**2.Algorithm: Lexical Analyzer Using C**

**Aim:  
To design and implement a lexical analyzer for a given language using C that ignores redundant spaces, tabs, and newlines and classifies tokens as keywords, operators, constants, or identifiers.**

**Step 1:** Start.

**Step 2:** Read the source file name from the user.

**Step 3:** Open the following files:

* fi → source code file in read mode.
* fo → intermediate file in write mode.
* fop → file containing operators list in read mode.
* fk → file containing keywords list in read mode.

**Step 4:** Read a character from fi.

**Step 5:** If the character is a letter, digit, [ , ] , or ., write it to fo.

**Step 6:** Else if the character is a newline, write "$" with tabs to fo.

**Step 7:** Else, write the character with tabs to fo.

**Step 8:** Repeat Steps 4–7 until the end of file is reached.

**Step 9:** Close fi and fo.

**Step 10:** Reopen the intermediate file fo in read mode.

**Step 11:** Set line counter i = 1 and print the first line number.

**Step 12:** Read each token from the intermediate file.

**Step 13:** If the token is "$", increment the line counter and print the new line number, then read the next token.

**Step 14:** Compare the token with each entry in the operators file fop.

* If matched, print token with operator type and set flag = 1.

**Step 15:** Compare the token with each entry in the keywords file fk.

* If matched, print token with keyword type and set flag = 1.

**Step 16:** If flag = 0, then:

* If the token starts with a digit, classify as constant.
* Else classify as identifier.

**Step 17:** Reset (rewind) the operator and keyword file pointers after each search.

**Step 18:** Repeat Steps 12–17 until the end of intermediate file is reached.

**Step 19:** Close all files.

**Step 20:** Stop.

**3.Algorithm: Replace Spaces, Tabs, and Newlines**

**Aim:**  
To implement a lexical analyzer using Lex tool to remove redundant spaces, tabs, and newlines.

**Step 1:** Start.

**Step 2:** Define a Lex rule to match one or more spaces " "\* and print a single space " ".

**Step 3:** Define a Lex rule to match a tab "\t" and print a single space " ".

**Step 4:** Define a Lex rule to match a newline "\n" and print an empty string "".

**Step 5:** In the main() function, open the input file "nw" in read mode and assign it to yyin.

**Step 6:** Call the function yylex() to start scanning and apply the above rules on the input text.

**Step 7:** Implement the function yywrap() to return 1 to indicate the end of file.

**Step 8:** Stop.

**4.Algorithm: Identification of Positive and Negative Integers**

**Aim:**  
To create a Lex program that identifies positive numbers, negative numbers, and identifiers.

**Step 1:** Start.

**Step 2:** Define a Lex rule to match one or more digits [0-9]+ and print " +ve no ".

**Step 3:** Define a Lex rule to match a minus sign followed by one or more digits -[0-9]+ and print " -ve no ".

**Step 4:** Define a Lex rule to match identifiers using the pattern [a-zA-Z]+[A-Z0-9]\* and print " identifier ".

**Step 5:** In the main() function, call yylex() to begin scanning the input.

**Step 6:** Implement the function yywrap() to return 1 to indicate the end of input.

**Step 7:** Positive integers are recognized and printed as +ve no. Negative integers are recognized and printed as -ve no. Valid identifiers are recognized and printed as identifier.

**Step 8:** Stop.

**5.Algorithm: Recognise the Language L = { 0ⁿ 1ᵐ , n ≥ 1, m ≥ 0 }**

**Aim:**  
To write a Lex program that accepts strings belonging to the language L = { 0ⁿ 1ᵐ , n ≥ 1, m ≥ 0 }.

**Step 1:** Start.

**Step 2:** Declare a global integer variable find and initialize it to 0.

**Step 3:** Define a Lex rule to match the pattern 0+1\* and set find = 1.

* This matches one or more 0s followed by zero or more 1s.

**Step 4:** Define a Lex rule to match the pattern 0\*1+0+ and set find = 0.

* This matches invalid sequences containing 1s followed by 0s.

**Step 5:** Define a rule for end of input (\n) to check:

* If find == 1, print "The string is accepted".
* Else, print "The string is not accepted".
* Exit the program.

**Step 6:** In the main() function, display "Enter the string:" and call yylex() to start scanning the input string.

**Step 7:** Implement the yywrap() function to return 1 to indicate the end of input.

**Step 8:** Execute the program and enter test strings.

**Step 9:** Strings of the form 0ⁿ1ᵐ, where n ≥ 1 and m ≥ 0, are recognized and accepted; otherwise, they are rejected.

**Step 10:** Stop.

**6.Algorithm: Counting of Characters, Words, and Lines**

**Aim:**  
To create a Lex program that counts the number of characters, words, and lines in a given file.

**Step 1:** Start.

**Step 2:** Initialize variables lines = 0, words = 0, s\_letters = 0, c\_letters = 0, num = 0, spl\_char = 0, total = 0.

**Step 3:** Define Lex rule for newline \n to increment lines by 1 and words by 1.

**Step 4:** Define Lex rule for space or tab [\t ''] to increment words by 1.

**Step 5:** Define Lex rule for capital letters [A-Z] to increment c\_letters by 1.

**Step 6:** Define Lex rule for small letters [a-z] to increment s\_letters by 1.

**Step 7:** Define Lex rule for digits [0-9] to increment num by 1.

**Step 8:** Define Lex rule for any other character . to increment spl\_char by 1.

**Step 9:** In the main() function, open the input file in read mode and assign it to yyin.

**Step 10:** Call yylex() to start scanning and applying the rules.

**Step 11:** After scanning, calculate total = s\_letters + c\_letters + num + spl\_char.

**Step 12:** Print the number of lines, words, small letters, capital letters, digits, special characters, and total characters.

