

**BTECH MTECH COMPUTER SCEIENCES AND ENGINEERING**

**(CYBERSECURITY)**

**COMPILER DESIGN PRACTICAL FILE**

**SUBMITTED TO:**

**Dr. Abinash Mishra**

**SUBMITTED BY:**

**VISHNULAL N**

**102CTBMCS2122025**

**SEMESTER -6**

**Table of contents**

|  |  |  |
| --- | --- | --- |
| **SL.no** | **Experiment** | **Sign** |
| **1** | **Make Case study on various types compiler such as C, C++, and Java Compiler** |  |
| **2** | **Design DFA and NFA using JFLAP for any given regular expression and regular language** |  |
| **3** | **Write a C++ program to find the number of tokens for any given arithmetic expression.** |  |
| **4** | **Write a program to design Lexical Analyzer in C++ (that recognize Keywords, Identifiers, Constants, Operators and Punctuation symbols)** |  |
| **5** | **Write a LEX program over the ∑ = {0,1,..,9} to recognize the following tokens**   * **Strings ending with 11** * **Strings containing 3 consecutive 2’s** * **10th symbol from right is 1** * **All four digit integer whose sum is 9** |  |
| **6** | **Write a Lex program to identify a floating-point number using file handling** |  |
| **7** | **Write a Lex program for the following:**   * **To identify keywords and convert to uppercase** * **To count no of vowels and consonants** * **To count no of identifiers and keywords and digits** |  |
| **8** | **Write a C++ program to find first and follow for any given CFG** |  |
| **9** | **Write a C++ program to design a LR(0) parser and SLR (1) parser and print the number of conflicts if any** |  |
| **10** | **Write a C++ program to design a CLR and LALR parser and print the parsing table** |  |

**1-Make Case study on various types compiler such as C, C++, and Java Compiler**

**Title**

Understanding the Functionality and Working of a C Compiler and C++ compiler

**Introduction**

C compilers play a crucial role in software development by translating human-readable C code into machine code that a computer can execute.

C++ compilers are essential tools in software development, extending the capabilities of C compilers to support object-oriented and generic programming features.

This case study explores the structure, working, and algorithms of a typical C & C++ compiler.

**How a C Compiler Works**

A C compiler transforms C / C++ source code into machine code through several stages, including lexical analysis, syntax analysis, semantic analysis, optimization, and code generation.

1. **Lexical Analysis**: The compiler reads the source code and converts it into tokens.
2. **Syntax Analysis**: The tokens are analyzed according to the grammar rules of C to form a syntax tree.
3. **Semantic Analysis**: The compiler checks the syntax tree for semantic errors.
4. **Optimization**: The compiler optimizes the code for better performance.
5. **Code Generation**: The optimized intermediate code is converted into machine code.
6. **Linking**: The machine code is linked with libraries to produce an executable.

**Flowchart of C /C++ Compiler Process**

Executable code

Code Generation

Optimization

Lexical Analysis

Semantic Analysis

Syntax Analysis

Source code ( C / C++)

Linking

**Algorithm of a C / C++ Compiler**

Input: Source code in C / C++

Output: Executable code

Steps:

1. Initialization
   * Read the source code from the input file.
2. Lexical Analysis
   * Convert the source code into a stream of tokens.
   * Each token represents a basic unit such as keywords, identifiers, operators, etc.
3. Syntax Analysis
   * Parse the stream of tokens to construct a syntax tree.
   * Ensure the tokens follow the grammatical structure of the C language.
4. Semantic Analysis
   * Traverse the syntax tree to check for semantic errors.
   * Ensure all identifiers are declared before use, types are consistent, etc.
   * Annotate the syntax tree with type information and other semantic details.
5. Optimization
   * Optimize the intermediate code generated from the annotated syntax tree.
   * Apply various optimization techniques to improve performance and reduce resource usage.
6. Code Generation
   * Generate machine code from the optimized intermediate code.
   * Translate high-level constructs into machine-level instructions.
7. Linking
   * Link the generated machine code with necessary libraries and runtime modules.
   * Resolve external references and generate the final executable.
8. Output
   * Write the executable code to an output file.

**Title**

Understanding the Functionality and Working of a Java compiler

**Introduction**

Java compilers are fundamental tools in software development, converting human-readable Java source code into bytecode that can be executed by the Java Virtual Machine (JVM). This case study explores the structure, working, and algorithms of a typical Java compiler.

**How a Java Compiler Works**

A Java compiler transforms Java source code into bytecode through several stages, including lexical analysis, syntax analysis, semantic analysis, optimization, and code generation.

1. **Lexical Analysis**: The compiler reads the source code and converts it into tokens.
2. **Syntax Analysis**: The tokens are analyzed according to the grammar rules of Java to form a syntax tree.
3. **Semantic Analysis**: The compiler checks the syntax tree for semantic errors.
4. **Optimization**: The compiler optimizes the code for better performance.
5. **Code Generation**: The optimized intermediate code is converted into bytecode.
6. **Linking**: The bytecode is linked with libraries to produce a complete class file

**Flowchart of JAVA Compiler Process**

Optimization

Lexical Analysis

Semantic Analysis

Syntax Analysis

Source code ( Java)

Code Generation

Linking

Bytecode ( Class )

**Algorithm of a Java Compiler**

**Input:** Source code in Java  
**Output:**  Bytecode (class file)

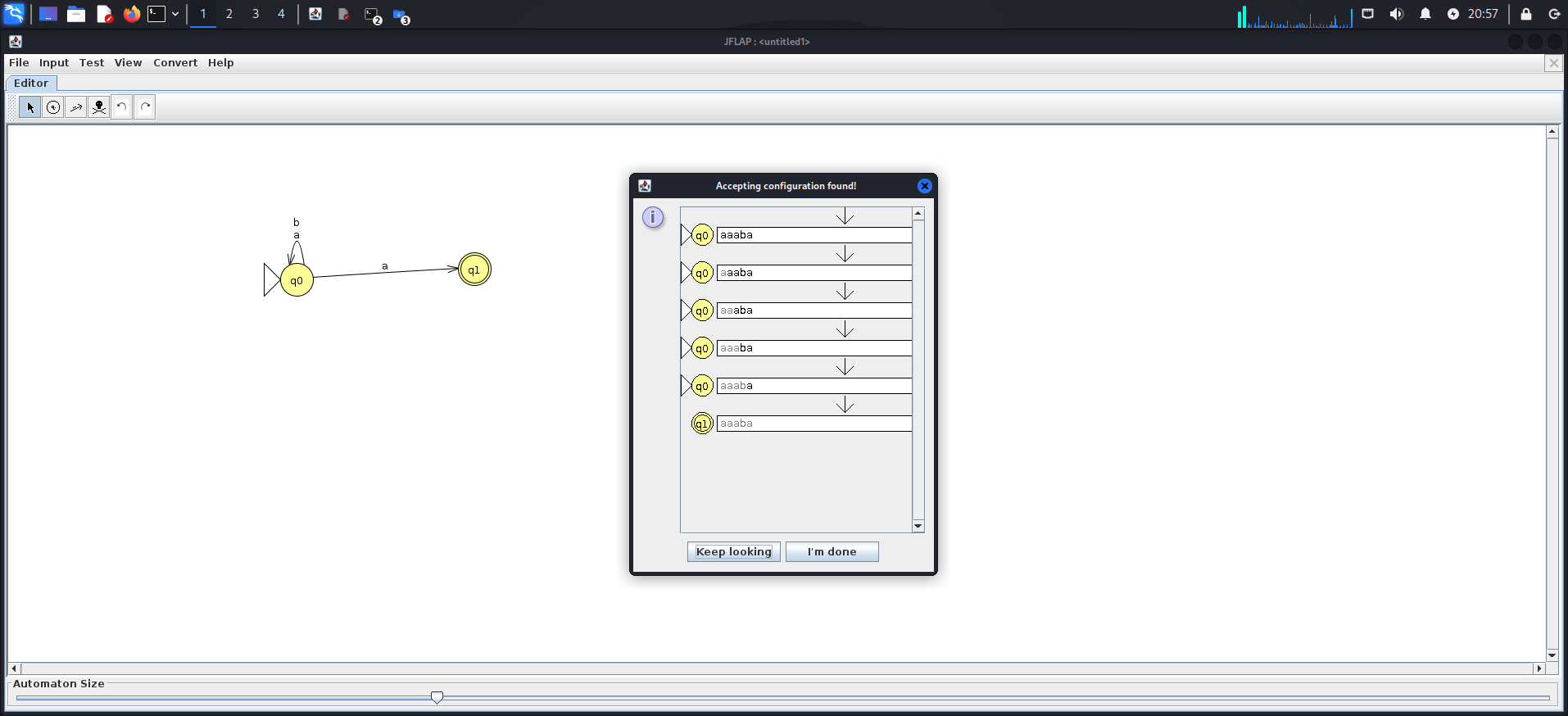
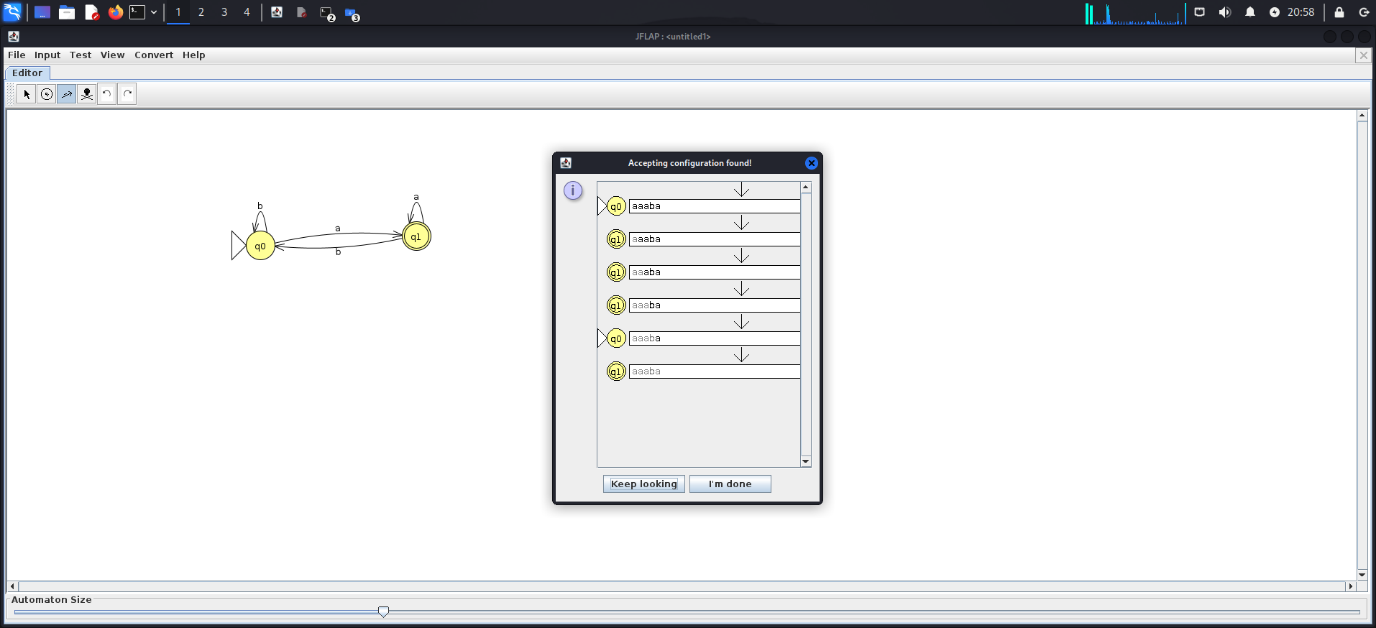
**Steps:**

