

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;
}

```



```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  #define MAX 100
6
7  // Stack implementation
8  char stack[MAX];
9  int top = -1;
10
11 void push(char symbol) {
12     if (top >= MAX - 1) {
13         printf("Stack Overflow!\n");
14         exit(1);
15     } else {
16         stack[++top] = symbol;
17     }
18 }
19
20 char pop() {
21     if (top < 0) {
22         printf("Stack Underflow!\n");
23         exit(1);
24     } else {
25         return stack[top--];
26     }
27 }
28
29 char peek() {
30     return stack[top];
31 }
32
33 // Function to check if a character is a terminal
34 int isTerminal(char c) {
35     return (c == 'i' || c == '+' || c == '*' || c == '(' || c == ')' || c == '$');
36 }
37
38 // Function to return production rule based on stack top and input symbol
39 const char* getProductionRule(char stackTop, char input) {
40     if (stackTop == 'E' && (input == 'i' || input == '(')) return "TZ";
41     if (stackTop == 'Z' && input == '+') return "+TZ";
42     if (stackTop == 'Z' && (input == ')' || input == '$')) return "0";
43     if (stackTop == 'T' && (input == 'i' || input == '(')) return "FX";
44     if (stackTop == 'X' && input == '*') return "*FX";
45     if (stackTop == 'X' && (input == '+' || input == ')') || input == '$')) return "0";
46     if (stackTop == 'F' && input == 'i') return "i";
47     if (stackTop == 'F' && input == '(') return "(E)";
48 }

```

```

39  const char* getProductionRule(char stackTop, char input) {
47      if (stackTop == 'F' && input == '(') return "(E)";
48
49      return NULL;
50  }
51
52  int main() {
53      char input[MAX], action[30];
54      int i = 0;
55
56      // Read input string
57      printf("Enter the input string to be parsed: ");
58      scanf("%s", input);
59      strcat(input, "$"); // Append end marker
60
61      // Initialize stack with start symbol and end marker
62      push('$');
63      push('E');
64
65      printf("\nStack\t\tInput\t\tAction\n");
66
67      while (stack[top] != '$') {
68          printf("\n");
69          for (int j = 0; j <= top; j++)
70              printf("%c", stack[j]);
71          printf("\t\t%s", (input + i));
72
73          char stackTop = pop();
74
75          if (isTerminal(stackTop)) {
76              if (stackTop == input[i]) {
77                  sprintf(action, "Matched %c", input[i]);
78                  i++;
79              } else {
80                  sprintf(action, "Error");
81                  printf("\n%s", action);
82                  exit(1);
83              }
84          } else {
85              const char* rule = getProductionRule(stackTop, input[i]);
86              if (rule != NULL) {
87                  sprintf(action, "Apply rule %s", rule);
88                  int k = strlen(rule) - 1;
89                  while (k >= 0) {
90                      if (rule[k] != '0' && rule[k] != ' ') push(rule[k]);
91                      k--;
92                  }

```

```

67     while (stack[top] != '$') {
68         ...
71         printf("\t\t%s", (input + i));
72
73         char stackTop = pop();
74
75         if (isTerminal(stackTop)) {
76             if (stackTop == input[i]) {
77                 sprintf(action, "Matched %c", input[i]);
78                 i++;
79             } else {
80                 sprintf(action, "Error");
81                 printf("\n%s", action);
82                 exit(1);
83             }
84         } else {
85             const char* rule = getProductionRule(stackTop, input[i]);
86             if (rule != NULL) {
87                 sprintf(action, "Apply rule %s", rule);
88                 int k = strlen(rule) - 1;
89                 while (k >= 0) {
90                     if (rule[k] != '0' && rule[k] != ' ') push(rule[k]);
91                     k--;
92                 }
93             } else {
94                 sprintf(action, "Error");
95                 printf("\n%s", action);
96                 exit(1);
97             }
98         }
99         printf("\t\t%s", action);
100     }
101
102     if (input[i] == '$') printf("\nInput string parsed successfully.\n");
103     else printf("\nError in parsing the input string.\n");
104
105     return 0;
106 }

```


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] == ')') {
```

