LAB 1. Write a program to find prefixes, suffixes and substring from given string.

Suffix of a string: A string s is called a suffix of a string w if it is obtained by removing 0 or more leading symbols in w. For example;
w = abcd
s = bcd is suffix of w. here s is proper suffix if s ≠ w.
Prefix of a string: A string s is called a prefix of a string w if it is obtained by removing 0 or more trailing symbols of w. For example;
w = abcd
s = abc is prefix of w,
Here, s is proper suffix i.e., s is proper suffix if s ≠ w.
Substring: A string s is called substring of a string w if it is obtained by removing 0 or more leading or trailing symbols in w. It is proper substring of w if s ≠ w.
If s is a string, then *Substr (s, i, j)* is substring of s beginning at i^{th} position & ending at j^{th} position both inclusive.

PROGRAM

```
# Program that takes input as a string and find its suffix, prefix and substring - lab1


# Function that returns the prefix of a string
def find_prefix(word):
    prefix_list = []
    for i in range(1,len(word)+1):
        prefix_list.append(word[:i])
    return prefix_list


# Function that returns the suffix of a string
def find_suffix(word):
    suffix_list = []
    for i in range(0,len(word)):
        suffix_list.append(word[i:])
    return suffix_list


# Function that returns the substring of a string
def find_substring(word):
    substring_list = []
    for i in range(0,len(word)):
        for j in range(i+1, len(word)+1):
            substring_list.append(word[i:j])
    return substring_list


# Function that sorts the list according to length of list item
```

```
def sort_list(given_list):
   final_list = sorted(given_list, key=len)
   return final_list


# User Input
word= input("Enter the string: ")
print("Prefix :",sort_list(find_prefix(word)),
     "\nSuffx :", sort_list(find_suffix(word)),
     "\nSubstring :", sort_list(find_substring(word)))
```

OUTPUT

```
Enter the string: roshan
Prefix : ['r', 'ro', 'ros', 'rosh', 'rosha', 'roshan']
Suffx : ['n', 'an', 'han', 'shan', 'oshan', 'roshan']
Substring : ['r', 'o', 's', 'h', 'a', 'n', 'ro', 'os', 'sh', 'ha', 'an', 'ros', 'osh', 'sha', 'han', 'rosh'
, 'osha', 'shan', 'rosha', 'oshan', 'roshan']
user@Roshans-MacBook-Pro lab files %
```

2. Write a program to implement DFA over {0, 1}, that accept all the string that start with 01.

Deterministic Finite Automata: A deterministic finite automaton is defined by a quintuple (5-tuple) as (Q, Σ, δ, q0, F). Where,
Q = Finite set of states,
Σ = Finite set of input symbols,
δ = A transition function that maps Q × Σ → Q q0 = A start state; q0 ∈Q
F = Set of final states; F Q.
A transition function δ that takes as arguments a state and an input symbol and returns a state.
In our diagram, δ is represented by arcs between states and the labels on the arcs.
For example

If s is a state and a is an input symbol then δ (p, a) is that state q such that there are arcs labeled
'a' from p to q.

PROGRAM

```
# Write a program to implement DFA over{0, 1 },
# that accept all the string that start with 01.

accept_states = ["q2"]
transition = {
    "q0": {"0": "q1", "1": "q3"},
    "q1": {"0": "q3", "1": "q2"},
    "q2": {"0": "q2", "1": "q2"},
    "q3": {"0": "q3", "1": "q3"},
}


word = input("Enter the string: ")
list_word = list(word)
index = len(word)

# Function for analize the string
def func():
    if index > 1:
        current_state = "q0"
        for i in range(0, index):
            new_state = transition[current_state][list_word[i]]
            print(current_state + " -> " + new_state)
            current_state = new_state
        return current_state

    else:
        return "string size is less"


# Function calling
```

```
x = func()


if x in accept_states:
    print("String is valid!")
else:
    print("String is invalid!")
```

OUTPUT

```
Enter the string: 01110
q0 -> q1
q1 -> q2
q2 -> q2
q2 -> q2
q2 -> q2
String is valid!
user@Roshans-MacBook-Pro lab files % python -u "/Users/user/Desktop/4th sem/toc/lab files/2dfaa.py"
Enter the string: 1010
q0 -> q3
q3 -> q3
q3 -> q3
q3 -> q3
String is invalid!
user@Roshans-MacBook-Pro lab files %
```

3. Write a program to implement DFA over {0, 1}, that accept all the string that ends with 101.

Answer

The below program implements DFA over {0,1} that accepts all the strings that ends with 101 and rejects otherwise.