1. **Initialization**
   * Read the source code from the input file.
2. **Lexical Analysis**
   * Convert the source code into a stream of tokens.
   * Each token represents a basic unit such as keywords, identifiers, operators, etc.
3. **Syntax Analysis**
   * Parse the stream of tokens to construct a syntax tree.
   * Ensure the tokens follow the grammatical structure of the Java language.
4. **Semantic Analysis**
   * Traverse the syntax tree to check for semantic errors.
   * Ensure all identifiers are declared before use, types are consistent, etc.
   * Annotate the syntax tree with type information and other semantic details.
5. **Optimization**
   * Optimize the intermediate code generated from the annotated syntax tree.
   * Apply various optimization techniques to improve performance and reduce resource usage.
6. **Code Generation**
   * Generate bytecode from the optimized intermediate code.
   * Translate high-level constructs into JVM instructions.
7. **Linking**
   * Link the generated bytecode with necessary libraries and runtime modules.
   * Resolve external references and generate the final class file.
8. **Output**
   * Write the bytecode to an output file.

**2.Design DFA and NFA using JFLAP for any given regular expression and regular language**

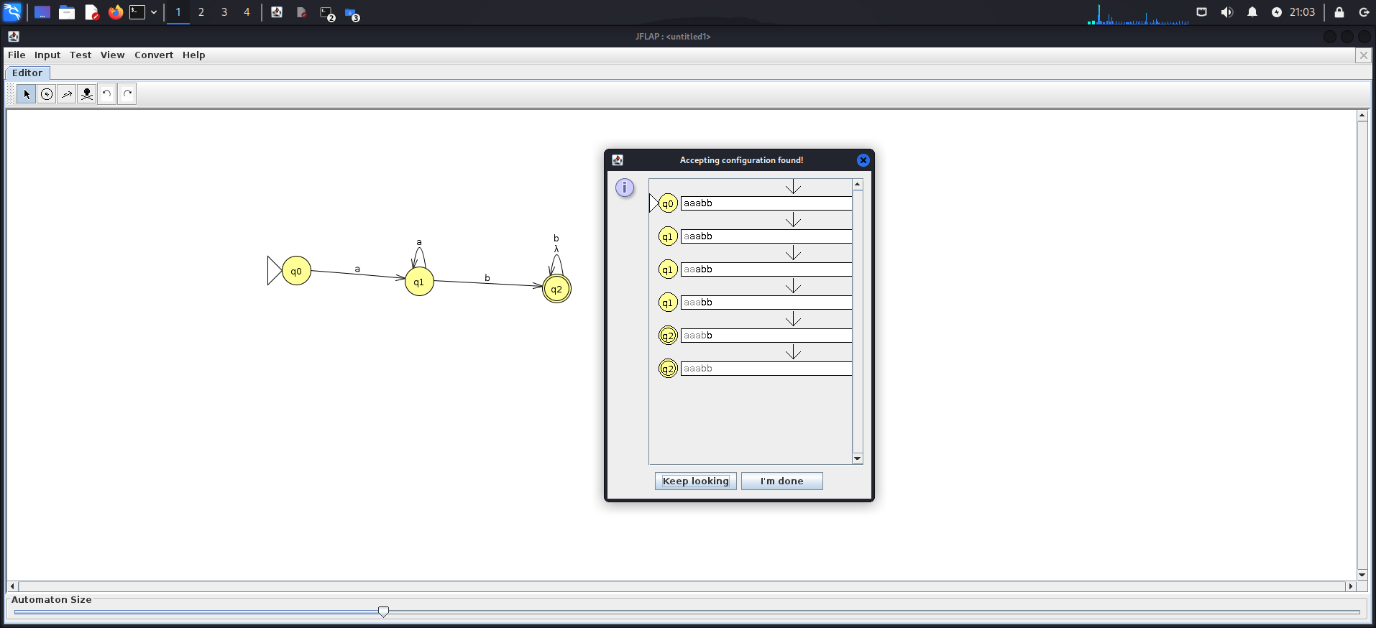
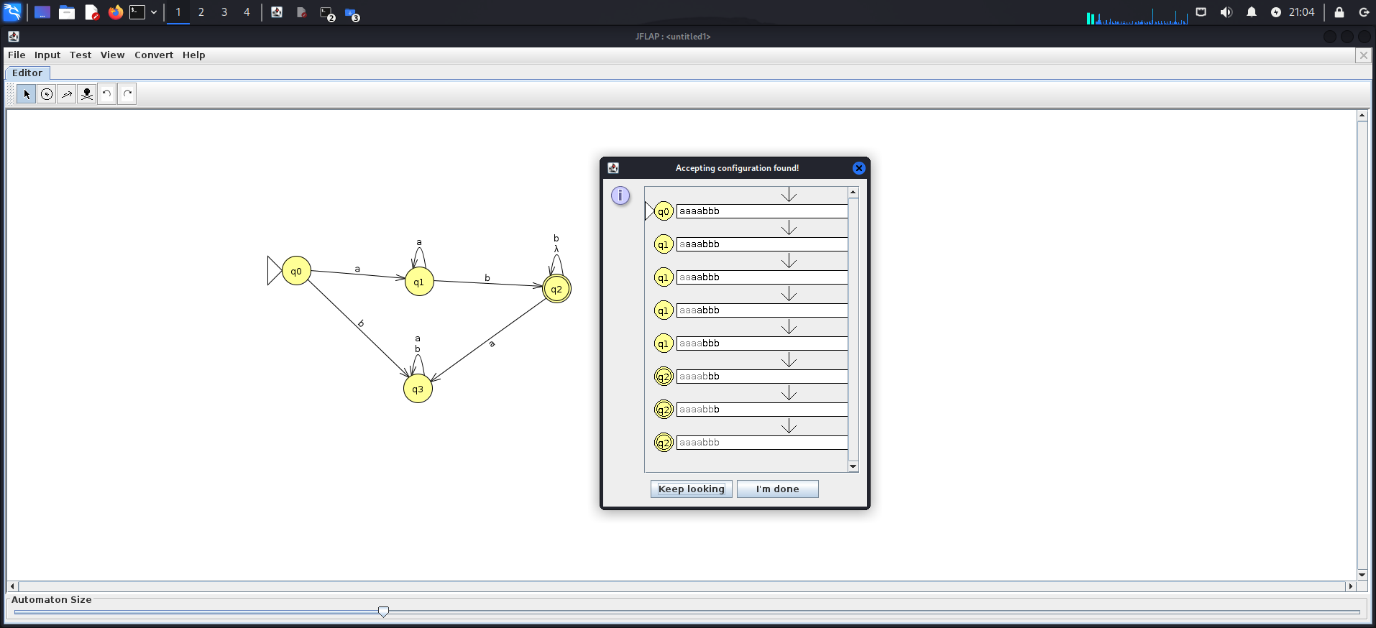
Regular expression:- (a+b)\*a