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

```

1  #include<stdio.h>
2  #include<string.h>
3
4  int main() {
5      char reg[20];
6      int q[20][3], i = 0, j = 1, len, a, b;
7
8      // Initialize the transition table with 0s
9      for (a = 0; a < 20; a++)
10         for (b = 0; b < 3; b++)
11             q[a][b] = 0;
12
13     // Read the regular expression input
14     scanf("%s", reg);
15     printf("Given regular expression: %s\n", reg);
16
17     len = strlen(reg);
18
19     // Process the regular expression
20     while (i < len) {
21         if (reg[i] == 'a' && reg[i + 1] != '|' && reg[i + 1] != '*' && reg[i + 1] != '(') {
22             q[j][0] = j + 1; // a transition
23             j++;
24         }
25         if (reg[i] == 'b' && reg[i + 1] != '|' && reg[i + 1] != '*' && reg[i + 1] != '(') {
26             q[j][1] = j + 1; // b transition
27             j++;
28         }
29         if (reg[i] == 'e' && reg[i + 1] != '|' && reg[i + 1] != '*' && reg[i + 1] != '(') {
30             q[j][2] = j + 1; // epsilon transition
31             j++;
32         }
33
34         // Handling a | b
35         if (reg[i] == 'a' && reg[i + 1] == '|' && reg[i + 2] == 'b') {
36             q[j][2] = ((j + 1) * 10) + (j + 3); // epsilon transition to two states
37             j++;
38             q[j][0] = j + 1; // a transition
39             j++;
40             q[j][2] = j + 3; // epsilon transition
41             j++;
42             q[j][1] = j + 1; // b transition
43             j++;
44             q[j][2] = j + 1; // epsilon transition
45             j++;
46             i += 2;
47         }

```


OUTPUT:-

```
(alb)*c
Given regular expression: (alb)*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.
```

```
(alb)c*
Given regular expression: (alb)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:-

```
#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: $T' \rightarrow * F T' \mid \$$

```

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: $F \rightarrow (E) \mid i$

```

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;  
}
```

```

1  #include <stdio.h>
2  #include <string.h>
3  //recursive parser
4  #define SUCCESS 1
5  #define FAILED 0
6  //22BCE3939
7
8  // Function prototypes
9  int E(), Edash(), T(), Tdash(), F();
10 const char *cursor;
11 char string[64];
12
13 int main() {
14     puts("Enter the string");
15     scanf("%s", string); // Read input from the user
16     cursor = string;
17     puts("");
18     puts("Input          Action");
19     puts("-----");
20
21     // Call the starting non-terminal E
22     if (E() && *cursor == '\0') { // If parsing is successful
23         puts("-----");
24         puts("String is successfully parsed");
25         return 0;
26     } else {
27         puts("-----");
28         puts("Error in parsing String");
29         return 1;
30     }
31 }
32
33 // Grammar rule: E -> T E'
34 int E() {
35     printf("%-16s E -> T E'\n", cursor);
36     if (T()) { // Call non-terminal T
37         if (Edash()) // Call non-terminal E'
38             return SUCCESS;
39         else
40             return FAILED;
41     } else
42         return FAILED;
43 }
44

```

