INDEX

|  |  |  |  |
| --- | --- | --- | --- |
| **Sr.No** | **Date** | **Practicals** | **Signature** |
| 1 |  | Write a program to convert the given NDFA to DFA |  |
| 2 |  | Write a program to convert the given Right Linear Grammar to Left Linear Grammar form. |  |
| 3 |  | Write a program to illustrate the generation on SPM for the input grammar. |  |
| 4 |  | Write a program to illustrate the generation on OPM for the input operator Grammar |  |
|  |  |  |  |

**PRACTICAL NO: 1**

**Aim:** Write a program to convert the given NDFA to DFA

**INPUT:-**

class NDFA:

def \_\_init\_\_(self, states, alphabet, start\_state, accept\_states, transitions):

self.states = states

self.alphabet = alphabet

self.start\_state = start\_state

self.accept\_states = accept\_states

self.transitions = transitions

class DFA:

def \_\_init\_\_(self, states, alphabet, start\_state, accept\_states, transitions):

self.states = states

self.alphabet = alphabet

self.start\_state = start\_state

self.accept\_states = accept\_states

self.transitions = transitions

def ndfa\_to\_dfa(ndfa):

start\_state = frozenset(epsilon\_closure(ndfa, {ndfa.start\_state}))

dfa\_states, unprocessed\_states = {start\_state}, [start\_state]

dfa\_transitions, dfa\_accept\_states = {}, set()

while unprocessed\_states:

current\_state = unprocessed\_states.pop()

dfa\_transitions[current\_state] = {}

if any(state in ndfa.accept\_states for state in current\_state):

dfa\_accept\_states.add(current\_state)

for symbol in ndfa.alphabet:

next\_state = frozenset(epsilon\_closure(ndfa, move(ndfa, current\_state, symbol)))

dfa\_transitions[current\_state][symbol] = next\_state

if next\_state not in dfa\_states:

dfa\_states.add(next\_state)

unprocessed\_states.append(next\_state)

return DFA(dfa\_states, ndfa.alphabet, start\_state, dfa\_accept\_states, dfa\_transitions)

def epsilon\_closure(ndfa, states):

closure, stack = set(states), list(states)

while stack:

state = stack.pop()

if (state, '') in ndfa.transitions:

for next\_state in ndfa.transitions[(state, '')]:

if next\_state not in closure:

closure.add(next\_state)

stack.append(next\_state)

return closure

def move(ndfa, states, symbol):

next\_states = set()

for state in states:

next\_states.update(ndfa.transitions.get((state, symbol), []))

return next\_states

# Example NDFA definition

ndfa = NDFA(

{'q0', 'q1', 'q2'}, # States

{'a', 'b'}, # Alphabet

'q0', # Start state

{'q2'}, # Accept states

{ # Transitions

('q0', 'a'): {'q0', 'q1'},

('q0', ''): {'q1'},

('q1', 'b'): {'q2'},

('q2', 'a'): {'q2'}

}

)

# Convert the NDFA to DFA

dfa = ndfa\_to\_dfa(ndfa)

# Print DFA details

print("DFA States:", dfa.states)

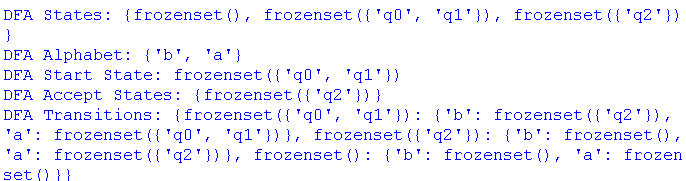
print("DFA Alphabet:", dfa.alphabet)

print("DFA Start State:", dfa.start\_state)

print("DFA Accept States:", dfa.accept\_states)

print("DFA Transitions:", dfa.transitions)

**OUTPUT:-**

****

**PRACTICAL NO: 2**

**Aim:** Write a program to convert the given Right Linear Grammar to Left Linear Grammar form.

**INPUT:-**

class Grammar:

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

self.non\_terminals = non\_terminals

self.terminals = terminals

self.start\_symbol = start\_symbol

self.productions = productions

def reverse\_production(production):

"""

Reverses the right-hand side of a production.