**Step 13:** Define yywrap() to return 1 to indicate the end of file.

**Step 14:** Run the executable and display the counts for the input file.

**Step 15:** Stop.

7. RECOGNIZE VALID ARITHMETIC EXPRESSION

AIM: - To recognize a valid arithmetic expression that uses operator +, -, \*, / etc.

**Step 1:** Start.

**Step 2:** In the Lex file, define a rule to match variable names [a-zA-Z]+ and return the token VARIABLE.

**Step 3:** Define a rule to match numbers [0-9] and return the token NUMBER.

**Step 4:** Ignore tabs [\t] and take no action.

**Step 5:** For newline \n, return 0 to indicate end of input.

**Step 6:** For any other character ., return the first character of the matched text.

**Step 7:** In the YACC file, declare tokens NUMBER and VARIABLE.

**Step 8:** Specify operator precedence: + and - lowest, \*, /, % higher, and parentheses (, ) highest.

**Step 9:** Define the grammar rule S : VARIABLE '=' E where E is an expression.

**Step 10:** Define the grammar rules for E as:  
E '+' E | E '-' E | E '\*' E | E '/' E | E '%' E | '(' E ')' | NUMBER | VARIABLE.

**Step 11:** In the action part of S, print "Entered arithmetic expression is valid" and return 0.

**Step 12:** In the main() function, display a message to enter an arithmetic expression and call yyparse() to parse input.

**Step 13:** Define yyerror() to print "Entered arithmetic expression is invalid".

**Step 14:** Run the executable and enter test expressions.

**Step 15:** If the expression satisfies the grammar, display "valid", otherwise display "invalid".

**Step 16:** Stop.

8.Recognize a valid variable

AIM:To recognize a valid variable which starts with a letter followed by any number of letters or digits.

**Step 1:** Start.

**Step 2:** In the Lex file, define a rule to match digits [0-9] and return the token DIGIT.

**Step 3:** Define a rule to match letters [a-zA-Z] and return the token LETTER.

**Step 4:** Ignore tabs [\t] and take no action.

**Step 5:** For newline \n, return 0 to indicate end of input.

**Step 6:** For any other character ., return the first character of the matched text.

**Step 7:** In the YACC file, declare tokens DIGIT and LETTER.

**Step 8:** Define the grammar rule stmt : LETTER tail.

**Step 9:** Define the grammar rule tail : LETTER tail | DIGIT tail | /\* empty \*/.

**Step 10:** In the main() function, display a message to enter a string and call yyparse() to start parsing.

**Step 11:** If the entered string satisfies the grammar rules, print "valid" and exit.

**Step 12:** Define the function yyerror() to print "invalid" and exit when the input does not satisfy the grammar.

**Step 13:** Run the executable and enter test strings.

**Step 14:** If the variable starts with a letter followed by any number of letters or digits, display "valid", otherwise display "invalid".

**Step 15:** Stop.

9.CALCULATOR

AIM:To implement a calculator using Lex and Yacc.

**Step 1:** Start.

**Step 2:** In the Lex file, define a macro DIGIT to match integers or floating‑point numbers.

**Step 3:** For the pattern {DIGIT}, convert yytext to a floating‑point value, assign it to yylval, and return the token NUM.

**Step 4:** For newline or any other character, return the character itself.

**Step 5:** In the YACC file, declare the token NUM and set YYSTYPE to double.

**Step 6:** Define operator precedence so that + and - have the lowest precedence, \* and / have higher precedence, and UMINUS has the highest precedence with right associativity.

**Step 7:** Define the grammar rule S for repeated expressions separated by newlines, allowing empty lines and error handling.

**Step 8:** Define the grammar rules for E to handle addition, subtraction, multiplication, division, parentheses, unary minus, and numbers.

**Step 9:** In the semantic actions of the grammar rules, perform the arithmetic operation and store the result in the left‑hand side variable.

**Step 10:** When an expression is successfully parsed, print the calculated result.

**Step 11:** In the main() function, display a prompt to enter an expression and call yyparse() to process the input.

**Step 12:** Define the yyerror() function to display an error message when invalid input is entered.

**Step 13:** Run the executable and enter test expressions.

**Step 14:** Display the calculated result for valid expressions and error messages for invalid expressions.

**Step 15:** Stop.

10.CONVERSION OF E NFA TO NFA

AIM:To convert NFA with epsilon transition to NFA without epsilon transition

**Step 1:** Start.

**Step 2:** Define constants for maximum number of states and symbols.

**Step 3:** Create two transition tables, NFAtab for the original NFA with ε‑transitions and NewNFAtab for the converted NFA without ε‑transitions.

**Step 4:** Implement a function string\_merge() to merge two state strings in sorted order without duplicates.

**Step 5:** Implement epsilon\_closure(state) to find all states reachable from a given state using only ε‑transitions.

**Step 6:** Implement epsilon\_closure\_set(states) to compute the ε‑closure for a set of states.

**Step 7:** Implement get\_next\_states(states, symbol) to find all states reachable from a set of states using a given input symbol and then take the ε‑closure of that result.

**Step 8:** Implement remove\_epsilon\_transitions() to iterate over each state, compute its ε‑closure, and for each input symbol store the reachable states in NewNFAtab.

**Step 9:** Implement a function print\_NFA(table, states, symbols) to display the state transition table.

**Step 10:** Initialize the NFA with ε‑transitions in init\_NFA\_with\_epsilon() by defining sample transitions.

**Step 11:** In main(), call init\_NFA\_with\_epsilon(), print the original NFA, call remove\_epsilon\_transitions(), and print the new NFA without ε‑transitions.

**Step 12:** Verify that all ε‑transitions have been removed and the new NFA transition table is correct.

**Step 13:** Stop.

Bottom of Form