22BCE3939
Karan Sehgal
Compiler Design
Lab Assessment 3

Q1:-

Design a Predictive Parser (Non-Recursive Descent Parser) for the given grammar:

Grammar G:

- $E \rightarrow TE'$
- $E' \rightarrow +TE' \mid \epsilon$
- $T \rightarrow FT'$
- $T' \rightarrow *FT' \mid \epsilon$
- $F \rightarrow (E) \mid id$

Outline of Key Functions:

1. **FIRST**: Compute the first set for each non-terminal.
2. **FOLLOW**: Compute the follow set for each non-terminal.
3. **PARSING TABLE**: Populate the table with appropriate production rules.
4. **PARSER**:
   - Simulate stack operations for top-down parsing.
   - Handle errors during mismatches or invalid input.

CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX 100

// Stack implementation
char stack[MAX];
int top = -1;

void push(char symbol) {
    if (top >= MAX - 1) {
        printf("Stack Overflow!\n");
        exit(1);
    } else {
        stack[++top] = symbol;
    }
}

char pop() {
    if (top < 0) {
        printf("Stack Underflow!\n");
        exit(1);
    } else {
        return stack[top--];
    }
}

char peek() {
    return stack[top];
}

// Function to check if a character is a terminal
int isTerminal(char c) {
    return (c == 'i' || c == '+' || c == '*' || c == '(' || c == ')' || c == '$');
}
```

```
// Function to return production rule based on stack top and input symbol
const char* getProductionRule(char stackTop, char input) {
    if (stackTop == 'E' && (input == 'i' || input == '(')) return "TZ";
    if (stackTop == 'Z' && input == '+') return "+TZ";
    if (stackTop == 'Z' && (input == ')' || input == '$')) return "0";
    if (stackTop == 'T' && (input == 'i' || input == '(')) return "FX";
    if (stackTop == 'X' && input == '*') return "*FX";
    if (stackTop == 'X' && (input == '+' || input == ')' || input == '$')) return
"0";
    if (stackTop == 'F' && input == 'i') return "i";
    if (stackTop == 'F' && input == '(') return "(E)";

    return NULL;
}

int main() {
    char input[MAX], action[30];
    int i = 0;

    // Read input string
    printf("Enter the input string to be parsed: ");
    scanf("%s", input);
    strcat(input, "$"); // Append end marker

    // Initialize stack with start symbol and end marker
    push('$');
    push('E');

    printf("\nStack\t\tInput\t\tAction\n");

    while (stack[top] != '$') {
        printf("\n");
        for (int j = 0; j <= top; j++)
            printf("%c", stack[j]);
        printf("\t\t%s", (input + i));

        char stackTop = pop();

        if (isTerminal(stackTop)) {
```

```
            if (stackTop == input[i]) {
                sprintf(action, "Matched %c", input[i]);
                i++;
            } else {
                sprintf(action, "Error");
                printf("\n%s", action);
                exit(1);
            }
        } else {
            const char* rule = getProductionRule(stackTop, input[i]);
            if (rule != NULL) {
                sprintf(action, "Apply rule %s", rule);
                int k = strlen(rule) - 1;
                while (k >= 0) {
                    if (rule[k] != '0' && rule[k] != ' ') push(rule[k]);
                    k--;
                }
            } else {
                sprintf(action, "Error");
                printf("\n%s", action);
                exit(1);
            }
        }
        printf("\t\t%s", action);
    }

    if (input[i] == '$') printf("\nInput string parsed successfully.\n");
    else printf("\nError in parsing the input string.\n");

    return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX 100

// Stack implementation
char stack[MAX];
int top = -1;

void push(char symbol) {
    if (top >= MAX - 1) {
        printf("Stack Overflow!\n");
        exit(1);
    } else {
        stack[++top] = symbol;
    }
}

char pop() {
    if (top < 0) {
        printf("Stack Underflow!\n");
        exit(1);
    } else {
        return stack[top--];
    }
}

char peek() {
    return stack[top];
}

// Function to check if a character is a terminal
int isTerminal(char c) {
    return (c == 'i' || c == '+' || c == '*' || c == '(' || c == ')' || c == '$');
}

// Function to return production rule based on stack top and input symbol
const char* getProductionRule(char stackTop, char input) {
    if (stackTop == 'E' && (input == 'i' || input == '(')) return "TZ";
    if (stackTop == 'Z' && input == '+') return "+TZ";
    if (stackTop == 'Z' && (input == ')' || input == '$')) return "0";
    if (stackTop == 'T' && (input == 'i' || input == '(')) return "FX";
    if (stackTop == 'X' && input == '*') return "*FX";
    if (stackTop == 'X' && (input == '+' || input == ')' || input == '$')) return "0";
    if (stackTop == 'F' && input == 'i') return "i";
    if (stackTop == 'F' && input == '(') return "(E)";
```

```c
const char* getProductionRule(char stackTop, char input) {
    if (stackTop == 'F' && input == '(') return "(E)";

    return NULL;
}

int main() {
    char input[MAX], action[30];
    int i = 0;