For example, A -> aB becomes A -> Ba

"""

lhs, rhs = production

reversed\_rhs = ''.join(reversed(rhs))

return lhs, reversed\_rhs

def convert\_rlg\_to\_llg(rlg):

"""

Converts a Right Linear Grammar (RLG) to a Left Linear Grammar (LLG).

"""

llg\_productions = []

for production in rlg.productions:

lhs, rhs = production

# Reverse each production's right-hand side

new\_rhs = ''.join(reversed(rhs))

# If the new right-hand side starts with a terminal followed by a non-terminal, swap them

if len(new\_rhs) == 2 and new\_rhs[0] in rlg.terminals and new\_rhs[1] in rlg.non\_terminals:

new\_rhs = new\_rhs[1] + new\_rhs[0]

llg\_productions.append((lhs, new\_rhs))

return Grammar(rlg.non\_terminals, rlg.terminals, rlg.start\_symbol, llg\_productions)

# Example Right Linear Grammar

rlg = Grammar(

non\_terminals={'S', 'A'},

terminals={'a', 'b'},

start\_symbol='S',

productions=[

('S', 'aS'),

('S', 'bA'),

('S', 'a'),

('A', 'aA'),

('A', 'b')

]

)

# Convert RLG to LLG

llg = convert\_rlg\_to\_llg(rlg)

# Print LLG details

print("LLG Non-terminals:", llg.non\_terminals)

print("LLG Terminals:", llg.terminals)

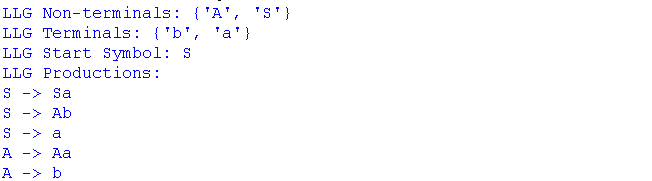
print("LLG Start Symbol:", llg.start\_symbol)

print("LLG Productions:")

for production in llg.productions:

print(f"{production[0]} -> {production[1]}")

**OUTPUT:-**

****

**PRACTICAL NO: 3**

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

**INPUT:-**

# Define a class to represent the grammar

class Grammar:

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

self.non\_terminals = non\_terminals

self.terminals = terminals

self.start\_symbol = start\_symbol

self.productions = productions

# Function to compute the first set for a given symbol

def compute\_first(symbol, productions, first\_sets):

if symbol not in first\_sets:

first\_sets[symbol] = set()

if symbol in grammar.terminals:

first\_sets[symbol].add(symbol)

return first\_sets[symbol]

for production in productions:

if production[0] == symbol:

rhs = production[1]

if rhs == '':

first\_sets[symbol].add('')

else:

for sym in rhs:

first\_sets[symbol].update(compute\_first(sym, productions, first\_sets))

if '' not in first\_sets[sym]:

break

return first\_sets[symbol]

# Function to compute the follow set for a given symbol

def compute\_follow(symbol, productions, follow\_sets, first\_sets, start\_symbol):

if symbol not in follow\_sets:

follow\_sets[symbol] = set()

if symbol == start\_symbol:

follow\_sets[symbol].add('$') # End of input marker

for production in productions:

lhs, rhs = production

for i, sym in enumerate(rhs):

if sym == symbol:

if i + 1 < len(rhs):

next\_sym = rhs[i + 1]

follow\_sets[symbol].update(first\_sets[next\_sym] - {''})

if '' in first\_sets[next\_sym]:

follow\_sets[symbol].update(compute\_follow(lhs, productions, follow\_sets, first\_sets,

start\_symbol))

else:

follow\_sets[symbol].update(compute\_follow(lhs, productions, follow\_sets, first\_sets,

start\_symbol))

return follow\_sets[symbol]

