1. String Manipulation

```cpp
#include <iostream>
#include <cctype>   // For toupper
#include <cstring>  // For standard string operations

// Function to calculate the length of a string
int string_length(const char* str) {
   int length = 0;
   while (str[length] != '\0') {
       length++;
   }
   return length;
}

// Function to copy one string into another
void string_copy(char* dest, const char* src) {
   while (*src != '\0') {
       *dest = *src;
       dest++;
       src++;
   }
   *dest = '\0'; // Null-terminate the destination string
}

// Function to convert a string to uppercase
void string_uppercase(char* str) {
   while (*str != '\0') {
       *str = toupper(*str); // Convert character to uppercase
       str++;
   }
}

// Function to concatenate two strings
void string_concatenate(char* dest, const char* src) {
   while (*dest != '\0') {
       dest++; // Move to the end of the destination string
   }
   while (*src != '\0') {
       *dest = *src;
       dest++;
       src++;
   }
   *dest = '\0'; // Null-terminate the result
}
```

```cpp
int main() {
    char str1[100], str2[100], str3[100];

    std::cout << "Enter the first string: ";
    std::cin.getline(str1, 100);

    std::cout << "Enter the second string: ";
    std::cin.getline(str2, 100);

    // Length of the first string
    std::cout << "Length of '" << str1 << "': " << string_length(str1) << std::endl;

    // Copy first string to a new string
    string_copy(str3, str1);
    std::cout << "Copied string: " << str3 << std::endl;

    // Convert first string to uppercase
    string_uppercase(str1);
    std::cout << "Uppercase string: " << str1 << std::endl;

    // Concatenate the two strings
    string_concatenate(str1, str2);
    std::cout << "Concatenated string: " << str1 << std::endl;

    return 0;
}
```

2. Token Count:

```cpp
#include <iostream>
#include <string>
#include <regex>
#include <algorithm> // For remove()

using namespace std;

int main() {
    string expression;
    cout << "Enter an arithmetic expression: ";
    getline(cin, expression);

    // Define regex for variables, constants, and operators
    regex variable_regex("[a-zA-Z]+");
    regex constant_regex("\\d+");
```

```
    regex operator_regex("[+\\-*/%=]");

    int variable_count = 0;
    int constant_count = 0;
    int operator_count = 0;

    // Remove spaces from the expression
    expression.erase(remove(expression.begin(), expression.end(), ' '),
expression.end());

    // Count variables
    sregex_iterator it(expression.begin(), expression.end(), variable_regex), end;
    while (it != end) {
        ++variable_count;
        ++it;
    }

    // Count constants
    it = sregex_iterator(expression.begin(), expression.end(), constant_regex);
    while (it != end) {
        ++constant_count;
        ++it;
    }

    // Count operators
    it = sregex_iterator(expression.begin(), expression.end(), operator_regex);
    while (it != end) {
        ++operator_count;
        ++it;
    }

    // Output results
    cout << "Variable count: " << variable_count << endl;
    cout << "Constant count: " << constant_count << endl;
    cout << "Operator count: " << operator_count << endl;

    return 0;
}
```

3. Token Specification:

```
#include <iostream>
#include <string>
```

```
#include <regex>
#include <algorithm> // For remove()

using namespace std;

int main() {
   string expression;
   cout << "Enter an arithmetic expression: ";
   getline(cin, expression);

   // Regular expression to match numbers and operators
   regex token_regex("(\\d+|[+\\-*/%=])");

   // Remove spaces from the expression
   expression.erase(remove(expression.begin(), expression.end(), ' '),
expression.end());

   // Tokenize the expression using regex
   sregex_iterator it(expression.begin(), expression.end(), token_regex), end;

   for (; it != end; ++it) {
       string token = it->str();
       if (isdigit(token[0])) {
           cout << "NUMBER: " << token << endl;
       } else if (token == "=") {
           cout << "EQUALS: " << token << endl;
       } else {
           cout << "OPERATOR: " << token << endl;
       }
   }

   return 0;
}
```

4. Lexical Analyzer

```
#include <iostream>
#include <cctype>
#include <cstring>

#define MAX_SYMBOLS 100
#define MAX_LENGTH 100

// Structure to represent a lexical token
```

```
struct LexicalToken {
   char lexeme[MAX_LENGTH];
   char tokenType[MAX_LENGTH];
};

// Function to add a token to the token array
void addToken(LexicalToken tokens[], int &tokenCount, const char *lexeme, const
char *tokenType) {
   strcpy(tokens[tokenCount].lexeme, lexeme);
   strcpy(tokens[tokenCount].tokenType, tokenType);
   tokenCount++;
}

// Function to analyze the input and tokenize it
void analyze(const char *input, LexicalToken tokens[], int &tokenCount) {
   int i = 0;
   while (input[i] != '\0') {
       if (isalpha(input[i])) { // Handle identifiers
           int start = i;
           while (isalnum(input[i])) i++; // Continue until alphanumeric ends
           char lexeme[MAX_LENGTH];
           strncpy(lexeme, &input[start], i - start);
           lexeme[i - start] = '\0';
           addToken(tokens, tokenCount, lexeme, "Identifier");
       } else if (isdigit(input[i])) { // Handle integer literals
           int start = i;
           while (isdigit(input[i])) i++; // Continue until non-digit character
           char lexeme[MAX_LENGTH];
           strncpy(lexeme, &input[start], i - start);
           lexeme[i - start] = '\0';
           addToken(tokens, tokenCount, lexeme, "Integer Literal");
       } else if (input[i] == '+' || input[i] == '-' || input[i] == '*' || input[i]
== '/' || input[i] == '=') { // Handle operators
           char lexeme[2] = {input[i], '\0'};
           const char *tokenType = (input[i] == '=') ? "Assignment Operator" :
"Operator";
           addToken(tokens, tokenCount, lexeme, tokenType);
           i++;
       } else if (input[i] == ';') { // Handle end of statement
           addToken(tokens, tokenCount, ";", "End of Statement");
           i++;
       } else { // Skip unrecognized characters or spaces
           i++;
       }
   }
```

```
}

int main() {
    char input[MAX_LENGTH];
    LexicalToken tokens[MAX_SYMBOLS];
    int tokenCount = 0;

    std::cout << "Enter a C++ expression (terminate with $): ";
    std::cin.getline(input, MAX_LENGTH);

    // Remove the terminating '$' character
    input[strcspn(input, "$")] = '\0';

    // Perform lexical analysis
    analyze(input, tokens, tokenCount);

    // Display the result of the lexical analysis
    std::cout << "\n--- Lexical Analysis Result ---\n";
    std::cout << "Lexeme\t\tToken Type\n";
    std::cout << "------------------------------\n";
    for (int i = 0; i < tokenCount; i++) {
        std::cout << tokens[i].lexeme << "\t\t" << tokens[i].tokenType << std::endl;
    }

    return 0;
}
```