NFA DFA



Regular language L={anbm|n,m>=1}

NFA DFA



**3.Write a C++ program to find the number of tokens for any given arithmetic expression**

#include <iostream>

#include <string>

#include <regex>

#include <vector>

#include <unordered\_map>

using namespace std;

// Function to tokenize the input string

vector<pair<string, string>> tokenize(const string& input\_string, unordered\_map<string, int>& token\_count) {

vector<pair<string, string>> tokens;

try {

// Remove comments from input string

string processed\_string = regex\_replace(input\_string, regex("/\\\*.\*?\\\*/"), "");

processed\_string = regex\_replace(processed\_string, regex("//.\*"), "");

// Regular expression patterns for different token types

vector<pair<string, string>> patterns = {

{"KEYWORD", "\\b(if|else|while|for|return|int|float|double|char|string|void|const|static|class|struct|union|enum)\\b"},

{"IDENTIFIER", "\\b(?!(if|else|while|for|return|int|float|double|char|string|void|const|static|class|struct|union|enum)\\b)[a-zA-Z\_][a-zA-Z0-9\_]\*\\b"},

{"NUMBER", "\\b\\d+\\b"},

{"STRING", "\"(\\\\.|[^\\\\\"])\*\""},

{"OPERATOR", "\\+\\+|\\-\\-|\\+|\\-|\\\*|\\/|\\=|\\!|\\<|\\>|\\&|\\||\\^|\\%|\\~|\\?|\\:"},

{"PUNCTUATION", "[\\(\\)\\{\\}\\[\\]\\;\\,\\.]"}

};

// Tokenize processed input string for each pattern separately

for (const auto& pattern : patterns) {

regex token\_regex(pattern.second);

auto words\_begin = sregex\_iterator(processed\_string.begin(), processed\_string.end(), token\_regex);

auto words\_end = sregex\_iterator();

for (sregex\_iterator it = words\_begin; it != words\_end; ++it) {

smatch match = \*it;

tokens.emplace\_back(pattern.first, match.str());

token\_count[pattern.first]++;

}

}

// Sort tokens by their position in the original string

stable\_sort(tokens.begin(), tokens.end(), [&processed\_string](const pair<string, string>& a, const pair<string, string>& b) {

return processed\_string.find(a.second) < processed\_string.find(b.second);

});

} catch (const std::exception &e) {

cerr << "Error tokenizing input string: " << e.what() << endl;

}

return tokens;

}

int main() {

string input\_string;

cout << "Enter a string: ";

getline(cin, input\_string);

unordered\_map<string, int> token\_count; // Map to store token identifiers and their counts

vector<pair<string, string>> tokens = tokenize(input\_string, token\_count);

cout << "Tokens found:\n";

for (const auto& token : tokens) {

cout << "\t" << token.first << ": " << token.second << endl;

}

cout << "Number of tokens: " << tokens.size() << endl;

cout << "Token counts:\n";

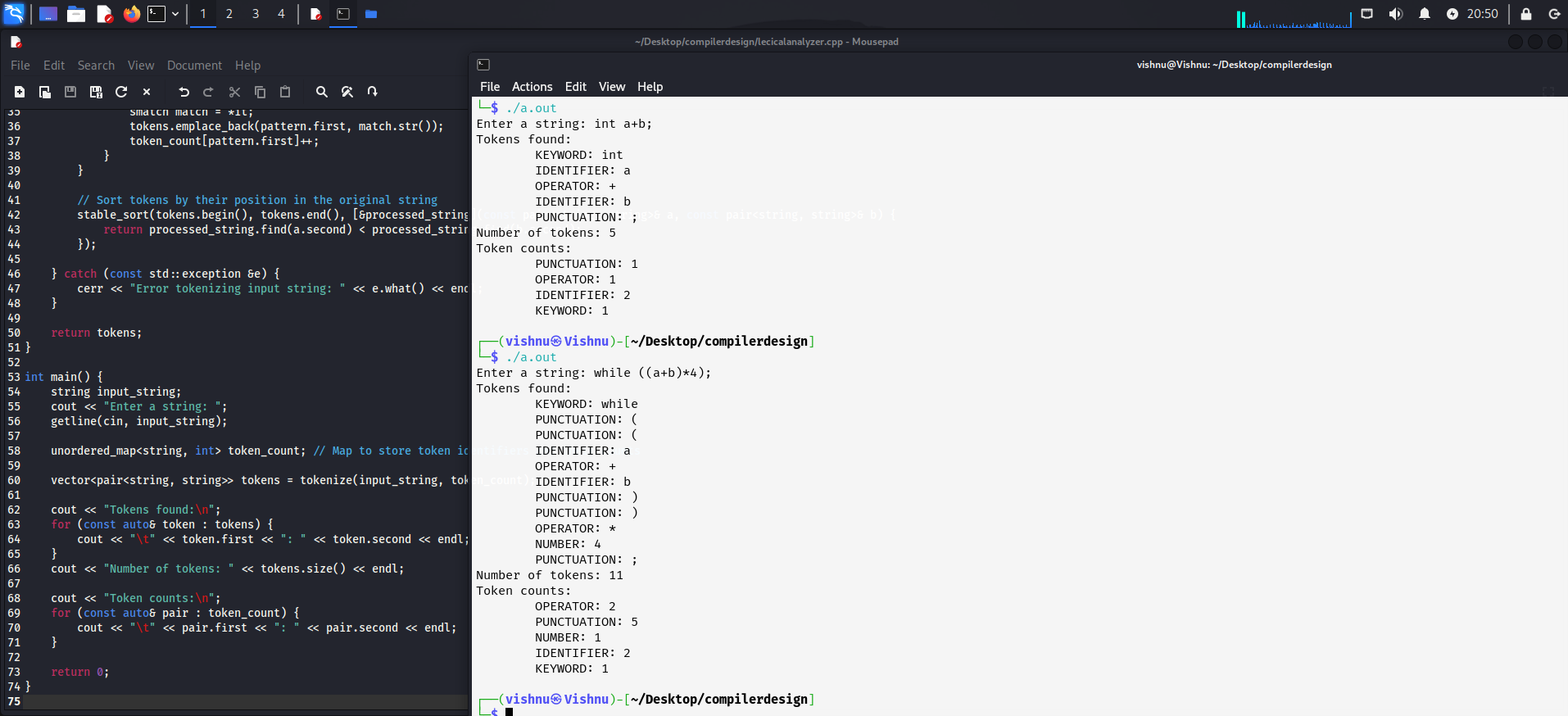
for (const auto& pair : token\_count) {

cout << "\t" << pair.first << ": " << pair.second << endl;

}

return 0;

}



**4.Write a program to design Lexical Analyzer in C++ (that recognize Keywords, Identifiers, Constants, Operators and Punctuation symbols)**

#include <iostream>

#include <string>

#include <regex>

#include <vector>

#include <unordered\_map>

using namespace std;

// Function to tokenize the input string

vector<pair<string, string>> tokenize(const string& input\_string, unordered\_map<string, int>& token\_count) {

vector<pair<string, string>> tokens;

try {

// Remove comments from input string

string processed\_string = regex\_replace(input\_string, regex("/\\\*.\*?\\\*/"), "");

processed\_string = regex\_replace(processed\_string, regex("//.\*"), "");

// Regular expression patterns for different token types

vector<pair<string, string>> patterns = {

{"KEYWORD", "\\b(if|else|while|for|return|int|float|double|char|string|void|const|static|class|struct|union|enum)\\b"},

{"IDENTIFIER", "\\b(?!(if|else|while|for|return|int|float|double|char|string|void|const|static|class|struct|union|enum)\\b)[a-zA-Z\_][a-zA-Z0-9\_]\*\\b"},

{"NUMBER", "\\b\\d+\\b"},

{"STRING", "\"(\\\\.|[^\\\\\"])\*\""},

{"OPERATOR", "\\+\\+|\\-\\-|\\+|\\-|\\\*|\\/|\\=|\\!|\\<|\\>|\\&|\\||\\^|\\%|\\~|\\?|\\:"},

{"PUNCTUATION", "[\\(\\)\\{\\}\\[\\]\\;\\,\\.]"}

};

// Tokenize processed input string for each pattern separately

for (const auto& pattern : patterns) {

regex token\_regex(pattern.second);

auto words\_begin = sregex\_iterator(processed\_string.begin(), processed\_string.end(), token\_regex);

auto words\_end = sregex\_iterator();

for (sregex\_iterator it = words\_begin; it != words\_end; ++it) {

smatch match = \*it;

tokens.emplace\_back(pattern.first, match.str());

token\_count[pattern.first]++;

