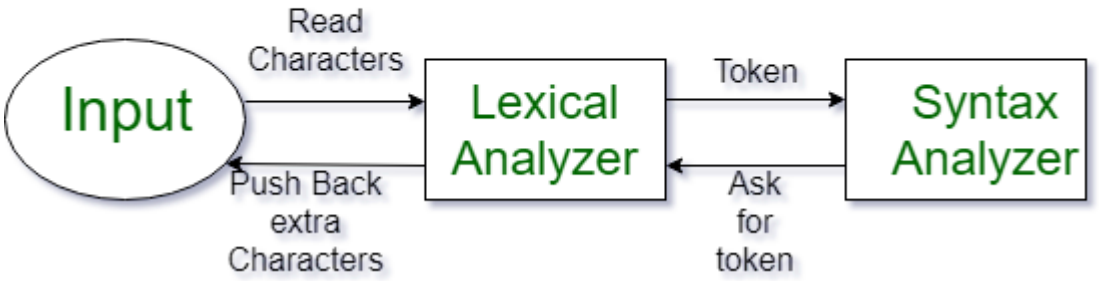
OBJECTIVES: On completion of this experiment student will able to…

I. role of lexical analyzer.

THEORY:

Lexical Analysis is the first phase of compiler also known as scanner. It converts the High level input program into a sequence of Tokens.

- Lexical Analysis can be implemented with the Deterministic finite Automata.
- The output is a sequence of tokens that is sent to the parser for syntax analysis

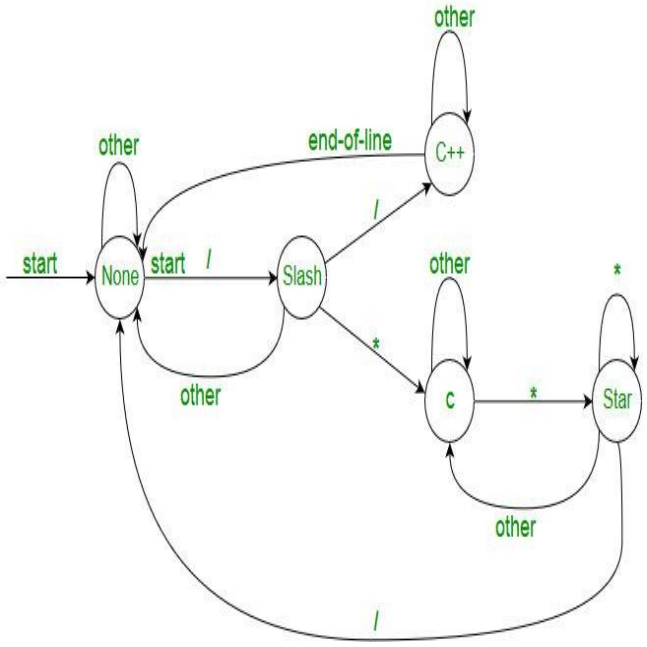


Lexeme:

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”, “;” .

HowLexical Analyzer functions

1. Tokenization .i.e Dividing the program into valid tokens.
2. Remove white space characters.
3. Remove comments.
4. It also provides help in generating error message by providing row number and column number.



- The lexical analyzer identifies the error with the help of 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 reference 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'  ';'  '}'
```

Above are the valid tokens.
You can observe that we have omitted comments.

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.

Exercise 2:

Count number of tokens :

int max(int i);

- Lexical analyzer first read **int** and finds it to be valid and accepts as token
- **max** is read by it and found to be valid function name after reading **(**
- **int** is also a token , then again **i** as another token and finally **;**

Answer: Total number of tokens 7:
int, max, (,int, i,), ;

EVALUATION:

Involvement (4)	Understanding / Problem solving (3)	Timely Completion (3)	**Total** **(10)**

Signature with date: _____

EXPERIMENT NO: 4 **DATE: / /**

TITLE: WRITE A PROGRAM TO GENERATE SYNTAX TREE

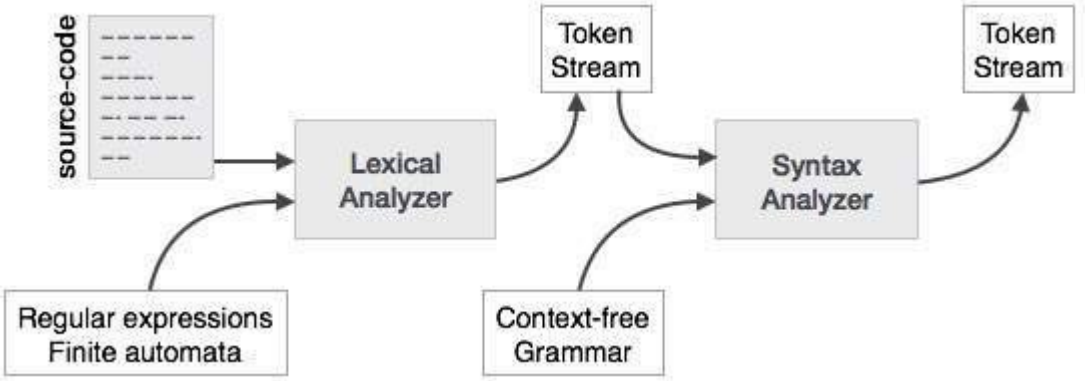
OBJECTIVES: On completion of this experiment student will able to…

- know the concept of syntax tree.

