BIT STUFFING:

```
import java.util.*;
public class BitStuffing {
   static String bitStuff(String data) {
      StringBuilder stuffed = new StringBuilder();
      int count = 0, i = 0;
      while (i < data.length()) {
         char bit = data.charAt(i);
         stuffed.append(bit);
         if (bit == '1') {
            count++;
         } else if (bit == '0' && count == 5) {
            stuffed.append('0');
            count = 0;
         } else {
            count = 0;
         }
         i++;
      }
      return stuffed.toString();
   }
   static String bitDestuff(String data) {
      StringBuilder destuffed = new StringBuilder();
      int count = 0, i = 0;
      while (i < data.length()) {
         char bit = data.charAt(i);
         destuffed.append(bit);
         if (bit == '1') {
            count++;
         } else if (bit == '0' && count == 5) {
            i++;
            count = 0;
            continue;
         } else {
            count = 0;
         }
         i++;
      }
      return destuffed.toString();
   }
   static String addFlags(String data) {
      return "01111110" + data + "01111110";
   }
```

```
static String removeFlags(String data) {
    if (data.length() >= 16) {
        return data.substring(8, data.length() - 8);
    } else {
        return "";
    }
}
static List<String> frameData(String data, int frameSize) {
    List<String> frames = new ArrayList<>();
    for (int i = 0; i < data.length(); i += frameSize) {
        int end = Math.min(i + frameSize, data.length());
        frames.add(data.substring(i, end));
    }
    return frames;
}
public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.print("Enter the data bits (e.g., 110111...): ");
    String dataBits = sc.nextLine();
    System.out.print("Enter the frame size (e.g., 10): ");
    int frameSize = sc.nextInt();
    List<String> framed = frameData(dataBits, frameSize);
    List<String> stuffedFrames;
    if (frameSize > 8) {
        stuffedFrames = new ArrayList<>();
        for (String f : framed) {
            stuffedFrames.add(bitStuff(f));
        }
    } else {
        stuffedFrames = framed;
    }
    List<String> flaggedFrames = new ArrayList<>();
    for (String f : stuffedFrames) {
        flaggedFrames.add(addFlags(f));
    }
    System.out.println("\n--- Transmitted Frames ---");
    for (int i = 0; i < flaggedFrames.size(); i++) {
        System.out.println("Frame " + (i + 1) + ": " + flaggedFrames.get(i));
    }
    StringBuilder receivedData = new StringBuilder();
    for (String f : flaggedFrames) {
        receivedData.append(removeFlags(f));
    }
```

```
        String destuffedData;
        if (frameSize > 8) {
            destuffedData = bitDestuff(receivedData.toString());
        } else {
            destuffedData = receivedData.toString();
        }
        System.out.println("\n--- Receiver Side ---");
        System.out.println("Destuffed Data: " + destuffedData);
        System.out.println("Matches original: " + destuffedData.equals(dataBits));
    }
}
```

Character Stuffing:

```
import java.util.Scanner;
public class Characterstuffing {
   public static String stuff(String s, String f, String e) {
      StringBuilder sd = new StringBuilder(f);
      for (int i = 0; i < s.length(); i++) {
         if (i + f.length() <= s.length() && s.substring(i, i + f.length()).equalsIgnoreCase(f)) {
            sd.append(e).append(f);
            i += f.length() - 1;
         } else if (i + e.length() <= s.length() && s.substring(i, i +
e.length()).equalsIgnoreCase(e)) {
            sd.append(e).append(e);
            i += e.length() - 1;
         } else {
            sd.append(s.charAt(i));
         }
      }
      sd.append(f);
      return sd.toString();
   }
   public static String destuff(String s, String f, String e) {
      StringBuilder dd = new StringBuilder();
      s = s.substring(f.length(), s.length() - f.length());
      for (int i = 0; i < s.length(); i++) {
         if (i + e.length() + f.length() <= s.length() && s.substring(i, i + e.length() +
f.length()).equalsIgnoreCase(e + f)) {
            dd.append(f);
            i += e.length() + f.length() - 1;
         } else if (i + e.length() + e.length() <= s.length() && s.substring(i, i + e.length() +
e.length()).equalsIgnoreCase(e + e)) {
            dd.append(e);
            i += e.length() + e.length() - 1;
         } else {
            dd.append(s.charAt(i));
         }
      }
      return dd.toString();
   }
   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      String f = "FLAG", e = "ESC";
      System.out.print("Enter packet: ");
      String s = sc.next();
```

```
        String sd = stuff(s, f, e);
        String dd = destuff(sd, f, e);
        System.out.println("Transmitted Data (Stuffed Data):\n" + sd);
        System.out.println("Receiver Data (De-Stuffed Data):\n" + dd);
        System.out.println("Matches Original: " + s.equals(dd));
        sc.close();
    }
}
```