}

}

// Sort tokens by their position in the original string

stable\_sort(tokens.begin(), tokens.end(), [&processed\_string](const pair<string, string>& a, const pair<string, string>& b) {

return processed\_string.find(a.second) < processed\_string.find(b.second);

});

} catch (const std::exception &e) {

cerr << "Error tokenizing input string: " << e.what() << endl;

}

return tokens;

}

int main() {

string input\_string;

cout << "Enter a string: ";

getline(cin, input\_string);

unordered\_map<string, int> token\_count; // Map to store token identifiers and their counts

vector<pair<string, string>> tokens = tokenize(input\_string, token\_count);

cout << "Tokens found:\n";

for (const auto& token : tokens) {

cout << "\t" << token.first << ": " << token.second << endl;

}

cout << "Number of tokens: " << tokens.size() << endl;

cout << "Token counts:\n";

for (const auto& pair : token\_count) {

cout << "\t" << pair.first << ": " << pair.second << endl;

}

return 0;

}

**Output**



5.**Write a LEX program over the ∑ = {0,1,..,9} to recognize the following tokens**

* **Strings ending with 11**

%{

#include <stdio.h>

%}

%%

[0-9]\*11 { printf("Matched: %s\n", yytext); }

%%

int main() {

yylex();

return 0;

}

int yywrap() {

return 1;

}

**Input.txt**

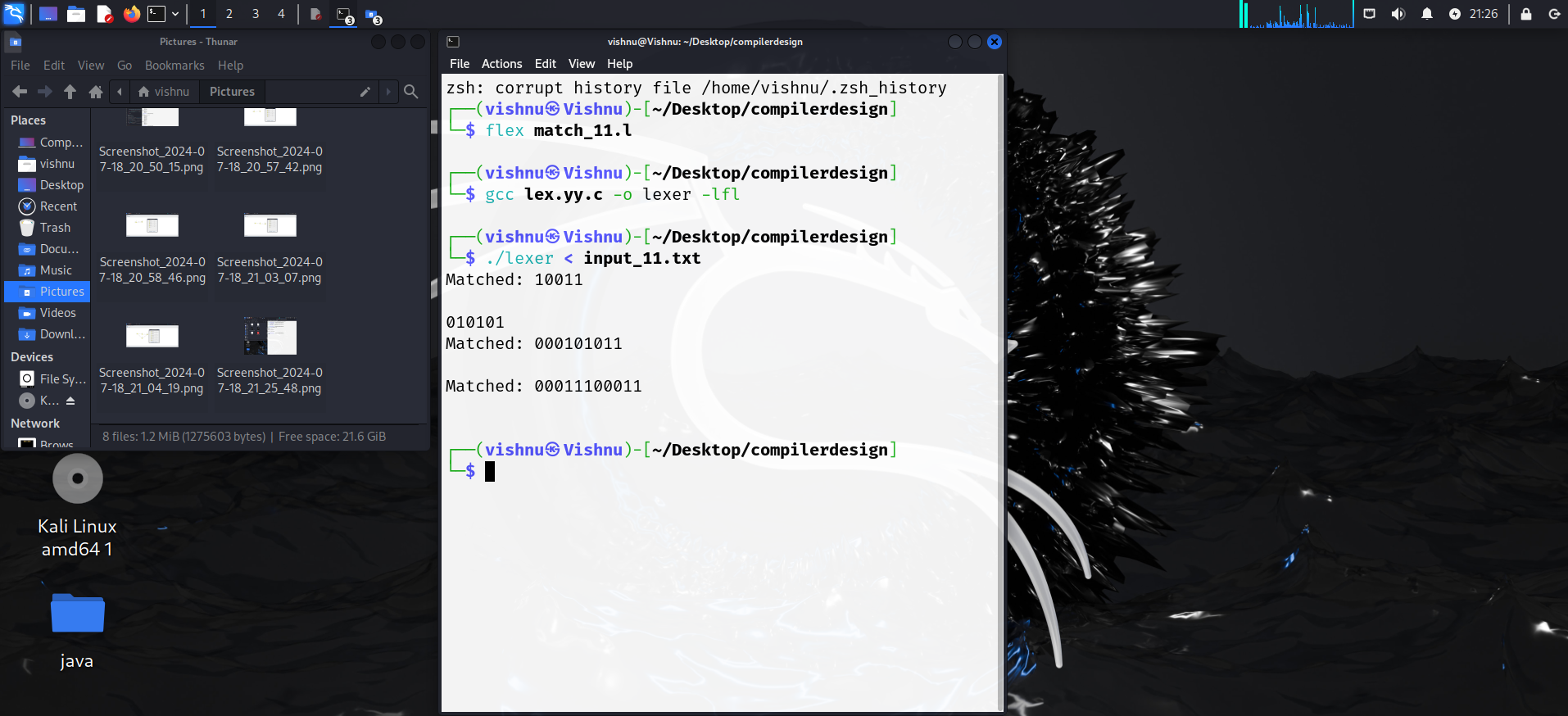
10011

010101

000101011

00011100011

**Output**



* **Strings containing 3 consecutive 2’s**

%{

#include <stdio.h>

int match\_found = 0; // Global variable to track if a match was found

int match\_count = 0; // Counter for the number of matches found

%}

%%

222 {

printf("Matched: %s\n", yytext);

match\_found = 1;

match\_count++; // Increment the counter each time a match is found

}

%%

int main() {

yylex();

if (!match\_found) {

printf("String not found.\n");

} else {

printf("-----------------------------------------------");

printf("\nNumber of matches found: %d\n", match\_count);

printf("-----------------------------------------------\n");

}

return 0;

}

int yywrap() {

return 1;

}

**Input.txt**

6383222

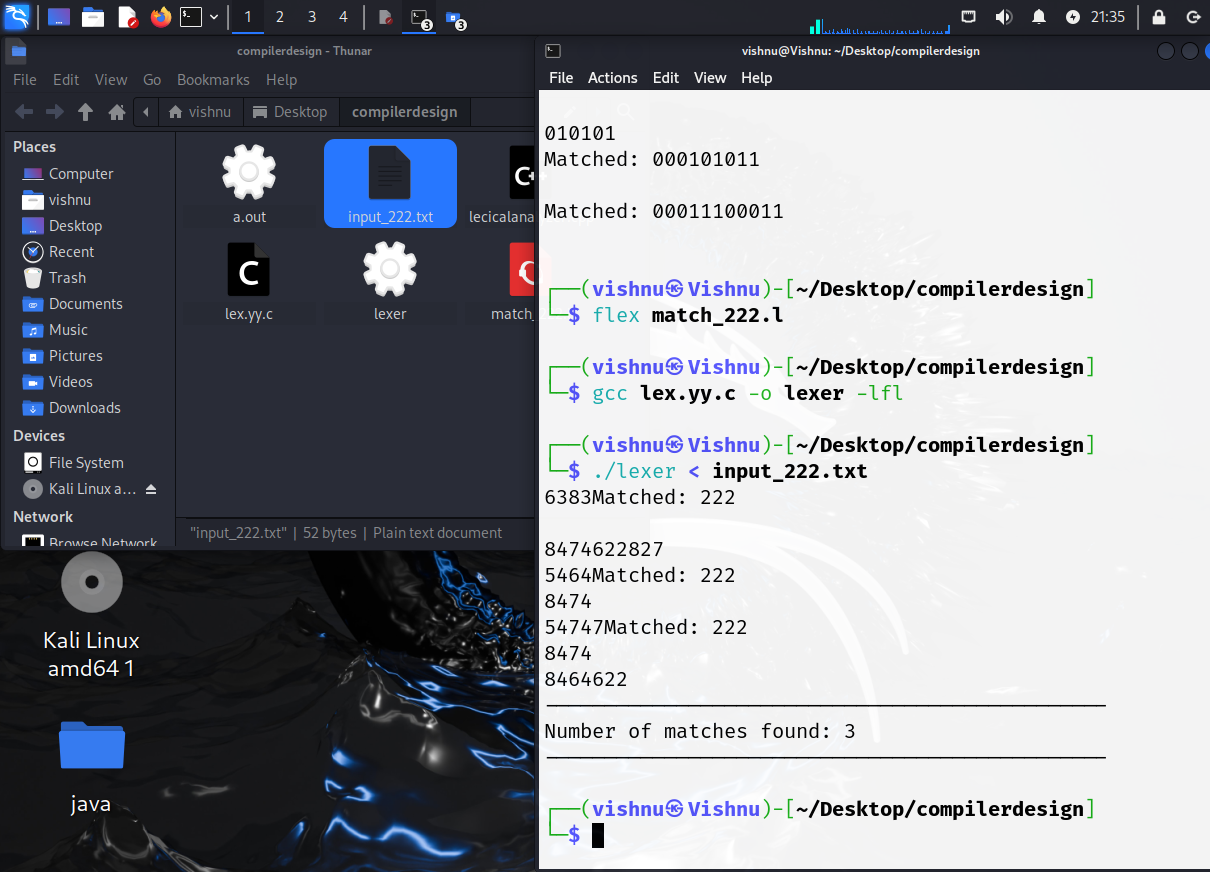
8474622827

54642228474

547472228474

8464622

**Output**



* **10th symbol from right is 1**

%{

#include <stdio.h>

%}

%%

[0-9]+ {

// Check if the 10th symbol from the right end is 1

int len = strlen(yytext);

if (len >= 10 && yytext[len - 10] == '1') {

printf("Matched: %s\n", yytext);