THEORY:

Syntax Analyzers

A syntax analyzer or parser takes the input from a lexical analyzer in the form of token streams. The parser analyzes the source code (token stream) against the production rules to detect any errors in the code. The output of this phase is a **parse tree**.



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

Parsers are expected to parse the whole code even if some errors exist in the program. Parsers use error recovering strategies, which we will learn later in this chapter.

Derivation

A derivation is basically a sequence of production rules, in order to get the input string. During parsing, we take two decisions for some sentential form of input:

- Deciding the non-terminal which is to be replaced.
- Deciding the production rule, by which, the non-terminal will be replaced.

To decide which non-terminal to be replaced with production rule, we can have two options.

Left-most Derivation

If the sentential form of an input is scanned and replaced from left to right, it is called left-most derivation. The sentential form derived by the left-most derivation is called the left-sentential form.

Right-most Derivation

If we scan and replace the input with production rules, from right to left, it is known as right-most derivation. The sentential form derived from the right-most derivation is called the right-sentential form.

Example

Production rules:

E → E + E
E → E * E
E → id

Input string: id + id * id

The left-most derivation is:

E → E * E
E → E + E * E
E → id + E * E
E → id + id * E
E → id + id * id

Notice that the left-most side non-terminal is always processed first.

The right-most derivation is:

E → E + E
E → E + E * E
E → E + E * id
E → E + id * id
E → id + id * id

Parse Tree

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

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

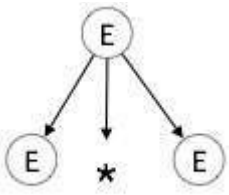
The left-most derivation is:

E → E * E
E → E + E * E
E → id + E * E
E → id + id * E

E → id + id * id

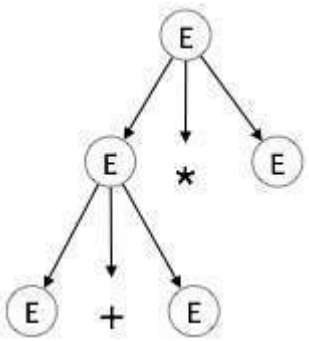
Step 1:

E → E * E



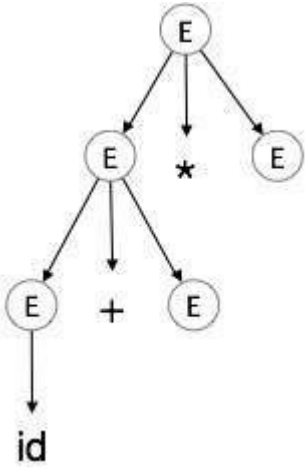
Step 2:

E → E + E * E



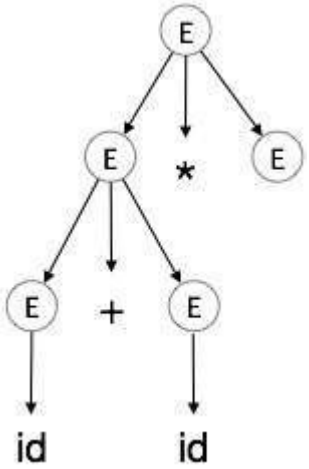
Step 3:

E → id + E * E



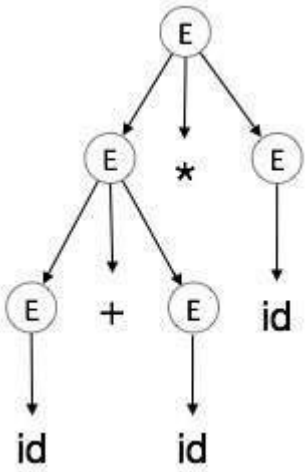
Step 4:

E → id + id * E



Step 5:

E → id + id * id



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.