5. Stack Dynamic

```
#include <iostream>
using namespace std;

class CustomStack {
private:
    int* data;          // Array to hold stack elements
    int capacity;       // Maximum size of the stack
    int topIndex;       // Index of the top element in the stack

public:
    // Constructor to initialize the stack with a given size, default size is 8
    CustomStack(int size = 8) {
        capacity = size;
        data = new int[capacity];
        topIndex = -1;   // Initially, stack is empty
    }
```

```cpp
    // Destructor to clean up memory
    ~CustomStack() {
        delete[] data;
    }

    // Push an element onto the stack
    void push(int value) {
        if (isFull()) {
            cout << "Stack is full, can't push " << value << "\n";
            return;
        }
        data[++topIndex] = value;
        cout << value << " added to the stack.\n";
    }

    // Pop the top element from the stack
    int pop() {
        if (isEmpty()) {
            cout << "Stack is empty, can't pop.\n";
            return -1;
        }
        cout << data[topIndex] << " removed from the stack.\n";
        return data[topIndex--];
    }

    // Check if the stack is empty
    bool isEmpty() {
        return topIndex == -1;
    }

    // Check if the stack is full
    bool isFull() {
        return topIndex == capacity - 1;
    }

    // Return the current number of elements in the stack
    int currentSize() {
        return topIndex + 1;
    }

    // Return the top element without removing it
    int peek() {
        if (!isEmpty()) {
            return data[topIndex];
```

```
        } else {
            cout << "No elements in the stack.\n";
            return -1;
        }
    }
};

int main() {
    CustomStack stack(6);

    // Push elements onto the stack
    stack.push(15);
    stack.push(25);
    stack.push(35);
    stack.push(45);
    stack.push(55);
    stack.push(65);  // Stack is full now

    // Show the current top item and total elements
    cout << "Top item: " << stack.peek() << "\n";
    cout << "Total elements: " << stack.currentSize() << "\n";

    // Pop two items from the stack
    stack.pop();
    stack.pop();

    // Show the new top item and total elements after removals
    cout << "Top item after removal: " << stack.peek() << "\n";
    cout << "Total elements after removal: " << stack.currentSize() << "\n";

    // Push another element to the stack
    stack.push(75);

    return 0;
}
```

6. Symbol table

```
#include <iostream>
#include <cctype>
#include <cstring>

#define MAX_SYMBOLS 15
#define MAX_LENGTH 100
```

```cpp
// Structure to represent a symbol in the symbol table
struct Symbol {
   char symbol[MAX_LENGTH];
   char type[12];
};

int main() {
   int i = 0, j = 0, x = 0, n;
   char b[MAX_LENGTH];
   Symbol symbols[MAX_SYMBOLS];

   std::cout << "Expression terminated by $: ";

   // Input the expression until '$' or max length is reached
   while ((b[i] = getchar()) != '$' && i < MAX_LENGTH - 1) {
       i++;
   }
   b[i] = '\0'; // Null-terminate the input string
   n = i;       // Length of the input string

   std::cout << "Given Expression: " << b << std::endl;
   std::cout << "\nSymbol Table\n";
   std::cout << "Symbol \t Type\n";

   // Parse the input string
   while (j < n) {
       char c = b[j];

       // Check for identifiers (alphabetic characters)
       if (isalpha(c)) {
           int start = j;
           while (j < n && isalpha(b[j])) {
               j++;
           }
           int length = j - start;

           // Copy the identifier into the symbol table
           strncpy(symbols[x].symbol, &b[start], length);
           symbols[x].symbol[length] = '\0';
           strcpy(symbols[x].type, "identifier");
           x++;
       }
       // Check for operators
       else if (c == '+' || c == '-' || c == '*' || c == '=') {
           symbols[x].symbol[0] = c;
```

```
            symbols[x].symbol[1] = '\0';
            strcpy(symbols[x].type, "operator");
            x++;
            j++;
        }
        // Skip other characters
        else {
            j++;
        }
    }

    // Print the symbol table
    for (i = 0; i < x; i++) {
        std::cout << symbols[i].symbol << " \t " << symbols[i].type << std::endl;
    }

    return 0;
}
```

7. Lexical analyzer using lex:

Lexer.l:

```
%{
#include <iostream>
#include <fstream>
#include <cstdio>
#include <cstdlib>
using namespace std;

int inCommentBlock = 0;
%}

identifier  [a-zA-Z][a-zA-Z0-9]*

%%
"#.*" {
   cout << "\n" << yytext << " is a Preprocessor Directive";
}

int|float|main|if|else|printf|scanf|for|char|getch|while {
   cout << "\n" << yytext << " is a Keyword";
}

"/*" {
   inCommentBlock = 1;
```

```
}

"*/" {
    inCommentBlock = 0;
}

{identifier}\( {
    if (!inCommentBlock)
        cout << "\nFunction:\t" << yytext;
}

"{" {
    if (!inCommentBlock)
        cout << "\nBlock Begins";
}

"}" {
    if (!inCommentBlock)
        cout << "\nBlock Ends";
}

{identifier}(\[[0-9]*\])? {
    if (!inCommentBlock)
        cout << "\n" << yytext << " is an Identifier";
}

\".*\" {
    if (!inCommentBlock)
        cout << "\n" << yytext << " is a String";
}

[0-9]+ {
    if (!inCommentBlock)
        cout << "\n" << yytext << " is a Number";
}

\)(\;)? {
    if (!inCommentBlock) {
        cout << "\t" << yytext << "\n";
    }
}

\( {
    cout << yytext;
}

"=" {
    if (!inCommentBlock)
```

```
        cout << "\n" << yytext << " is an Assignment Operator";
}

"<="|">="|"<"|"=="|">" {
    if (!inCommentBlock)
        cout << "\n" << yytext << " is a Relational Operator";
}

.|\n {
    // Do nothing for unrecognized characters
}
%%

int main(int argc, char **argv) {
    if (argc > 1) {
        ifstream file(argv[1]);
        if (!file.is_open()) {
            cerr << "\nCould not open the file: " << argv[1] << endl;
            exit(1);
        }
        yyin = fopen(argv[1], "r");
    }

    yylex();
    cout << "\n\n";
    return 0;
}

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

Cli code:

```
lex lexer.l
g++ lex.yy.c -o lexer
./lexer sample.c
```

8. First and Follow

```
productions = {}
firsts = {}
```

```python
follows = {}

def find_first(symbol):
    # If symbol is a terminal, return it as the first set
    if not symbol.isupper():
        return {symbol}

    # If already computed, return cached result
    if firsts[symbol]:
        return firsts[symbol]

    first = set()

    # Iterate through each production for the given non-terminal symbol
    for production in productions[symbol]:
        if production == 'ε':
            first.add('ε')
        else:
            for char in production:
                first_of_char = find_first(char)
                first.update(first_of_char - {'ε'})
                if 'ε' not in first_of_char:
                    break
            else:
                first.add('ε')

    firsts[symbol] = first
    return first

def find_follow(symbol):
    # If follow is already computed, return it
    if follows[symbol]:
        return follows[symbol]

    follow = set()

