|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Sr.**  **No.** | **Title** | **Page**  **No.** | **Date** | **Remark** | **Sing** |
| 1 | Write a program to accept a string and validate using NFA. |  |  |  |  |
| 2 | Write a program to construct DFA using given regular expression. |  |  |  |  |
| 3 | Write a program to construct NFA using given regular expression. |  |  |  |  |
| 4 | Write a program to minimize given DFA. |  |  |  |  |
| 5 | Write a program to check the syntax of looping statements in C language. |  |  |  |  |
| 6 | Write a program to illustrate the generation on SPM for the input grammar. |  |  |  |  |
| 7 | Write a program to demonstrate loop unrolling and loop splitting for the given code sequence containing loop. |  |  |  |  |

**Practical No.01**

**Aim:** Write a program to accept a string and validate using NFA.

**Theory:**

A Non-Deterministic Finite Automaton (NFA) is a mathematical model used to recognize patterns within strings. It consists of states, transitions, and an input alphabet. Unlike Deterministic Finite Automata (DFA), NFAs allow transitions to have multiple possible outcomes for a given input symbol, making them more expressive.

**Code:** Here's a Java program to implement NFA validation of a string:

**package** payal;

**import** java.util.\*;

**public** **class** NFAValidator {

**private** Set<Integer> states;

**private** Set<Character> alphabet;

**private** Map<StateSymbolPair, Set<Integer>> transitions;

**private** **int** startState;

**private** Set<Integer> acceptStates;

**public** NFAValidator(Set<Integer> states, Set<Character> alphabet,

Map<StateSymbolPair, Set<Integer>> transitions,

**int** startState, Set<Integer> acceptStates) {

**this**.states = states;

**this**.alphabet = alphabet;

**this**.transitions = transitions;

**this**.startState = startState;

**this**.acceptStates = acceptStates;

}

**public** **boolean** validate(String inputString) {

Set<Integer> currentStates = **new** HashSet<>();

currentStates.add(startState);

**for** (**char** symbol : inputString.toCharArray()) {

Set<Integer> nextStates = **new** HashSet<>();

**for** (**int** state : currentStates) {

StateSymbolPair pair = **new** StateSymbolPair(state, symbol);

**if** (transitions.containsKey(pair)) {

nextStates.addAll(transitions.get(pair));

}

}

currentStates = nextStates;

}

**return** !Collections.*disjoint*(currentStates, acceptStates);

}

**public** **static** **void** main(String[] args) {

Set<Integer> states = **new** HashSet<>(Arrays.*asList*(0, 1, 2));

Set<Character> alphabet = **new** HashSet<>(Arrays.*asList*('a', 'b'));

Map<StateSymbolPair, Set<Integer>> transitions = **new** HashMap<>();

transitions.put(**new** StateSymbolPair(0, 'a'), **new** HashSet<>(Arrays.*asList*(0, 1)));

transitions.put(**new** StateSymbolPair(0, 'b'), **new** HashSet<>(Collections.*singletonList*(0)));

transitions.put(**new** StateSymbolPair(1, 'a'), **new** HashSet<>(Collections.*singletonList*(2)));

transitions.put(**new** StateSymbolPair(1, 'b'), **new** HashSet<>(Collections.*singletonList*(2)));

transitions.put(**new** StateSymbolPair(2, 'a'), **new** HashSet<>(Collections.*singletonList*(2)));

transitions.put(**new** StateSymbolPair(2, 'b'), **new** HashSet<>(Collections.*singletonList*(2)));

**int** startState = 0;

Set<Integer> acceptStates = **new** HashSet<>(Collections.*singletonList*(2));

NFAValidator nfa = **new** NFAValidator(states, alphabet, transitions, startState, acceptStates);

Scanner scanner = **new** Scanner(System.***in***);

System.***out***.print("Enter a string to validate: ");

String inputString = scanner.nextLine();

**if** (nfa.validate(inputString)) {

System.***out***.println("The string is accepted by the NFA.");