CRC(Cyclic Redundancy Check):

```
import java.util.Scanner;
public class CRC {
    public static String xorOp(String a, String b) {
        StringBuilder result = new StringBuilder();
        for (int i = 1; i < b.length(); i++) {
            result.append(a.charAt(i) == b.charAt(i) ? '0' : '1');
        }
        return result.toString();
    }
    public static String mod2div(String dividend, String divisor) {
        int pick = divisor.length();
        String tmp = dividend.substring(0, pick);
        int n = dividend.length();
        while (pick < n) {
            if (tmp.charAt(0) == '1') {
                tmp = xorOp(divisor, tmp) + dividend.charAt(pick);
            } else {
                tmp = xorOp("0".repeat(pick), tmp) + dividend.charAt(pick);
            }
            pick++;
        }
        if (tmp.charAt(0) == '1') {
            tmp = xorOp(divisor, tmp);
        } else {
            tmp = xorOp("0".repeat(pick), tmp);
        }
        return tmp;
    }
    public static String encodeData(String data, String key) {
        int keyLen = key.length();
        String appendedData = data + "0".repeat(keyLen - 1);
        String remainder = mod2div(appendedData, key);
        return data + remainder;
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter the dataword (binary): ");
        String data = sc.nextLine();
        System.out.print("Enter the divisor (binary): ");
        String key = sc.nextLine();
        String codeword = encodeData(data, key);
        System.out.println("Generated Codeword: " + codeword);
```

```
        System.out.print("Enter the received codeword (binary): ");
        String received = sc.nextLine();
        String remainder = mod2div(received, key);
        if (remainder.contains("1")) {
            System.out.println("Error detected in received codeword");
        } else {
            System.out.println("No error in received codeword");
        }
        sc.close();
    }
}
```

Dijkstra’s Algorithm:

```
import java.util.*;
public class DijkstraRouting {
   static final int INF = Integer.MAX_VALUE;
   public static int[] dijkstra(int[][] adjMatrix, int start) {
      int n = adjMatrix.length;
      int[] distances = new int[n];
      boolean[] visited = new boolean[n];
      Arrays.fill(distances, INF);
      distances[start] = 0;
      PriorityQueue<int[]> pq = new PriorityQueue<>(Comparator.comparingInt(a -> a[0]));
      pq.offer(new int[]{0, start});
      while (!pq.isEmpty()) {
         int[] current = pq.poll();
         int currentDistance = current[0];
         int u = current[1];
         if (visited[u]) continue;
         visited[u] = true;
         for (int v = 0; v < n; v++) {
            int weight = adjMatrix[u][v];
            if (weight != -1 && !visited[v]) {
               int distance = currentDistance + weight;
               if (distance < distances[v]) {
                  distances[v] = distance;
                  pq.offer(new int[]{distance, v});
               }
            }
         }
      }
      return distances;
   }
   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      System.out.print("Enter number of nodes: ");
      int n = sc.nextInt();
      int[][] adjMatrix = new int[n][n];
      System.out.println("Enter adjacency matrix row by row (-1 for no edge, 0 for self-
distance):");
      for (int i = 0; i < n; i++) {
         System.out.print("Row " + (i + 1) + ": ");
         for (int j = 0; j < n; j++) {
            adjMatrix[i][j] = sc.nextInt();
         }
```

```
        }
        System.out.print("Enter starting node index (0 to n-1): ");
        int startNode = sc.nextInt();
        int[] shortestDistances = dijkstra(adjMatrix, startNode);
        System.out.println("\nShortest distances from node " + startNode + ":");
        for (int i = 0; i < n; i++) {
            if (shortestDistances[i] == INF)
                System.out.println("Node " + i + ": Unreachable");
            else
                System.out.println("Node " + i + ": " + shortestDistances[i]);
        }
        sc.close();
    }
}
```

Lexical Analyzer:

