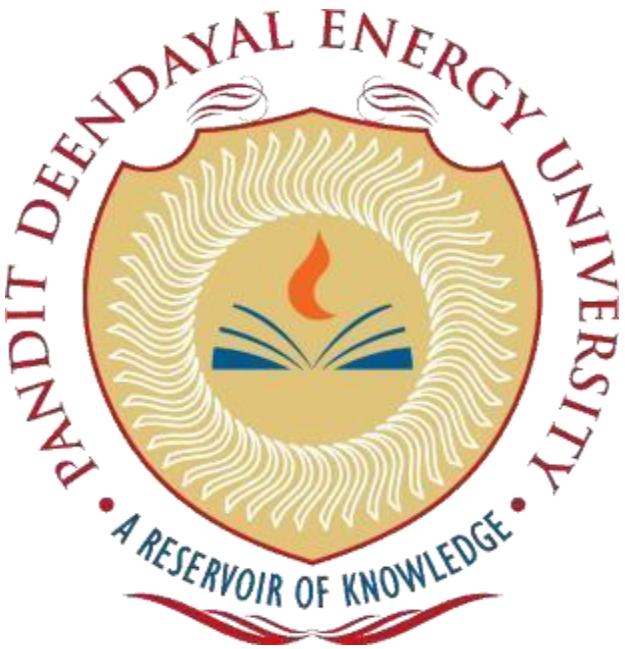
Pandit Deendayal Energy University, Gandhinagar

School of Technology

Department of Computer Science & Engineering

System Software & Compiler Design Lab (20CP302P)



Name: **Sutariya Smit Dharmendrabhai**

Enrolment No: **21BCP142**

Semester: **V**

Division: **3 (G5)**

Branch: **Computer Science Engineering**

Practical: 6

Aim: WAP to construct operator precedence parsing table for the given grammar and check the validity of the string.