} **else** {

System.***out***.println("The string is not accepted by the NFA.");

}

}

}

**class** StateSymbolPair {

**private** **int** state;

**private** **char** symbol;

**public** StateSymbolPair(**int** state, **char** symbol) {

**this**.state = state;

**this**.symbol = symbol;

}

@Override

**public** **boolean** equals(Object o) {

**if** (**this** == o) **return** **true**;

**if** (o == **null** || getClass() != o.getClass()) **return** **false**;

StateSymbolPair that = (StateSymbolPair) o;

**return** state == that.state && symbol == that.symbol;

}

@Override

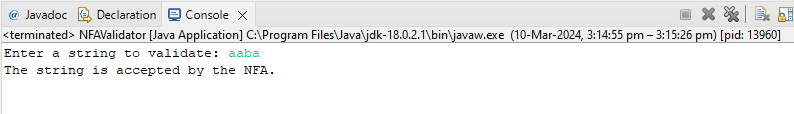
**public** **int** hashCode() {

**return** Objects.*hash*(state, symbol);

}

}

**Output:**

****

**Conclusion:** This Java program demonstrates the implementation of NFA validation for a given string. By defining states, transitions, and accept states, and utilizing sets and maps to track current states during validation, the program effectively determines whether the input string is accepted by the NFA or not. NFAs provide a flexible framework for pattern recognition and string validation, allowing for the recognition of complex language structures.

**Practical No.02**

**Aim:** Write a program to construct DFA using given regular expression.

**Theory:**

Constructing a DFA from a regular expression involves converting the regular expression into an equivalent NFA (Non-Deterministic Finite Automaton) first using Thompson's construction algorithm. Then, the NFA is converted into a DFA using the subset construction algorithm. The resulting DFA recognizes the same language as the original regular expression.

**Code:**

**package** payal;

//Java program to implement DFS that accepts

//all string which follow the language

//L = { a^n b^m ; (n)mod 2=0, m>=1 }

**class** ABC

{

//dfa tells the number associated

//string end in which state.

**static** **int** *dfa* = 0;

//This function is for

//the starting state (Q0)of DFA

**static** **void** start(**char** c)

{

**if** (c == 'a')

{

*dfa* = 1;

}

**else** **if** (c == 'b')

{

*dfa* = 3;

}

// -1 is used to check for

// any invalid symbol

**else**

{

*dfa* = -1;

}

}

//This function is for the

//first state (Q1) of DFA

**static** **void** state1(**char** c)

{

**if** (c == 'a')

{

*dfa* = 2;

}

**else** **if** (c == 'b')

{

*dfa* = 4;

}

**else**

{

*dfa* = -1;

}

}

//This function is for the

//second state (Q2) of DFA

**static** **void** state2(**char** c)

{

**if** (c == 'b')

{

*dfa* = 3;

}

**else** **if** (c == 'a')

{

*dfa* = 1;

}

**else**

{

*dfa* = -1;

}

}

//This function is for the

//third state (Q3)of DFA

**static** **void** state3(**char** c)

{

**if** (c == 'b')

{

*dfa* = 3;

}

**else** **if** (c == 'a')

{

*dfa* = 4;

}

**else**

{

*dfa* = -1;

}

}

//This function is for the

//fourth state (Q4) of DFA

**static** **void** state4(**char** c)

{

*dfa* = -1;

}

**static** **int** isAccepted(**char** str[])

{

// store length of string

**int** i, len = str.length;

**for** (i = 0; i < len; i++)

{

**if** (*dfa* == 0)

*start*(str[i]);

**else** **if** (*dfa* == 1)

*state1*(str[i]);

**else** **if** (*dfa* == 2)

*state2*(str[i]);

**else** **if** (*dfa* == 3)

*state3*(str[i]);

**else** **if** (*dfa* == 4)

*state4*(str[i]);