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#define MAX_ID_LEN 31
#define MAX_TOKENS 500
const char *keywords[] = {
   "auto","break","case","char","const","continue","default","do",
   "double","else","enum","extern","float","for","goto","if",
   "int","long","register","return","short","signed","sizeof","static",
   "struct","switch","typedef","union","unsigned","void","volatile","while"
};
int keywordCount = sizeof(keywords) / sizeof(keywords[0]);
const char *operators[] = {
   "+","-","*","/","%","=","==","!=","<",">","<=",">=",
   "++","--","&&","||","!","+=","-=","*=","/="
};
int operatorCount = sizeof(operators) / sizeof(operators[0]);
const char *delimiters[] = {
   ";",",","(",")","{","}","[","]"
};
int delimiterCount = sizeof(delimiters) / sizeof(delimiters[0]);
const char *specialSymbols[] = {
   "#","\"","'","\\",".",":","?"
};
int specialCount = sizeof(specialSymbols) / sizeof(specialSymbols[0]);
char keywordsFound[MAX_TOKENS][MAX_ID_LEN];
int kwIndex = 0;
char identifiersFound[MAX_TOKENS][MAX_ID_LEN];
int idIndex = 0;
char numbersFound[MAX_TOKENS][64];
int numIndex = 0;
char operatorsFound[MAX_TOKENS][4];
int opIndex = 0;
char delimitersFound[MAX_TOKENS][4];
int delIndex = 0;
char specialsFound[MAX_TOKENS][4];
int spIndex = 0;
char stringsFound[MAX_TOKENS][256];
int strIndex = 0;
char charsFound[MAX_TOKENS][4];
int chIndex = 0;
int isKeyword(const char *str) {
```

```
    for (int i = 0; i < keywordCount; i++) {
        if (strcmp(str, keywords[i]) == 0) return 1;
    }
    return 0;
}
int isOperator(const char *str) {
    for (int i = 0; i < operatorCount; i++) {
        if (strcmp(str, operators[i]) == 0) return 1;
    }
    return 0;
}
int isDelimiter(char c) {
    for (int i = 0; i < delimiterCount; i++) {
        if (c == delimiters[i][0]) return 1;
    }
    return 0;
}
int isSpecial(char c) {
    for (int i = 0; i < specialCount; i++) {
        if (c == specialSymbols[i][0]) return 1;
    }
    return 0;
}
void skipWhitespaceAndComments(FILE *fp) {
    int c;
    while ((c = fgetc(fp)) != EOF) {
        if (isspace(c)) continue;
        if (c == '/') {
            int next = fgetc(fp);
            if (next == '/') {
                while ((c = fgetc(fp)) != '\n' && c != EOF);
            } else if (next == '*') {
                while ((c = fgetc(fp)) != EOF) {
                    if (c == '*' && (c = fgetc(fp)) == '/') break;
                }
            } else {
                ungetc(next, fp);
                ungetc(c, fp);
                return;
            }
        } else {
            ungetc(c, fp);
            return;
        }
```

```
    }
}

void getToken(FILE *fp) {
    int c;
    skipWhitespaceAndComments(fp);
    c = fgetc(fp);
    if (c == EOF) return;
    if (isalpha(c) || c == '_') {
        char buffer[MAX_ID_LEN + 1];
        int i = 0;
        buffer[i++] = c;
        while ((c = fgetc(fp)) != EOF && (isalnum(c) || c == '_')) {
            if (i < MAX_ID_LEN) buffer[i++] = c;
        }
        buffer[i] = '\0';
        ungetc(c, fp);
        if (isKeyword(buffer))
            strcpy(keywordsFound[kwIndex++], buffer);
        else
            strcpy(identifiersFound[idIndex++], buffer);
    }
    else if (isdigit(c)) {
        char buffer[64];
        int i = 0;
        buffer[i++] = c;
        while ((c = fgetc(fp)) != EOF && (isdigit(c) || c == '.')) {
            buffer[i++] = c;
        }
        buffer[i] = '\0';
        ungetc(c, fp);
        strcpy(numbersFound[numIndex++], buffer);
    }
    else if (c == '"') {
        char buffer[256];
        int i = 0;
        while ((c = fgetc(fp)) != EOF && c != '"') {
            buffer[i++] = c;
        }
        buffer[i] = '\0';
        strcpy(stringsFound[strIndex++], buffer);
    }
    else if (c == '\'') {
        char ch = fgetc(fp);
```