```

44
45 // Grammar rule: E' -> + T E' | $
46 int Edash() {
47     if (*cursor == '+') {
48         printf("%-16s E' -> + T E'\n", cursor);
49         cursor++;
50         if (T()) { // Call non-terminal T
51             if (Edash()) // Call non-terminal E'
52                 return SUCCESS;
53             else
54                 return FAILED;
55         } else
56             return FAILED;
57     } else {
58         printf("%-16s E' -> $\n", cursor);
59         return SUCCESS;
60     }
61 }
62
63 // Grammar rule: T -> F T'
64 int T() {
65     printf("%-16s T -> F T'\n", cursor);
66     if (F()) { // Call non-terminal F
67         if (Tdash()) // Call non-terminal T'
68             return SUCCESS;
69         else
70             return FAILED;
71     } else
72         return FAILED;
73 }
74
75 // Grammar rule: T' -> * F T' | $
76 int Tdash() {
77     if (cursor == ' ') {
78         printf("%-16s T' -> * F T'\n", cursor);
79         cursor++;
80         if (F()) { // Call non-terminal F
81             if (Tdash()) // Call non-terminal T'
82                 return SUCCESS;
83             else
84                 return FAILED;
85         } else
86             return FAILED;
87     } else {
88         printf("%-16s T' -> $\n", cursor);
89         return SUCCESS;
90     }
91 }

```

```

76  int Tdash() {
77      if (cursor == ' ') {
87  } else {
88      printf("%-16s T' -> $\n", cursor);
89      return SUCCESS;
90  }
91  }
92
93  // Grammar rule: F -> ( E ) | i
94  int F() {
95      if (*cursor == '(') {
96          printf("%-16s F -> ( E )\n", cursor);
97          cursor++;
98          if (E()) { // Call non-terminal E
99              if (*cursor == ')') {
100                  cursor++;
101                  return SUCCESS;
102              } else
103                  return FAILED;
104          } else
105              return FAILED;
106      } else if (*cursor == 'i') {
107          printf("%-16s F -> i\n", cursor);
108          cursor++;
109          return SUCCESS;
110      } else
111          return FAILED;
112  }

```


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

```
1  #include <ctype.h>
2  #include <stdio.h>
3  #include <string.h>
4  //First and FLOW
5  //22BCE3939
6
7  // Functions to calculate Follow
8  void followfirst(char, int, int);
9  void follow(char c);
10
11 // Function to calculate First
12 void findfirst(char, int, int);
13
14 int count, n = 0;
15 // Stores the final result of the First Sets
16 char calc_first[10][100];
17 // Stores the final result of the Follow Sets
18 char calc_follow[10][100];
19 int m = 0;
20 // Stores the production rules
21 char production[10][10];
22 char f[10], first[10];
23 int k;
24 char ck;
25 int e;
26
27 int main() {
28     int i, choice;
29     char c, ch;
30     count = 0; // Initialize count for production rules
31
32     printf("Enter the number of production rules: ");
33     scanf("%d", &count);
34
35     // Take production rules input
36     printf("Enter the grammar (e.g., S=AB | A=a):\n");
37     for (i = 0; i < count; i++) {
38         printf("Production %d: ", i + 1);
39         scanf("%s", production[i]);
40     }
41
42     int kay;
43     char done[count];
44     int ptr = -1;
45
46     // Initializing the calc_first array
47     for (k = 0; k < count; k++) {
```

home > karan > C k.c

```
27  int main() {
54      for (k = 0; k < count; k++) {
90          printf("}\n");
91          n = 0;  // Reset for the next Non-Terminal
92          point1++;
93      }
94      printf("\n");
95      printf("-----\n\n");
96
97      char donee[count];
98      ptr = -1;
99
100     // Initializing the calc_follow array
101     for (k = 0; k < count; k++) {
102         for (kay = 0; kay < 100; kay++) {
103             calc_follow[k][kay] = '!';
104         }
105     }
106     point1 = 0;
107     int land = 0;
108     for (e = 0; e < count; e++) {
109         ck = production[e][0];
110         point2 = 0;
111         xxx = 0;
112
113         // Checking if Follow of ck has already been calculated
114         for (kay = 0; kay <= ptr; kay++)
115             if (ck == donee[kay])
116                 xxx = 1;
117
118         if (xxx == 1)
119             continue;
120         land += 1;
121
122         // Function call
123         follow(ck);
124         ptr += 1;
125
126         // Adding ck to the calculated list
127         donee[ptr] = ck;
128         printf(" Follow(%c) = { ", ck);
129         calc_follow[point1][point2++] = ck;
130
131         // Printing the Follow Sets of the grammar
132         for (i = 0; i < m; i++) {
133             int lark = 0, chk = 0;
134             for (lark = 0; lark < point2; lark++) {
135                 if (f[i] == calc_follow[point1][lark]) {
```

home > karan > **C** k.c

