1. Start.
2. Declare all variables & file pointers.
3. Display the I/p programs.
4. Seperate the keyword in the prgm & display it.
5. Display the header ~~pogms~~ files of the I/p prgm.
6. seperate operators of the I/p prgm & display it.
7. Print the punctuation marks.
8. Print the costant that are present in I/p prgm.
9. Print the identifiers of the I/p prgm.
10. Stop.

Calculator Using Lex

1. Take the I/p string
2. Call the function yylex(). The control will given to rule section.
3. Define the rules.

```
dig [0-9]+([0-9]*)"."([0-9]+)
add "+"        div "/"
sub "-"        pow "^"
mul "*"
```

4. Check the rules as

```
{dig} {digi();}        {div} {op=4;}
{add} {op=1;}          {pow} {op=5;}
{sub} {op=2;}
{mul} {op=3;}

{\n} {printf("\n The Answer : %f \n\n", a);}
Switch (op)
  { case1 :  a=a+b;
             break
    case2 : a=a-b;          case4 : a=a/b;
            break;                  break.
    Case3 : a=a*b;
            break
    Case5 : for (i=a; b>1; b--)
              a=a*i;
                break;
  } op=0; }}
```

5. call yywrap() to wind up the session.
6. Stop

Counting Vowels & Consonants

1. Take the I/p String.
2. Call the funct^n yylex(). The control will given to rule section.
3. Check rule if input is [aeiou AEIOU] then increment count as V++. Else,
4. Increment Count as C++.
5. Output the no: of vowels & Consonants.

Counting no: of Words, lines

1. Take a string as I/p & Store it in the array of characters.
2. Create 3 counter variable for the count of words, lines & characters in the string.
3. Using for loop search for a ~~search~~ space ' ' in the string & consecutively increment the variable count for words.
4. Using for loop search for a next line \n in the string & consecutively increment a variable count for next ~~line~~ line.
5. Using forloop search for a characters except space & new line in the string & consecutively increment a variable count for characters.
6. Repeat Step 3, 4, 5 until the loop reaches to the end of the string.
7. Check for the corner ~~access~~ cases (discussed above) & do accordingly.
8. Print all the values of the counter.

NFA to DFA

1. Start.
2. Input the required array ie, set of alphabets, set of initial state, set of final states, transitions.
3. Initially $Q = \phi$
4. Add q_0 of NFA to Q'. Then find the transition from this start state.
5. In Q', find the possible set of states for each I/p symbol. If this ~~a~~ set of states is not in Q', then add it to Q'.
6. In DFA, the final state will be all the states which contain f (final state of NFA.
7. Stop.

Intermediate Code Generation

Lex prgm

1. Start.
2. In rule section, compare yytext with regular expression if yytext match [0-9].
   return DIGIT
   If yytext match [A-Z a-z] [A-Z a-z 0-9].
   return ID
   If yytext match [+ - / * \n]
   return * yytext.
3. Stop.

YACC Prgm

1. Start.
2. Take token generated in the lex prgm.
3. Apply product^n rule to generate three-address code by storing them in temporary variables.

```
S : S ID '=' E '\n' / S E '\n' / S '\n' ;
E : DIGIT / ID / E '+' E / E '-' E / E '*' E / E / E / E / -
    E / ('E');
```

4. In main funct^n call yyparse(). 5. Stop

Backend of Compiler

1. start.
2. Read the address code.
3. Check the operation in it & copy to pr variable
4. Print MOV & variable R1.
5. Print operatn second variable R;
6. Print MOV R; , result variable.
7. stop.

Prgms

1)
2) Lexical Analyser.
3) Calculator using Lex.
4) Counting Vowels & Consonants, Lex.
5) Counting no: of words, lmB.
6) Intermediate code ~~xxx~~ generatn
7) NFA to DFA.
8) prgm for Constant propagation.
9) Yacc specificatn to recognize a valid arithmetic expressn
10) Backend of a Compiler.
11) First & Follow
12) Shift reduce paser.