```
        fgetc(fp);
        charsFound[chIndex][0] = ch;
        charsFound[chIndex][1] = '\0';
        chIndex++;
    }
    else if (isDelimiter(c)) {
        char d[2] = {c, '\0'};
        strcpy(delimitersFound[delIndex++], d);
    }
    else if (isSpecial(c)) {
        char s[2] = {c, '\0'};
        strcpy(specialsFound[spIndex++], s);
    }
    else {
        char op[3] = {c, '\0', '\0'};
        int next = fgetc(fp);
        if (next != EOF) {
            op[1] = next;
            if (!isOperator(op)) {
                op[1] = '\0';
                ungetc(next, fp);
            }
        }
        if (isOperator(op)) strcpy(operatorsFound[opIndex++], op);
    }
}
int main() {
    FILE *fp = fopen("input.c", "r");
    if (!fp) {
        printf("Could not open file.\n");
        return 1;
    }
    while (!feof(fp)) {
        getToken(fp);
    }
    fclose(fp);
    printf("\nKeywords:\n");
    for (int i = 0; i < kwIndex; i++) printf("%s\n", keywordsFound[i]);
    printf("\nIdentifiers:\n");
    for (int i = 0; i < idIndex; i++) printf("%s\n", identifiersFound[i]);
    printf("\nNumbers:\n");
    for (int i = 0; i < numIndex; i++) printf("%s\n", numbersFound[i]);
    printf("\nStrings:\n");
    for (int i = 0; i < strIndex; i++) printf("\"%s\"\n", stringsFound[i]);
```

```
    printf("\nChars:\n");
    for (int i = 0; i < chIndex; i++) printf("'%s'\n", charsFound[i]);
    printf("\nOperators:\n");
    for (int i = 0; i < opIndex; i++) printf("%s\n", operatorsFound[i]);
    printf("\nDelimiters:\n");
    for (int i = 0; i < delIndex; i++) printf("%s\n", delimitersFound[i]);
    printf("\nSpecial Symbols:\n");
    for (int i = 0; i < spIndex; i++) printf("%s\n", specialsFound[i]);
    return 0;
}
```

Lexical Analyzer using lex:

Lexical_analyzer.l

```
%{
#include <stdio.h>
#include <stdlib.h>

int COMMENT = 0;
%}

identifier [a-zA-Z_][a-zA-Z0-9_]*
number [0-9]+

%%
"#".*                          { printf("\n%s is a PREPROCESSOR DIRECTIVE", yytext); }

"int"|"float"|"char"|"double"|"while"|"for"|"struct"|"typedef"|"do"|"if"|"break"|
"continue"|"void"|"switch"|"return"|"else"|"goto"
                               { if(!COMMENT) printf("\n%s is a KEYWORD", yytext); }

"/*"                           { COMMENT = 1; printf("\n%s begins a COMMENT", yytext); }
"*/"                           { COMMENT = 0; printf("\n%s ends a COMMENT", yytext); }

{identifier}\(                 { if(!COMMENT) printf("\nFUNCTION: %s", yytext); }

"{"                            { if(!COMMENT) printf("\nBLOCK BEGINS"); }
"}"                            { if(!COMMENT) printf("\nBLOCK ENDS"); }

{identifier}(\[[0-9]*\])?      { if(!COMMENT) printf("\n%s is an IDENTIFIER", yytext); }

\".*\"                         { if(!COMMENT) printf("\n%s is a STRING", yytext); }

{number}                       { if(!COMMENT) printf("\n%s is a NUMBER", yytext); }

"="                            { if(!COMMENT) printf("\n%s is an ASSIGNMENT OPERATOR", yytext);
}

"<="|">="|"<"|"=="|">"         { if(!COMMENT) printf("\n%s is a RELATIONAL OPERATOR",
yytext); }

[ \t\n]+                       ;   /* Ignore whitespace */
.                              { if(!COMMENT) printf("\nUNKNOWN TOKEN: %s", yytext); }

%%
```

```
int main(int argc, char **argv) {
   FILE *file = fopen("var.c", "r");
   if (!file) {
      printf("Could not open the file.\n");
      exit(0);
   }
   yyin = file;
   yylex();
   printf("\n");
   return 0;
}

int yywrap() {
   return 1;
}
```

Var.c

```
#include<stdio.h>
#include<conio.h>

void main() {
   int a, b, c;
   a = 1;
   b = 2;
   c = a + b;
   printf("Sum: %d", c);
}
```

YAAC program:

Calc.l

```
%{
#include <stdio.h>
#include <stdlib.h>
#include "y.tab.h"
extern int yylval;
%}