# Function to generate the Simple Precedence Matrix (SPM)

def generate\_spm(grammar):

first\_sets = {}

follow\_sets = {}

for non\_terminal in grammar.non\_terminals:

compute\_first(non\_terminal, grammar.productions, first\_sets)

for non\_terminal in grammar.non\_terminals:

compute\_follow(non\_terminal, grammar.productions, follow\_sets, first\_sets,

grammar.start\_symbol)

spm = {}

symbols = list(grammar.non\_terminals) + list(grammar.terminals) + ['$']

for sym1 in symbols:

spm[sym1] = {}

for sym2 in symbols:

spm[sym1][sym2] = ' '

for lhs, rhs in grammar.productions:

for i in range(len(rhs) - 1):

if rhs[i] in grammar.terminals and rhs[i+1] in grammar.terminals:

spm[rhs[i]][rhs[i+1]] = '='

if rhs[i] in grammar.terminals and rhs[i+1] in grammar.non\_terminals:

for sym in first\_sets[rhs[i+1]]:

if sym:

spm[rhs[i]][sym] = '<'

if rhs[i] in grammar.non\_terminals and rhs[i+1] in grammar.terminals:

for sym in follow\_sets[rhs[i]]:

if sym:

spm[sym][rhs[i+1]] = '>'

return spm

# Example Grammar

non\_terminals = {'S', 'A'}

terminals = {'a', 'b'}

start\_symbol = 'S'

productions = [

('S', 'aA'),

('A', 'b'),

('A', '')

]

grammar = Grammar(non\_terminals, terminals, start\_symbol, productions)

# Generate the Simple Precedence Matrix

spm = generate\_spm(grammar)

# Print the SPM

print("Simple Precedence Matrix:")

symbols = list(non\_terminals) + list(terminals) + ['$']

print(" ", " ".join(symbols))

for sym1 in symbols:

row = [sym1]

for sym2 in symbols:

row.append(spm[sym1][sym2])

print(" ".join(row))

**OUTPUT:-**



**PRACTICAL NO: 4**

**Aim:** Write a program to illustrate the generation on OPM for the input operator Grammar

**INPUT:-**

# Define operator precedence and associativity

precedence = {

'+': 1,

'-': 1,

'\*': 2,

'/': 2,

'^': 3

}

associativity = {

'+': 'left',

'-': 'left',

'\*': 'left',

'/': 'left',

'^': 'right'

}

# Function to generate Operator Precedence Matrix (OPM)

def generate\_opm(operators):

n = len(operators)

opm = [[None] \* n for \_ in range(n)] # Matrix size is n x n now

# Fill diagonal with '='

for i in range(n):

opm[i][i] = '='

# Fill matrix based on precedence and associativity

for i in range(n):

for j in range(n):

if i == j:

continue

op\_i = operators[i]

op\_j = operators[j]

if precedence[op\_i] > precedence[op\_j]:

opm[i][j] = '>'

elif precedence[op\_i] < precedence[op\_j]:

opm[i][j] = '<'

else: # Same precedence, check associativity

if associativity[op\_i] == 'left':

opm[i][j] = '>'

else:

opm[i][j] = '<'

return opm

# Example usage:

operators = ['+', '-', '\*', '/', '^']

# Generate Operator Precedence Matrix

opm = generate\_opm(operators)

# Print Operator Precedence Matrix

print("Operator Precedence Matrix (OPM):")

header\_row = ' ' + ' '.join(operators)

print(header\_row)

print('-' \* len(header\_row))

for i in range(len(opm)):

row = operators[i] + ' '

for j in range(len(opm[i])):

if opm[i][j] is None:

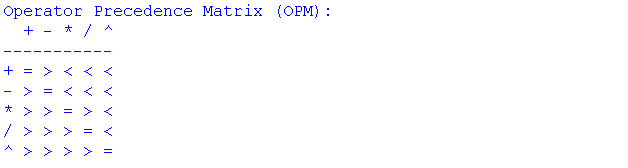
row += ' ' # Add empty space for None values

else:

row += opm[i][j] + ' '

print(row)

**OUTPUT:**

****