    // Read input string
    printf("Enter the input string to be parsed: ");
    scanf("%s", input);
    strcat(input, "$"); // Append end marker

    // Initialize stack with start symbol and end marker
    push('$');
    push('E');

    printf("\nStack\t\tInput\t\tAction\n");

    while (stack[top] != '$') {
        printf("\n");
        for (int j = 0; j <= top; j++)
            printf("%c", stack[j]);
        printf("\t\t%s", (input + i));

        char stackTop = pop();

        if (isTerminal(stackTop)) {
            if (stackTop == input[i]) {
                sprintf(action, "Matched %c", input[i]);
                i++;
            } else {
                sprintf(action, "Error");
                printf("\n%s", action);
                exit(1);
            }
        } else {
            const char* rule = getProductionRule(stackTop, input[i]);
            if (rule != NULL) {
                sprintf(action, "Apply rule %s", rule);
                int k = strlen(rule) - 1;
                while (k >= 0) {
                    if (rule[k] != '0' && rule[k] != ' ') push(rule[k]);
                    k--;
                }
```

```c
int main() {
    while (stack[top] != '$') {
        printf("\t\t%s", (input + i));

        char stackTop = pop();

        if (isTerminal(stackTop)) {
            if (stackTop == input[i]) {
                sprintf(action, "Matched %c", input[i]);
                i++;
            } else {
                sprintf(action, "Error");
                printf("\n%s", action);
                exit(1);
            }
        } else {
            const char* rule = getProductionRule(stackTop, input[i]);
            if (rule != NULL) {
                sprintf(action, "Apply rule %s", rule);
                int k = strlen(rule) - 1;
                while (k >= 0) {
                    if (rule[k] != '0' && rule[k] != ' ') push(rule[k]);
                    k--;
                }
            } else {
                sprintf(action, "Error");
                printf("\n%s", action);
                exit(1);
            }
        }
        printf("\t\t%s", action);
    }

    if (input[i] == '$') printf("\nInput string parsed successfully.\n");
    else printf("\nError in parsing the input string.\n");

    return 0;
}
```

OUTPUT:

```
Enter the input string to be parsed: i*i+i

Stack           Input           Action

$E              i*i+i$          Apply rule TZ
$ZT             i*i+i$          Apply rule FX
$ZXF            i*i+i$          Apply rule i
$ZXi            i*i+i$          Matched i
$ZX             *i+i$           Apply rule *FX
$ZXF*           *i+i$           Matched *
$ZXF            i+i$            Apply rule i
$ZXi            i+i$            Matched i
$ZX             +i$             Apply rule 0
$Z              +i$             Apply rule +TZ
$ZT+            +i$             Matched +
$ZT             i$              Apply rule FX
$ZXF            i$              Apply rule i
$ZXi            i$              Matched i
$ZX             $               Apply rule 0
$Z              $               Apply rule 0
Input string parsed successfully.

Process returned 0 (0x0)   execution time : 12.993 s
Press ENTER to continue.
^[[23~
```

```
Enter the input string to be parsed: i+i+i*i

Stack           Input           Action

$E              i+i+i*i$                Apply rule TZ
$ZT             i+i+i*i$                Apply rule FX
$ZXF            i+i+i*i$                Apply rule i
$ZXi            i+i+i*i$                Matched i
$ZX             +i+i*i$         Apply rule 0
$Z              +i+i*i$         Apply rule +TZ
$ZT+            +i+i*i$         Matched +
$ZT             i+i*i$          Apply rule FX
$ZXF            i+i*i$          Apply rule i
$ZXi            i+i*i$          Matched i
$ZX             +i*i$           Apply rule 0
$Z              +i*i$           Apply rule +TZ
$ZT+            +i*i$           Matched +
$ZT             i*i$            Apply rule FX
$ZXF            i*i$            Apply rule i
$ZXi            i*i$            Matched i
$ZX             *i$             Apply rule *FX
$ZXF*           *i$             Matched *
$ZXF            i$              Apply rule i
$ZXi            i$              Matched i
$ZX             $               Apply rule 0
$Z              $               Apply rule 0
Input string parsed successfully.

Process returned 0 (0x0)   execution time : 21.276 s
Press ENTER to continue.
^[[24~
```

Q2:Write a C program to construct a Non-Deterministic Finite Automata (NFA) from a given regular expression.

The objective is to implement an algorithm that converts a regular expression over a given alphabet into its equivalent Non-Deterministic Finite Automaton. The NFA should be constructed using a combination of transitions for concatenation, union (alternation), and Kleene star operations. The program should finally display the NFA transitions for each state and show how different components of the regular expression are mapped to the NFA states.

CODE:-

```
#include<stdio.h>

#include<string.h>

int main() {

    char reg[20];

    int q[20][3], i = 0, j = 1, len, a, b;

    // Initialize the transition table with 0s

    for (a = 0; a < 20; a++)

        for (b = 0; b < 3; b++)

            q[a][b] = 0;

    // Read the regular expression input

    scanf("%s", reg);

    printf("Given regular expression: %s\n", reg);


    len = strlen(reg);