Ambiguity

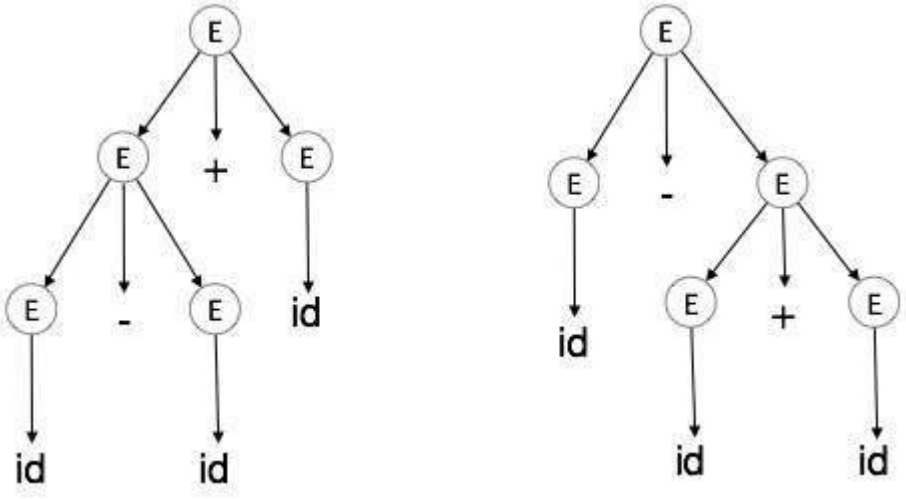
A grammar G is said to be ambiguous if it has more than one parse tree (left or right derivation) for at least one string.

Example

E → E + E
E → E – E

E → id

For the string id + id – id, the above grammar generates two parse trees:



The language generated by an ambiguous grammar is said to be **inherently ambiguous**. Ambiguity in grammar is not good for a compiler construction. No method can detect and remove ambiguity automatically, but it can be removed by either re-writing the whole grammar without ambiguity, or by setting and following associativity and precedence constraints.

Associativity

If an operand has operators on both sides, the side on which the operator takes this operand is decided by the associativity of those operators. If the operation is left-associative, then the operand will be taken by the left operator or if the operation is right-associative, the right operator will take the operand.

Example

Operations such as Addition, Multiplication, Subtraction, and Division are left associative. If the expression contains:

id op id op id

it will be evaluated as:

(id op id) op id

For example, (id + id) + id

Operations like Exponentiation are right associative, i.e., the order of evaluation in the same expression will be:

id op (id op id)

For example, id ^ (id ^ id)

Precedence

If two different operators share a common operand, the precedence of operators decides which will take the operand. That is, 2+3*4 can have two different parse trees, one corresponding to (2+3)*4 and another corresponding to 2+(3*4). By setting precedence among operators, this problem can be easily removed. As in the previous example, mathematically * (multiplication) has precedence over + (addition), so the expression 2+3*4 will always be interpreted as:

2 + (3 * 4)

EVALUATION:

Involvement (4)	Understanding / Problem solving (3)	Timely Completion (3)	**Total (10)**

Signature with date: _____

EXPERIMENT NO: 5 **DATE: / /**

TITLE: WRITE A PROGRAM TO CONSTRUCT NFA FOR THE GIVEN REGULAR EXPRESSION

OBJECTIVES: On completion of this experiment student will able to…

- know the concept of NFA.

THEORY:

From Regular Expression to NFA to DFA

It is proven (Kleene's Theorem) that RE and FA are equivalent language definition methods. Based on this theoretical result practical algorithms have been developed enabling us actually to construct FA's from RE's and simulate the FA with a computer program using Transition Tables. In following this progression an NFA is constructed first from a regular expression, then the NFA is reconstructed to a DFA, and finally a Transition Table is built. The Thompson's Construction Algorithm is one of the algorithms that can be used to build a Nondeterministic Finite Automaton (NFA) from RE, and Subset construction Algorithm can be applied to convert the NFA into a Deterministic Finite Automaton (DFA). The last step is to generate a transition table.

We need a finite state machine that is a deterministic finite automaton (DFA) so that each state has one unique edge for an input alphabet element. So that for code generation there is no ambiguity. But a nondeterministic finite automaton (NFA) with more than one edge for an input alphabet element is easier to construct using a general algorithm - Thompson's construction. Then following a standard procedure, we convert the NFA to a DFA for coding.

1. Regular expression

Consider the regular expression r = (a|b)*abb, that matches

{abb, aabb, babb, aaabb, bbabb, ababb, aababb,.....}

To construct a NFA from this, use Thompson's construction.

This method constructs a regular expression from its components using ε-transitions. The ε transitions act as "glue or mortar" for the subcomponent NFA's. An ε-transition adds nothing since concatenation with the empty string leaves a regular expression unchanged (concatenation with ε is the identity operation).

Step 1.

Parse the regular expression into its sub expressions involving alphabet symbols a and b and ε:

ε, a, b, a|b, ()*, ab, abb

These describe

a. a regular expression for single characters ε, a, b

b. alternation between a and b representing the union of the sets: L(a) U L(b)

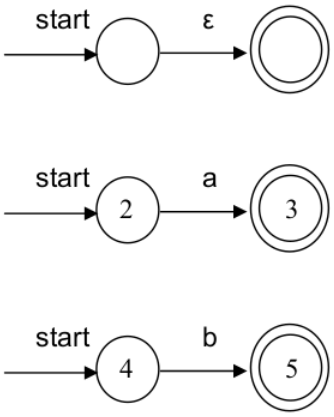
c. Kleene star ()*

d. concatenation of a and b: ab, and also abb

Subexpressions of these kinds have their own Nondeterministic Finite Automata from which the overall NFA is constructed. Each component NFA has its own start and end accepting states. A Nondeterministic Finite Automata (NFA) has a transition diagram with possibly more than one edge for a symbol (character of the alphabet) that has a start state and an accepting state. The NFA definitely provides an accepting state for the symbol.