PROGRAM

```
# Write a program to implement DFA over{0, 1 },
# that accept all the string that ends with 101.

accept_states = ["q3"]
transition = {
    "q0": {"0": "q0", "1": "q1"},
    "q1": {"0": "q2", "1": "q1"},
    "q2": {"0": "q0", "1": "q3"},
    "q3": {"0": "q2", "1": "q1"},
}


word = input("Enter the string: ")
list_word = list(word)
index = len(word)

# Function for analize the string
def func():
    if index > 2:
        current_state = "q0"
        for i in range(0, index):
            new_state = transition[current_state][list_word[i]]
            print(current_state + " (" + str(list_word[i]) + ") " + " -> " +
new_state)
            current_state = new_state
        return current_state

    else:
        return "string size is less"


# Function calling
x = func()


if x in accept_states:
    print("String is valid!")
else:
    print("String is invalid!")
```

OUTPUT

```
Enter the string: 1100101
q0 (1)  -> q1
q1 (1)  -> q1
q1 (0)  -> q2
q2 (0)  -> q0
q0 (1)  -> q1
q1 (0)  -> q2
q2 (1)  -> q3
String is valid!
user@Roshans-MacBook-Pro lab files % python -u "/Users/user/Desktop/4th sem/toc/lab
files/3dfa.py"
Enter the string: 1001010
q0 (1)  -> q1
q1 (0)  -> q2
q2 (0)  -> q0
q0 (1)  -> q1
q1 (0)  -> q2
q2 (1)  -> q3
q3 (0)  -> q2
String is invalid!
```

4. Write a program to implement DFA over {0, 1}, that contain substring 0001.
Answer
The below program implements DFA over {0, 1}, that contains substring 0001.

PROGRAM

```
# Write a program to implement DFA over{0, 1 },
# that contain substring 0001.

accept_states = ["q4"]
transition = {
    "q0": {"0": "q1", "1": "q0"},
    "q1": {"0": "q2", "1": "q0"},
    "q2": {"0": "q3", "1": "q0"},
    "q3": {"0": "q2", "1": "q4"},
    "q4": {"0": "q4", "1": "q4"},
}


word = input("Enter the string: ")
list_word = list(word)
index = len(word)

# Function for analize the string
def func():
    if index > 3:
        current_state = "q0"
        for i in range(0, index):
            new_state = transition[current_state][list_word[i]]
            print(current_state + " (" + str(list_word[i]) + ") " + " -> " +
new_state)
            current_state = new_state
        return current_state

    else:
        return "string size is less"


# Function calling
x = func()


if x in accept_states:
    print("String is valid!")
else:
    print("String is invalid!")
```

OUTPUT

```
Enter the string: 10001110
q0 (1)  -> q0
q0 (0)  -> q1
q1 (0)  -> q2
q2 (0)  -> q3
q3 (1)  -> q4
q4 (1)  -> q4
q4 (1)  -> q4
q4 (0)  -> q4
String is valid!
user@Roshans-MacBook-Pro lab files % python -u "/Users/user/Desktop/4th sem/toc/lab
files/4dfa.py"
Enter the string: 10101
q0 (1)  -> q0
q0 (0)  -> q1
q1 (1)  -> q0
q0 (0)  -> q1
q1 (1)  -> q0
String is invalid!
```

5. Write a program to validate C identifiers

Identifier refers to name given to entities such as variables, functions, structures etc. Identifiers must be unique. They are created to give a unique name to an entity to identify it during the execution of the program.