    // Process the regular expression

    while (i < len) {

        if (reg[i] == 'a' && reg[i + 1] != '|' && reg[i + 1] != '*' && reg[i + 1] != '(') {

            q[j][0] = j + 1; // a transition

            j++;

        }

        if (reg[i] == 'b' && reg[i + 1] != '|' && reg[i + 1] != '*' && reg[i + 1] != '(') {
```

```
    q[j][1] = j + 1; // b transition
    j++;
}
if (reg[i] == 'e' && reg[i + 1] != '|' && reg[i + 1] != '*' && reg[i + 1] != '(') {
    q[j][2] = j + 1; // epsilon transition
    j++;
}

// Handling a | b
if (reg[i] == 'a' && reg[i + 1] == '|' && reg[i + 2] == 'b') {
    q[j][2] = ((j + 1) * 10) + (j + 3); // epsilon transition to two states
    j++;
    q[j][0] = j + 1; // a transition
    j++;
    q[j][2] = j + 3; // epsilon transition
    j++;
    q[j][1] = j + 1; // b transition
    j++;
    q[j][2] = j + 1; // epsilon transition
    j++;
    i += 2;
}

// Handling b | a
if (reg[i] == 'b' && reg[i + 1] == '|' && reg[i + 2] == 'a') {
    q[j][2] = ((j + 1) * 10) + (j + 3); // epsilon transition to two states
    j++;
    q[j][1] = j + 1; // b transition
```

```
        j++;
        q[j][2] = j + 3; // epsilon transition
        j++;
        q[j][0] = j + 1; // a transition
        j++;
        q[j][2] = j + 1; // epsilon transition
        j++;
        i += 2;
    }
   // Handling a* (Kleene star)
    if (reg[i] == 'a' && reg[i + 1] == '*') {
        q[j][2] = ((j + 1) * 10) + (j + 3); // epsilon transition
        j++;
        q[j][0] = j + 1; // a transition
        j++;
        q[j][2] = ((j + 1) * 10) + (j - 1); // epsilon transition looping back
        j++;
    }

    // Handling b* (Kleene star)
    if (reg[i] == 'b' && reg[i + 1] == '*') {
        q[j][2] = ((j + 1) * 10) + (j + 3); // epsilon transition
        j++;
        q[j][1] = j + 1; // b transition
        j++;
        q[j][2] = ((j + 1) * 10) + (j - 1); // epsilon transition looping back
        j++;
    }
```

```
        // Handling (a|b)* (Kleene star over parentheses)
        if (reg[i] == '(' && reg[i + 1] == 'a' && reg[i + 2] == '|' && reg[i + 3] == 'b' &&
reg[i + 4] == ')' && reg[i + 5] == '*') {
            q[j][2] = ((j + 1) * 10) + (j + 3); // epsilon transition
            j++;
            q[j][2] = ((j + 1) * 10) + (j + 5); // epsilon transition to next choice
            j++;
            q[j][0] = j + 1; // a transition
            j++;
            q[j][2] = j + 3; // epsilon transition
            j++;
            q[j][1] = j + 1; // b transition
            j++;
            q[j][2] = ((j + 1) * 10) + (j - 5); // epsilon transition looping back
            j++;
            i += 5;
        }

        i++;
    }

    // Print the transition table
    printf("\n\tTransition Table \n");
    printf("_\n");
    printf("Current State |\tInput |\tNext State");
    printf("\n_\n");
    for (i = 0; i <= j; i++) {
        if (q[i][0] != 0) printf("\n  q[%d]\t      |   a   |  q[%d]", i, q[i][0]);
```

```
if (q[i][1] != 0) printf("\n  q[%d]\t      |   b   |  q[%d]", i, q[i][1]);
if (q[i][2] != 0) {
```

```c
#include<stdio.h>
#include<string.h>

int main() {
    char reg[20];
    int q[20][3], i = 0, j = 1, len, a, b;

    // Initialize the transition table with 0s
    for (a = 0; a < 20; a++)
        for (b = 0; b < 3; b++)
            q[a][b] = 0;

    // Read the regular expression input
    scanf("%s", reg);
    printf("Given regular expression: %s\n", reg);

    len = strlen(reg);

    // Process the regular expression
    while (i < len) {
        if (reg[i] == 'a' && reg[i + 1] != '|' && reg[i + 1] != '*' && reg[i + 1] != '(') {
            q[j][0] = j + 1; // a transition
            j++;
        }
        if (reg[i] == 'b' && reg[i + 1] != '|' && reg[i + 1] != '*' && reg[i + 1] != '(') {
            q[j][1] = j + 1; // b transition
            j++;
        }
        if (reg[i] == 'e' && reg[i + 1] != '|' && reg[i + 1] != '*' && reg[i + 1] != '(') {
            q[j][2] = j + 1; // epsilon transition
            j++;
        }

        // Handling a | b
        if (reg[i] == 'a' && reg[i + 1] == '|' && reg[i + 2] == 'b') {
            q[j][2] = ((j + 1) * 10) + (j + 3); // epsilon transition to two states
            j++;
            q[j][0] = j + 1; // a transition
            j++;
            q[j][2] = j + 3; // epsilon transition
            j++;
            q[j][1] = j + 1; // b transition
            j++;
            q[j][2] = j + 1; // epsilon transition
            j++;
            i += 2;
        }
```