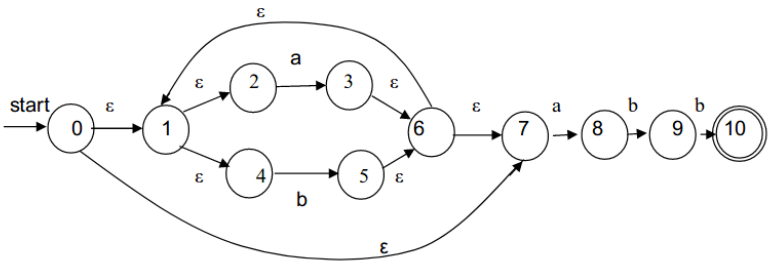
Take these NFA's in turn:

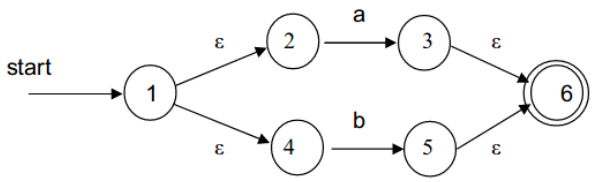
a. the NFA's for single character regular expressions ε, a, b



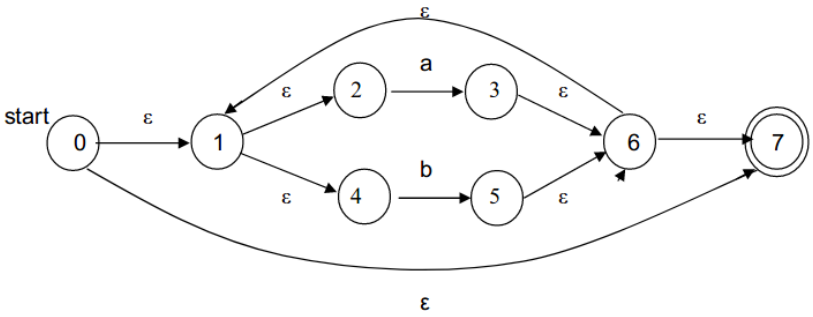
the NFA for the union of a and b: a|b is constructed from the individual NFA's using the εNFA as "glue". Remove the individual accepting states and replace with the overall accepting state

d. concatenate with abb





c. Kleene star on (a|b)*. The NFA accepts ε in addition to (a|b)*



This is the complete NFA. It describes the regular expression (a|b)*abb. The problem is that it is not suitable as the basis of a DFA transition table since there are multiple εedges leaving many states (0, 1, 6).

ALGORITHM:

1. Get the regular expression.
2. Mark the initial state with a line and final state with double circle.
3. Mark the intermediate states with a single circle and a line connecting previous state and current state with an arrow towards the current state.
4. Display the input above the arrow for every transition.
5. Initial state must have no incoming states and final state should not have any outgoing states.
6. The input is, A REGULAR EXPRESSION ‘r’ OVER AN ALPHABET ‘E’.
7. The output is, AN NFA ‘N’ ACCEPTING L(r).
8 Parse ‘r’ into the sub expressions.
9. Using the rules construct NFA for each of the basic symbols in ‘r’.

EVALUATION:

Involvement (4)	Understanding / Problem solving (3)	Timely Completion (3)	**Total** **(10)**

Signature with date: _____

EXPERIMENT NO: 6 **DATE: / /**

TITLE: WRITE A PROGRAM TO CONSTRUCT DFA FOR THE GIVEN REGULAR EXPRESSION.

OBJECTIVES: On completion of this experiment student will able to…

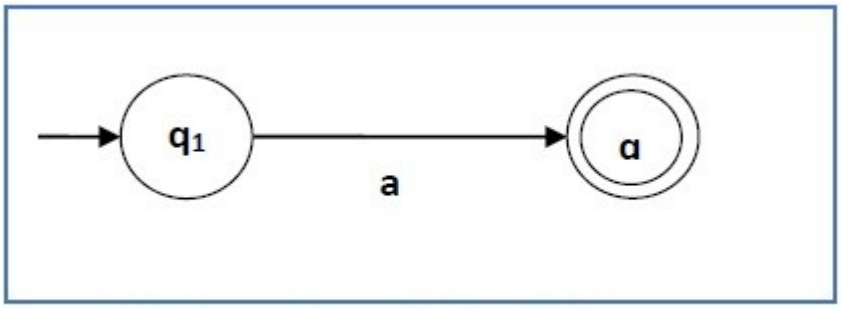
- Know the concept of DFA.

THEORY:

We can use Thompson's Construction to find out a Finite Automaton from a Regular Expression. We will reduce the regular expression into smallest regular expressions and converting these to NFA and finally to DFA.