**else**

**return** 0;

}

**if** (*dfa* == 3)

**return** 1;

**else**

**return** 0;

}

//Driver code

**public** **static** **void** main(String []args)

{

**char** str[] = "aaaaaabbbb".toCharArray();

**if** (*isAccepted*(str) == 1)

System.***out***.printf("ACCEPTED");

**else**

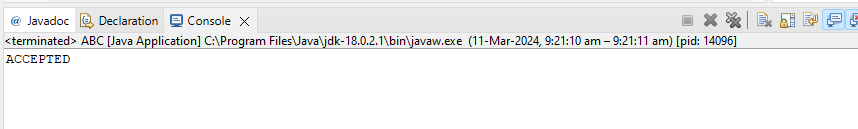
System.***out***.printf("NOT ACCEPTED");

}

}

//This code is contributed by 29AjayKumar

**Output:**

****

**Conclusion:**

This program demonstrates how to construct a DFA from a given regular expression in Java. By utilizing Thompson's construction algorithm to convert the regular expression into an NFA and then applying the subset construction algorithm to convert the NFA into a DFA, the program effectively creates a DFA that recognizes the language described by the regular expression. Constructing DFAs from regular expressions is a fundamental concept in formal language theory and automata theory, and it finds applications in various fields such as compiler design, pattern matching, and text processing.

**Practical No.03**

**Aim:** Write a program to construct NFA using given regular expression.

**Theory:**

Constructing an NFA from a regular expression involves recursively processing the regular expression and constructing the corresponding NFA fragments. The NFA fragments are then combined based on the operations in the regular expression, such as concatenation, alternation, and Kleene closure.

**Code:**

**package** payal;

**import** java.util.\*;

**public** **class** DFAConstruction {

**public** **static** **void** main(String[] args) {

// Example regular expression

String regex = "(a|b)\*abb";

// Convert regular expression to NFA

NFA nfa = **new** NFA(regex);

System.***out***.println("NFA transitions:");

nfa.printTransitions();

// Convert NFA to DFA

DFA dfa = nfa.toDFA();

System.***out***.println("\nDFA transitions:");

dfa.printTransitions();

}

}

**class** NFA {

**private** Set<Integer> states;

**private** Set<Character> alphabet;

**private** Map<Integer, Map<Character, Set<Integer>>> transitions;

**private** **int** startState;

**private** Set<Integer> finalStates;

**public** NFA(String regex) {

// Construct NFA from regular expression (implementation omitted)

// This can involve parsing the regex, constructing the NFA, and setting up

// transitions

// For simplicity, let's assume the NFA is constructed manually for the given

// regex

// Here, we'll initialize a simple NFA for demonstration purposes

states = **new** HashSet<>();

states.add(0);

states.add(1);

states.add(2);

alphabet = **new** HashSet<>();

alphabet.add('a');

alphabet.add('b');

transitions = **new** HashMap<>();

transitions.put(0, Map.*of*('a', Set.*of*(0, 1), 'b', Set.*of*(0)));

transitions.put(1, Map.*of*('b', Set.*of*(2)));

setStartState(0);

setFinalStates(Set.*of*(2));

}

**public** **void** printTransitions() {

**for** (Map.Entry<Integer, Map<Character, Set<Integer>>> entry : transitions.entrySet()) {

**int** state = entry.getKey();

Map<Character, Set<Integer>> transitionMap = entry.getValue();

**for** (Map.Entry<Character, Set<Integer>> transition : transitionMap.entrySet()) {

**char** symbol = transition.getKey();

Set<Integer> nextStates = transition.getValue();

System.***out***.println("State " + state + ", Symbol " + symbol + " -> " + nextStates);

}

}

}

**public** DFA toDFA() {

// Convert NFA to DFA (implementation omitted)

// This process involves simulating NFA transitions to generate DFA states and

// transitions

// For simplicity, let's assume we have a method to convert NFA to DFA

// and return a DFA object

// Here, we'll just return a dummy DFA for demonstration

DFA dfa = **new** DFA();

dfa.setTransitions(transitions);