```c
int main() {
    while (i < len) {
        }

        // Handling b | a
        if (reg[i] == 'b' && reg[i + 1] == '|' && reg[i + 2] == 'a') {
            q[j][2] = ((j + 1) * 10) + (j + 3); // epsilon transition to two states
            j++;
            q[j][1] = j + 1; // b transition
            j++;
            q[j][2] = j + 3; // epsilon transition
            j++;
            q[j][0] = j + 1; // a transition
            j++;
            q[j][2] = j + 1; // epsilon transition
            j++;
            i += 2;
        }

        // Handling a* (Kleene star)
        if (reg[i] == 'a' && reg[i + 1] == '*') {
            q[j][2] = ((j + 1) * 10) + (j + 3); // epsilon transition
            j++;
            q[j][0] = j + 1; // a transition
            j++;
            q[j][2] = ((j + 1) * 10) + (j - 1); // epsilon transition looping back
            j++;
        }

        // Handling b* (Kleene star)
        if (reg[i] == 'b' && reg[i + 1] == '*') {
            q[j][2] = ((j + 1) * 10) + (j + 3); // epsilon transition
            j++;
            q[j][1] = j + 1; // b transition
            j++;
            q[j][2] = ((j + 1) * 10) + (j - 1); // epsilon transition looping back
            j++;
        }

        // Handling (a|b)* (Kleene star over parentheses)
        if (reg[i] == '(' && reg[i + 1] == 'a' && reg[i + 2] == '|' && reg[i + 3] == 'b' && reg[i + 4] == ')' && reg[i + 5] == '*') {
            q[j][2] = ((j + 1) * 10) + (j + 3); // epsilon transition
            j++;
            q[j][2] = ((j + 1) * 10) + (j + 5); // epsilon transition to next choice
            j++;
            q[j][0] = j + 1; // a transition
            j++;
            q[j][2] = j + 3; // epsilon transition
```

```c
        // Handling (a|b)* (Kleene star over parentheses)
        if (reg[i] == '(' && reg[i + 1] == 'a' && reg[i + 2] == '|' && reg[i + 3] == 'b' && reg[i + 4] == ')' && reg[i + 5] == '*') {
            q[j][2] = ((j + 1) * 10) + (j + 3); // epsilon transition
            j++;
            q[j][2] = ((j + 1) * 10) + (j + 5); // epsilon transition to next choice
            j++;
            q[j][0] = j + 1; // a transition
            j++;
            q[j][2] = j + 3; // epsilon transition
            j++;
            q[j][1] = j + 1; // b transition
            j++;
            q[j][2] = ((j + 1) * 10) + (j - 5); // epsilon transition looping back
            j++;
            i += 5;
        }

        i++;
    }

    // Print the transition table
    printf("\n\tTransition Table \n");
    printf("_\n");
    printf("Current State |\tInput |\tNext State");
    printf("\n_\n");
    for (i = 0; i <= j; i++) {
        if (q[i][0] != 0) printf("\n  q[%d]\t      |   a   |  q[%d]", i, q[i][0]);
        if (q[i][1] != 0) printf("\n  q[%d]\t      |   b   |  q[%d]", i, q[i][1]);
        if (q[i][2] != 0) {
            if (q[i][2] < 10) printf("\n  q[%d]\t      |   e   |  q[%d]", i, q[i][2]);
            else printf("\n  q[%d]\t      |   e   |  q[%d] , q[%d]", i, q[i][2] / 10, q[i][2] % 10);
        }
    }
    printf("\n_\n");

    return 0;
}
```

OUTPUT:-

```
(a|b)*c
Given regular expression: (a|b)*c

        Transition Table

_
Current State | Input | Next State
-

  q[1]        |   e   |  q[2] , q[4]
  q[2]        |   e   |  q[3] , q[7]
  q[3]        |   a   |  q[4]
  q[4]        |   e   |  q[7]
  q[5]        |   b   |  q[6]
  q[6]        |   e   |  q[7] , q[1]
-

Process returned 0 (0x0)   execution time : 9.907 s
Press ENTER to continue.
```

```
(a|b)c*
Given regular expression: (a|b)c*

        Transition Table

_
Current State | Input | Next State
-

  q[1]        |   e   |  q[2] , q[4]
  q[2]        |   a   |  q[3]
  q[3]        |   e   |  q[6]
  q[4]        |   b   |  q[5]
  q[5]        |   e   |  q[6]
-

Process returned 0 (0x0)   execution time : 49.295 s
Press ENTER to continue.
```

Q3: Write a C program to implement a Recursive Descent Parser for a given Context-Free Grammar (CFG).

The objective is to develop a Recursive Descent Parser that can handle a grammar without left recursion. The parser will analyze an input string and determine if it can be derived from the grammar by following a recursive parsing strategy. The program should output the sequence of production rules applied to generate the input string, allowing the user to observe how the grammar processes and parses the input.

CODE:-