%%
[0-9]+      { yylval = atoi(yytext); return NUMBER; }
[ \t]+      { /* ignore spaces */ }
\n          { return 0; }
.           { return yytext[0]; }
%%

int yywrap(void) { return 1; }
```

parse.y

```
%{
#include <stdio.h>
int flag = 0;
int yylex(void);
void yyerror(const char *s);
%}

%token NUMBER
%left '+' '-'
%left '*' '/' '%'
%left '(' ')'

%%
ArithmeticExpression
    : E { printf("Result = %d\n", $$); return 0; }
    ;

E   : E '+' E   { $$ = $1 + $3; }
    | E '-' E   { $$ = $1 - $3; }
    | E '*' E   { $$ = $1 * $3; }
    | E '/' E   { $$ = $1 / $3; }
    | E '%' E   { $$ = $1 % $3; }
    | '(' E ')' { $$ = $2; }
    | NUMBER    { $$ = $1; }
```

```
   ;
%%

int main(void) {
   printf("Enter any arithmetic expression (+, -, *, /, %% and parentheses):\n");
   yyparse();
   if (flag == 0)
      printf("Entered arithmetic expression is Valid\n");
   return 0;
}

void yyerror(const char *s) {
   (void)s;
   printf("Entered arithmetic expression is Invalid\n");
   flag = 1;
}
```

Compile and run:

```
yacc -d parse.y
lex calc.l
gcc lex.yy.c y.tab.c -w
./a.out
```

First and Follow:

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#define MAX 20
char productions[MAX][MAX];
char first[MAX][MAX];
char follow[MAX][MAX];
int prod_count = 0;
int first_computed[26] = {0};
int follow_computed[26] = {0};
int follow_visited[26] = {0}; // Prevent infinite recursion
void findFirst(char *result, char c);
void findFollow(char *result, char c);
int isNonTerminal(char c);
void addToResult(char *result, char c);
int contains(char *str, char c);
int main() {
   int i;
   printf("Enter number of productions: ");
   scanf("%d", &prod_count);
   getchar();
   printf("Enter productions (e.g., E->E+T). Use '#' for epsilon:\n");
   for (i = 0; i < prod_count; i++) {
      fgets(productions[i], MAX, stdin);
      productions[i][strcspn(productions[i], "\n")] = '\0'; // Remove newline
   }
   printf("\nFIRST sets:\n");
   for (i = 0; i < prod_count; i++) {
      char nonTerminal = productions[i][0];
      if (!first_computed[nonTerminal - 'A']) {
         findFirst(first[nonTerminal - 'A'], nonTerminal);
         first_computed[nonTerminal - 'A'] = 1;
      }
   }
   // Print FIRST sets
   for (i = 0; i < 26; i++) {
      if (first_computed[i]) {
         printf("FIRST(%c) = { ", (char)(i + 'A'));
         for (int k = 0; first[i][k] != '\0'; k++) {
            if (first[i][k] == 'i')
               printf("id");
            else if (first[i][k] == '#')
```

```
                printf("e");
            else
                printf("%c", first[i][k]);
            if (first[i][k + 1] != '\0') printf(", ");
        }
        printf(" }\n");
    }
  }
  printf("\nFOLLOW sets:\n");
  for (i = 0; i < prod_count; i++) {
    char nonTerminal = productions[i][0];
    if (!follow_computed[nonTerminal - 'A']) {
      findFollow(follow[nonTerminal - 'A'], nonTerminal);
      follow_computed[nonTerminal - 'A'] = 1;
    }
  }
  // Print FOLLOW sets
  for (i = 0; i < 26; i++) {
    if (follow_computed[i]) {
      printf("FOLLOW(%c) = { ", (char)(i + 'A'));
      for (int k = 0; follow[i][k] != '\0'; k++) {
        if (follow[i][k] == 'i')
          printf("id");
        else if (follow[i][k] == '#')
          printf("e");
        else
          printf("%c", follow[i][k]);
        if (follow[i][k + 1] != '\0') printf(", ");
      }
      printf(" }\n");
    }
  }
  return 0;
}
void findFirst(char *result, char c) {
  if (!isNonTerminal(c)) {
    addToResult(result, c);
    return;
  }
  for (int i = 0; i < prod_count; i++) {
    if (productions[i][0] == c) {
      int rhs_index = 3;
      while (productions[i][rhs_index] != '\0') {
        char next = productions[i][rhs_index];
```

```
            if (next == '#') { // epsilon
                addToResult(result, '#');
                break;
            } else if (!isNonTerminal(next)) {
                addToResult(result, next);
                break;
            } else {
                char temp[MAX] = "";
                findFirst(temp, next);
                int containsEpsilon = 0;
                for (int k = 0; temp[k] != '\0'; k++) {
                    if (temp[k] != '#') {
                        addToResult(result, temp[k]);
                    } else {
                        containsEpsilon = 1;
                    }
                }
                if (!containsEpsilon) {
                    break;
                } else {
                    rhs_index++;
                    if (productions[i][rhs_index] == '\0') {
                        addToResult(result, '#');
                    }
                }
            }
        }
    }
}
void findFollow(char *result, char c) {
    if (follow_visited[c - 'A']) return; // Avoid infinite recursion
    follow_visited[c - 'A'] = 1;
    if (productions[0][0] == c) {
        addToResult(result, '$'); // Add $ to start symbol
    }
    for (int i = 0; i < prod_count; i++) {
        int len = strlen(productions[i]);
        for (int j = 3; j < len; j++) {
            if (productions[i][j] == c) {
                int nextIndex = j + 1;
                int addedFollowFromNext = 0;
```

```
            while (nextIndex < len) {
                char next = productions[i][nextIndex];
                if (!isNonTerminal(next)) {
                    addToResult(result, next);
                    addedFollowFromNext = 1;
                    break;
                } else {
                    char temp[MAX] = "";
                    findFirst(temp, next);
                    int containsEpsilon = 0;
                    for (int k = 0; temp[k] != '\0'; k++) {
                        if (temp[k] != '#') {
                            addToResult(result, temp[k]);
                        } else {
                            containsEpsilon = 1;
                        }
                    }
                    if (containsEpsilon) {
                        nextIndex++;
                    } else {
                        addedFollowFromNext = 1;
                        break;
                    }
                }
            }
            if (!addedFollowFromNext && productions[i][0] != c) {
                findFollow(result, productions[i][0]);
            }
        }
    }
    follow_visited[c - 'A'] = 0;
}
int isNonTerminal(char c) {
    return (c >= 'A' && c <= 'Z');
}
void addToResult(char *result, char c) {
    if (c == '\0') return;
    if (!contains(result, c)) {
        int len = strlen(result);
        result[len] = c;
        result[len + 1] = '\0';
    }
}
```

```
int contains(char *str, char c) {
    for (int i = 0; str[i] != '\0'; i++) {
        if (str[i] == c) return 1;
    }
    return 0;
}
```