    # Add $ to the follow of the start symbol
    if symbol == start_symbol:
        follow.add('$')

    # Iterate through the productions to compute the follow set
    for lhs, rhs_list in productions.items():
        for rhs in rhs_list:
            for i, char in enumerate(rhs):
                if char == symbol:
```

```
                    if i + 1 < len(rhs):
                        follow_of_next = find_first(rhs[i + 1])
                        follow.update(follow_of_next - {'ε'})
                        if 'ε' in follow_of_next:
                            follow.update(find_follow(lhs))
                    else:
                        if lhs != symbol:
                            follow.update(find_follow(lhs))

    follows[symbol] = follow
    return follow

def compute_firsts_and_follows():
    # Compute firsts and follows for each non-terminal
    for non_terminal in productions:
        find_first(non_terminal)
        find_follow(non_terminal)

# Input the number of productions
num_productions = int(input("Enter the number of productions: "))
for _ in range(num_productions):
    lhs, rhs = input("Enter production (A=BC|d): ").split('=')
    productions[lhs] = rhs.split('|')
    firsts[lhs] = set()
    follows[lhs] = set()

# Set the start symbol (usually the first non-terminal)
start_symbol = list(productions.keys())[0]

# Compute the first and follow sets
compute_firsts_and_follows()

# Output the computed first and follow sets
for non_terminal in productions:
    print(f"First({non_terminal}) = {{ {', '.join(sorted(firsts[non_terminal]))}
}}")
    print(f"Follow({non_terminal}) = {{ {', '.join(sorted(follows[non_terminal]))}
}}")
```

9. NFA from Regex

```
def print_transition_table(q):
    print("\n\tTransition Table \n")
    print("_____________________________________________\n")
```

```
    print("Current State |\tInput |\tNext State")
    print("\n_____________________________________\n")
    for i in range(len(q)):
        if q[i][0] != 0:
            print(f" q[{i}]\t\t| a\t| q[{q[i][0]}]")
        if q[i][1] != 0:
            print(f" q[{i}]\t\t| b\t| q[{q[i][1]}]")
        if q[i][2] != 0:
            if q[i][2] < 10:
                print(f" q[{i}]\t\t| e\t| q[{q[i][2]}]")
            else:
                print(f" q[{i}]\t\t| e\t| q[{q[i][2] // 10}] , q[{q[i][2] % 10}]")
    print("\n_____________________________________\n")

def convert_regex_to_dfa(regex):
    q = [[0, 0, 0] for _ in range(20)]
    j = 0
    i = 0
    len_reg = len(regex)

    while i < len_reg:
        if regex[i] == 'a' and (i + 1 >= len_reg or regex[i + 1] not in ['|', '*']):
            q[j][0] = j + 1
            j += 1
        elif regex[i] == 'b' and (i + 1 >= len_reg or regex[i + 1] not in ['|',
'*']):
            q[j][1] = j + 1
            j += 1
        elif regex[i] == 'e' and (i + 1 >= len_reg or regex[i + 1] not in ['|',
'*']):
            q[j][2] = j + 1
            j += 1
        elif regex[i] == 'a' and i + 2 < len_reg and regex[i + 1] == '|' and regex[i
+ 2] == 'b':
            q[j][2] = ((j + 1) * 10) + (j + 3)
            j += 1
            q[j][0] = j + 1
            j += 1
            q[j][2] = j + 3
            j += 1
            q[j][1] = j + 1
            j += 1
            q[j][2] = j + 1
            j += 1
            i += 2
```

```
        elif regex[i] == 'b' and i + 2 < len_reg and regex[i + 1] == '|' and regex[i
+ 2] == 'a':
            q[j][2] = ((j + 1) * 10) + (j + 3)
            j += 1
            q[j][1] = j + 1
            j += 1
            q[j][2] = j + 3
            j += 1
            q[j][0] = j + 1
            j += 1
            q[j][2] = j + 1
            j += 1
            i += 2
        elif regex[i] == 'a' and i + 1 < len_reg and regex[i + 1] == '*':
            q[j][2] = ((j + 1) * 10) + (j + 3)
            j += 1
            q[j][0] = j + 1
            j += 1
            q[j][2] = ((j + 1) * 10) + (j - 1)
            j += 1
        elif regex[i] == 'b' and i + 1 < len_reg and regex[i + 1] == '*':
            q[j][2] = ((j + 1) * 10) + (j + 3)
            j += 1
            q[j][1] = j + 1
            j += 1
            q[j][2] = ((j + 1) * 10) + (j - 1)
            j += 1
        elif regex[i] == ')' and i + 1 < len_reg and regex[i + 1] == '*':
            q[0][2] = ((j + 1) * 10) + 1
            q[j][2] = ((j + 1) * 10) + 1
            j += 1
        i += 1

    print(f"Given regular expression: {regex}")
    print_transition_table(q)

# Example usage
regex = "(a|b)*"
convert_regex_to_dfa(regex)
```

10. Predictive parser

```
import copy

def removeLeftRecursion(rulesDiction):
```

```python
    store = {}
    for lhs in rulesDiction:
        alphaRules = []
        betaRules = []
        allrhs = rulesDiction[lhs]

        for subrhs in allrhs:
            if subrhs[0] == lhs:
                alphaRules.append(subrhs[1:])
            else:
                betaRules.append(subrhs)

        if len(alphaRules) != 0:
            lhs_ = lhs + "'"
            while (lhs_ in rulesDiction.keys()) or (lhs_ in store.keys()):
                lhs_ += "'"

            for b in range(len(betaRules)):
                betaRules[b].append(lhs_)

            rulesDiction[lhs] = betaRules

            for a in range(len(alphaRules)):
                alphaRules[a].append(lhs_)
            alphaRules.append(['0'])
            store[lhs_] = alphaRules

    for left in store:
        rulesDiction[left] = store[left]
    return rulesDiction


def LeftFactoring(rulesDiction):
    newDict = {}
    for lhs in rulesDiction:
        allrhs = rulesDiction[lhs]
        temp = dict()

        for subrhs in allrhs:
            if subrhs[0] not in list(temp.keys()):
                temp[subrhs[0]] = [subrhs]
            else:
                temp[subrhs[0]].append(subrhs)

        new_rule = []
```

```
        tempo_dict = {}

        for term_key in temp:
            allStartingWithTermKey = temp[term_key]

            if len(allStartingWithTermKey) > 1:
                lhs_ = lhs + "'"
                while (lhs_ in rulesDiction.keys()) or (lhs_ in tempo_dict.keys()):
                    lhs_ += "'"

                new_rule.append([term_key, lhs_])
                ex_rules = []
                for g in temp[term_key]:
                    ex_rules.append(g[1:])
                tempo_dict[lhs_] = ex_rules
            else:
                new_rule.append(allStartingWithTermKey[0])

        newDict[lhs] = new_rule
        for key in tempo_dict:
            newDict[key] = tempo_dict[key]

    return newDict


def first(rule):
    global diction, firsts
    if len(rule) != 0 and rule is not None:
        if rule[0] in term_userdef:
            return [rule[0]]
        elif rule[0] == '0':
            return ['0']
        if len(rule) != 0:
            if rule[0] in list(diction.keys()):
                fres = []
                rhs_rules = diction[rule[0]]
                for itr in rhs_rules:
                    indivRes = first(itr)
                    if '0' in indivRes:
                        indivRes.remove('0')
                        if len(rule) > 1:
                            ansNew = first(rule[1:])
                            if ansNew is not None:
                                if type(ansNew) is list:
                                    indivRes += ansNew
```