```c
#include <stdio.h>
#include <string.h>
//recursive parser
#define SUCCESS 1
#define FAILED 0
//22BCE3939

// Function prototypes
int E(), Edash(), T(), Tdash(), F();
const char *cursor;
char string[64];

int main() {
    puts("Enter the string");
    scanf("%s", string);  // Read input from the user
    cursor = string;
    puts("");
    puts("Input            Action");
    puts("--------------------------------");

    // Call the starting non-terminal E
    if (E() && *cursor == '\0') {  // If parsing is successful and the cursor has reached the end
        puts("--------------------------------");
        puts("String is successfully parsed");
        return 0;
    } else {
        puts("--------------------------------");
```

```
        puts("Error in parsing String");
        return 1;
    }
}

// Grammar rule: E -> T E'
int E() {
    printf("%-16s E -> T E'\n", cursor);
    if (T()) {  // Call non-terminal T
        if (Edash())  // Call non-terminal E'
            return SUCCESS;
        else
            return FAILED;
    } else
        return FAILED;
}

// Grammar rule: E' -> + T E' | $
int Edash() {
    if (*cursor == '+') {
        printf("%-16s E' -> + T E'\n", cursor);
        cursor++;
        if (T()) {  // Call non-terminal T
            if (Edash())  // Call non-terminal E'
                return SUCCESS;
            else
                return FAILED;
        } else
            return FAILED;
    } else {
        printf("%-16s E' -> $\n", cursor);
        return SUCCESS;
    }
}

// Grammar rule: T -> F T'
int T() {
    printf("%-16s T -> F T'\n", cursor);
    if (F()) {  // Call non-terminal F
```

```
        if (Tdash())  // Call non-terminal T'
            return SUCCESS;
        else
            return FAILED;
    } else
        return FAILED;
}

// Grammar rule: T' -> * F T' | $
int Tdash() {
    if (cursor == ' ') {
        printf("%-16s T' -> * F T'\n", cursor);
        cursor++;
        if (F()) {  // Call non-terminal F
            if (Tdash())  // Call non-terminal T'
                return SUCCESS;
            else
                return FAILED;
        } else
            return FAILED;
    } else {
        printf("%-16s T' -> $\n", cursor);
        return SUCCESS;
    }
}

// Grammar rule: F -> ( E ) | i
int F() {
    if (*cursor == '(') {
        printf("%-16s F -> ( E )\n", cursor);
        cursor++;
        if (E()) {  // Call non-terminal E
            if (*cursor == ')') {
                cursor++;
                return SUCCESS;
            } else
                return FAILED;
        } else
            return FAILED;
```

```
    } else if (*cursor == 'i') {
        printf("%-16s F -> i\n", cursor);
        cursor++;
        return SUCCESS;
    } else
        return FAILED;
}
```

```c
#include <stdio.h>
#include <string.h>
//recursive parser
#define SUCCESS 1
#define FAILED 0
//22BCE3939

// Function prototypes
int E(), Edash(), T(), Tdash(), F();
const char *cursor;
char string[64];

int main() {
    puts("Enter the string");
    scanf("%s", string);  // Read input from the user
    cursor = string;
    puts("");
    puts("Input             Action");
    puts("--------------------------------");

    // Call the starting non-terminal E
    if (E() && *cursor == '\0') {  // If parsing is successful
        puts("--------------------------------");
        puts("String is successfully parsed");
        return 0;
    } else {
        puts("--------------------------------");
        puts("Error in parsing String");
        return 1;
    }
}

// Grammar rule: E -> T E'
int E() {
    printf("%-16s E -> T E'\n", cursor);
    if (T()) {  // Call non-terminal T
        if (Edash())  // Call non-terminal E'
            return SUCCESS;
        else
            return FAILED;
    } else
        return FAILED;
}

```

```c

// Grammar rule: E' -> + T E' | $
int Edash() {
    if (*cursor == '+') {
        printf("%-16s E' -> + T E'\n", cursor);
        cursor++;
        if (T()) {  // Call non-terminal T
            if (Edash())  // Call non-terminal E'
                return SUCCESS;
            else
                return FAILED;
        } else
            return FAILED;
    } else {
        printf("%-16s E' -> $\n", cursor);
        return SUCCESS;
    }
}

// Grammar rule: T -> F T'
int T() {
    printf("%-16s T -> F T'\n", cursor);
    if (F()) {  // Call non-terminal F
        if (Tdash())  // Call non-terminal T'
            return SUCCESS;
        else
            return FAILED;
    } else
        return FAILED;
}

// Grammar rule: T' -> * F T' | $
int Tdash() {
    if (cursor == ' ') {
        printf("%-16s T' -> * F T'\n", cursor);
        cursor++;
        if (F()) {  // Call non-terminal F
            if (Tdash())  // Call non-terminal T'
                return SUCCESS;
            else
                return FAILED;
        } else
            return FAILED;
    } else {
        printf("%-16s T' -> $\n", cursor);
        return SUCCESS;
    }
}
```