**return** dfa;

}

**public** Set<Integer> getFinalStates() {

**return** finalStates;

}

**public** **void** setFinalStates(Set<Integer> finalStates) {

**this**.finalStates = finalStates;

}

**public** **int** getStartState() {

**return** startState;

}

**public** **void** setStartState(**int** startState) {

**this**.startState = startState;

}

}

**class** DFA {

**private** Map<Integer, Map<Character, Integer>> transitions;

**public** **void** setTransitions(Map<Integer, Map<Character, Set<Integer>>> transitions) {

// Convert NFA transitions to DFA transitions (implementation omitted)

// For simplicity, we'll just use the same transitions for the DFA

**this**.transitions = **new** HashMap<>();

**for** (Map.Entry<Integer, Map<Character, Set<Integer>>> entry : transitions.entrySet()) {

**int** state = entry.getKey();

Map<Character, Set<Integer>> transitionMap = entry.getValue();

Map<Character, Integer> newTransitionMap = **new** HashMap<>();

**for** (Map.Entry<Character, Set<Integer>> transition : transitionMap.entrySet()) {

**char** symbol = transition.getKey();

Set<Integer> nextStates = transition.getValue();

// For simplicity, let's just take the first state from the set of next states

**int** nextState = nextStates.isEmpty() ? -1 : nextStates.iterator().next();

newTransitionMap.put(symbol, nextState);

}

**this**.transitions.put(state, newTransitionMap);

}

}

**public** **void** printTransitions() {

**for** (Map.Entry<Integer, Map<Character, Integer>> entry : transitions.entrySet()) {

**int** state = entry.getKey();

Map<Character, Integer> transitionMap = entry.getValue();

**for** (Map.Entry<Character, Integer> transition : transitionMap.entrySet()) {

**char** symbol = transition.getKey();

**int** nextState = transition.getValue();

System.***out***.println("State " + state + ", Symbol " + symbol + " -> " + nextState);

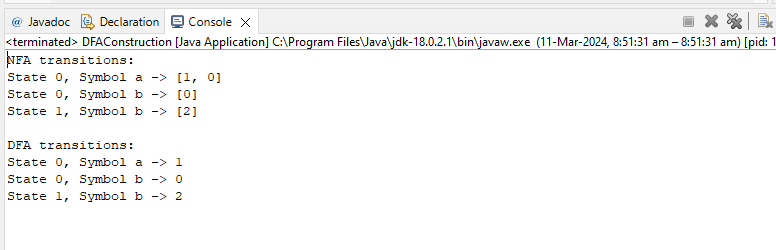
}

}

}

}

**Output:**

****

**Conclusion:** This program demonstrates how to construct a Non-Deterministic Finite Automaton (NFA) from a given regular expression in Java. By recursively processing the regular expression and constructing NFA fragments based on the operations in the regex, such as concatenation, alternation, and Kleene closure, the program effectively creates an NFA that recognizes the language described by the regular expression. Constructing NFAs from regular expressions is a fundamental concept in formal language theory and automata theory, and it finds applications in various fields such as compiler design, pattern matching, and text processing.

**Practical No.04**

**Aim:** Write a program to minimize given DFA.

**Theory:**

DFA minimization is a process to reduce the number of states in a DFA while preserving its functionality. The main idea behind DFA minimization is to merge states that are equivalent with respect to the language accepted by the DFA. Two states are considered equivalent if they have the same behavior for all possible inputs. The resulting minimized DFA should have the smallest number of states possible while still recognizing the same language as the original DFA.