```
                    else:
                        indivRes.append(ansNew)
                if type(indivRes) is list:
                    fres.extend(indivRes)
                else:
                    fres.append(indivRes)
            return fres


def follow(nt):
   global start_symbol, diction, firsts, follows
   solset = set()

   if nt == start_symbol:
       solset.add('$')

   for curNT in diction:
       rhs = diction[curNT]
       for subrule in rhs:
           if nt in subrule:
               index_nt = subrule.index(nt)
               subrule = subrule[index_nt + 1:]

               if len(subrule) != 0:
                   res = first(subrule)
                   if '0' in res:
                       res.remove('0')
                   if len(subrule) > 1:
                       res += first(subrule[1:])
                   res.append('0')
                   solset.update(res)
               else:
                   if nt != curNT:
                       res = follow(curNT)
                       if res is not None:
                           solset.update(res)
   return list(solset)


def computeAllFirsts():
   global diction, firsts
   for rule in rules:
       k = rule.split("->")
       k[0] = k[0].strip()
       k[1] = k[1].strip()
```

```
        rhs = k[1]
        multirhs = rhs.split('|')

        for i in range(len(multirhs)):
            multirhs[i] = multirhs[i].strip()
            multirhs[i] = multirhs[i].split()

        diction[k[0]] = multirhs

    diction = removeLeftRecursion(diction)
    diction = LeftFactoring(diction)

    for y in list(diction.keys()):
        t = set()
        for sub in diction.get(y):
            res = first(sub)
            if res != None:
                if type(res) is list:
                    t.update(res)
                else:
                    t.add(res)
        firsts[y] = t


def computeAllFollows():
    global start_symbol, diction, firsts, follows
    for NT in diction:
        solset = set()
        sol = follow(NT)
        if sol is not None:
            solset.update(sol)
        follows[NT] = solset


def createParseTable():
    import copy
    global diction, firsts, follows, term_userdef
    mx_len_first = 0
    mx_len_fol = 0

    for u in diction:
        k1 = len(str(firsts[u]))
        k2 = len(str(follows[u]))

        if k1 > mx_len_first:
```

```
            mx_len_first = k1
        if k2 > mx_len_fol:
            mx_len_fol = k2

    print(f"{{:<{10}}} {{:<{mx_len_first + 5}}} {{:<{mx_len_fol +
5}}}".format("Non-T", "FIRST", "FOLLOW"))

    for u in diction:
        print(f"{{:<{10}}} {{:<{mx_len_first + 5}}} {{:<{mx_len_fol +
5}}}".format(u, str(firsts[u]), str(follows[u])))

    ntlist = list(diction.keys())
    terminals = copy.deepcopy(term_userdef)
    terminals.append('$')
    mat = []

    for x in diction:
        row = []
        for y in terminals:
            row.append('')
        mat.append(row)

    grammar_is_LL = True
    for lhs in diction:
        rhs = diction[lhs]
        for y in rhs:
            res = first(y)
            if '0' in res:
                if type(res) == str:
                    res = [res]
                res.remove('0')
                res += follows[lhs]

            if type(res) is str:
                res = [res]

            for c in res:
                if c in terminals:
                    xnt = ntlist.index(lhs)
                    yt = terminals.index(c)
                    if mat[xnt][yt] == '':
                        mat[xnt][yt] = f"{lhs} -> {' '.join(y)}"
                    else:
                        if f"{lhs} -> {' '.join(y)}" in mat[xnt][yt]:
                            continue
```

```
                        else:
                            grammar_is_LL = False
                            mat[xnt][yt] += f", {lhs} -> {' '.join(y)}"

   print("\nGenerated parsing table:\n")
   frmt = "{:>12}" * len(terminals)
   print(frmt.format(*terminals))

   j = 0
   for y in mat:
       frmt1 = "{:>12}" * len(y)
       print(f"{ntlist[j]} {frmt1.format(*y)}")
       j += 1

   return (mat, grammar_is_LL, terminals)


rules = [
   "E -> T E'",
   "E' -> + T E' | 0",
   "T -> F T'",
   "T' -> * F T' | 0",
   "F -> ( E ) | id"
]

term_userdef = ['id', '+', '*', '(', ')']
diction = {}
firsts = {}
follows = {}

computeAllFirsts()
start_symbol = list(diction.keys())[0]
computeAllFollows()

(parsing_table, result, tabTerm) = createParseTable()
```

11. Recursive descent parser:

```
SUCCESS = 1
FAILED = 0
cursor = 0
string = ""

def E():
   global cursor
```

```python
    print(f"{string[cursor:]} E -> T E'")
    if T():
        if Edash():
            return SUCCESS
    return FAILED

def Edash():
    global cursor
    if cursor < len(string) and string[cursor] == '+':
        print(f"{string[cursor:]} E' -> + T E'")
        cursor += 1
        if T():
            if Edash():
                return SUCCESS
    else:
        print(f"{string[cursor:]} E' -> ε")
        return SUCCESS
    return FAILED

def T():
    global cursor
    print(f"{string[cursor:]} T -> F T'")
    if F():
        if Tdash():
            return SUCCESS
    return FAILED

def Tdash():
    global cursor
    if cursor < len(string) and string[cursor] == '*':
        print(f"{string[cursor:]} T' -> * F T'")
        cursor += 1
        if F():
            if Tdash():
                return SUCCESS
    else:
        print(f"{string[cursor:]} T' -> ε")
        return SUCCESS
    return FAILED

def F():
    global cursor
    if cursor < len(string) and string[cursor] == '(':
        print(f"{string[cursor:]} F -> ( E )")
        cursor += 1
```

```
        if E():
            if cursor < len(string) and string[cursor] == ')':
                cursor += 1
                return SUCCESS
    elif cursor < len(string) and string[cursor] == 'i':
        print(f"{string[cursor:]} F -> i")
        cursor += 1
        return SUCCESS
    return FAILED

if __name__ == "__main__":
    string = input("Enter the string: ")
    cursor = 0
    print("\nInput\t\tAction")
    print("--------------------------------")
    if E() and cursor == len(string):
        print("--------------------------------")
        print("String is successfully parsed")
    else:
        print("--------------------------------")
        print("Error in parsing String")
```

12. LALR