}

}

%%

int main() {

yylex();

return 0;

}

int yywrap() {

return 1;

}

**Input\_10**

0000000001

1101010101

1010101011

1111111111

0010101011

1000000001

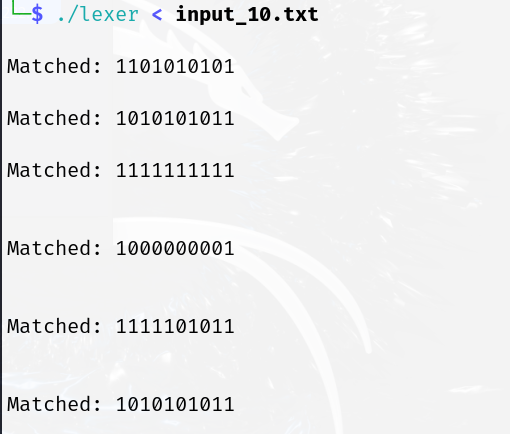
0101010111

1111101011

0000010101

1010101011

**Output**



* **All four digit integer whose sum is 9**

%{

#include <stdio.h>

%}

%%

[0-9]{4} {

int sum = 0;

for (int i = 0; yytext[i] != '\0'; i++) {

sum += yytext[i] - '0'; // Convert char to int and add to sum

}

if (sum == 9) {

printf("Matched: %s\n", yytext);

}

}

%%

int main() {

yylex();

return 0;

}

int yywrap() {

return 1;

}

**Input**

3321

2133

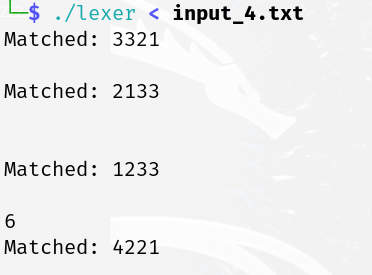
5342

1233

53526

4221

**Output**



**6 - Write a Lex program to identify a floating-point number using file handling**

%{

#include <stdio.h>

%}

%%

[+-]?([0-9]\*\.[0-9]+|[0-9]+\.[0-9]\*)([eE][+-]?[0-9]+)? {

printf("Float = {%s}\n", yytext);

}

.|\n {

/\* Ignore other characters \*/

}

%%

int main() {

yylex();

return 0;

}

int yywrap() {

return 1;

}

**Input**

27.73

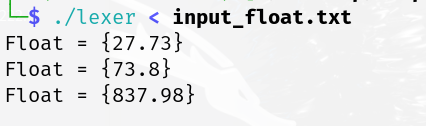
73.8

54

766

837.98

**Output**

****

7-**Write a Lex program for the following:**

* **To identify keywords and convert to uppercase**

%{

#include <stdio.h>

#include <string.h>

#include <ctype.h>

%}

%option noyywrap

%%

"if"|"else"|"while"|"for"|"return"|"int"|"float"|"double"|"char"|"string"|"void"|"const"|"static"|"class"|"struct"|"union"|"enum" {

for (int i = 0; yytext[i]; i++)

putchar(toupper(yytext[i]));

putchar('\n');

}

.|\n // catch-all rule to skip any other characters

%%

int main() {

FILE \*fp = fopen("input.txt", "r");

if (!fp) {

perror("Error opening file");

return 1;

}

yyin = fp;

yylex();

fclose(fp);

return 0;

}

**Input**

if (x > 0) {

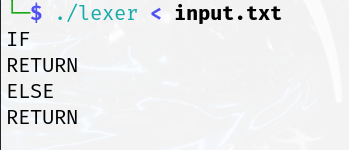
return x;

} else {

return -x;

}

**Output**



* **To count no of vowels and consonants**

%{

#include <stdio.h>

#include <ctype.h>

%}

%option noyywrap

%%

[a-zA-Z] {

if (strchr("aeiouAEIOU", yytext[0]) != NULL)

printf("Vowel: %s\n", yytext);

else if (isalpha(yytext[0]))

printf("Consonant: %s\n", yytext);

}

.|\n // catch-all rule to skip any other characters

%%

int main() {

FILE \*fp = fopen("input.txt", "r");

if (!fp) {

perror("Error opening file");

return 1;

}

yyin = fp;

yylex();

fclose(fp);

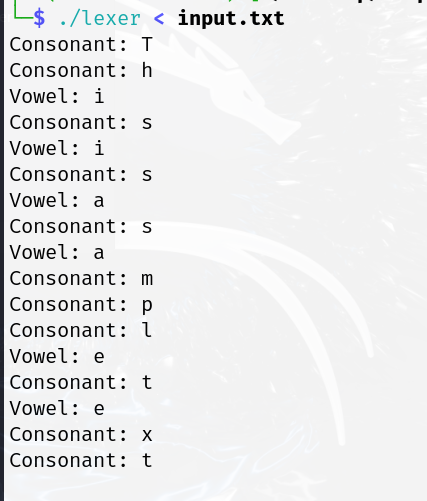
return 0;

}

**Input**

This is a sample text ! , >

**Output**



* **To count no of identifiers and keywords and digit**

%{

#include <stdio.h>

#include <string.h>

#include <ctype.h>

int keyword\_count = 0;

int identifier\_count = 0;

int digit\_count = 0;

%}

%option noyywrap

%%

"if"|"else"|"while"|"for"|"return"|"int"|"float"|"double"|"char"|"string"|"void"|"const"|"static"|"class"|"struct"|"union"|"enum" {

printf("Keyword: %s\n", yytext);

keyword\_count++;

}

[a-zA-Z\_][a-zA-Z0-9\_]\* {

printf("Identifier: %s\n", yytext);

identifier\_count++;

}

[0-9]+ {

printf("Digit: %s\n", yytext);

digit\_count++;

}

.|\n // catch-all rule to skip any other characters

%%

int main() {

FILE \*fp = fopen("input.txt", "r");

if (!fp) {

perror("Error opening file");

return 1;

}

yyin = fp;

yylex();

fclose(fp);

// Print total counts

printf("\nTotal counts:\n");

printf("Keywords: %d\n", keyword\_count);

printf("Identifiers: %d\n", identifier\_count);

printf("Digits: %d\n", digit\_count);

return 0;

}

**Input**

int main() {

int x = 10;

if (x > 5) {

return x;

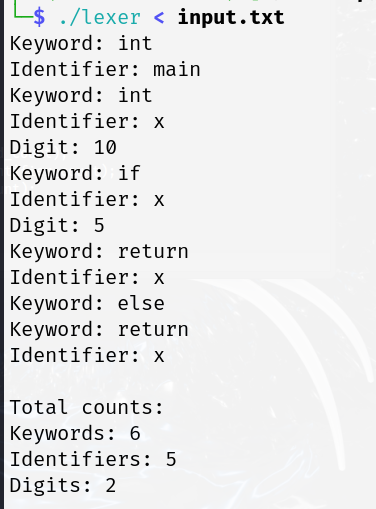
} else {

return -x;

}

}

**Output**



**8-Write a C++ program to find first and follow for any given CFG**

#include <iostream>

#include <vector>

#include <unordered\_map>

#include <unordered\_set>

#include <sstream>

using namespace std;

class GrammarAnalyzer {

private:

unordered\_map<char, vector<string>> productions;

unordered\_set<char> first(char nonTerminal);

unordered\_set<char> follow(char nonTerminal);

public:

void addProduction(char nonTerminal, const vector<string>& symbols);

void computeFirstSets();

void computeFollowSets();

};

void GrammarAnalyzer::addProduction(char nonTerminal, const vector<string>& symbols) {

productions[nonTerminal] = symbols;

}

void GrammarAnalyzer::computeFirstSets() {

cout << "First sets:\n";

for (auto prod : productions) {

cout << "First(" << prod.first << ") = ";

unordered\_set<char> firstSet = first(prod.first);

for (char terminal : firstSet) {

cout << terminal << " ";

}

cout << endl;

}

}

void GrammarAnalyzer::computeFollowSets() {

cout << "\nFollow sets:\n";

for (auto prod : productions) {

cout << "Follow(" << prod.first << ") = ";

unordered\_set<char> followSet = follow(prod.first);

for (char terminal : followSet) {

cout << terminal << " ";

}

cout << endl;

}

}

unordered\_set<char> GrammarAnalyzer::first(char nonTerminal) {

unordered\_set<char> result;

if (!isupper(nonTerminal)) {

result.insert(nonTerminal);

return result;