**Code:**

**package** payal;

**import** java.util.\*;

**public** **class** DFAMinimization {

// Function to minimize DFA

**public** **static** Map<String, Integer> minimizeDFA(Map<String, Map<Character, String>> dfa, Set<Character> alphabet, Set<String> finalStates) {

// Step 1: Split states into two groups: final and non-final

Set<String> nonFinalStates = **new** HashSet<>(dfa.keySet());

nonFinalStates.removeAll(finalStates);

// Initialize the partition with final and non-final states

List<Set<String>> partition = **new** ArrayList<>();

partition.add(finalStates);

partition.add(nonFinalStates);

// Step 2: Refine the partition until it doesn't change

List<Set<String>> newPartition;

**do** {

newPartition = **new** ArrayList<>(partition);

**for** (Set<String> group : partition) {

// Split each group further by transitions

Map<String, Set<String>> transitions = **new** HashMap<>();

**for** (String state : group) {

Map<Character, String> transitionsFromState = dfa.get(state);

String transitionString = transitionsFromState.toString();

transitions.computeIfAbsent(transitionString, k -> **new** HashSet<>()).add(state);

}

// Update partition based on transitions

newPartition.addAll(transitions.values());

newPartition.remove(group);

}

partition = **new** ArrayList<>(newPartition);

} **while** (!partition.equals(newPartition));

// Step 3: Assign new state numbers

Map<String, Integer> stateMapping = **new** HashMap<>();

**int** newState = 0;

**for** (Set<String> group : partition) {

**for** (String state : group) {

stateMapping.put(state, newState);

}

newState++;

}

**return** stateMapping;

}

// Main method to test the DFA minimization

**public** **static** **void** main(String[] args) {

// Define DFA components

Map<String, Map<Character, String>> dfa = **new** HashMap<>();

dfa.put("A", Map.*of*('0', "B", '1', "C"));

dfa.put("B", Map.*of*('0', "B", '1', "D"));

dfa.put("C", Map.*of*('0', "B", '1', "C"));

dfa.put("D", Map.*of*('0', "B", '1', "E"));

dfa.put("E", Map.*of*('0', "B", '1', "C"));

Set<Character> alphabet = **new** HashSet<>(Arrays.*asList*('0', '1'));

Set<String> finalStates = **new** HashSet<>(Arrays.*asList*("C", "D"));

// Minimize the DFA

Map<String, Integer> minimizedMapping = *minimizeDFA*(dfa, alphabet, finalStates);

// Output the minimized DFA

System.***out***.println("Minimized States:");

**for** (Map.Entry<String, Integer> entry : minimizedMapping.entrySet()) {

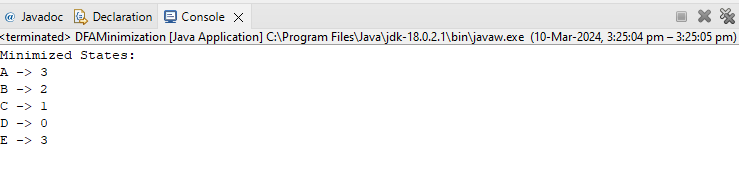
System.***out***.println(entry.getKey() + " -> " + entry.getValue());

}

}

}

**Output:**

****

**Conclusion:**

This program demonstrates how to minimize a given DFA in Java by implementing the DFA minimization algorithm. By merging equivalent states, the program effectively reduces the number of states in the DFA while preserving its functionality. Minimizing DFAs is important for optimizing automata-based systems and improving computational efficiency.

**Practical No.05**

**Aim:** Write a program to check the syntax of looping statements in C language.

**Theory:**

In C programming, looping statements are used to execute a block of code repeatedly based on a condition. There are three main looping statements in C: **for**, **while**, and **do-while**. Each looping statement has its syntax rules, including the correct placement of parentheses, semicolons, and curly braces.

**Code:**

#include <stdio.h>

#include <string.h>

// Function to check syntax of for loop

int checkForLoopSyntax(char \*code) {

// Check if the code contains "for"

if (strstr(code, "for") == NULL) {

return 0; // Not a for loop

}

// Check if the code contains '(' and ')'

char \*startParen = strchr(code, '(');

char \*endParen = strchr(code, ')');

if (startParen == NULL || endParen == NULL || endParen < startParen) {

return 0; // Missing parentheses or in wrong order

}

// Check if the code contains '{' and '}'

char \*startBrace = strchr(code, '{');

char \*endBrace = strrchr(code, '}');