```
MAX = 100
NUM_STATES = 5
NUM_SYMBOLS = 4

# Action Table
action = [
    [2, 3, -1, -1],
    [-1, -1, -1, 0],
    [2, 3, -1, -1],
    [-2, -2, -2, -2],
    [-1, -1, -1, -1],
]

# Goto Table
goto_table = [
    [1],
    [-1],
    [4],
    [-1],
    [-1],
```

```python
]

# Symbol Constants
C, D, B, DOLLAR = 0, 1, 2, 3

class Stack:
    def __init__(self):
        self.items = []

    def push(self, item):
        if len(self.items) < MAX:
            self.items.append(item)
        else:
            print("Stack overflow")

    def pop(self):
        if self.items:
            return self.items.pop()
        else:
            print("Stack underflow")
            return -1

    def peek(self):
        if self.items:
            return self.items[-1]
        else:
            return -1

    def display(self):
        print("Stack:", self.items)

def print_parsing_table():
    print("Parsing Action Table:")
    print("State | c | d | b | $")
    print("---------------------")
    for i in range(NUM_STATES):
        print(f"{i:2d}", end="")
        for j in range(NUM_SYMBOLS):
            if action[i][j] == -1:
                print("    . ", end="")
            elif action[i][j] < 0:
                print(f" R{-action[i][j]} ", end="")
            else:
                print(f" S{action[i][j]} ", end="")
        print()
```

```
    print("\nParsing Goto Table:")
    print("State | A")
    print("-------------")
    for i in range(NUM_STATES):
        print(f"{i:2d}", end="")
        if goto_table[i][0] == -1:
            print("   . ")
        else:
            print(f" {goto_table[i][0]:2d} ")

def LALR_parser(input_string):
    s = Stack()
    s.push(0)
    i = 0

    while i < len(input_string):
        # Map the input characters to symbol codes (C, D, B, DOLLAR)
        symbol = C if input_string[i] == 'c' else D if input_string[i] == 'd' else B
if input_string[i] == 'b' else DOLLAR
        state = s.peek()
        action_code = action[state][symbol]

        if action_code > 0:  # Shift operation
            print(f"Shift: {input_string[i]}")
            s.push(action_code)
            i += 1
        elif action_code < 0:  # Reduce operation
            prod = -action_code
            print("Reduce by ", end="")
            if prod == 1:
                print("S -> A b")
            elif prod == 2:
                print("A -> c A")
            elif prod == 3:
                print("A -> d")

            # Pop two elements (based on the production rules)
            s.pop()
            s.pop()
            # Push the goto state (if any)
            s.push(goto_table[s.peek()][0])
        elif action_code == 0:  # Accept input
            print("Input accepted.")
            return
```

```
        else:  # Error in parsing
            print(f"Error: Unexpected symbol {input_string[i]}")
            return

    s.display()
    if s.peek() == 0:
        print("Input accepted.")
    else:
        print("Input rejected.")

if __name__ == "__main__":
    print_parsing_table()
    input_string = input("Enter the input string (e.g., cd or cdcdb): ")
    input_string += "$"  # Append the dollar symbol
    LALR_parser(input_string)
```

13. SLR

```
import copy

def grammarAugmentation(rules, nonterm_userdef,
                        start_symbol):

    newRules = []

    newChar = start_symbol + "'"
    while (newChar in nonterm_userdef):
        newChar += "'"
    newRules.append([newChar,
                     ['.', start_symbol]])

    for rule in rules:

        k = rule.split("->")
        lhs = k[0].strip()
        rhs = k[1].strip()

        multirhs = rhs.split('|')
        for rhs1 in multirhs:
            rhs1 = rhs1.strip().split()

            rhs1.insert(0, '.')
            newRules.append([lhs, rhs1])
    return newRules
```

```python
def findClosure(input_state, dotSymbol):
   global start_symbol, \
       separatedRulesList, \
       statesDict

   closureSet = []
   if dotSymbol == start_symbol:
       for rule in separatedRulesList:
           if rule[0] == dotSymbol:
               closureSet.append(rule)
   else:
       closureSet = input_state

   prevLen = -1
   while prevLen != len(closureSet):
       prevLen = len(closureSet)

       tempClosureSet = []
       for rule in closureSet:
           indexOfDot = rule[1].index('.')
           if rule[1][-1] != '.':
               dotPointsHere = rule[1][indexOfDot + 1]
               for in_rule in separatedRulesList:
                   if dotPointsHere == in_rule[0] and \
                           in_rule not in tempClosureSet:
                       tempClosureSet.append(in_rule)

       for rule in tempClosureSet:
           if rule not in closureSet:
               closureSet.append(rule)
   return closureSet


def compute_GOTO(state):
   global statesDict, stateCount

   generateStatesFor = []
   for rule in statesDict[state]:

       if rule[1][-1] != '.':
           indexOfDot = rule[1].index('.')
           dotPointsHere = rule[1][indexOfDot + 1]
           if dotPointsHere not in generateStatesFor:
               generateStatesFor.append(dotPointsHere)
```

```python
    if len(generateStatesFor) != 0:
        for symbol in generateStatesFor:
            GOTO(state, symbol)
    return


def GOTO(state, charNextToDot):
    global statesDict, stateCount, stateMap

    newState = []
    for rule in statesDict[state]:
        indexOfDot = rule[1].index('.')
        if rule[1][-1] != '.':
            if rule[1][indexOfDot + 1] == \
                    charNextToDot:
                shiftedRule = copy.deepcopy(rule)
                shiftedRule[1][indexOfDot] = \
                    shiftedRule[1][indexOfDot + 1]
                shiftedRule[1][indexOfDot + 1] = '.'
                newState.append(shiftedRule)

    addClosureRules = []
    for rule in newState:
        indexDot = rule[1].index('.')
        if rule[1][-1] != '.':
            closureRes = \
                findClosure(newState, rule[1][indexDot + 1])
            for rule in closureRes:
                if rule not in addClosureRules \
                        and rule not in newState:
                    addClosureRules.append(rule)

    for rule in addClosureRules:
        newState.append(rule)

    stateExists = -1
    for state_num in statesDict:
        if statesDict[state_num] == newState:
            stateExists = state_num
            break

    if stateExists == -1:

        stateCount += 1
```

```python
            statesDict[stateCount] = newState
            stateMap[(state, charNextToDot)] = stateCount
    else:

            stateMap[(state, charNextToDot)] = stateExists
    return


def generateStates(statesDict):
    prev_len = -1
    called_GOTO_on = []

    while (len(statesDict) != prev_len):
        prev_len = len(statesDict)
        keys = list(statesDict.keys())

        for key in keys:
            if key not in called_GOTO_on:
                called_GOTO_on.append(key)
                compute_GOTO(key)
    return

def first(rule):
    global rules, nonterm_userdef, \
        term_userdef, diction, firsts

    if len(rule) != 0 and (rule is not None):
        if rule[0] in term_userdef:
            return rule[0]
        elif rule[0] == '#':
            return '#'

    if len(rule) != 0:
        if rule[0] in list(diction.keys()):

            fres = []
            rhs_rules = diction[rule[0]]

            for itr in rhs_rules:
                indivRes = first(itr)
                if type(indivRes) is list:
                    for i in indivRes:
                        fres.append(i)
                else:
                    fres.append(indivRes)
```