}

for (string production : productions[nonTerminal]) {

if (production[0] == nonTerminal) continue; // Avoid direct recursion

unordered\_set<char> temp = first(production[0]);

result.insert(temp.begin(), temp.end());

// Check for epsilon in the first set of the production

size\_t i = 0;

while (temp.find('@') != temp.end() && i < production.size()) {

temp.erase('@');

++i;

if (i < production.size() && isupper(production[i])) {

unordered\_set<char> nextFirst = first(production[i]);

temp.insert(nextFirst.begin(), nextFirst.end());

} else if (i < production.size()) {

temp.insert(production[i]);

}

}

result.insert(temp.begin(), temp.end());

}

return result;

}

unordered\_set<char> GrammarAnalyzer::follow(char nonTerminal) {

unordered\_set<char> result;

if (nonTerminal == 'S') result.insert('$'); // Start symbol

for (auto prod : productions) {

for (string production : prod.second) {

size\_t found = production.find(nonTerminal);

if (found != string::npos) {

if (found == production.size() - 1) { // If non-terminal is the last symbol

if (prod.first != nonTerminal) { // To avoid infinite recursion

unordered\_set<char> temp = follow(prod.first);

result.insert(temp.begin(), temp.end());

}

} else { // If non-terminal is followed by a terminal or non-terminal

char nextSymbol = production[found + 1];

if (isupper(nextSymbol)) { // If next symbol is non-terminal

unordered\_set<char> temp = first(nextSymbol);

if (temp.find('@') != temp.end()) { // If next symbol can derive epsilon

temp.erase('@');

unordered\_set<char> followSet = follow(prod.first);

temp.insert(followSet.begin(), followSet.end());

}

result.insert(temp.begin(), temp.end());

} else { // If next symbol is terminal

result.insert(nextSymbol);

}

}

}

}

}

return result;

}

int main() {

GrammarAnalyzer analyzer;

string input;

cout << "Enter the grammar productions (Enter 'done' to finish):\n";

while (true) {

getline(cin, input);

if (input == "done") break;

stringstream ss(input);

string production, arrow;

char nonTerminal;

ss >> nonTerminal >> arrow;

vector<string> symbols;

while (ss >> production) {

symbols.push\_back(production);

}

analyzer.addProduction(nonTerminal, symbols);

}

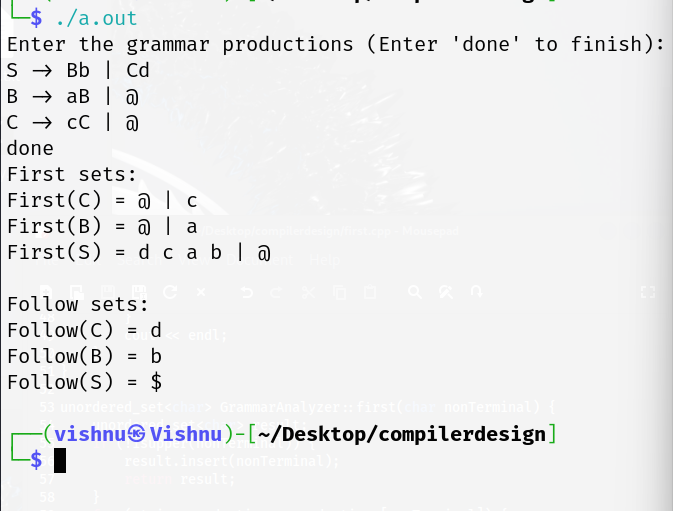
analyzer.computeFirstSets();

analyzer.computeFollowSets();

return 0;

}

**Output**

****

**9-Write a Python program to design a LR(0) parser and SLR (1) parser and print the number of conflicts if any**

**Lr(0)**

class Grammar:

def \_\_init\_\_(self, terminals, nonterminals, productions, start\_symbol):

self.terminals = terminals

self.nonterminals = nonterminals

self.productions = productions

self.start\_symbol = start\_symbol

def get\_productions(self, nonterminal):

return [prod for prod in self.productions if prod[0] == nonterminal]

class Item:

def \_\_init\_\_(self, production, dot\_position):

self.production = (production[0], tuple(production[1])) # Convert list to tuple for hashing

self.dot\_position = dot\_position

def \_\_eq\_\_(self, other):

return (self.production == other.production and self.dot\_position == other.dot\_position)

def \_\_hash\_\_(self):

return hash((self.production, self.dot\_position))

def \_\_repr\_\_(self):

return f"{self.production[0]} -> {' '.join(self.production[1][:self.dot\_position])} . {' '.join(self.production[1][self.dot\_position:])}"

def closure(items, grammar):

closure\_set = set(items)

while True:

new\_items = set(closure\_set)

for item in closure\_set:

if item.dot\_position < len(item.production[1]):

symbol = item.production[1][item.dot\_position]

if symbol in grammar.nonterminals:

for prod in grammar.get\_productions(symbol):

new\_items.add(Item(prod, 0))

if new\_items == closure\_set:

break

closure\_set = new\_items

return closure\_set

def goto(items, symbol, grammar):

goto\_set = set()

for item in items:

if item.dot\_position < len(item.production[1]) and item.production[1][item.dot\_position] == symbol:

goto\_set.add(Item(item.production, item.dot\_position + 1))

return closure(goto\_set, grammar)

def items(grammar):

start\_item = Item((grammar.start\_symbol, [grammar.productions[0][0]]), 0)

states = [closure({start\_item}, grammar)]

transitions = {}

state\_to\_id = {frozenset(states[0]): 0}

while True:

new\_states = list(states)

for state in states:

for symbol in grammar.terminals + grammar.nonterminals:

new\_state = goto(state, symbol, grammar)

if new\_state:

frozen\_new\_state = frozenset(new\_state)

if frozen\_new\_state not in state\_to\_id:

state\_to\_id[frozen\_new\_state] = len(new\_states)

new\_states.append(new\_state)

transitions[(state\_to\_id[frozenset(state)], symbol)] = state\_to\_id[frozen\_new\_state]

if new\_states == states:

break

states = new\_states

return states, transitions, state\_to\_id

def build\_parsing\_table(grammar):

states, transitions, state\_to\_id = items(grammar)

action = {}

goto\_table = {}

for i, state in enumerate(states):

for item in state:

if item.dot\_position == len(item.production[1]):

if item.production[0] == grammar.start\_symbol:

action[(i, '$')] = ('accept',)

else:

for terminal in grammar.terminals:

action[(i, terminal)] = ('reduce', item.production)

elif item.production[1][item.dot\_position] in grammar.terminals:

symbol = item.production[1][item.dot\_position]

if (i, symbol) in transitions:

next\_state = transitions[(i, symbol)]

action[(i, symbol)] = ('shift', next\_state)

else:

symbol = item.production[1][item.dot\_position]

if (i, symbol) in transitions:

next\_state = transitions[(i, symbol)]

goto\_table[(i, symbol)] = next\_state

return action, goto\_table, states, transitions

def print\_dfa(states, transitions):

print("DFA States and Transitions:")

for i, state in enumerate(states):

print(f"State {i}:")

for item in state:

print(f" {item}")

print()

print("Transitions:")

for (state, symbol), next\_state in transitions.items():

print(f" State {state} --{symbol}--> State {next\_state}")

print()

def print\_parsing\_table(action, goto\_table, grammar):

print("Action Table:")

for key in sorted(action.keys()):

print(f"{key}: {action[key]}")

print("\nGoto Table:")

for key in sorted(goto\_table.keys()):

print(f"{key}: {goto\_table[key]}")

if \_\_name\_\_ == "\_\_main\_\_":

terminals = ['a', 'b', '$']

nonterminals = ['S', 'A']

productions = [

('S', ['A', 'A']),

('A', ['a', 'A']),

('A', ['b'])

]

start\_symbol = 'S'

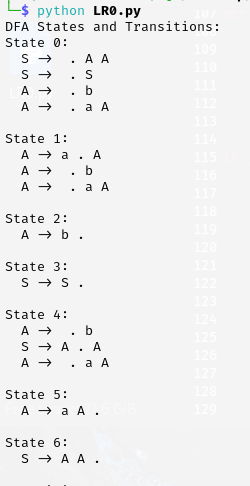
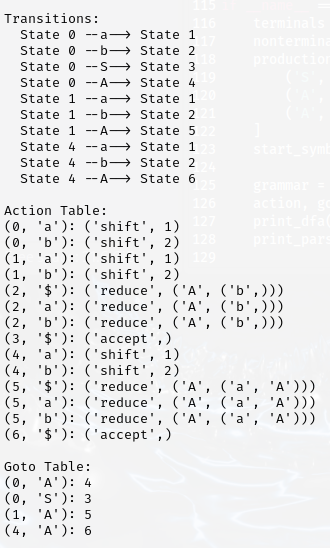
grammar = Grammar(terminals, nonterminals, productions, start\_symbol)

action, goto\_table, states, transitions = build\_parsing\_table(grammar)

print\_dfa(states, transitions)

print\_parsing\_table(action, goto\_table, grammar)

