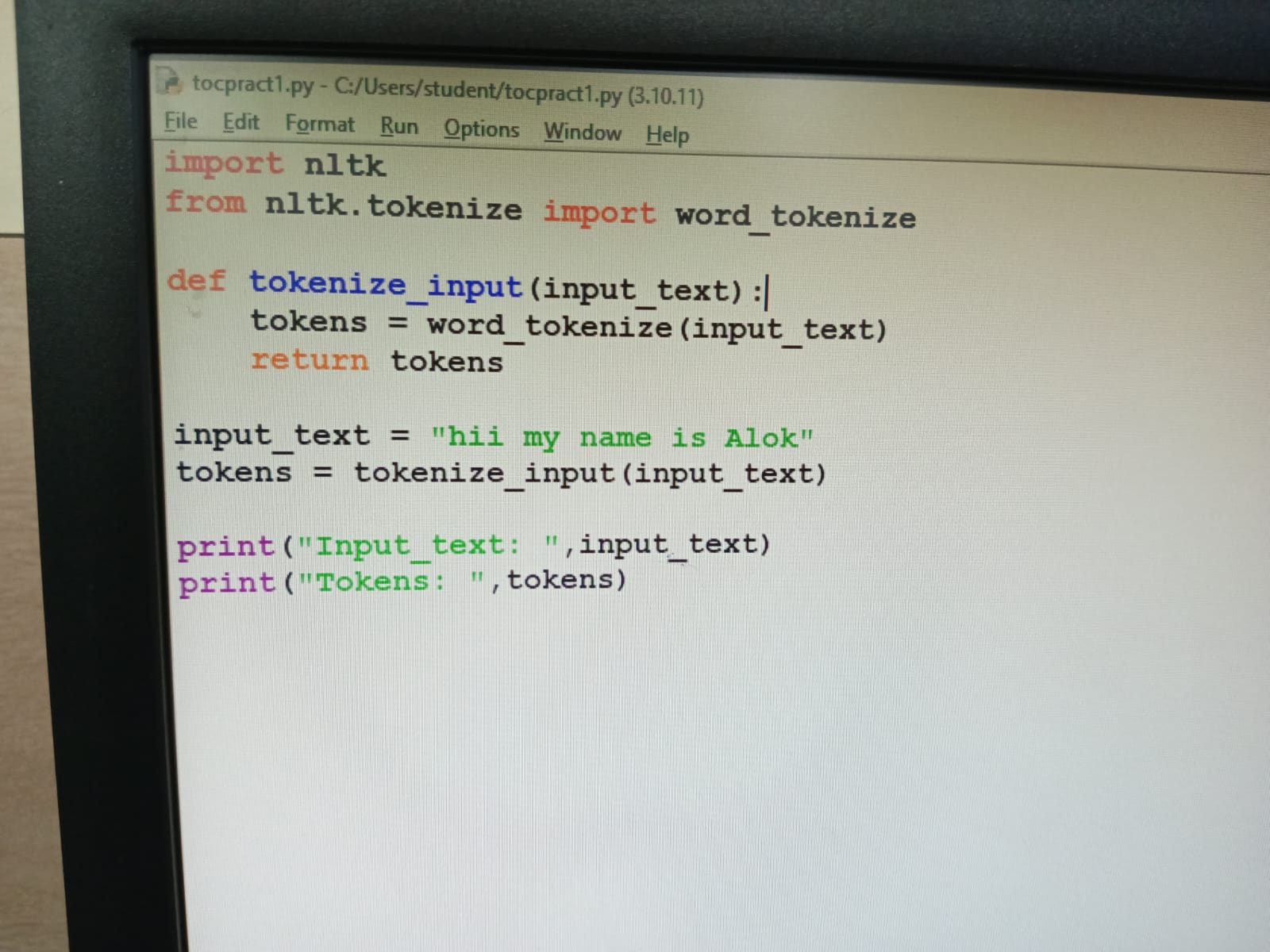
Theory of Computation

Practical 1



exp 2 :

import re

def generate\_regex\_from\_grammar(grammar):

regex\_dict = {}

for rule in grammar:

non\_terminal, production = rule.split(' -> ')

if non\_terminal not in regex\_dict:

regex\_dict[non\_terminal] = set()

for symbol in production.split('|'):

# Handle terminals (characters)

if symbol.isalpha() and len(symbol) == 1:

regex\_dict[non\_terminal].add(re.escape(symbol))

# Handle non-terminals

else:

regex\_dict[non\_terminal].add(symbol)

# Convert the sets to regular expressions

for non\_terminal, expressions in regex\_dict.items():

regex\_dict[non\_terminal] = '|'.join(expressions)

return regex\_dict

# Example usage:

grammar = [

"S -> aA|bB",

"A -> a|cA",

"B -> b|d"

]

regex\_dict = generate\_regex\_from\_grammar(grammar)

# Output the generated regular expressions

for non\_terminal, regex in regex\_dict.items():

print(f"{non\_terminal} -> {regex}")

exp 3 :

def generate\_derivation(grammar, start\_symbol, sequence, max\_depth=10):

derivation\_sequences = []

def derive(current\_sequence, depth):

if depth == 0:

return

for rule in grammar:

non\_terminal, production = rule.split(' -> ')

alternatives = production.split('|')

for alt in alternatives:

if current\_sequence.startswith(alt):

new\_sequence = current\_sequence.replace(alt, non\_terminal, 1)

derivation\_sequences.append(new\_sequence)

derive(new\_sequence, depth - 1)

derive(sequence, max\_depth)

return derivation\_sequences

# Example usage:

grammar = [

"S -> AB",

"A -> a|ε",

"B -> b"

