3. Implementation of lexical analyzer using lex tool.

Flex (Fast Lexical Analyzer) is a tool for generating lexical analyzers in C or C++ based on regular expressions.

Steps for implementing a lexical analyzer using Flex:

1. Install Flex:
   - You can install Flex on Linux with the following command:

     sudo apt install flex

2. Create a Lex Specification File (lexer.l): Write a .l file that defines the regular expressions (patterns) for each token type.

Example of lexer.l file:

```
%{
#include <stdio.h>
#include <stdlib.h>
%}
%%
// Regular Expressions (token definitions)
int     [a-zA-Z_][a-zA-Z0-9_]*
float   [0-9]+\.[0-9]+
digit   [0-9]+
keyword int|float|if|else|for|while|return
%%
// Rules and actions
{keyword}       { printf("Token: %s | Type: KEYWORD\n", yytext); }
{int}           { printf("Token: %s | Type: IDENTIFIER\n", yytext); }
{float}         { printf("Token: %s | Type: CONSTANT\n", yytext); }
{digit}         { printf("Token: %s | Type: CONSTANT\n", yytext); }
"+"|"-"|"*"|"/" { printf("Token: %s | Type: OPERATOR\n", yytext); }
"(" | ")" | ";" | "," { printf("Token: %s | Type: PUNCTUATION\n", yytext); }
"//".*          { printf("Token: %s | Type: COMMENT\n", yytext); }
"/*".* "*/"     { printf("Token: %s | Type: COMMENT\n", yytext); }
[ \t\n\r]+      { /* Ignore whitespace */ }

.               { printf("Token: %s | Type: UNKNOWN\n", yytext); }
%%

int main() {
   yylex();  // Invoke the lexical analyzer
   return 0;
}
```

Explanation:

- Regular Expressions: Similar to JLex, we define token patterns such as identifiers, constants, operators, comments, etc.
- Actions: Each pattern has an associated action. When a pattern is matched, it prints the matched token and its type.

3. Generate the Lexer with Flex: After creating the lexer.l file, run Flex to generate the C code for the lexical analyzer:
   flex lexer.l

   This will create a file called lex.yy.c containing the lexer code.

4. Compile the C Code: Compile the generated lex.yy.c file using GCC:
   gcc lex.yy.c -o lexer -lfl

   The -lfl flag is used to link the Flex library.

5. Run the Lexer: After compiling, you can run your lexer on a text input file:
   ./lexer < input.c

   This will read the file input.c, process it, and output the recognized tokens.

Example Output:

For the input:

```
int x = 100;  // This is a comment
float y = 3.14;
x = x + y;
```

5. Convert the bnf rules into yacc form and write code to generate abstract syntax tree.

To convert BNF (Backus-Naur Form) rules into Yacc form and write code to generate an Abstract Syntax Tree (AST), let's break it down into steps:

1. Understanding the BNF Grammar: We'll first take a simple arithmetic expression grammar in BNF form.
2. Convert BNF to Yacc: Convert the BNF rules to Yacc's syntax and add actions to build the AST.
3. Define AST Nodes: Create structs to represent the AST nodes and their relationships.
4. Actions in Yacc: During parsing, we will create AST nodes for each operation.

Let's implement the solution for a simple arithmetic expression that uses the operators +, -, *, / and constructs an Abstract Syntax Tree.

1. BNF Grammar

A simple BNF for arithmetic expressions:

```
<expression> ::= <expression> '+' <term>
           | <expression> '-' <term>
           | <term>
<term>       ::= <term> '*' <factor>
           | <term> '/' <factor>
           | <factor>
<factor>     ::= <number>
           | '(' <expression> ')'
<number>     ::= [0-9]+
```

This is the arithmetic grammar for addition, subtraction, multiplication, division, and parentheses.

2. Convert BNF to Yacc

We will now translate this BNF grammar to Yacc form and include actions to create the Abstract Syntax Tree.

3. Define Abstract Syntax Tree (AST) Structure

An AST is represented using structs. For arithmetic expressions, the nodes can represent operations (like addition or multiplication) and numbers.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
// Abstract Syntax Tree (AST) structures
typedef struct node {

   enum { NUM, ADD, SUB, MUL, DIV } type; // Node types: NUM (number), ADD
(addition), SUB (subtraction), MUL (multiplication), DIV (division)
   union {
      int value; // Holds the number for NUM nodes
      struct {   // Holds pointers to child nodes for operator nodes
         struct node *left;
         struct node *right;
      };
   };
} Node;
// Functions to create nodes for the AST
Node *new_number(int value) {
   Node *n = malloc(sizeof(Node));
   n->type = NUM;
   n->value = value;
   return n;
}
Node *new_operator(int type, Node *left, Node *right) {
   Node *n = malloc(sizeof(Node));
   n->type = type;
   n->left = left;
   n->right = right;
   return n;
}

void print_ast(Node *n) {
   if (!n) return;
   if (n->type == NUM) {
      printf("%d", n->value);
```