Rules for naming identifiers

1. A valid identifier can have letters (both uppercase and lowercase letters), digits and underscores.
2. The first letter of an identifier should be either a letter or an underscore.
3. You cannot use keywords like int, while etc. as identifiers.
4. There is no rule on how long an identifier can be. However, you may run into problems in some compilers if the identifier is longer than 31 characters.

PROGRAM

```
# Python program to identify the identifier

# import re module

# re module provides support
# for regular expressions
import re

# Make a regular expression
# for identify valid identifier
regex = "^[A-Za-z_][A-Za-z0-9_]*"

# Define a function for
# identifying valid identifier
def check(word):
    keywords = [
        "int",
        "double",
        "auto",
        "break",
        "case",
        "char",
        "const",
        "continue",
        "default",
        "do",
        "else",
        "enum",
        "extern",
        "float",
        "for",
        "goto",
        "if",
        "long",
        "register",
```

```
        "return",
        "short",
        "signed",
        "sizeof",
        "static",
        "struct",
        "switch",
        "typedef",
        "union",
        "unsigned",
        "void",
        "volatile",
        "while",
    ]
    # pass the regular expression
    # and the string in search() method
    if re.search(regex, word):
        if word in keywords:
            print("It is a c keyword")
        else:
            print("Valid Identifier")

    else:
        print("Invalid Identifier")


# Driver Code
if __name__ == "__main__":
    character = input("Enter a string: ")
    check(character)
```

OUTPUT

```
Enter a string: roshan
Valid Identifier
user@Roshans-MacBook-Pro lab files % python -u "/Users/user/Desktop/4th sem/toc/lab
files/5cidentifiers.py"
Enter a string: #roshan
Invalid Identifier
```

6. Implement NFA over {0, 1}, that accept the string starting with 01

NFA stands for non-deterministic finite automata. It is easy to construct an NFA than DFA for a given regular language.
The finite automata are called NFA when there exist many paths for specific input from the current state to the next state.
Every NFA is not DFA, but each NFA can be translated into DFA.

An NFA can be represented by digraphs called state diagram. In which:

1. The state is represented by vertices.
2. The arc labeled with an input character show the transitions.
3. The initial state is marked with an arrow.
4. The final state is denoted by the double circle.

PROGRAM

```
# Implement NFA over{0, 1 }, that accept the string starting with 01

accept_state = ["q2"]
transition = {
    "q0": {"0": ["q1"], "1": ["don't care"]},
    "q1": {"0": ["don't care"], "1": ["q2"]},
    "q2": {"0": ["q2"], "1": ["q2"]},
}

# User input
word = input("Enter the string: ")
word_list = list(word)
word_length = len(word)
current_state = ["q0"]

# Returns the final states of the nfa
def get_final_states(current_state):
    if word_length > 1:
        fi = []
        # For each charater of the given word
        for i in range(0, word_length):
            li = []
            ch = word_list[i]
            print("for ", word_list[i])  # Each character

            # For every state in the current_state find their next_state
            for c in current_state:
                if c == "don't care":
                    pass
                else:
                    new_state = transition[c][ch]
                    if new_state not in fi:
                        fi.append(new_state)
```

```
                    print(str(c) + " -> " + str(new_state))
            # print('states:', new_state)
            current_state = new_state
            print("-----------")
        return fi

    else:
        print("String size is less")
        quit()


x = get_final_states(current_state)
print("Final state ", x)

# If the accept_state if present in the final_states
if accept_state in x:
    print("String is valid!")
else:
    print("String is invalid!")
```

OUTPUT

```
Enter the string: 01001
for  0
q0 -> ['q1']
-----------
for  1
q1 -> ['q2']
-----------
for  0
q2 -> ['q2']
-----------
for  0
q2 -> ['q2']
-----------
for  1
q2 -> ['q2']
-----------
Final state  [['q1'], ['q2']]
String is valid!
user@Roshans-MacBook-Pro lab files % python -u "/Users/user/Desktop/4th sem/toc/lab
files/6nfa.py"
Enter the string: 1011
for  1
q0 -> ["don't care"]
-----------
for  0
-----------
for  1
-----------
for  1
-----------
Final state  [["don't care"]]
String is invalid!
```

7. Write a program to implement NFA over {0, 1}, that accept all the string that ends with 11.

Answer

The below program implements NFA over {0, 1}, that accept all the string that ends with 11 and rejects otherwise.