```c
int Tdash() {
    if (cursor == ' ') {
    } else {
        printf("%-16s T' -> $\n", cursor);
        return SUCCESS;
    }
}

// Grammar rule: F -> ( E ) | i
int F() {
    if (*cursor == '(') {
        printf("%-16s F -> ( E )\n", cursor);
        cursor++;
        if (E()) {  // Call non-terminal E
            if (*cursor == ')') {
                cursor++;
                return SUCCESS;
            } else
                return FAILED;
        } else
            return FAILED;
    } else if (*cursor == 'i') {
        printf("%-16s F -> i\n", cursor);
        cursor++;
        return SUCCESS;
    } else
        return FAILED;
}
```

OUTPUT:-

```
Enter the string
i+i

Input              Action
--------------------------------
i+i               E -> T E'
i+i               T -> F T'
i+i               F -> i
+i                T' -> $
+i                E' -> + T E'
i                 T -> F T'
i                 F -> i
                  T' -> $
                  E' -> $
--------------------------------
String is successfully parsed

Process returned 0 (0x0)   execution time : 1.944 s
Press ENTER to continue.
```

```
Enter the string
(i+i)*i

Input              Action
--------------------------------
(i+i)*i           E -> T E'
(i+i)*i           T -> F T'
(i+i)*i           F -> ( E )
i+i)*i            E -> T E'
i+i)*i            T -> F T'
i+i)*i            F -> i
+i)*i             T' -> $
+i)*i             E' -> + T E'
i)*i              T -> F T'
i)*i              F -> i
)*i               T' -> $
)*i               E' -> $
*i                T' -> $
*i                E' -> $
--------------------------------
Error in parsing String

Process returned 1 (0x1)   execution time : 9.338 s
Press ENTER to continue.
```

Q4: Write a C program that computes the First and Follow sets for a given context-free grammar (CFG).

The objective of this program is to:

1. Implement a C program that takes as input a context-free grammar (CFG) in standard notation.
2. Compute the **First** set for each non-terminal, which consists of terminals that begin the strings derivable from the non-terminal.
3. Compute the **Follow** set for each non-terminal, which consists of terminals that can appear immediately to the right of the non-terminal in any sentential form derived from the start symbol.

CODE:-

```
#include <ctype.h>
#include <stdio.h>
#include <string.h>
//First and FLow
//22BCE3939

// Functions to calculate Follow
void followfirst(char, int, int);
void follow(char c);

// Function to calculate First
void findfirst(char, int, int);

int count, n = 0;
// Stores the final result of the First Sets
char calc_first[10][100];
// Stores the final result of the Follow Sets
char calc_follow[10][100];
int m = 0;
// Stores the production rules
char production[10][10];
char f[10], first[10];
int k;
char ck;
```

```
int e;

int main() {
    int i, choice;
    char c, ch;
    count = 0; // Initialize count for production rules

    printf("Enter the number of production rules: ");
    scanf("%d", &count);

    // Take production rules input
    printf("Enter the grammar (e.g., S=AB | A=a):\n");
    for (i = 0; i < count; i++) {
        printf("Production %d: ", i + 1);
        scanf("%s", production[i]);
    }

    int kay;
    char done[count];
    int ptr = -1;

    // Initializing the calc_first array
    for (k = 0; k < count; k++) {
        for (kay = 0; kay < 100; kay++) {
            calc_first[k][kay] = '!';
        }
    }

    int point1 = 0, point2, xxx;
    for (k = 0; k < count; k++) {
        c = production[k][0];
        point2 = 0;
        xxx = 0;

        // Checking if First of c has already been calculated
        for (kay = 0; kay <= ptr; kay++)
            if (c == done[kay])
                xxx = 1;
```

```
        if (xxx == 1)
            continue;

        // Function call
        findfirst(c, 0, 0);
        ptr += 1;

        // Adding c to the calculated list
        done[ptr] = c;
        printf("\n First(%c) = { ", c);
        calc_first[point1][point2++] = c;

        // Printing the First Sets of the grammar
        for (i = 0; i < n; i++) {
            int lark = 0, chk = 0;
            for (lark = 0; lark < point2; lark++) {
                if (first[i] == calc_first[point1][lark]) {
                    chk = 1;
                    break;
                }
            }
            if (chk == 0) {
                printf("%c, ", first[i]);
                calc_first[point1][point2++] = first[i];
            }
        }
        printf("}\n");
        n = 0;  // Reset for the next Non-Terminal
        point1++;
    }
    printf("\n");
    printf("-----------------------------------------------\n\n");

    char donee[count];
    ptr = -1;

    // Initializing the calc_follow array
    for (k = 0; k < count; k++) {
        for (kay = 0; kay < 100; kay++) {
```

```
            calc_follow[k][kay] = '!';
        }
    }
    point1 = 0;
    int land = 0;
    for (e = 0; e < count; e++) {
        ck = production[e][0];
        point2 = 0;
        xxx = 0;

        // Checking if Follow of ck has already been calculated
        for (kay = 0; kay <= ptr; kay++)
            if (ck == donee[kay])
                xxx = 1;

        if (xxx == 1)
            continue;
        land += 1;