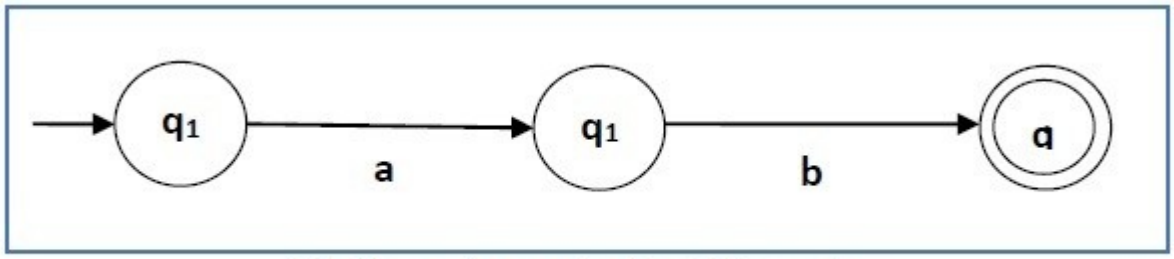
Some basic RA expressions are the following −

Case 1 − For a regular expression ‘a’, we can construct the following FA −



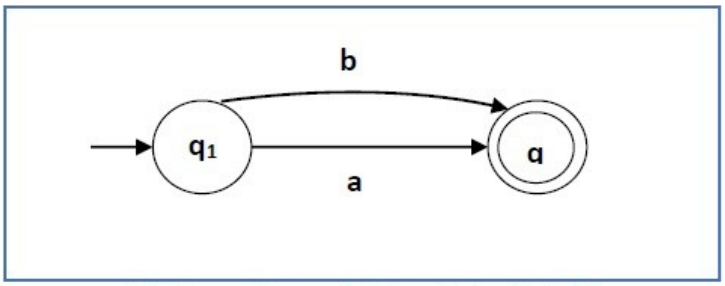
Finite automata for RE = a

Case 2 − For a regular expression ‘ab’, we can construct the following FA −



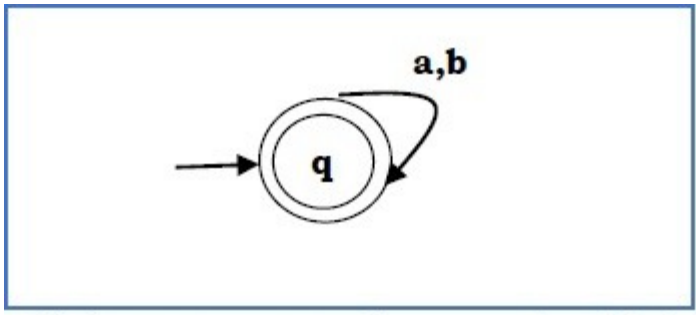
Finite automata for RE = ab

Case 3 − For a regular expression (a+b), we can construct the following FA −



Finite automata for RE= (a+b)

Case 4 − For a regular expression (a+b)*, we can construct the following FA −



Finite automata for RE= (a+b)*

Method

Step 1 Construct an NFA with Null moves from the given regular expression.

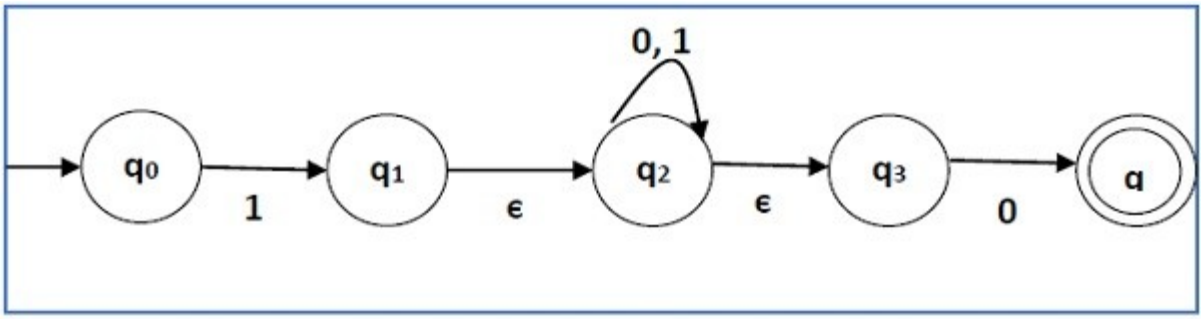
Step 2 Remove Null transition from the NFA and convert it into its equivalent DFA.