Convert BNF rules to YACC:

```
import ply.lex as lex
import ply.yacc as yacc
from graphviz import Digraph

# ---------------- Lexer ----------------
tokens = ('NUMBER',)
literals = ['+', '-', '*', '/', '(', ')']
t_ignore = ' \t\n'


def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t


def t_error(t):
    print(f"Illegal character '{t.value[0]}'")
    t.lexer.skip(1)


lexer = lex.lex()

# ---------------- AST Nodes ----------------
class ASTNode:
    pass


class BinOp(ASTNode):
    def __init__(self, left, op, right):
        self.left = left
        self.op = op
        self.right = right

    def __repr__(self):
        return f"({self.left} {self.op} {self.right})"


class Num(ASTNode):
    def __init__(self, value):
        self.value = value
```

```
    def __repr__(self):
        return str(self.value)


# ---------------- Convert BNF to YACC rules ----------------
def bnf_to_yacc(bnf_grammar):
    rules = {}
    for line in bnf_grammar.splitlines():
        if '::=' not in line:
            continue
        left, right = line.split('::=')
        rules[left.strip()] = right.strip()
    return rules


# ---------------- Parser ----------------
def generate_parser(yacc_rules):
    def p_expr(p):
        '''expr : expr '+' term
                | expr '-' term
                | term'''
        if len(p) == 4:
            p[0] = BinOp(p[1], p[2], p[3])
        else:
            p[0] = p[1]

    def p_term(p):
        '''term : term '*' factor
                | term '/' factor
                | factor'''
        if len(p) == 4:
            p[0] = BinOp(p[1], p[2], p[3])
        else:
            p[0] = p[1]

    def p_factor(p):
        '''factor : '(' expr ')'
                  | NUMBER'''
        if len(p) == 4:
            p[0] = p[2]
        else:
            p[0] = Num(p[1])

    def p_error(p):
```

```
        print("Syntax error!")

    return yacc.yacc()


# ---------------- AST to GraphViz ----------------
def ast_to_graph(node, graph=None, counter=[0]):
    if graph is None:
        graph = Digraph()
        graph.attr('node', shape='circle')

    counter[0] += 1
    node_id = str(counter[0])

    if isinstance(node, BinOp):
        graph.node(node_id, node.op)
        left_id = ast_to_graph(node.left, graph, counter)
        right_id = ast_to_graph(node.right, graph, counter)
        graph.edge(node_id, left_id)
        graph.edge(node_id, right_id)

    elif isinstance(node, Num):
        graph.node(node_id, str(node.value))

    else:
        graph.node(node_id, "?")

    return node_id


# ---------------- Main Program ----------------
def main():
    print("Enter your BNF grammar (end with an empty line):")
    lines = []
    while True:
        line = input()
        if line.strip() == '':
            break
        lines.append(line)

    bnf_grammar = '\n'.join(lines)

    # Step 1: Convert BNF to YACC dictionary
    yacc_rules = bnf_to_yacc(bnf_grammar)
```

```
    print("\nConverted YACC rules:")
    for k, v in yacc_rules.items():
        print(f"{k} -> {v}")

    # Step 2: Build parser
    parser = generate_parser(yacc_rules)

    # Step 3: Interactive expression input
    print("\nEnter arithmetic expressions (type 'exit' to quit):")
    while True:
        try:
            expr_input = input(">>> ")
            if expr_input.lower() == 'exit':
                break

            result = parser.parse(expr_input)
            print("AST:", result)

            # Step 4: Generate AST diagram
            dot = Digraph()
            ast_to_graph(result, dot)
            dot.render('ast_tree', format='png', cleanup=True)
            print("AST diagram saved as ast_tree.png")

        except Exception as e:
            print("Error:", e)


if __name__ == "__main__":
    main()
```