        // Function call
        follow(ck);
        ptr += 1;

        // Adding ck to the calculated list
        donee[ptr] = ck;
        printf(" Follow(%c) = { ", ck);
        calc_follow[point1][point2++] = ck;

        // Printing the Follow Sets of the grammar
        for (i = 0; i < m; i++) {
            int lark = 0, chk = 0;
            for (lark = 0; lark < point2; lark++) {
                if (f[i] == calc_follow[point1][lark]) {
                    chk = 1;
                    break;
                }
            }
            if (chk == 0) {
                printf("%c, ", f[i]);
```

```
                calc_follow[point1][point2++] = f[i];
            }
        }
        printf(" }\n\n");
        m = 0;  // Reset for the next Non-Terminal
        point1++;
    }
}

void follow(char c) {
    int i, j;
    // Adding "$" to the follow set of the start symbol
    if (production[0][0] == c) {
        f[m++] = '$';
    }
    for (i = 0; i < count; i++) {
        for (j = 2; j < 10; j++) {
            if (production[i][j] == c) {
                if (production[i][j + 1] != '\0') {
                    // Calculate the first of the next Non-Terminal in the
production
                    followfirst(production[i][j + 1], i, (j + 2));
                }
                if (production[i][j + 1] == '\0' && c != production[i][0]) {
                    // Calculate the follow of the Non-Terminal in the L.H.S. of
the production
                    follow(production[i][0]);
                }
            }
        }
    }
}

void findfirst(char c, int q1, int q2) {
    int j;
    // The case where we encounter a Terminal
    if (!(isupper(c))) {
        first[n++] = c;
    }
```

```
    for (j = 0; j < count; j++) {
        if (production[j][0] == c) {
            if (production[j][2] == '#') {
                if (production[q1][q2] == '\0')
                    first[n++] = '#';
                else if (production[q1][q2] != '\0' && (q1 != 0 || q2 != 0)) {
                    // Recursion to calculate First of New Non-Terminal we
encounter after epsilon
                    findfirst(production[q1][q2], q1, (q2 + 1));
                } else
                    first[n++] = '#';
            } else if (!isupper(production[j][2])) {
                first[n++] = production[j][2];
            } else {
                // Recursion to calculate First of New Non-Terminal we
encounter at the beginning
                findfirst(production[j][2], j, 3);
            }
        }
    }
}

void followfirst(char c, int c1, int c2) {
    int k;
    // The case where we encounter a Terminal
    if (!(isupper(c)))
        f[m++] = c;
    else {
        int i = 0, j = 1;
        for (i = 0; i < count; i++) {
            if (calc_first[i][0] == c)
                break;
        }
        // Including the First set of the Non-Terminal in the Follow of the
original query
        while (calc_first[i][j] != '!') {
            if (calc_first[i][j] != '#') {
                f[m++] = calc_first[i][j];
            } else {
```

```
                if (production[c1][c2] == '\0') {
                    // Case where we reach the end of a production
                    follow(production[c1][0]);
                } else {
                    // Recursion to the next symbol in case we encounter a "#"
                    followfirst(production[c1][c2], c1, c2 + 1);
                }
            }
            j++;
        }
    }
}
```

home > karan > C k.c

```c
#include <ctype.h>
#include <stdio.h>
#include <string.h>
//First and FLow
//22BCE3939

// Functions to calculate Follow
void followfirst(char, int, int);
void follow(char c);

// Function to calculate First
void findfirst(char, int, int);

int count, n = 0;
// Stores the final result of the First Sets
char calc_first[10][100];
// Stores the final result of the Follow Sets
char calc_follow[10][100];
int m = 0;
// Stores the production rules
char production[10][10];
char f[10], first[10];
int k;
char ck;
int e;

int main() {
    int i, choice;
    char c, ch;
    count = 0; // Initialize count for production rules

    printf("Enter the number of production rules: ");
    scanf("%d", &count);

    // Take production rules input
    printf("Enter the grammar (e.g., S=AB | A=a):\n");
    for (i = 0; i < count; i++) {
        printf("Production %d: ", i + 1);
        scanf("%s", production[i]);
    }

    int kay;
    char done[count];
    int ptr = -1;

    // Initializing the calc_first array
    for (k = 0; k < count; k++) {
```

home > karan > C k.c

```c
int main() {
    for (k = 0; k < count; k++) {
            printf("}\n");
            n = 0;  // Reset for the next Non-Terminal
            point1++;
        }
        printf("\n");
        printf("-----------------------------------------------\n\n");

        char donee[count];
        ptr = -1;

        // Initializing the calc_follow array
        for (k = 0; k < count; k++) {
            for (kay = 0; kay < 100; kay++) {
                calc_follow[k][kay] = '!';
            }
        }
        point1 = 0;
        int land = 0;
        for (e = 0; e < count; e++) {
            ck = production[e][0];
            point2 = 0;
            xxx = 0;