Problem

Convert the following RA into its equivalent DFA − 1 (0 + 1)* 0

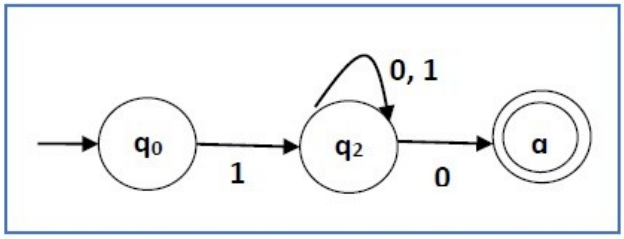
Solution

We will concatenate three expressions "1", "(0 + 1)*" and "0"



NDFA with NULL transition for RA: 1 (0 + 1)* 0

Now we will remove the **ε** transitions. After we remove the **ε** transitions from the NDFA, we get the following −



NDFA without NULL transition for RA: 1 (0 + 1)* 0

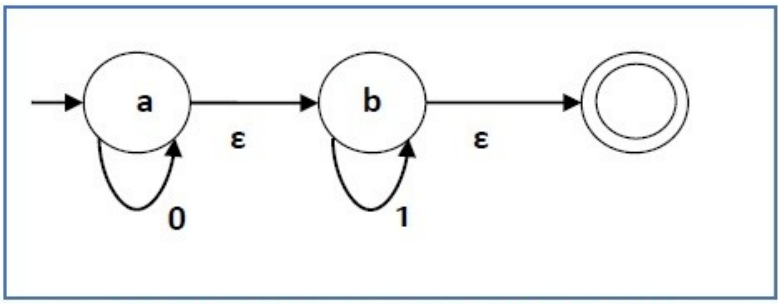
It is an NDFA corresponding to the RE − 1 (0 + 1)* 0. If you want to convert it into a DFA, simply apply the method of converting NDFA to DFA discussed in Chapter 1.

Finite Automata with Null Moves (NFA-ε)

A Finite Automaton with null moves (FA-ε) does transit not only after giving input from the alphabet set but also without any input symbol. This transition without input is called a **null move**.

An NFA-ε is represented formally by a 5-tuple (Q, ∑, δ, q_0, F), consisting of

- **Q** − a finite set of states
- ∑ − a finite set of input symbols
- **δ** − a transition function $\delta : Q \times (\sum \cup \{\varepsilon\}) \rightarrow 2^Q$
- **q_0** − an initial state $q_0 \in Q$
- **F** − a set of final state/states of Q ($F \subseteq Q$).



Finite automata with Null Moves

The above **(FA-ε)** accepts a string set − {0, 1, 01}

Removal of Null Moves from Finite Automata

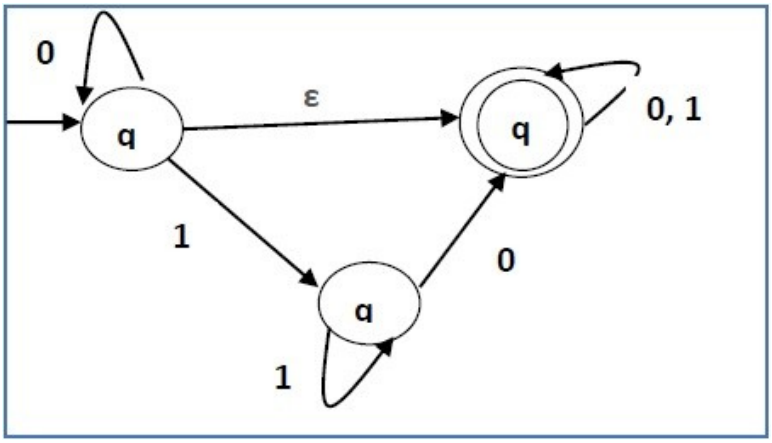
If in an NDFA, there is ϵ-move between vertex X to vertex Y, we can remove it using the following steps −

- Find all the outgoing edges from Y.

- Copy all these edges starting from X without changing the edge labels.
- If X is an initial state, make Y also an initial state.
- If Y is a final state, make X also a final state.

Problem

Convert the following NFA-ε to NFA without Null move.



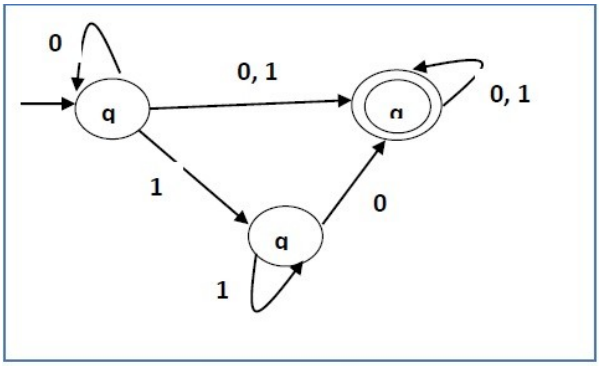
Solution

Step 1 −

Here the ε transition is between $\mathbf{q_1}$ and $\mathbf{q_2}$, so let $\mathbf{q_1}$ is **X** and $\mathbf{q_f}$ is **Y**. Here the outgoing edges from q_f is to q_f for inputs 0 and 1.