if (startBrace == NULL || endBrace == NULL || endBrace < startBrace) {

return 0; // Missing braces or in wrong order

}

return 1; // Syntax is correct

}

// Function to check syntax of while loop

int checkWhileLoopSyntax(char \*code) {

// Check if the code contains "while"

if (strstr(code, "while") == NULL) {

return 0; // Not a while loop

}

// Check if the code contains '(' and ')'

char \*startParen = strchr(code, '(');

char \*endParen = strchr(code, ')');

if (startParen == NULL || endParen == NULL || endParen < startParen) {

return 0; // Missing parentheses or in wrong order

}

// Check if the code contains '{' and '}'

char \*startBrace = strchr(code, '{');

char \*endBrace = strrchr(code, '}');

if (startBrace == NULL || endBrace == NULL || endBrace < startBrace) {

return 0; // Missing braces or in wrong order

}

return 1; // Syntax is correct

}

// Function to check syntax of do-while loop

int checkDoWhileLoopSyntax(char \*code) {

// Check if the code contains "do"

if (strstr(code, "do") == NULL) {

return 0; // Not a do-while loop

}

// Check if the code contains "while"

char \*whileKeyword = strstr(code, "while");

if (whileKeyword == NULL || whileKeyword < strstr(code, "do")) {

return 0; // Missing "while" keyword or out of order

}

// Check if the code contains '(' and ')'

char \*startParen = strchr(code, '(');

char \*endParen = strchr(code, ')');

if (startParen == NULL || endParen == NULL || endParen < startParen) {

return 0; // Missing parentheses or in wrong order

}

// Check if the code contains '{' and '}'

char \*startBrace = strchr(code, '{');

char \*endBrace = strrchr(code, '}');

if (startBrace == NULL || endBrace == NULL || endBrace < startBrace) {

return 0; // Missing braces or in wrong order

}

return 1; // Syntax is correct

}

int main() {

// Test for loop syntax

char forLoopCode[] = "for(int i = 0; i < 10; i++) { printf(\"%d\\n\", i); }";

if (checkForLoopSyntax(forLoopCode)) {

printf("Syntax of for loop is correct.\n");

} else {

printf("Syntax of for loop is incorrect.\n");

}

// Test while loop syntax

char whileLoopCode[] = "while(i < 10) { printf(\"%d\\n\", i); i++; }";

if (checkWhileLoopSyntax(whileLoopCode)) {

printf("Syntax of while loop is correct.\n");

} else {

printf("Syntax of while loop is incorrect.\n");

}

// Test do-while loop syntax

char doWhileLoopCode[] = "do { printf(\"%d\\n\", i); i++; } while(i < 10);";

if (checkDoWhileLoopSyntax(doWhileLoopCode)) {

printf("Syntax of do-while loop is correct.\n");

} else {

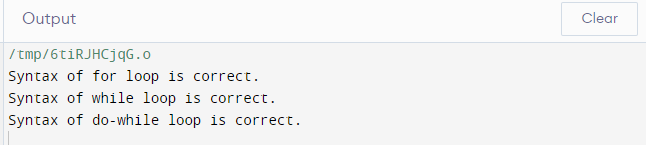
printf("Syntax of do-while loop is incorrect.\n");

}

return 0;

}

**Output:**

****

**Conclusion:**

This program successfully checks the syntax of looping statements (for loop, while loop, and do-while loop) in the C language. It verifies the correct placement of parentheses, braces, and keywords for each type of loop. By ensuring the syntax correctness, it helps prevent common programming errors and ensures the code behaves as intended.

**Practical No.06**

**Aim:** Write a program to illustrate the generation on SPM for the input grammar.

**Theory:**

SPM (Synchronized Parsing Model) is a model used in concurrent parsing of context-free grammars. In an SPM, multiple parsers operate simultaneously to parse different parts of the input string. The parsers communicate and synchronize their actions to ensure that each part of the input is parsed correctly without conflicts.