**Output**

**** ****

**SLR(1)**

# SLR(1)

import copy

# perform grammar augmentation

def grammarAugmentation(rules, nonterm\_userdef,

start\_symbol):

# newRules stores processed output rules

newRules = []

# create unique 'symbol' to

# - represent new start symbol

newChar = start\_symbol + "'"

while (newChar in nonterm\_userdef):

newChar += "'"

# adding rule to bring start symbol to RHS

newRules.append([newChar,

['.', start\_symbol]])

# new format => [LHS,[.RHS]],

# can't use dictionary since

# - duplicate keys can be there

for rule in rules:

# split LHS from RHS

k = rule.split("->")

lhs = k[0].strip()

rhs = k[1].strip()

# split all rule at '|'

# keep single derivation in one rule

multirhs = rhs.split('|')

for rhs1 in multirhs:

rhs1 = rhs1.strip().split()

# ADD dot pointer at start of RHS

rhs1.insert(0, '.')

newRules.append([lhs, rhs1])

return newRules

# find closure

def findClosure(input\_state, dotSymbol):

global start\_symbol, \

separatedRulesList, \

statesDict

# closureSet stores processed output

closureSet = []

# if findClosure is called for

# - 1st time i.e. for I0,

# then LHS is received in "dotSymbol",

# add all rules starting with

# - LHS symbol to closureSet

if dotSymbol == start\_symbol:

for rule in separatedRulesList:

if rule[0] == dotSymbol:

closureSet.append(rule)

else:

# for any higher state than I0,

# set initial state as

# - received input\_state

closureSet = input\_state

# iterate till new states are

# - getting added in closureSet

prevLen = -1

while prevLen != len(closureSet):

prevLen = len(closureSet)

# "tempClosureSet" - used to eliminate

# concurrent modification error

tempClosureSet = []

# if dot pointing at new symbol,

# add corresponding rules to tempClosure

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)

# add new closure rules to closureSet

for rule in tempClosureSet:

if rule not in closureSet:

closureSet.append(rule)

return closureSet

def compute\_GOTO(state):

global statesDict, stateCount

# find all symbols on which we need to

# make function call - GOTO

generateStatesFor = []

for rule in statesDict[state]:

# if rule is not "Handle"

if rule[1][-1] != '.':

indexOfDot = rule[1].index('.')

dotPointsHere = rule[1][indexOfDot + 1]

if dotPointsHere not in generateStatesFor:

generateStatesFor.append(dotPointsHere)

# call GOTO iteratively on all symbols pointed by dot

if len(generateStatesFor) != 0:

for symbol in generateStatesFor:

GOTO(state, symbol)

return

def GOTO(state, charNextToDot):

global statesDict, stateCount, stateMap

# newState - stores processed new state

newState = []

for rule in statesDict[state]:

indexOfDot = rule[1].index('.')

if rule[1][-1] != '.':

if rule[1][indexOfDot + 1] == \

charNextToDot:

# swapping element with dot,

# to perform shift operation

shiftedRule = copy.deepcopy(rule)

shiftedRule[1][indexOfDot] = \

shiftedRule[1][indexOfDot + 1]

shiftedRule[1][indexOfDot + 1] = '.'

newState.append(shiftedRule)

# add closure rules for newState

# call findClosure function iteratively

# - on all existing rules in newState

# addClosureRules - is used to store

# new rules temporarily,

# to prevent concurrent modification error

addClosureRules = []

for rule in newState:

indexDot = rule[1].index('.')

# check that rule is not "Handle"

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)

# add closure result to newState

for rule in addClosureRules:

newState.append(rule)

# find if newState already present

# in Dictionary

stateExists = -1

for state\_num in statesDict:

if statesDict[state\_num] == newState:

stateExists = state\_num

break

# stateMap is a mapping of GOTO with

# its output states

if stateExists == -1:

# if newState is not in dictionary,

# then create new state

stateCount += 1

statesDict[stateCount] = newState

stateMap[(state, charNextToDot)] = stateCount

else:

# if state repetition found,

# assign that previous state number

stateMap[(state, charNextToDot)] = stateExists

return

def generateStates(statesDict):

prev\_len = -1

called\_GOTO\_on = []

# run loop till new states are getting added

while (len(statesDict) != prev\_len):

prev\_len = len(statesDict)

keys = list(statesDict.keys())

# make compute\_GOTO function call

# on all states in dictionary

for key in keys:

if key not in called\_GOTO\_on:

called\_GOTO\_on.append(key)

compute\_GOTO(key)

return

# calculation of first

# epsilon is denoted by '#' (semi-colon)

# pass rule in first function

def first(rule):

global rules, nonterm\_userdef, \

term\_userdef, diction, firsts

# recursion base condition

# (for terminal or epsilon)

if len(rule) != 0 and (rule is not None):

if rule[0] in term\_userdef:

return rule[0]

elif rule[0] == '#':

return '#'

# condition for Non-Terminals

if len(rule) != 0:

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

# fres temporary list of result

fres = []

rhs\_rules = diction[rule[0]]

# call first on each rule of RHS

# fetched (& take union)

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 no epsilon in result

# - received return fres

if '#' not in fres:

return fres

else:

# apply epsilon

# rule => f(ABC)=f(A)-{e} U f(BC)

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

# if result is not already returned

# - control reaches here

# lastly if eplison still persists

# - keep it in result of first

fres.append('#')

return fres

# calculation of follow

def follow(nt):

global start\_symbol, rules, nonterm\_userdef, \

term\_userdef, diction, firsts, follows

# for start symbol return $ (recursion base case)

solset = set()

if nt == start\_symbol:

# return '$'

solset.add('$')

# check all occurrences

# solset - is result of computed 'follow' so far

# For input, check in all rules

for curNT in diction:

rhs = diction[curNT]

# go for all productions of NT

for subrule in rhs:

if nt in subrule:

# call for all occurrences on

# - non-terminal in subrule

while nt in subrule:

index\_nt = subrule.index(nt)

subrule = subrule[index\_nt + 1:]

# empty condition - call follow on LHS

if len(subrule) != 0:

# compute first if symbols on

# - RHS of target Non-Terminal exists

res = first(subrule)

# if epsilon in result apply rule

# - (A->aBX)- follow of -

# - follow(B)=(first(X)-{ep}) U follow(A)

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:

# when nothing in RHS, go circular

# - and take follow of LHS

# only if (NT in LHS)!=curNT

if nt != curNT:

res = follow(curNT)

# add follow result in set form

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

# create rows and cols

rows = list(statesDict.keys())

cols = T+['$']+NT

# create empty table

Table = []

tempRow = []

for y in range(len(cols)):

tempRow.append('')

for x in range(len(rows)):

Table.append(copy.deepcopy(tempRow))

# make shift and GOTO entries in table

for entry in stateMap:

state = entry[0]

symbol = entry[1]

# get index

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]} "

# start REDUCE procedure

# number the separated rules

numbered = {}

key\_count = 0

for rule in separatedRulesList:

tempRule = copy.deepcopy(rule)

tempRule[1].remove('.')

numbered[key\_count] = tempRule

key\_count += 1

# start REDUCE procedure

# format for follow computation

addedR = f"{separatedRulesList[0][0]} -> " \

f"{separatedRulesList[0][1][1]}"

rules.insert(0, addedR)

for rule in rules:

k = rule.split("->")

# remove un-necessary spaces

k[0] = k[0].strip()

k[1] = k[1].strip()

rhs = k[1]

multirhs = rhs.split('|')

# remove un-necessary spaces

for i in range(len(multirhs)):

multirhs[i] = multirhs[i].strip()

multirhs[i] = multirhs[i].split()

diction[k[0]] = multirhs

# find 'handle' items and calculate follow.

for stateno in statesDict:

for rule in statesDict[stateno]:

if rule[1][-1] == '.':

# match the item

temp2 = copy.deepcopy(rule)

temp2[1].remove('.')

for key in numbered:

if numbered[key] == temp2:

# put Rn in those ACTION symbol columns,

# who are in the follow of

# LHS of current Item.

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} "

# printing table

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]}")

# \*\*\* MAIN \*\*\* - Driver Code

# uncomment any rules set to test code

# follow given format to add -

# user defined grammar rule set

# rules section - \*START\*

# example sample set 01

rules = ["S -> L = R",

"S -> R",

"L -> \* R",

"L -> id",

"R -> L"

]

nonterm\_userdef = ['S', 'L', 'R']

term\_userdef = ['id', '=', '\*' ]

start\_symbol = nonterm\_userdef[0]

# example sample set 02

# rules = ["S -> a X d | b Y d | a Y e | b X e",

# "X -> c",

# "Y -> c"

# ]

# nonterm\_userdef = ['S','X','Y']

# term\_userdef = ['a','b','c','d','e']

# start\_symbol = nonterm\_userdef[0]

# rules section - \*END\*

print("\nOriginal grammar input:\n")

for y in rules:

print(y)