Code:

```
from tabulate import tabulate

# S->x A y |x B y |x A z
# A->a S|q
# B->q

firstop = {}
lastop = {}
productions = []
prod_dict = {}
table_list = []

def add_to_firstop(nterm, symbol):
    if nterm not in firstop:
        firstop[nterm] = set()
    firstop[nterm].add(symbol)

def add_to_lastop(nterm, symbol):
    if nterm not in lastop:
        lastop[nterm] = set()
    lastop[nterm].add(symbol)

def replace_err(table):
    for i in range(len(table)):
        for j in range(len(table[i])):
            if table[i][j] == ' ':
                table[i][j] = 'err'
    return table

def parse_expression(str):
    stack = ['$']  # Initialize the stack with '$'
    string = str.split()
    input_buffer = list(string) + ['$']  # Append '$' to the input string
    print(input_buffer)
    index = 0  # Index to traverse the input buffer

    while len(stack) > 0:
```

```
        top_stack = stack[-1]
        print(top_stack)
        current_input = input_buffer[index]

        top_stack_index = terminals.index(top_stack)
        current_input_index = terminals.index(current_input)

        relation = terminal_matrix[top_stack_index][current_input_index]

        if relation == '<' or relation == '=':
            stack.append(current_input)
            index += 1
        elif relation == '>':
            popped = ''
            while relation != '<':
                popped = stack.pop()  # Pop elements from the stack until '<'
relation is found
                top_stack = stack[-1] if stack else None
                top_stack_index = terminals.index(top_stack) if top_stack else
None
                relation =
terminal_matrix[top_stack_index][terminals.index(popped)]
        elif relation == 'acc':
            print("Input string is accepted.")
            return
        else:
            print("Input string is not accepted.")
            return

no_of_terminals = int(input("Enter no. of terminals: "))
terminals = []
print("Enter the terminals:")
for _ in range(no_of_terminals):
    terminals.append(input())

no_of_non_terminals = int(input("Enter no. of non-terminals: "))
non_terminals = []
print("Enter the non-terminals:")
for _ in range(no_of_non_terminals):
    non_terminals.append(input())

starting_symbol = input("Enter the starting symbol: ")

no_of_productions = int(input("Enter no of productions: "))
```

```
print("Enter the productions:")
for _ in range(no_of_productions):
    productions.append(input())

for nT in non_terminals:
    prod_dict[nT] = []

for production in productions:
    nonterm_to_prod = production.split("->")
    alternatives = nonterm_to_prod[1].split("|")
    for alternative in alternatives:
        prod_dict[nonterm_to_prod[0]].append(alternative)

print("Populated prod_dict:")
for non_terminal, prods in prod_dict.items():
    print(f"{non_terminal} -> {prods}")

parsing_string = input("Enter an expression to parse: ")

# Compute firstop for each non-terminal
for non_terminal in non_terminals:
    for production in prod_dict[non_terminal]:
        symbols = production.split()
        print(symbols)
        for symbol in symbols:
            if symbol in non_terminals:
                add_to_firstop(non_terminal, symbol)
            elif symbol in terminals:
                add_to_firstop(non_terminal, symbol)
                break

# Compute lastop for each non-terminal
for non_terminal in non_terminals:
    for production in prod_dict[non_terminal]:
        symbols = production.split()
        for symbol in reversed(symbols):
            if symbol in non_terminals:
                add_to_lastop(non_terminal, symbol)
            elif symbol in terminals:
                add_to_lastop(non_terminal, symbol)
                break

# Print the firstop and lastop sets
print("firstop:")
for non_terminal, first_set in firstop.items():
```

```
    print(f'firstop({non_terminal}) = {{{", ".join(first_set)}}}')

print("lastop:")
for non_terminal, last_set in lastop.items():
    print(f'lastop({non_terminal}) = {{{", ".join(last_set)}}}')

counter=0
while counter<no_of_productions:
    for non_terminal, first_set in firstop.items():
        first_set_copy = first_set.copy()  # Create a copy of the set to iterate
over
        for symbol in first_set_copy:
            if symbol in non_terminals:
                firstop[non_terminal] |= firstop[symbol]
    counter+=1


# Remove non-terminals from lastop sets
counter=0
while counter<no_of_productions:
    for non_terminal, last_set in lastop.items():
        last_set_copy = last_set.copy()  # Create a copy of the set to iterate
over
        for symbol in last_set_copy:
            if symbol in non_terminals:
                lastop[non_terminal] |= lastop[symbol]
    counter+=1

# Remove non-terminals from firstop sets
for non_terminal, first_set in firstop.items():
    first_set_copy = first_set.copy()  # Create a copy of the set to iterate over
    for symbol in first_set_copy:
        if symbol in non_terminals:
            first_set.remove(symbol)

# Remove non-terminals from lastop sets
for non_terminal, last_set in lastop.items():
    last_set_copy = last_set.copy()  # Create a copy of the set to iterate over
    for symbol in last_set_copy:
        if symbol in non_terminals:
            last_set.remove(symbol)

# Print the modified firstop and lastop sets
print("Firstop:")
for non_terminal, first_set in firstop.items():
```

```
    print(f'Firstop({non_terminal}) = {{{", ".join(first_set)}}}')

print("Lastop:")
for non_terminal, last_set in lastop.items():
    print(f'Lastop({non_terminal}) = {{{", ".join(last_set)}}}')

terminals.append('$')

terminal_matrix = [[' ' for _ in range(len(terminals))] for _ in
range(len(terminals))]

# Rule 1: Whenever terminal a immediately precedes non-terminal B in any
production, put a <·α where α is any terminal in the firstop+ list of B
for non_terminal in non_terminals:
    for productions in prod_dict[non_terminal]:
        production = productions.split()
        for i in range(len(production) - 1):
            if production[i] in terminals and production[i + 1] in non_terminals:
                for alpha in firstop[production[i + 1]]:
                    row_index = terminals.index(production[i])
                    col_index = terminals.index(alpha)
                    terminal_matrix[row_index][col_index] = '<'

# Rule 2: Whenever terminal b immediately follows non-terminal C in any
production, put β ·>b where β is any terminal in the lastop+ list of C
for non_terminal in non_terminals:
    for productions in prod_dict[non_terminal]:
        production = productions.split()
        for i in range(1, len(production)):
            if production[i - 1] in non_terminals and production[i] in terminals:
                for beta in lastop[production[i - 1]]:
                    row_index = terminals.index(beta)
                    col_index = terminals.index(production[i])
                    terminal_matrix[row_index][col_index] = '>'

# Rule 3: Whenever a sequence aBc or ac occurs in any production, put a ≐ c
for non_terminal in non_terminals:
    for productions in prod_dict[non_terminal]:
        production = productions.split()
        for i in range(1, len(production) - 1):
            if production[i - 1] in terminals and production[i + 1] in terminals:
                row_index = terminals.index(production[i - 1])
                col_index = terminals.index(production[i + 1])
                terminal_matrix[row_index][col_index] = '='
```

```
# Rule 4: Add relations $<· a and a ·> $ for all terminals in the firstop+ and
lastop+ lists, respectively of S
for alpha in firstop[starting_symbol]:
    col_index = terminals.index(alpha)
    terminal_matrix[-1][col_index] = '<'
for beta in lastop[starting_symbol]:
    row_index = terminals.index(beta)
    terminal_matrix[row_index][-1] = '>'

dollar_index = terminals.index('$')
terminal_matrix[-1][dollar_index] = 'acc'
terminal_matrix = replace_err(terminal_matrix)

for i in range(len(terminals)):
    row = [terminals[i]]
    row.extend([terminal_matrix[i][j] for j in range(len(terminals))])
    table_list.append(row)

headers = [''] + terminals

Operator_Precedence_table = tabulate(table_list, headers, tablefmt="grid")

print("Operator Precedence Table:")
print(Operator_Precedence_table)
parse_expression(parsing_string)
```