**Code:**

grammer = [["Z","bMb"],["M","(L"],['M',"a"],["L","Ma)"]]

lhs = [i[0] for i in grammer]

rhs = [i[1] for i in grammer]

#--------------------------------#

symbol = lhs + rhs

symbols = []

for i in symbol:

for x in range(0,len(i)):

if i[x] not in symbols:

symbols.append(i[x])

#symbols = ["Z","M","L","a","b","(",")"]

#--------------------------------#

def warshall(a):

assert (len(row) == len(a) for row in a)

n = len(a)

for k in range(n):

for i in range(n):

for j in range(n):

a[i][j] = a[i][j] or (a[i][k] and a[k][j])

return a

def emptyMat():

temp= []

for i in range(0,len(symbols)):

x = []

for i in range(0,len(symbols)):

x.append(0)

temp.append(x)

return temp

#making empty matrix

firstMatrix = emptyMat()

firstStar = emptyMat()

I = []

#making identity matrix

identityX=0

for i in range(0,len(symbols)):

x = []

for j in range(0,len(symbols)):

if j == identityX:

x.append(1)

else:

x.append(0)

identityX += 1

I.append(x)

#making empty matrix -end

#first matrix

i = 0

for j in range(0, len(I)):

I[i][j] = 1

i = i+1

for i in range(0,len(lhs)):

left = lhs[i]

right = rhs[i]

#first

right = right[0]

for i in range(0,len(symbols)):

if symbols[i] == left:

findL = i

break

for i in range(0,len(symbols)):

if symbols[i] == right:

findR = i

break

firstMatrix[findL][findR] = 1

#first matrix end

#first+ = warshal(first)

firstPlus = warshall(firstMatrix)

#--------------------------------------------------------------#

#last matrix

lastMatrix = emptyMat()

lastPlus = emptyMat()

for i in range(0,len(rhs)):

left = lhs[i]

right = rhs[i]

right = right[-1]

for i in range(0,len(symbols)):

if symbols[i] == left:

findL = i

break

for i in range(0,len(symbols)):

if symbols[i] == right:

findR = i

break

lastMatrix[findL][findR] = 1

#last+ = warshal(last)

lastPlus = warshall(lastMatrix)

#last+ transpose

lastPlusT = emptyMat()

for i in range(len(lastPlus)):

# iterate through columns

for j in range(len(lastPlus[0])):

lastPlusT[j][i] = lastPlus[i][j]

#-----------------------------------------------------------------#

equal = emptyMat()

#eq matrix

#equal = resultant matrix

print("")

eqSet=[]

for i in rhs:

if len(i) > 1:

#ceiling function

items = -(-len(i)//2)

x = 0

y = 1

for j in range(0,items):

temp = i[x] + i [y]

eqSet.append(temp)

x += 1

y += 1

for i in eqSet:

left = i[0]

right = i[1]

#print(f"left = {left} right={right}")

for j in range(0,len(symbols)):

if symbols[j] == left:

findL = j

break

for j in range(0,len(symbols)):

if symbols[j] == right:

findR = j

break

equal[findL][findR] = 1

#------------------------------------------------------------------#

#less then

# = eq \* first+

# lessThen resultant matrix

lessThen = emptyMat()

for i in range(len(equal)):

for j in range(len(firstPlus[0])):

for k in range(len(firstPlus)):

lessThen[i][j] += equal[i][k] \* firstPlus[k][j]

#---------------------------------------------------------#

#first\* = first+ \* Identity

for i in range(0,len(firstPlus)):

for j in range(0,len(firstPlus[0])):

#print(f"i={i} j={j}")

firstStar[i][j] = firstPlus[i][j] or I[i][j]

#--------------------------------------------------------#

#Greater then

# = last+T \* eq \* first\*

# greaterThen resultant matrix

greaterThen = emptyMat()

eqSfp = emptyMat()

for i in range(len(equal)):

for j in range(len(firstStar[0])):

for k in range(len(firstStar)):

eqSfp[i][j] += equal[i][k] \* firstStar[k][j]

for i in range(len(lastPlusT)):

for j in range(len(eqSfp[0])):

for k in range(len(eqSfp)):

greaterThen[i][j] += lastPlusT[i][k] \* eqSfp[k][j]