            // Checking if Follow of ck has already been calculated
            for (kay = 0; kay <= ptr; kay++)
                if (ck == donee[kay])
                    xxx = 1;

            if (xxx == 1)
                continue;
            land += 1;

            // Function call
            follow(ck);
            ptr += 1;

            // Adding ck to the calculated list
            donee[ptr] = ck;
            printf(" Follow(%c) = { ", ck);
            calc_follow[point1][point2++] = ck;

            // Printing the Follow Sets of the grammar
            for (i = 0; i < m; i++) {
                int lark = 0, chk = 0;
                for (lark = 0; lark < point2; lark++) {
                    if (f[i] == calc_follow[point1][lark]) {
```

home > karan > C k.c

```c
int main() {
    for (e = 0; e < count; e++) {
        // Adding ck to the calculated list
        donee[ptr] = ck;
        printf(" Follow(%c) = { ", ck);
        calc_follow[point1][point2++] = ck;

        // Printing the Follow Sets of the grammar
        for (i = 0; i < m; i++) {
            int lark = 0, chk = 0;
            for (lark = 0; lark < point2; lark++) {
                if (f[i] == calc_follow[point1][lark]) {
                    chk = 1;
                    break;
                }
            }
            if (chk == 0) {
                printf("%c, ", f[i]);
                calc_follow[point1][point2++] = f[i];
            }
        }
        printf(" }\n\n");
        m = 0;  // Reset for the next Non-Terminal
        point1++;
    }
}

void follow(char c) {
    int i, j;
    // Adding "$" to the follow set of the start symbol
    if (production[0][0] == c) {
        f[m++] = '$';
    }
    for (i = 0; i < count; i++) {
        for (j = 2; j < 10; j++) {
            if (production[i][j] == c) {
                if (production[i][j + 1] != '\0') {
                    // Calculate the first of the next Non-Terminal in the production
                    followfirst(production[i][j + 1], i, (j + 2));
                }
                if (production[i][j + 1] == '\0' && c != production[i][0]) {
                    // Calculate the follow of the Non-Terminal in the L.H.S. of the production
                    follow(production[i][0]);
                }
            }
        }
    }
}
```

home > karan > k.c

```c
void findfirst(char c, int q1, int q2) {
    int j;
    // The case where we encounter a Terminal
    if (!(isupper(c))) {
        first[n++] = c;
    }
    for (j = 0; j < count; j++) {
        if (production[j][0] == c) {
            if (production[j][2] == '#') {
                if (production[q1][q2] == '\0')
                    first[n++] = '#';
                else if (production[q1][q2] != '\0' && (q1 != 0 || q2 != 0)) {
                    // Recursion to calculate First of New Non-Terminal we encounter after epsilon
                    findfirst(production[q1][q2], q1, (q2 + 1));
                } else
                    first[n++] = '#';
            } else if (!isupper(production[j][2])) {
                first[n++] = production[j][2];
            } else {
                // Recursion to calculate First of New Non-Terminal we encounter at the beginning
                findfirst(production[j][2], j, 3);
            }
        }
    }
}

void followfirst(char c, int c1, int c2) {
    int k;
    // The case where we encounter a Terminal
    if (!(isupper(c)))
        f[m++] = c;
    else {
        int i = 0, j = 1;
        for (i = 0; i < count; i++) {
            if (calc_first[i][0] == c)
                break;
        }
        // Including the First set of the Non-Terminal in the Follow of the original query
        while (calc_first[i][j] != '!') {
            if (calc_first[i][j] != '#') {
                f[m++] = calc_first[i][j];
            } else {
                if (production[c1][c2] == '\0') {
                    // Case where we reach the end of a production
                    follow(production[c1][0]);
                } else {
                    // Recursion to the next symbol in case we encounter a "#"
```

```c
void followfirst(char c, int c1, int c2) {
    int k;
    // The case where we encounter a Terminal
    if (!(isupper(c)))
        f[m++] = c;
    else {
        int i = 0, j = 1;
        for (i = 0; i < count; i++) {
            if (calc_first[i][0] == c)
                break;
        }
        // Including the First set of the Non-Terminal in the Follow of the original query
        while (calc_first[i][j] != '!') {
            if (calc_first[i][j] != '#') {
                f[m++] = calc_first[i][j];
            } else {
                if (production[c1][c2] == '\0') {
                    // Case where we reach the end of a production
                    follow(production[c1][0]);
                } else {
                    // Recursion to the next symbol in case we encounter a "#"
                    followfirst(production[c1][c2], c1, c2 + 1);
                }
            }
            j++;
        }
    }
}
```

OUTPUT:-

```
Enter the number of production rules: 6
Enter the grammar (e.g., S=AB | A=a):
Production 1: S=Bb
Production 2: S=Cd
Production 3: B=aB
Production 4: B=#
Production 5: C=cC
Production 6: C=#

 First(S) = { a, b, c, d, }

 First(B) = { a, #, }

 First(C) = { c, #, }

-----------------------------------------------

 Follow(S) = { $,  }

 Follow(B) = { b,  }

 Follow(C) = { d,  }


Process returned 0 (0x0)   execution time : 42.532 s
Press ENTER to continue.
```