Three Address code for given code fragment:

```
import java.util.*;
import java.util.regex.*;

public class Main {

   // Tokenize the expression into identifiers, operators, and parentheses
   public static List<String> tokenize(String expr) {
      List<String> tokens = new ArrayList<>();
      Matcher matcher = Pattern.compile("[a-zA-Z_][\\w]*|\\d+|[()+\\-*/]").matcher(expr);
      while (matcher.find()) {
         tokens.add(matcher.group());
      }
      return tokens;
   }

   static final Map<String, Integer> precedence = new HashMap<>() {{
      put("+", 1);
      put("-", 1);
      put("*", 2);
      put("/", 2);
   }};

   // Convert infix to postfix (Reverse Polish Notation)
   public static List<String> infixToPostfix(List<String> tokens) {
      Stack<String> stack = new Stack<>();
      List<String> output = new ArrayList<>();

      for (String token : tokens) {
         if (token.matches("[a-zA-Z_][\\w]*|\\d+")) {
            output.add(token);
         } else if (token.equals("(")) {
            stack.push(token);
         } else if (token.equals(")")) {
            while (!stack.isEmpty() && !stack.peek().equals("(")) {
               output.add(stack.pop());
            }
            if (!stack.isEmpty()) stack.pop(); // Remove '('
         } else { // operator
            while (!stack.isEmpty() && !stack.peek().equals("(")
                  && precedence.get(stack.peek()) >= precedence.get(token)) {
               output.add(stack.pop());
            }
```

```
            stack.push(token);
        }
    }

    while (!stack.isEmpty()) {
        output.add(stack.pop());
    }

    return output;
}

// Generate Quadruples, Triples, and Indirect Triples
public static void generateTACStructures(List<String> postfix) {
    int tempCount = 1;
    Stack<String> stack = new Stack<>();
    List<String[]> quadruples = new ArrayList<>();
    List<String[]> triples = new ArrayList<>();

    for (String token : postfix) {
        if (token.matches("[a-zA-Z_][\\w]*|\\d+")) {
            stack.push(token);
        } else {
            String op2 = stack.pop();
            String op1 = stack.pop();
            String tempVar = "t" + tempCount;

            // Quadruple entry: (op, arg1, arg2, result)
            quadruples.add(new String[]{token, op1, op2, tempVar});

            // Triple entry: (op, arg1, arg2)
            String arg1Triple = (op1.startsWith("t"))
                    ? "(" + (Integer.parseInt(op1.substring(1)) - 1) + ")"
                    : op1;
            String arg2Triple = (op2.startsWith("t"))
                    ? "(" + (Integer.parseInt(op2.substring(1)) - 1) + ")"
                    : op2;

            triples.add(new String[]{token, arg1Triple, arg2Triple});
            stack.push(tempVar);
            tempCount++;
        }
    }

    // Indirect triples (Pointer Table)
```

```
        List<Integer> indirectTriples = new ArrayList<>();
        for (int i = 0; i < triples.size(); i++) {
            indirectTriples.add(i);
        }

        printQuadruples(quadruples);
        printTriples(triples);
        printIndirectTriples(indirectTriples);
    }

    // Print Quadruples
    public static void printQuadruples(List<String[]> quadruples) {
        System.out.println("Quadruples:");
        System.out.printf("%-5s %-3s %-8s %-8s %-5s\n", "Index", "Op", "Arg1", "Arg2",
"Result");
        for (int i = 0; i < quadruples.size(); i++) {
            String[] quad = quadruples.get(i);
            System.out.printf("%-5d %-3s %-8s %-8s %-5s\n", i, quad[0], quad[1], quad[2],
quad[3]);
        }
    }

    // Print Triples
    public static void printTriples(List<String[]> triples) {
        System.out.println("\nTriples:");
        System.out.printf("%-5s %-3s %-8s %-8s\n", "Index", "Op", "Arg1", "Arg2");
        for (int i = 0; i < triples.size(); i++) {
            String[] triple = triples.get(i);
            System.out.printf("%-5d %-3s %-8s %-8s\n", i, triple[0], triple[1], triple[2]);
        }
    }

    // Print Indirect Triples
    public static void printIndirectTriples(List<Integer> indirectTriples) {
        System.out.println("\nIndirect Triples (Pointer Table):");
        System.out.printf("%-5s %-20s\n", "Index", "Pointer to Triple Index");
        for (int i = 0; i < indirectTriples.size(); i++) {
            System.out.printf("%-5d %-20d\n", i, indirectTriples.get(i));
        }
    }

    // Main conversion process
    public static void convertExpressionToTACStructures(String expression) {
        List<String> tokens = tokenize(expression);
```

```
        List<String> postfix = infixToPostfix(tokens);
        generateTACStructures(postfix);
    }

    // Main method
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter an arithmetic expression: ");
        String expr = scanner.nextLine();

        System.out.println("\nExpression: " + expr + "\n");
        convertExpressionToTACStructures(expr);
    }
}
```