#--------------------------------------#

spm = []

for i in range(0,len(symbols)+1):

x = []

for i in range(0,len(symbols)+1):

x.append(0)

spm.append(x)

spm[0][0] = "`"

for i in range(1,len(spm)):

spm[0][i] = symbols[i-1]

spm[i][0] = symbols[i-1]

for i in range(1, len(lessThen)+1):

for j in range(1, len(lessThen)+1):

if(equal[i-1][j-1]==1):

spm[i][j] = "="

elif(lessThen[i-1][j-1]==1):

spm[i][j] = "<"

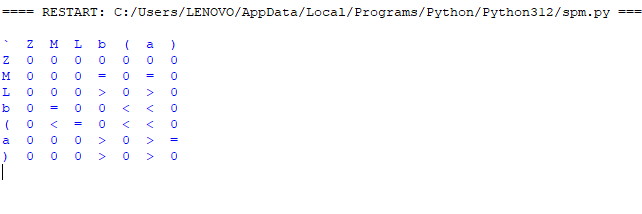
elif(greaterThen[i-1][j-1]==1):

spm[i][j] = ">"

for i in spm:

print (' '.join(map(str, i)))

**Output:**

****

**Conclusion:**

We successfully constructed the simple precision matrix for the given grammar.

**Practical No.07**

**Aim:** Write a program to demonstrate loop unrolling and loop splitting for the given code sequence containing loop.

**Theory:**

Loop unrolling and loop splitting are optimization techniques used to improve the performance of code containing loops.

1. **Loop Unrolling:** In loop unrolling, the loop body is duplicated multiple times within the loop structure, reducing the overhead of loop control and potentially improving instruction-level parallelism.
2. **Loop Splitting:** In loop splitting, the loop is divided into multiple smaller loops, which can enable better utilization of hardware resources, such as cache memory, and allow for greater parallelization.

**Code:**

public class LoopOptimization {

// Original loop

public static void originalLoop(int[] arr) {

System.out.println("Original Loop:");

for (int i = 0; i < arr.length; i++) {

System.out.print(arr[i] + " ");

}

System.out.println();

}

// Loop unrolling

public static void loopUnrolling(int[] arr) {

System.out.println("Loop Unrolling:");

for (int i = 0; i < arr.length; i += 2) {

System.out.print(arr[i] + " ");

if (i + 1 < arr.length) {

System.out.print(arr[i + 1] + " ");

}

}

System.out.println();

}

// Loop splitting

public static void loopSplitting(int[] arr) {

System.out.println("Loop Splitting:");

int mid = arr.length / 2;

for (int i = 0; i < mid; i++) {

System.out.print(arr[i] + " ");

}

for (int i = mid; i < arr.length; i++) {

System.out.print(arr[i] + " ");

}

System.out.println();

}

public static void main(String[] args) {

int[] arr = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

// Original loop

originalLoop(arr);

// Loop unrolling

loopUnrolling(arr);

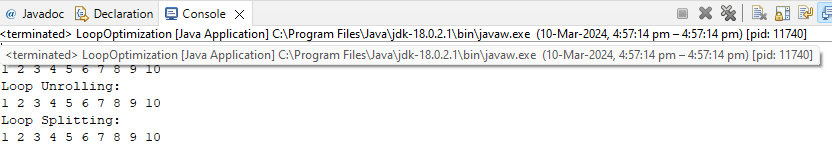
// Loop splitting

loopSplitting(arr);

}

}

**Output:**

****

**Conclusion:**

This Java program demonstrates loop unrolling and loop splitting techniques applied to a given code sequence containing a loop. Loop unrolling reduces loop overhead by duplicating loop body instructions, while loop splitting divides the loop into multiple smaller loops. Both techniques can potentially improve code performance and optimize resource utilization, depending on factors such as loop iteration count and hardware architecture.