```
            if '#' not in fres:
                return fres
            else:

                newList = []
                fres.remove('#')
                if len(rule) > 1:
                    ansNew = first(rule[1:])
                    if ansNew != None:
                        if type(ansNew) is list:
                            newList = fres + ansNew
                        else:
                            newList = fres + [ansNew]
                    else:
                        newList = fres
                    return newList


                fres.append('#')
                return fres


def follow(nt):
    global start_symbol, rules, nonterm_userdef, \
        term_userdef, diction, firsts, follows

    solset = set()
    if nt == start_symbol:

        solset.add('$')

    for curNT in diction:
        rhs = diction[curNT]

        for subrule in rhs:
            if nt in subrule:

                while nt in subrule:
                    index_nt = subrule.index(nt)
                    subrule = subrule[index_nt + 1:]

                    if len(subrule) != 0:

                        res = first(subrule)
```

```
                        if '#' in res:
                            newList = []
                            res.remove('#')
                            ansNew = follow(curNT)
                            if ansNew != None:
                                if type(ansNew) is list:
                                    newList = res + ansNew
                                else:
                                    newList = res + [ansNew]
                            else:
                                newList = res
                            res = newList
                    else:
                        if nt != curNT:
                            res = follow(curNT)
                    if res is not None:
                        if type(res) is list:
                            for g in res:
                                solset.add(g)
                        else:
                            solset.add(res)
    return list(solset)


def createParseTable(statesDict, stateMap, T, NT):
    global separatedRulesList, diction
    rows = list(statesDict.keys())
    cols = T+['$']+NT
    Table = []
    tempRow = []
    for y in range(len(cols)):
        tempRow.append('')
    for x in range(len(rows)):
        Table.append(copy.deepcopy(tempRow))
    for entry in stateMap:
        state = entry[0]
        symbol = entry[1]
        a = rows.index(state)
        b = cols.index(symbol)
        if symbol in NT:
            Table[a][b] = Table[a][b]\
                + f"{stateMap[entry]} "
        elif symbol in T:
            Table[a][b] = Table[a][b]\
                + f"S{stateMap[entry]} "
```

```python
numbered = {}
key_count = 0
for rule in separatedRulesList:
    tempRule = copy.deepcopy(rule)
    tempRule[1].remove('.')
    numbered[key_count] = tempRule
    key_count += 1
addedR = f"{separatedRulesList[0][0]} -> " \
    f"{separatedRulesList[0][1][1]}"
rules.insert(0, addedR)
for rule in rules:
    k = rule.split("->")
    k[0] = k[0].strip()
    k[1] = k[1].strip()
    rhs = k[1]
    multirhs = rhs.split('|')
    for i in range(len(multirhs)):
        multirhs[i] = multirhs[i].strip()
        multirhs[i] = multirhs[i].split()
    diction[k[0]] = multirhs
for stateno in statesDict:
    for rule in statesDict[stateno]:
        if rule[1][-1] == '.':
            temp2 = copy.deepcopy(rule)
            temp2[1].remove('.')
            for key in numbered:
                if numbered[key] == temp2:
                    follow_result = follow(rule[0])
                    for col in follow_result:
                        index = cols.index(col)
                        if key == 0:
                            Table[stateno][index] = "Accept"
                        else:
                            Table[stateno][index] =\
                                Table[stateno][index]+f"R{key} "

print("\nSLR(1) parsing table:\n")
frmt = "{:>8}" * len(cols)
print(" ", frmt.format(*cols), "\n")
ptr = 0
j = 0
for y in Table:
    frmt1 = "{:>8}" * len(y)
    print(f"{{:>3}} {frmt1.format(*y)}"
        .format('I'+str(j)))
```

```
        j += 1

def printResult(rules):
    for rule in rules:
        print(f"{rule[0]} ->"
            f" {' '.join(rule[1])}")

def printAllGOTO(diction):
    for itr in diction:
        print(f"GOTO ( I{itr[0]} ,"
            f" {itr[1]} ) = I{stateMap[itr]}")
rules = ["E -> E + T | T",
        "T -> T * F | F",
        "F -> ( E ) | id"
        ]
nonterm_userdef = ['E', 'T', 'F']
term_userdef = ['id', '+', '*', '(', ')']
start_symbol = nonterm_userdef[0]
print("\nOriginal grammar input:\n")
for y in rules:
    print(y)
print("\nGrammar after Augmentation: \n")
separatedRulesList = \
    grammarAugmentation(rules,
                        nonterm_userdef,
                        start_symbol)
printResult(separatedRulesList)
start_symbol = separatedRulesList[0][0]
print("\nCalculated closure: I0\n")
I0 = findClosure(0, start_symbol)
printResult(I0)
statesDict = {}
stateMap = {}
statesDict[0] = I0
stateCount = 0
generateStates(statesDict)
print("\nStates Generated: \n")
for st in statesDict:
    print(f"State = I{st}")
    printResult(statesDict[st])
    print()

print("Result of GOTO computation:\n")
printAllGOTO(stateMap)
diction = {}
```

```
createParseTable(statesDict, stateMap,
               term_userdef,
               nonterm_userdef)
```

14. Shift reduce parser;

```
a = "a*a/b"
stk = []
act = "SHIFT"

def check():
   global stk, a
   ac = "REDUCE TO E -> "

   # Rule 1: a -> E
   if len(stk) >= 1 and stk[-1] == 'a':
       print(f"${''.join(stk[:-1])}a\t{a}$\t{ac}a")
       stk[-1] = 'E'
       print(f"${''.join(stk)}\t{a}$")

   # Rule 2: b -> E
   if len(stk) >= 1 and stk[-1] == 'b':
       print(f"${''.join(stk[:-1])}b\t{a}$\t{ac}b")
       stk[-1] = 'E'
       print(f"${''.join(stk)}\t{a}$")

   # Rule 3: E + E -> E
   i = 0
   while i < len(stk) - 2:
       if stk[i] == 'E' and stk[i + 1] == '+' and stk[i + 2] == 'E':
           print(f"${''.join(stk[:i])}E+E{''.join(stk[i+3:])}\t{a}$\t{ac}E+E")
           stk[i] = 'E'
           del stk[i + 1:i + 3]
           print(f"${''.join(stk)}\t{a}$")
           i = max(i - 2, 0)
       else:
           i += 1

   # Rule 4: E * E -> E
   i = 0
   while i < len(stk) - 2:
       if stk[i] == 'E' and stk[i + 1] == '*' and stk[i + 2] == 'E':
           print(f"${''.join(stk[:i])}E*E{''.join(stk[i+3:])}\t{a}$\t{ac}E*E")
           stk[i] = 'E'
           del stk[i + 1:i + 3]
```