Three Address Code to Assembly code:

```
#include <bits/stdc++.h>
using namespace std;

struct ThreeAC {
   string op, arg1, arg2, result;
};

string getRegister(string var, unordered_map<string, string>& regMap, int& regCount) {
   if (regMap.find(var) == regMap.end())
      regMap[var] = "R" + to_string(regCount++);
   return regMap[var];
}

int main() {
   int n;
   cout << "Enter number of 3-address code statements: ";
   cin >> n;
   cin.ignore();

   vector<ThreeAC> code(n);
   cout << "Enter each 3-address code statement (e.g., t1 = a + b):\n";
   for (int i = 0; i < n; ++i) {
      cout << "Statement " << i + 1 << ": ";
      string line;
      getline(cin, line);
      stringstream ss(line);
      string eq;
      ss >> code[i].result >> eq >> code[i].arg1 >> code[i].op >> code[i].arg2;
      if (code[i].op.empty()) code[i].op = "=";
   }

   cout << "\nGenerated Assembly Code:\n";

   unordered_map<string, string> regMap;
   int regCount = 0;

   for (auto& stmt : code) {
      string r1 = getRegister(stmt.arg1, regMap, regCount);
      string r2 = getRegister(stmt.arg2, regMap, regCount);
      string r3 = getRegister(stmt.result, regMap, regCount);

      if (stmt.op == "+")
```

```
            cout << "MOV " << r3 << ", " << r1 << "\nADD " << r3 << ", " << r2 << "\n";
        else if (stmt.op == "-")
            cout << "MOV " << r3 << ", " << r1 << "\nSUB " << r3 << ", " << r2 << "\n";
        else if (stmt.op == "*")
            cout << "MOV " << r3 << ", " << r1 << "\nMUL " << r3 << ", " << r2 << "\n";
        else if (stmt.op == "/")
            cout << "MOV " << r3 << ", " << r1 << "\nDIV " << r3 << ", " << r2 << "\n";
        else if (stmt.op == "=")
            cout << "MOV " << r3 << ", " << r1 << "\n";
    }

    return 0;
}
```