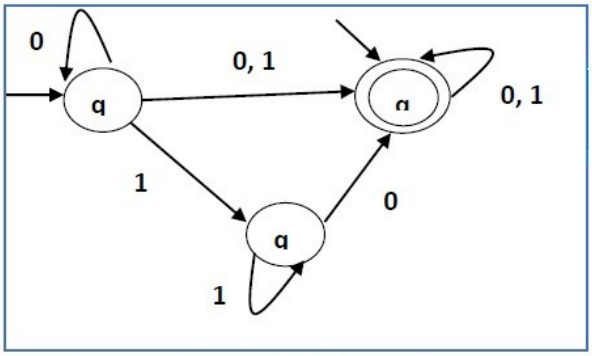
Step 2 −

Now we will Copy all these edges from q_1 without changing the edges from q_f and get the following FA −



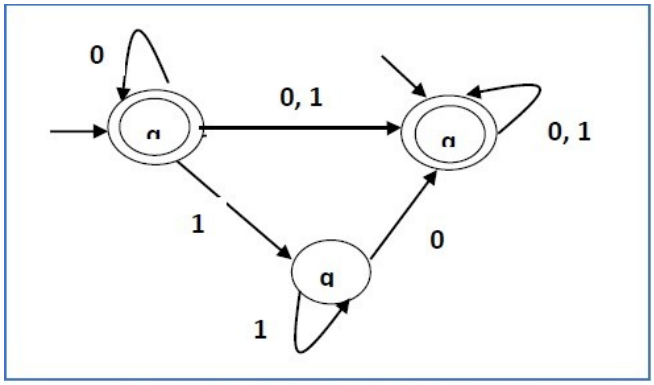
NDFA after step 2

Step 3 −Here q_1 is an initial state, so we make q_f also an initial state.So the FA becomes −



NDFA after Step 3

Step 4 −Here q_f is a final state, so we make q_1 also a final state. So the FA becomes −



Final NDFA without NULL moves

EVALUATION:

Involvement (4)	Understanding / Problem solving (3)	Timely Completion (3)	**Total** **(10)**

Signature with date: _____

EXPERIMENT NO: 7 **DATE: / /**

TITLE: WRITE A PROGRAM TO IMPLEMENT SYMBOL TABLE USING C LANGUAGE.

OBJECTIVES: On completion of this experiment student will able to…

- concept of symbol table.

THEORY:

ALGORITHM:

1. Open the file pointer for the input file.
2.Read each string and separate the token.
3. For identifiers use ‘id’,’k’ for keywords and‘s’ for symbols.
4. Write each string in the above form to an output file.
5. For each identifier read make an entry in symbol table.
6. For each symbol mention its type, value, relocatable location and size.
7. If there is an assignment expression updates the table.

Symbol Table is an important data structure created and maintained by the compiler in order to keep track of semantics of variable i.e. it stores information about scope and binding information about names, information about instances of various entities such as variable and function names, classes, objects, etc.

- It is built in lexical and syntax analysis phases.
- The information is collected by the analysis phases of compiler and is used by synthesis phases of compiler to generate code.
- It is used by compiler to achieve compile time efficiency.
- It is used by various phases of compiler .

Symbol Table entries – Each entry in symbol table is associated with attributes that support compiler in different phases.

Items stored in Symbol table:

- Variable names and constants
- Procedure and function names
- Literal constants and strings

- Compiler generated temporaries
- Labels in source languages

Information used by compiler from Symbol table:

- Data type and name
- Declaring procedures
- Offset in storage
- If structure or record then, pointer to structure table.
- For parameters, whether parameter passing by value or by reference
- Number and type of arguments passed to function
- Base Address

Operations of Symbol table – The basic operations defined on a symbol table include:

Operation	Function
allocate	to allocate a new empty symbol table
free	to remove all entries and free storage of symbol table
lookup	to search for a name and return pointer to its entry
insert	to insert a name in a symbol table and return a pointer to its entry
set_attribute	to associate an attribute with a given entry
get_attribute	to get an attribute associated with a given entry

EVALUATION:

Involvement (4)	Understanding / Problem solving (3)	Timely Completion (3)	**Total (10)**

Signature with date: _____

EXPERIMENT NO: 8 **DATE: / /**

TITLE: WRITE A PROGRAM TO FIND FIRST & FOLLOW FROM A GRAMMAR.

OBJECTIVES: On completion of this experiment student will able to…

- concept of first and follow .

THEORY:

- **Why FIRST?**

We saw the need of backtrack in the Introduction to Syntax Analysis, which is really a complex process to implement. There can be easier way to sort out this problem:
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".

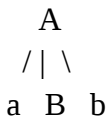
Thus, in the example above, if it knew that after reading character 'c' in the input string and applying S->cAd, next character in the input string is 'a', then it would have ignored the production rule A->bc (because 'b' is the first character of the string produced by this production rule, not 'a'),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

(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'). 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.

- **Why FOLLOW?**
  The parser faces one more problem. Let us consider below grammar to understand this problem.

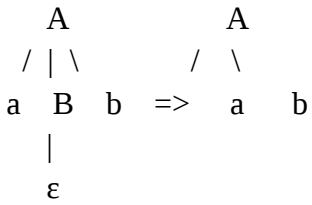
  A -> aBb
  B -> c | ε

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.



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 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.



So FOLLOW can make a Non-terminal to 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.