```
            print(f"${''.join(stk)}\t{a}$")
            i = max(i - 2, 0)
        else:
            i += 1

    # Rule 5: E / E -> E
    i = 0
    while i < len(stk) - 2:
        if stk[i] == 'E' and stk[i + 1] == '/' and stk[i + 2] == 'E':
            print(f"${''.join(stk[:i])}E/E{''.join(stk[i+3:])}\t{a}$\t{ac}E/E")
            stk[i] = 'E'
            del stk[i + 1:i + 3]
            print(f"${''.join(stk)}\t{a}$")
            i = max(i - 2, 0)
        else:
            i += 1

def main():
    global stk, a, act
    print("stack\tinput\taction")

    # Initial state
    print(f"${''.join(stk)}\t{a}$\t{act}")

    # Shift operations (Push input to stack)
    for char in a:
        stk.append(char)
        a = a[1:]
        print(f"${''.join(stk)}\t{a}$\t{act}")
        check()  # Check for reductions after each shift

    # Check the final acceptance/rejection
    check()  # Final check after all shifts
    if len(stk) == 1 and stk[0] == 'E':
        print(f"${''.join(stk)}\t{a}$\tAccept")
    else:
        print(f"${''.join(stk)}\t{a}$\tReject")

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

15. DAG

```
#include <iostream>
```

```
#include <stack>
#include <map>
#include <vector>
#include <cctype>

using namespace std;

struct Node {
   int left, right;
   char label;
   char op;
};

vector<Node> dag;
map<char, int> node_map;

void printDAG() {
   cout << "PTR\tLEFT PTR\tRIGHT PTR\tLABEL\tOPERATOR\n";
   for (int i = 0; i < dag.size(); ++i) {
       cout << i << "\t" << dag[i].left << "\t\t" << dag[i].right << "\t\t"
            << dag[i].label << "\t" << dag[i].op << "\n";
   }
}

int main() {
   string expression;
   cout << "Enter the expression: ";
   cin >> expression;

   stack<int> operandStack;
   stack<char> operatorStack;

   for (char ch : expression) {
       if (isalnum(ch)) { // Operand (either number or variable)
           Node node = {-1, -1, ch, '\0'};
           int index = dag.size();
           dag.push_back(node);
           node_map[ch] = index;
           operandStack.push(index);
       }
       else if (ch == '(') { // Left parenthesis
           operatorStack.push(ch);
       }
       else if (ch == ')') { // Right parenthesis
           while (!operatorStack.empty() && operatorStack.top() != '(') {
```

```
            char op = operatorStack.top();
            operatorStack.pop();

            int right = operandStack.top();
            operandStack.pop();
            int left = operandStack.top();
            operandStack.pop();

            Node node = {left, right, '\0', op};
            int index = dag.size();
            dag.push_back(node);
            operandStack.push(index);
        }
        operatorStack.pop(); // Pop the '('
    }
    else { // Operator (like +, -, *, /)
        while (!operatorStack.empty() && operatorStack.top() != '(') {
            char op = operatorStack.top();
            operatorStack.pop();

            int right = operandStack.top();
            operandStack.pop();
            int left = operandStack.top();
            operandStack.pop();

            Node node = {left, right, '\0', op};
            int index = dag.size();
            dag.push_back(node);
            operandStack.push(index);
        }
        operatorStack.push(ch);
    }
}

// Final processing for remaining operators
while (!operatorStack.empty()) {
    char op = operatorStack.top();
    operatorStack.pop();

    int right = operandStack.top();
    operandStack.pop();
    int left = operandStack.top();
    operandStack.pop();

    Node node = {left, right, '\0', op};
```

```
        int index = dag.size();
        dag.push_back(node);
        operandStack.push(index);
    }

    printDAG();

    return 0;
}
```

16. Three address code to target code

```
#include <stdio.h>
#include <string.h>
//#include <conio.h>  // Uncomment this if using DOS-based systems with conio.h
void main() {
    char icode[10][30], str[20], opr[10];
    int i = 0;

    // Accepting intermediate code input from the user
    printf("\nEnter the set of intermediate code (terminated by exit):\n");
    do {
        scanf("%s", icode[i]);
    } while (strcmp(icode[i++], "exit") != 0); // Continue reading until 'exit' is
encountered

    // Start target code generation
    printf("\nTarget Code Generation");
    printf("\n***********************");

    // Reset i for target code generation
    i = 0;

    // Generate target code based on intermediate code
    do {
        strcpy(str, icode[i]);

        // Handle the operator in the intermediate code
        switch (str[3]) {
            case '+': strcpy(opr, "ADD "); break;
            case '-': strcpy(opr, "SUB "); break;
            case '*': strcpy(opr, "MUL "); break;
            case '/': strcpy(opr, "DIV "); break;
            default:
```

```
            strcpy(opr, "UNKNOWN ");
            break;
        }

        // Printing assembly instructions based on the intermediate code
        printf("\n\tMOV %c, R%d", str[2], i); // Move operand into a register
        printf("\n\t%s%c, R%d", opr, str[4], i); // Perform the operation
        printf("\n\tMOV R%d, %c", i, str[0]); // Move the result to the left

    } while (strcmp(icode[++i], "exit") != 0); // Loop until 'exit' is encountered

    // getch();  // Uncomment this if using DOS-based systems to wait for user input
}
```

17. Valid Variable using lex:

Valid.l:

```
%{
#include "y.tab.h"
%}
%%
[a-zA-Z] { return ALPHA; }
[0-9]+   { return NUMBER; }
"\n"     { return ENTER; }
.        { return ER; }
%%
int yywrap() { // Explicit return type
    return 1;
}
```

Valid.y:

```
%{
#include <stdio.h>
#include <stdlib.h>
%}
%token ALPHA NUMBER ENTER ER
%%
var: v ENTER {
    printf("Valid Variable\n");
    exit(0);
}
```

```
v: ALPHA exp1
exp1: ALPHA exp1
   | NUMBER exp1
   | /* empty */
%%
void yyerror(const char *s) { // Explicit return type and parameter
   printf("Invalid Variable: %s\n", s);
}

int main() {
   printf("Enter the expression: ");
   yyparse();
   return 0; // Ensure main returns an int
}
```

18. Recognize Arithmetic

Recog.l:

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

%%
[a-zA-Z]+    { yylval.var = strdup(yytext); return VARIABLE; }
[0-9]+       { yylval.num = atoi(yytext); return NUMBER; }
[\t]         ; // ignore tabs
[\n]         { return 0; } // newline (end of input)
.            { return yytext[0]; } // return any other character as is
%%

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

Recog.y:

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

```
void yyerror(const char *s);
int yylex(void);

%}

%union {
    int num;
    char *var;
}

%token <num> NUMBER
%token <var> VARIABLE

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

%%

S: VARIABLE '=' E {
    printf("\nEntered arithmetic expression is Valid\n\n");
}
;

E: E '+' E
  | E '-' E
  | E '*' E
  | E '/' E
  | E '%' E
  | '(' E ')'
  | NUMBER
  | VARIABLE
  ;

%%

int main() {
    printf("\nEnter Any Arithmetic Expression which can have operations Addition,
Subtraction, Multiplication, Division, Modulus and Round brackets:\n");
    yyparse();
    return 0;
}

void yyerror(const char *s) {
    printf("\nEntered arithmetic expression is Invalid: %s\n\n", s);
}
```

