**Lexical Analysis**

**Aim**: To implement a lexical analyser based on the given problem.

**Algorithm**:

• Tokenization i.e., Dividing the program into valid tokens.

• Remove white space characters.

• Remove comments.

• It also provides help in generating error messages by providing row numbers and column numbers.

**EXP 2 - RE to NFA**

**AIM:** To convert a given regular expression into its non- deterministic finite automata representation using C++

# REQUIREMENTS:

1. Knowledge of languages and regular expression
2. Knowledge of NFA and its state diagram
3. Online GDB compiler to execute and implement code

# ALGORITHM:

**STEP 1:** A structure is created for each kind of regular expression that we want to create into NDFA.

**STEP 2:** Under each structure, we define 3 variables - start state, alphabet input and n-state.

**STEP 3:** Each state changes into another state based on the alphabet input symbol. All these are defined in the structures

**STEP 4:** A switch case is made with 4 choices (4 different regular expressions) and based on the choice selected, an NFA diagram is shown as output.

**ELIMINATION OF LEFT RECURSION**

**EX. NO. 4(a)**

**AIM:** A program for Elimination of Left Recursion.

**ALGORITHM:**

* Start the program.
* Initialize the arrays for taking input from the user.
* Prompt the user to input the no. of non-terminals having left recursion and no. of productions for these non-terminals.
* Prompt the user to input the production for non-terminals.
* Eliminate left recursion using the following rules: -
  + A->Aα1| Aα2 | |Aαm
  + A->β1| β2| | βn
* Then replace it by
  + A-> βi A’ i=1,2,3, … m
  + A’-> αj A’ j=1,2,3, … n
  + A’-> Ɛ
* After eliminating the left recursion by applying these rules, display the productions without left recursion.
* Stop.

**ELIMINATION OF LEFT FACTORING**

**EX. NO. 4(b)**

**AIM:** A program for implementation Of Left Factoring

**ALGORITHM:**

* Start
* Ask the user to enter the set of productions
* Check for common symbols in the given set of productions by comparing with:
  + A->aB1|aB2
* If found, replace the particular productions with:
  + A->aA’
  + A’->B1 | B2|ɛ
* Display the output
* Exit

**FIRST and FOLLOW Computation**

**Aim** : To write a C program to find the FIRST and the FOLLOW of a grammar.

**Algorithm:**

1. If X is a terminal then FIRST(X) = {X} Example: F -> I | id We can write it as FIRST(F) -> { ( , id )
2. 2 If X is a non-terminal like E -> T then to get FIRSTI substitute T with other productions until you get a terminal as the first symbol
3. . If X -> ε then add ε to FIRST(X). For computing the follow:
4. 1. Always check the right side of the productions for a non-terminal, whose FOLLOW set is being found. (never see the left side).
5. 2. (a) If that non-terminal (S,A,B…) is followed by any terminal (a,b…,\*,+,(,)…) , then add that terminal into the FOLLOW set. (b) If that non-terminal is followed by any other non-terminal then add FIRST of other nonterminal into the FOLLOW set..

**EXPERIMENT- PREDICTIVE PARSING**

**Aim:** A program for Predictive Parsing

**Algorithm: -**

1. Start the program.
2. Initialize the required variables.
3. Get the number of coordinates and productions from the user.
4. Perform the following
   1. for (each production A → α in G) { for (each terminal a in FIRST(α)) add A → α to M[A, a];
   2. if (ε is in FIRST(α))
   3. for (each symbol b in FOLLOW(A)) add A → α to M[A, b];
5. Print the resulting stack.
6. Print if the grammar is accepted or not.
7. Exit the program.

# Computation of

# LR (0) Items

**Aim:** A program to implement LR (0) items

# Algorithm: -

1. Start.
2. Create structure for production with LHS and RHS.
3. Open file and read input from file.
4. Build state 0 from extra grammar Law S' -> S $ that is all start symbol of grammar and one Dot (.) before S symbol.
5. If Dot symbol is before a non-terminal, add grammar laws that this non-terminal is in Left Hand Side of that Law and set Dot in before of first part of Right-Hand Side.
6. If state exists (a state with this Laws and same Dot position), use that instead.
7. Now find set of terminals and non-terminals in which Dot exist in before.
8. If step 7 Set is non-empty go to 9, else go to 10.
9. For each terminal/non-terminal in set step 7 create new state by using all grammar law that Dot position is before of that terminal/non-terminal in reference state by increasing Dot point to next part in Right Hand Side of that laws.
10. Go to step 5.
11. End of state building.
12. Display the output.
13. End.

**EXP 8: LEADING AND TRAILING**

**AIM:** A program to implement Leading and Trailing

**ALGORITHM:**

1. For Leading, check for the first non-terminal.
2. If found, print it.
3. Look for next production for the same non-terminal.
4. If not found, recursively call the procedure for the single non-terminal present before the comma or End of Production String.
5. Include its results in the result of this non-terminal.
6. For trailing, we compute same as leading but we start from the end of the production to the beginning.
7. Stop

**EXP 8: LEADING AND TRAILING**

**AIM:** A program to implement Leading and Trailing

**ALGORITHM:**

1. For Leading, check for the first non-terminal.
2. If found, print it.
3. Look for next production for the same non-terminal.
4. If not found, recursively call the procedure for the single non-terminal present before the comma or End of Production String.
5. Include its results in the result of this non-terminal.
6. For trailing, we compute same as leading but we start from the end of the production to the beginning.
7. Stop

**Exp9: Computation of LR(0) Items**

**Aim:** A program to implement LR(0) items

**Algorithm:-**

1. Start.
2. Create structure for production with LHS and RHS.
3. Open file and read input from file.
4. Build state 0 from extra grammar Law S' -> S $ that is all start symbol of grammar and one Dot ( . ) before S symbol.
5. If Dot symbol is before a non-terminal, add grammar laws that this non-terminal is in Left Hand Side of that Law and set Dot in before of first part of Right Hand Side.
6. If state exists (a state with this Laws and same Dot position), use that instead.
7. Now find set of terminals and non-terminals in which Dot exist in before.
8. If step 7 Set is non-empty go to 9, else go to 10.
9. For each terminal/non-terminal in set step 7 create new state by using all grammar law that Dot position is before of that terminal/non-terminal in reference state by increasing Dot point to next part in Right Hand Side of that laws.
10. Go to step 5.
11. End of state building.
12. Display the output.
13. End.

**NFA to DFA**

**Aim** : To write a program for conversion of NFA to DFA.

**Algorithm**: 1. Start

2. Get the input from the user

3. Set the only state in SDFA to “unmarked”.

4. while SDFA contains an unmarked state do: i Let T be that unmarked state ii for each a in % do S = e-Closure(MoveNFA(T,a)) iii if S is not in SDFA already then, add S to SDFA (as an “unmarked” state) iv Set MoveDFA(T,a) to S

5. For each S in SDFA if any s & S is a final state in the NFA then, mark S an a final state in the DFA

6. Print the result.

7. Stop the program