**TEST CASE 1 (simple)**

S->AA

A->aA

A->b

**TEST CASE 2 (epsilon production)**

S->A

A->aA

A->ε

**TEST CASE 3 (different start symbol)**

E->T+E

E->T

T->i

**TEST CASE 4 (epsilon production)**

S->AaAb

S->BbBa

A->ε

B->ε

**TEST CASE 5 (left recursive)**

S -> S + S

S -> a

**TEST CASE 6 (grammar with conflicts)**

S->A

S->B

A->a

B->a

**CODE**

import termtables as tt

import graphviz

# Adds a dot after "->" in a production rule string.

def append\_dot(a):

if "ε" in a:

return a # Epsilon productions don't need a dot inside

return a.replace("->", "->.")

# Computes the closure of an item set for a given grammar, handling epsilon.

def closure(a, prod):

temp = [a]

for it in temp:

if '.' not in it:

continue # Skip items that don't have a dot

pos = it.index(".")

if pos != len(it) - 1:

jj = it[pos + 1]

if jj.isupper(): # Check if non-terminal

for k in prod:

if k.startswith(jj + "->") and append\_dot(k) not in temp:

if k.endswith("->ε") or k.endswith("->.ε"):

temp.append(k) # Directly add epsilon production

else:

temp.append(append\_dot(k))

return temp

# Swaps the dot's position in an item to the next position.

def swap(new, pos):

new = list(new)

if pos != len(new) - 1:

new[pos], new[pos + 1] = new[pos + 1], new[pos]

return "".join(new)

return "".join(new)

# Extracts terminal symbols from the grammar.

def get\_terminals(gram):

terms = set()

for p in gram:

x1 = p.split('->')

for t in x1[1].strip():

if not t.isupper() and t != '.' and t != '' and t != 'ε':

terms.add(t)

terms.add('$')

return terms

# Extracts non-terminal symbols from the grammar.

def get\_non\_terminals(gram):

non\_terms = set()

for p in gram:

non\_terms.add(p.split('->')[0])

return non\_terms

# Calculates the goto set for an item and a symbol.

def goto1(x1, prod):

if '.' not in x1:

return [] # Return an empty list if the dot is not found to avoid errors

pos = x1.index(".")

if pos != len(x1) - 1:

kk = swap(x1, pos)

return closure(kk, prod)

return []

# Main code to ensure only valid items are processed

if \_\_name\_\_ == '\_\_main\_\_':

grammar\_file = input("Enter the grammar file path (e.g., 'grammar.txt'): ")

prod = []

set\_of\_items = []

c = []

with open(grammar\_file, 'r') as fp:

for line in fp.readlines():

prod.append(line.strip())

start\_symbol = prod[0].split('->')[0]

augmented\_start = f"X->{start\_symbol}"

prod.insert(0, append\_dot(augmented\_start))

print("---------------------------------------------------------------")

print("Augmented Grammar")

print(prod)

prod\_num = {prod[i]: i for i in range(1, len(prod))}

j = closure(append\_dot(augmented\_start), prod)

set\_of\_items.append(j)

state\_numbers = {}

dfa\_prod = {}

items = 0

while set\_of\_items:

jk = set\_of\_items.pop(0)

c.append(jk)

state\_numbers[str(jk)] = items

items += 1

symbols = sorted(get\_terminals(prod).union(get\_non\_terminals(prod)), key=lambda x: (x != 'S', x))

for sym in symbols:

new\_items = set()

for item in jk:

if '.' in item:

pos = item.index('.')

if pos + 1 < len(item) and item[pos + 1] == sym:

moved\_item = swap(item, pos)

new\_items.update(closure(moved\_item, prod))

if new\_items:

new\_items = list(new\_items)

if new\_items not in c and new\_items not in set\_of\_items:

set\_of\_items.append(new\_items)

dfa\_prod[f"{state\_numbers[str(jk)]} {sym}"] = new\_items

print("---------------------------------------------------------------")

print("Total States: ", len(c))

for i, items in enumerate(c):

print(f"State {i}:")

for item in items:

print(f" {item}")

print("---------------------------------------------------------------")

print("---------------------------------------------------------------")

print("Flow of the States (Transitions):")

for key, value in dfa\_prod.items():

current\_state, transition = key.split()

next\_state = state\_numbers.get(str(value), None)

if next\_state is not None:

print(f"From State {current\_state} on symbol {transition} -> Go to State {next\_state}")

print("---------------------------------------------------------------")

# Generate Parsing Table with Reduce Moves and Accept State

table = []

term = sorted(list(get\_terminals(prod)))

non\_term = sorted(list(get\_non\_terminals(prod) - {'X'}))

header = [''] + term + non\_term

table.append(header)

table\_dic = {}

for i in range(len(c)):

data = [''] \* (len(term) + len(non\_term))

samp = {}

# Action

for item in c[i]:

if '.' in item and item.index('.') == len(item) - 1: # If dot is at the end

# This indicates a reduce move

production = item.replace('.', '')

if production == "X->S": # Check for Accept state

data[term.index('$')] = 'Accept'

samp['$'] = 'Accept'

elif production in prod\_num:

reduce\_index = prod\_num[production]

for t in term:

data[term.index(t)] = f'r{reduce\_index}' # Add reduce move

samp[t] = f'r{reduce\_index}'

for j in dfa\_prod:

current\_state, symbol = j.split()

if int(current\_state) == i:

if symbol in term:

data[term.index(symbol)] = 'S' + str(state\_numbers.get(str(dfa\_prod[j]), ''))

samp[symbol] = 'S' + str(state\_numbers.get(str(dfa\_prod[j]), ''))

# Goto

for j in dfa\_prod:

current\_state, symbol = j.split()

if int(current\_state) == i and symbol in non\_term:

data[len(term) + non\_term.index(symbol)] = str(state\_numbers.get(str(dfa\_prod[j]), ''))

samp[symbol] = str(state\_numbers.get(str(dfa\_prod[j]), ''))

table\_dic[i] = samp

table.append([i] + data)

# Print parsing table using termtables

final\_table = tt.to\_string(data=table, style=tt.styles.ascii\_thin\_double, padding=(0, 1))

print("\nParsing Table:\n")

print(final\_table)

# Printing DFA

def visualize\_dfa(dfa\_prod, state\_numbers):

dot = graphviz.Digraph(format='png')

dot.attr(rankdir='LR', size='8,5')

dot.attr('node', shape='circle')

for key, value in dfa\_prod.items():

current\_state, transition = key.split()

next\_state = state\_numbers.get(str(value), None)

if next\_state is not None:

dot.edge(f'S{current\_state}', f'S{next\_state}', label=transition)

dot.render('dfa', view=True, cleanup=True)

print("DFA has been generated and saved as 'dfa.png'.")

visualize\_dfa(dfa\_prod, state\_numbers)