19. Calculator:

Calc.l:

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

%%
[0-9]+ {
                yylval=atoi(yytext);
                return NUMBER;

        }
[\t] ;

[\n] return 0;

. return yytext[0];

%%

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

Calc.y:

```
%{
#include <stdio.h>
#include <stdlib.h> // Required for atoi
int flag = 0;
%}

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


%%
ArithmeticExpression:
   E {
```

```
        printf("\nResult = %d\n", $1);
        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() {
    printf("Enter any arithmetic expression (Addition, Subtraction, Multiplication, Division, Modulus, and Round brackets are supported):\n");
    yyparse();
    if (flag == 0)
        printf("\nEntered arithmetic expression is Valid\n\n");
    return 0;
}

void yyerror(const char *msg) {
    printf("\nError: %s\n", msg);
    flag = 1;
}
```

20. Three address code generation:

Add3.l:

```
%{
#include "y.tab.h"
extern char yyval;
%}

%%

[0-9]+ {
    yylval.symbol = (char)(yytext[0]);
    return NUMBER;
}
```

```
[a-z] {
    yylval.symbol = (char)(yytext[0]);
    return LETTER;
}

. {
    return yytext[0];
}

\n {
    return 0;
}

%%
```

Add3.y:

```
%{
#include "y.tab.h"
#include <ctype.h>
#include <stdio.h>

char addtotable(char, char, char);

int index1 = 0;
char temp = 'A' - 1;

struct expr {
    char operand1;
    char operand2;
    char operator;
    char result;
};
%}

%union{
    char symbol;
}

%left '+' '-'
%left '/' '*'

%token <symbol> LETTER NUMBER
%type <symbol> exp
```

```
%%

statement: LETTER '=' exp ';' { addtotable((char)$1, (char)$3, '='); };

exp:
    exp '+' exp { $$ = addtotable((char)$1, (char)$3, '+'); }
    | exp '-' exp { $$ = addtotable((char)$1, (char)$3, '-'); }
    | exp '/' exp { $$ = addtotable((char)$1, (char)$3, '/'); }
    | exp '*' exp { $$ = addtotable((char)$1, (char)$3, '*'); }
    | '(' exp ')' { $$ = (char)$2; }
    | NUMBER { $$ = (char)$1; }
    | LETTER { $$ = (char)$1; };

%%

struct expr arr[20];

void yyerror(char *s) {
    printf("Errror %s", s);
}

char addtotable(char a, char b, char o) {
    temp++;
    arr[index1].operand1 = a;
    arr[index1].operand2 = b;
    arr[index1].operator = o;
    arr[index1].result = temp;
    index1++;
    return temp;
}

void threeAdd() {
    int i = 0;
    char temp = 'A';
    while (i < index1) {
        printf("%c :=\t", arr[i].result);
        printf("%c\t", arr[i].operand1);
        printf("%c\t", arr[i].operator);
        printf("%c\t", arr[i].operand2);
        i++;
        temp++;
        printf("\n");
    }
}

void fouradd() {
    int i = 0;
    char temp = 'A';
```

```
    while (i < index1) {
        printf("%c\t", arr[i].operator);
        printf("%c\t", arr[i].operand1);
        printf("%c\t", arr[i].operand2);
        printf("%c", arr[i].result);
        i++;
        temp++;
        printf("\n");
    }
}

int find(char l) {
    int i;
    for (i = 0; i < index1; i++)
        if (arr[i].result == l) break;
    return i;
}

void triple() {
    int i = 0;
    char temp = 'A';
    while (i < index1) {
        printf("%c\t", arr[i].operator);
        if (!isupper(arr[i].operand1))
            printf("%c\t", arr[i].operand1);
        else {
            printf("pointer");
            printf("%d\t", find(arr[i].operand1));
        }
        if (!isupper(arr[i].operand2))
            printf("%c\t", arr[i].operand2);
        else {
            printf("pointer");
            printf("%d\t", find(arr[i].operand2));
        }
        i++;
        temp++;
        printf("\n");
    }
}

int yywrap() {
    return 1;
}

int main() {
    printf("Enter the expression: ");
    yyparse();
```

```c
    threeAdd();
    printf("\n");
    fouradd();
    printf("\n");
    triple();
    return 0;
}
```

21. Infix to postfix

Intopo.l:

```
%{
#include <stdio.h>
#include "y.tab.h"
extern int yylval;
%}
op "+"|"-"|"*"|"/"
%%
[a-z] { yylval=*yytext; return id; }
{op} { return (int) yytext[0]; }
\n { return(0); }
. { return err; }
%%
```

Intopo.y:

```
%{
#include <stdio.h>
#include <ctype.h>
#define YYSTYPE char
int f=0;
%}
%token id err
%left '-' '+'
%left '*' '/'
%%
input: /* empty string */
     | input exp {}
     | error {f=1;}
   ;
exp: exp '+' exp { printf("+"); }
     | exp '-' exp { printf("-"); }
     | exp '*' exp { printf("*"); }
     | exp '/' exp { printf("/");}
```

```
    | id { printf("%c",yylval); }
  ;
%%
int main()
{
  printf("\nEnter an arithmetic expression:\n\n");
  yyparse();
  printf("\n");
  if(f==1)
    printf("Invalid Expression\n");
  return 0;
}
int yywrap()
{
  return 1;
}
int yyerror(char *mes) {
  return 0;
}
```

22. Code optimization:

```c
#include <stdio.h>

// Function to calculate factorial using a for loop
int factorial_for(int n) {
   int fact = 1;
   // Unused variable removed, it was not serving any purpose
   for (int i = 1; i <= n; i++) {
       fact *= i; // Multiply fact by i for each iteration
   }
   return fact;
}

// Function to calculate factorial using a do-while loop
int factorial_do_while(int n) {
   int fact = 1, i = 1;
   do {
       fact *= i; // Multiply fact by i for each iteration
       i++;         // Increment i
   } while (i <= n);
   return fact;
}

// Optimized approach to calculate factorial
int optimized_factorial(int n) {
```

```
    int fact = 1;
    for (int i = 1; i <= n; i++) {
        fact = fact * i; // Multiply fact by i for each iteration
    }
    return fact;
}

int main() {
    int n;

    // Ask the user to input a number
    printf("Enter a number to calculate its factorial: ");
    scanf("%d", &n);

    // Print the factorial calculated using different methods
    printf("Factorial using for loop: %d\n", factorial_for(n));
    printf("Factorial using do-while loop: %d\n", factorial_do_while(n));
    printf("Factorial using optimized approach: %d\n", optimized_factorial(n));

    return 0;
}
```