PROGRAM

```
# Write a program to implement NFA over {0, 1},
# that accept all the string that ends with 11.


accept_state = ["q2"]
transition = {
    "q0": {"0": ["q0"], "1": ["q1"]},
    "q1": {"0": ["don't care"], "1": ["q2"]},
    "q2": {"0": ["don't care"], "1": ["don't care"]},
}


# User input
word = input("Enter the string: ")
word_list = list(word)
word_length = len(word)
current_state = ["q0"]


# Returns the final states of the nfa
def get_final_states(current_state):
    if word_length > 1:
        fi = []
        # For each charater of the given word
        for i in range(0, word_length):
            li = []
            ch = word_list[i]
            print("for ", word_list[i])  # Each character

            # For every state in the current_state find their next_state
            for c in current_state:
                if c == "don't care":
                    pass
                else:
                    new_state = transition[c][ch]
                    if new_state not in fi:
                        fi.append(new_state)
                    print(str(c) + " -> " + str(new_state))
                    # print('states:', new_state)
            current_state = new_state
            print("----------")
        return fi
```

```
    else:
        print("String size is less")
        quit()


x = get_final_states(current_state)
print("Final state ", x)


# If the accept_state if present in the final_states
if accept_state in x:
    print("String is valid!")
else:
    print("String is invalid!")
```

OUTPUT

```
4th sem/toc/lab files/7nfa.py"
Enter the string: 00111
for  0
q0 -> ['q0']
------------
for  0
q0 -> ['q0']
------------
for  1
q0 -> ['q1']
------------
for  1
q1 -> ['q2']
------------
for  1
q2 -> ["don't care"]
------------
Final state  [['q0'], ['q1'], ['q2'], ["don't care"]]
String is valid!
user@Roshans-MacBook-Pro lab files %
```

8. Take a Grammar of your choice and implement parse tree for the string produced by the grammar.

Parse tree is the graphical representation of symbol. The symbol can be terminal or non-terminal.
In parsing, the string is derived using the start symbol. The root of the parse tree is that start symbol.
It is the graphical representation of symbol that can be terminals or non-terminals.
Parse tree follows the precedence of operators. The deepest sub-tree traversed first. So, the operator in the parent node has less precedence over the operator in the sub-tree.

The parse tree follows these points:

- All leaf nodes have to be terminals.
- All interior nodes have to be non-terminals.
- In-order traversal gives original input string.

PROGRAM

```
# Write a program to implement DFA over{0, 1 },
# that accept all the string that ends with 101.

accept_states = ["q3"]
transition = {
   "q0": {"0": "q0", "1": "q1"},
   "q1": {"0": "q2", "1": "q1"},
   "q2": {"0": "q0", "1": "q3"},
   "q3": {"0": "q2", "1": "q1"},
}

word = input("Enter the string: ")
list_word = list(word)
index = len(word)

# Function for analize the string
def func():
   if index > 2:
      current_state = "q0"
      for i in range(0, index):
         new_state = transition[current_state][list_word[i]]
         print(current_state + " (" + str(list_word[i]) + ") " + " -> " + new_state)
         current_state = new_state
      return current_state

   else:
      return "string size is less"

# Function calling
```

```
x = func()

if x in accept_states:
   print("String is valid!")
else:
   print("String is invalid!")
```

OUTPUT

```
user@Roshans-MacBook-Pro lab files % python -u "/Users/user/Desktop/4th sem/toc/lab
files/8parsetree.py"
S  ->  sAB
updated string  sAB
A  ->  a
updated string  saB
B  ->  b
updated string  sab
sab
user@Roshans-MacBook-Pro lab files %
```