EVALUATION:

Involvement (4)	Understanding / Problem solving (3)	Timely Completion (3)	**Total (10)**

Signature with date: _____

EXPERIMENT NO: 9 **DATE: / /**

TITLE: WRITE A PROGRAM TO IMPLEMENT CONSTRUCTION OF OPERATOR PRECEDENCE PARSE TABLE

OBJECTIVES: On completion of this experiment student will able to…

- solve multiplication of two numbers

THEORY:

A grammar that satisfies the following 2 conditions is called as **Operator Precedence Grammar**–

- There exists no production rule which contains ε on its RHS.
- There exists no production rule which contains two non-terminals adjacent to each other on its RHS.
- It represents a small class of grammar.
- But it is an important class because of its widespread applications.

Examples-

Designing Operator Precedence Parser-

In operator precedence parsing,

- Firstly, we define precedence relations between every pair of terminal symbols.
- Secondly, we construct an operator precedence table.

Defining Precedence Relations-

The precedence relations are defined using the following rules-

Rule-01:

- If precedence of b is higher than precedence of a, then we define a < b
- If precedence of b is same as precedence of a, then we define a = b
- If precedence of b is lower than precedence of a, then we define a > b

Rule-02:

- An identifier is always given the higher precedence than any other symbol.
- $ symbol is always given the lowest precedence.

Rule-03:

- If two operators have the same precedence, then we go by checking their associativity.

Parsing A Given String- The given input string is parsed using the following steps-

Step-01:

Insert the following-

- $ symbol at the beginning and ending of the input string.
- Precedence operator between every two symbols of the string by referring the operator precedence table.

Step-02:

- Start scanning the string from LHS in the forward direction until > symbol is encountered.
- Keep a pointer on that location.

Step-03:

- Start scanning the string from RHS in the backward direction until < symbol is encountered.
- Keep a pointer on that location.

Step-04:

- Everything that lies in the middle of < and > forms the handle.
- Replace the handle with the head of the respective production.

Step-05:

Keep repeating the cycle from Step-02 to Step-04 until the start symbol is reached.

Operator Precedence Functions-

Precedence functions perform the mapping of terminal symbols to the integers.

- To decide the precedence relation between symbols, a numerical comparison is performed.
- It reduces the space complexity to a large extent.

PRACTICE PROBLEM BASED ON OPERATOR PRECEDENCE PARSING-

Problem-01 : Consider the following grammar-

$$E \rightarrow EAE \mid id$$

$$A \rightarrow + \mid x$$

Construct the operator precedence parser and parse the string id + id x id.

Solution- Step-01:We convert the given grammar into operator precedence grammar. The equivalent operator precedence grammar is-
$E \rightarrow E + E \mid E \times E \mid id$

	id	+	x	$
id		>	>	>
+	<	>	<	>
x	<	>	>	>
$	<	<	<	

Operator Precedence Table

Step-02: The terminal symbols in the grammar are { id, + , x , $ } We construct the operator precedence table as-

Parsing Given String-

Given string to be parsed is **id + id x id**. We follow the following steps to parse the given string-

Step-01: We insert $ symbol at both ends of the string as- $ id + id x id $

We insert precedence operators between the string symbols as- **$ < id > + < id > x < id > $**

Step-02: We scan and parse the string as-

$ **< id >** + < id > x < id > $

$ E + **< id >** x < id > $

$ E + E x **< id >** $

$ E + E x E $

$ + x $

$ < + **< x >** $

$ **< + >** $

$ $

EVALUATION:

Involvement (4)	Understanding / Problem solving (3)	Timely Completion (3)	**Total (10)**

Signature with date: _____

EXPERIMENT NO: 10 **DATE: / /**

TITLE: **WRITE A C PROGRAM TO IMPLEMENT SIMPLE LR PARSING ALGORITHM .**

OBJECTIVES: On completion of this experiment student will able to…

- clear the concept of LR Parser

THEORY: ALGORITHM:

Input: An input string w and an LR parsing table with functions action and goto for a grammar G.

Output: If w is in L(G), a bottom – up parse for w; otherwise an error indication.

Method:

Initially, the parser has s0 on its stack, s0 is the initial state, and w$ in the input buffer. The parser then executes the program until accept or error action is encountered.

```
set ip to point to the first symbol of w$;
repeat forever begin
let s be the state on the top of the stack and
a the symbol pointed to by ip;
if action [s, a] = shift s□ then begin
push a then s□ on the top of the stack;
advance ip to the next input symbol . end
else if action [s, a] = reduce A □ □ then begin
pop 2*|□| symbols off the stack;
let s□ be the state now on top of the stack;
push A then goto [s□, A] on top of the stack;
output the production A □ □
end
else if action [s, a] = accept then
return
else error() end
```

EVALUATION:

Involvement (4)	Understanding / Problem solving (3)	Timely Completion (3)	**Total (10)**

Signature with date: _____