]

start\_symbol = 'S'

target\_sequence = 'ab'

derivation\_sequences = generate\_derivation(grammar, start\_symbol, target\_sequence)

# Output the generated derivation sequences

for i, deriv\_seq in enumerate(derivation\_sequences, 1):

print(f"Derivation Sequence {i}: {deriv\_seq}")

exp 4 :

def accepts\_three\_consecutive\_ones(input\_str):

current\_state = 'q0'

for symbol in input\_str:

if current\_state == 'q0' and symbol == '1':

current\_state = 'q1'

elif current\_state == 'q1' and symbol == '1':

current\_state = 'q2'

elif current\_state == 'q2' and symbol == '1':

return True

else:

current\_state = 'q0'

return False

# Example usage:

input\_str = input("Enter a binary string: ")

result = accepts\_three\_consecutive\_ones(input\_str)

if result:

print("Accepted: The string contains three consecutive '1's.")

else:

print("Rejected: The string does not contain three consecutive '1's.")

exp5 :

def accepts\_string\_ending\_with\_101(input\_str):

current\_state = 'q0'

for symbol in input\_str:

if current\_state == 'q0' and symbol == '1':

current\_state = 'q1'

elif current\_state == 'q1' and symbol == '0':

current\_state = 'q2'

elif current\_state == 'q2' and symbol == '1':

current\_state = 'q3'

else:

current\_state = 'q0'

return current\_state == 'q3'

# Example usage:

input\_str = input("Enter a binary string: ")

result = accepts\_string\_ending\_with\_101(input\_str)

if result:

print("Accepted: The string ends with '101'.")

else:

print("Rejected: The string does not end with '101'.")

exp 6 :

def is\_divisible\_by\_2(input\_str):

current\_state = 'q0'

for digit in input\_str:

if current\_state == 'q0' and digit in '02468':

current\_state = 'q1'

elif current\_state == 'q1' and digit in '02468':

current\_state = 'q0'

else:

return False

return current\_state == 'q0'

# Example usage:

input\_str = input("Enter a decimal number: ")

# Check if the entered number is divisible by 2

result = is\_divisible\_by\_2(input\_str)

if result:

print("Accepted: The decimal number is divisible by 2.")

else:

print("Rejected: The decimal number is not divisible by 2.")

exp 7 :

def accepts\_equal\_01(input\_str):

count\_1 = 0

count\_0 = 0

for symbol in input\_str:

if symbol == '1':

count\_1 += 1

elif symbol == '0':

count\_0 += 1

return count\_1 == count\_0

# Example usage:

input\_str = input("Enter a binary string: ")

result = accepts\_equal\_01(input\_str)

if result:

print("Accepted: The string has an equal number of '1's and '0's.")

else:

print("Rejected: The string does not have an equal number of '1's and '0's.")

exp 8 :

def count\_ones\_and\_zeros(input\_str):

count\_1 = 0

count\_0 = 0

for symbol in input\_str:

if symbol == '1':

count\_1 += 1

elif symbol == '0':

count\_0 += 1

return count\_1, count\_0

# Example usage:

input\_str = input("Enter a binary string: ")

count\_1, count\_0 = count\_ones\_and\_zeros(input\_str)

print(f"Number of '1's: {count\_1}")

print(f"Number of '0's: {count\_0}")

exp 9 :

from pyformlang.pda import State, PDA, Symbol, StackSymbol

def create\_wcwr\_pda():

# Define states

q0 = State("q0")

q1 = State("q1")

q\_accept = State("q\_accept")

q\_reject = State("q\_reject")

# Define stack symbols

sigma = set([Symbol("0"), Symbol("1"), Symbol("c")])

gamma = set([StackSymbol("Z")])

# Define transitions

transitions = [

((q0, Symbol("0"), StackSymbol("Z")), (q0, [StackSymbol("0"), StackSymbol("Z")])),

((q0, Symbol("1"), StackSymbol("Z")), (q0, [StackSymbol("1"), StackSymbol("Z")])),

((q0, Symbol("0"), StackSymbol("0")), (q0, [StackSymbol("0"), StackSymbol("0")])),

((q0, Symbol("1"), StackSymbol("1")), (q0, [StackSymbol("1"), StackSymbol("1")])),

((q0, Symbol("c"), StackSymbol("0")), (q1, [])),

((q1, Symbol("0"), StackSymbol("0")), (q1, [])),

((q1, Symbol("1"), StackSymbol("1")), (q1, [])),

((q1, Symbol(""), StackSymbol("Z")), (q\_accept, [])),

((q0, Symbol(""), StackSymbol("Z")), (q\_reject, [])),

]

pda = PDA(

states={q0, q1, q\_accept, q\_reject},

input\_symbols=sigma,

stack\_symbols=gamma,

transitions=transitions,

initial\_state=q0,

initial\_stack\_symbol=StackSymbol("Z"),

final\_states={q\_accept},

)

return pda

def test\_wcwr\_pda(input\_str):

pda = create\_wcwr\_pda()

result = pda.accepts\_input(list(input\_str))

return result

# Example usage:

input\_str = input("Enter a string: ")

result = test\_wcwr\_pda(input\_str)

if result:

print("Accepted: The string is in the language WCWR.")

else:

print("Rejected: The string is not in the language WCWR.")