Output:

```
PS D:\Sem-5\compiler> python -u "d:\Sem-5\compiler\Lab6\Opp6.py"
Enter no. of terminals: 5
Enter the terminals:
x
y
z
a
q
Enter no. of non-terminals: 3
Enter the non-terminals:
S
A
B
Enter the starting symbol: S
Enter no of productions: 3
Enter the productions:
S->x A y |x B y |x A z
A->a S|q
B->q
Populated prod_dict:
S -> ['x A y ', 'x B y ', 'x A z ']
A -> ['a S', 'q']
B -> ['q']
Enter an expression to parse: x q y
['x', 'A', 'y']
['x', 'B', 'y']
['x', 'A', 'z']
['a', 'S']
['q']
['q']
firstop:
firstop(S) = {x}
firstop(A) = {a, q}
firstop(B) = {q}
lastop:
lastop(S) = {y, z}
lastop(A) = {S, a, q}
lastop(B) = {q}
Firstop:
Firstop(S) = {x}
Firstop(A) = {a, q}
Firstop(B) = {q}
Lastop:
Lastop(S) = {y, z}
Lastop(A) = {y, z, a, q}
Lastop(B) = {q}
```

```
Operator Precedence Table:
+----+-----+-----+-----+-----+-----+-----+
|    | x   | y   | z   | a   | q   | $   |
+====+=====+=====+=====+=====+=====+=====+
| x  | err | =   | =   | <   | <   | err |
+----+-----+-----+-----+-----+-----+-----+
| y  | err | >   | >   | err | err | >   |
+----+-----+-----+-----+-----+-----+-----+
| z  | err | >   | >   | err | err | >   |
+----+-----+-----+-----+-----+-----+-----+
| a  | <   | >   | >   | err | err | err |
+----+-----+-----+-----+-----+-----+-----+
| q  | err | >   | >   | err | err | err |
+----+-----+-----+-----+-----+-----+-----+
| $  | <   | err | err | err | err | acc |
+----+-----+-----+-----+-----+-----+-----+
['x', 'q', 'y', '$']
$
x
q
x
y
$
Input string is accepted.
PS D:\Sem-5\compiler>
```