# print processed rules

print("\nGrammar after Augmentation: \n")

separatedRulesList = \

grammarAugmentation(rules,

nonterm\_userdef,

start\_symbol)

printResult(separatedRulesList)

# find closure

start\_symbol = separatedRulesList[0][0]

print("\nCalculated closure: I0\n")

I0 = findClosure(0, start\_symbol)

printResult(I0)

# use statesDict to store the states

# use stateMap to store GOTOs

statesDict = {}

stateMap = {}

# add first state to statesDict

# and maintain stateCount

# - for newState generation

statesDict[0] = I0

stateCount = 0

# computing states by GOTO

generateStates(statesDict)

# print goto states

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)

# "follow computation" for making REDUCE entries

diction = {}

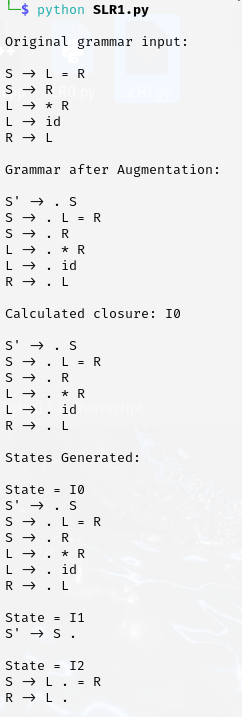
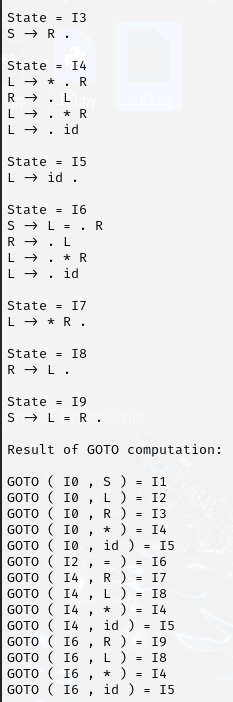
# call createParseTable function

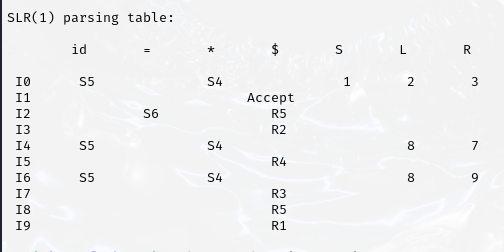
createParseTable(statesDict, stateMap,

term\_userdef,

nonterm\_userdef)

**Output**

**** ****



**10.Write a C++ program to design a CLR and LALR parser and print the parsing table**

**CLR**

class Grammar:

def \_\_init\_\_(self, terminals, nonterminals, productions, start\_symbol):

self.terminals = terminals

self.nonterminals = nonterminals

self.productions = productions

self.start\_symbol = start\_symbol

def get\_productions(self, nonterminal):

return [prod for prod in self.productions if prod[0] == nonterminal]

class Item:

def \_\_init\_\_(self, production, dot\_position, lookahead):

self.production = (production[0], tuple(production[1])) # Convert list to tuple for hashing

self.dot\_position = dot\_position

self.lookahead = lookahead

def \_\_eq\_\_(self, other):

return (self.production == other.production and

self.dot\_position == other.dot\_position and

self.lookahead == other.lookahead)

def \_\_hash\_\_(self):

return hash((self.production, self.dot\_position, self.lookahead))

def \_\_repr\_\_(self):

return f"{self.production[0]} -> {' '.join(self.production[1][:self.dot\_position])} . {' '.join(self.production[1][self.dot\_position:])}, {self.lookahead}"

def first(symbol, grammar, first\_cache):

if symbol in first\_cache:

return first\_cache[symbol]

first\_set = set()

if symbol in grammar.terminals:

first\_set.add(symbol)

else:

for prod in grammar.get\_productions(symbol):

if prod[1][0] in grammar.terminals:

first\_set.add(prod[1][0])

else:

for sym in prod[1]:

first\_set.update(first(sym, grammar, first\_cache))

if 'ε' not in first\_cache[sym]:

break

else:

first\_set.add('ε')

first\_cache[symbol] = first\_set

return first\_set

def follow(nonterminal, grammar, first\_cache, follow\_cache):

if nonterminal in follow\_cache:

return follow\_cache[nonterminal]

follow\_set = set()

if nonterminal == grammar.start\_symbol:

follow\_set.add('$')

for prod in grammar.productions:

for idx, sym in enumerate(prod[1]):

if sym == nonterminal:

if idx + 1 < len(prod[1]):

next\_symbol = prod[1][idx + 1]

follow\_set.update(first(next\_symbol, grammar, first\_cache) - {'ε'})

if 'ε' in first\_cache[next\_symbol]:

follow\_set.update(follow(prod[0], grammar, first\_cache, follow\_cache))

else:

if prod[0] != nonterminal:

follow\_set.update(follow(prod[0], grammar, first\_cache, follow\_cache))

follow\_cache[nonterminal] = follow\_set

return follow\_set

def closure(items, grammar, first\_cache):

closure\_set = set(items)

while True:

new\_items = set(closure\_set)

for item in closure\_set:

if item.dot\_position < len(item.production[1]):

symbol = item.production[1][item.dot\_position]

if symbol in grammar.nonterminals:

for prod in grammar.get\_productions(symbol):

lookaheads = set()

for sym in item.production[1][item.dot\_position+1:]:

lookaheads.update(first(sym, grammar, first\_cache))

if 'ε' not in first\_cache[sym]:

break

else:

lookaheads.update({item.lookahead})

for lookahead in lookaheads:

new\_items.add(Item(prod, 0, lookahead))

if new\_items == closure\_set:

break

closure\_set = new\_items

return closure\_set

def goto(items, symbol, grammar, first\_cache):

goto\_set = set()

for item in items:

if item.dot\_position < len(item.production[1]) and item.production[1][item.dot\_position] == symbol:

goto\_set.add(Item(item.production, item.dot\_position + 1, item.lookahead))

return closure(goto\_set, grammar, first\_cache)

def items(grammar):

first\_cache = {}

follow\_cache = {}

start\_item = Item((grammar.start\_symbol, [grammar.productions[0][0]]), 0, '$')

states = [closure({start\_item}, grammar, first\_cache)]

transitions = {}

state\_to\_id = {frozenset(states[0]): 0}

while True:

new\_states = list(states)

for state in states:

for symbol in grammar.terminals + grammar.nonterminals:

new\_state = goto(state, symbol, grammar, first\_cache)

if new\_state:

frozen\_new\_state = frozenset(new\_state)

if frozen\_new\_state not in state\_to\_id:

state\_to\_id[frozen\_new\_state] = len(new\_states)

new\_states.append(new\_state)

transitions[(state\_to\_id[frozenset(state)], symbol)] = state\_to\_id[frozen\_new\_state]

if new\_states == states:

break

states = new\_states

return states, transitions, state\_to\_id

def build\_parsing\_table(grammar):

states, transitions, state\_to\_id = items(grammar)

action = {}

goto\_table = {}

follow\_cache = {}

first\_cache = {}

for i, state in enumerate(states):

for item in state:

if item.dot\_position == len(item.production[1]):

if item.production[0] == grammar.start\_symbol:

action[(i, '$')] = ('accept',)

else:

for follow\_symbol in follow(item.production[0], grammar, first\_cache, follow\_cache):

action[(i, follow\_symbol)] = ('reduce', item.production)

elif item.production[1][item.dot\_position] in grammar.terminals:

symbol = item.production[1][item.dot\_position]

if (i, symbol) in transitions:

next\_state = transitions[(i, symbol)]

action[(i, symbol)] = ('shift', next\_state)

else:

symbol = item.production[1][item.dot\_position]

if (i, symbol) in transitions:

next\_state = transitions[(i, symbol)]

goto\_table[(i, symbol)] = next\_state

return action, goto\_table, states, transitions

def print\_dfa(states, transitions):

print("DFA States and Transitions:")

for i, state in enumerate(states):

print(f"State {i}:")

for item in state:

print(f" {item}")

print()

print("Transitions:")

for (state, symbol), next\_state in transitions.items():

print(f" State {state} --{symbol}--> State {next\_state}")

print()

def print\_parsing\_table(action, goto\_table, grammar):

print("Action Table:")

for key in sorted(action.keys()):

print(f"{key}: {action[key]}")

print("\nGoto Table:")

for key in sorted(goto\_table.keys()):

print(f"{key}: {goto\_table[key]}")

if \_\_name\_\_ == "\_\_main\_\_":

terminals = ['a', 'b', '$']

nonterminals = ['S', 'A']

productions = [

('S', ['A', 'A']),

('A', ['a', 'A']),

('A', ['b'])

]

start\_symbol = 'S'

grammar = Grammar(terminals, nonterminals, productions, start\_symbol)

action, goto\_table, states, transitions = build\_parsing\_table(grammar)

print\_dfa(states, transitions)

print\_parsing\_table(action, goto\_table, grammar)

**Output**