```
27  int main() {
108      for (e = 0; e < count; e++) {
126          // Adding ck to the calculated list
127          donee[ptr] = ck;
128          printf(" Follow(%c) = { ", ck);
129          calc_follow[point1][point2++] = ck;
130
131          // Printing the Follow Sets of the grammar
132          for (i = 0; i < m; i++) {
133              int lark = 0, chk = 0;
134              for (lark = 0; lark < point2; lark++) {
135                  if (f[i] == calc_follow[point1][lark]) {
136                      chk = 1;
137                      break;
138                  }
139              }
140              if (chk == 0) {
141                  printf("%c, ", f[i]);
142                  calc_follow[point1][point2++] = f[i];
143              }
144          }
145          printf(" }\n\n");
146          m = 0; // Reset for the next Non-Terminal
147          point1++;
148      }
149  }
150
151  void follow(char c) {
152      int i, j;
153      // Adding "$" to the follow set of the start symbol
154      if (production[0][0] == c) {
155          f[m++] = '$';
156      }
157      for (i = 0; i < count; i++) {
158          for (j = 2; j < 10; j++) {
159              if (production[i][j] == c) {
160                  if (production[i][j + 1] != '\0') {
161                      // Calculate the first of the next Non-Terminal in the production
162                      followfirst(production[i][j + 1], i, (j + 2));
163                  }
164                  if (production[i][j + 1] == '\0' && c != production[i][0]) {
165                      // Calculate the follow of the Non-Terminal in the L.H.S. of the production
166                      follow(production[i][0]);
167                  }
168              }
169          }
170      }
171  }
```

```

home > karan > C k.c
172
173 void findfirst(char c, int q1, int q2) {
174     int j;
175     // The case where we encounter a Terminal
176     if (!(isupper(c))) {
177         first[n++] = c;
178     }
179     for (j = 0; j < count; j++) {
180         if (production[j][0] == c) {
181             if (production[j][2] == '#') {
182                 if (production[q1][q2] == '\0')
183                     first[n++] = '#';
184                 else if (production[q1][q2] != '\0' && (q1 != 0 || q2 != 0)) {
185                     // Recursion to calculate First of New Non-Terminal we encounter after epsilon
186                     findfirst(production[q1][q2], q1, (q2 + 1));
187                 } else
188                     first[n++] = '#';
189             } else if (!isupper(production[j][2])) {
190                 first[n++] = production[j][2];
191             } else {
192                 // Recursion to calculate First of New Non-Terminal we encounter at the beginning
193                 findfirst(production[j][2], j, 3);
194             }
195         }
196     }
197 }
198
199 void followfirst(char c, int c1, int c2) {
200     int k;
201     // The case where we encounter a Terminal
202     if (!(isupper(c)))
203         f[m++] = c;
204     else {
205         int i = 0, j = 1;
206         for (i = 0; i < count; i++) {
207             if (calc_first[i][0] == c)
208                 break;
209         }
210         // Including the First set of the Non-Terminal in the Follow of the original query
211         while (calc_first[i][j] != '!') {
212             if (calc_first[i][j] != '#') {
213                 f[m++] = calc_first[i][j];
214             } else {
215                 if (production[c1][c2] == '\0') {
216                     // Case where we reach the end of a production
217                     follow(production[c1][0]);
218                 } else {
219                     // Recursion to the next symbol in case we encounter a "#"

```



```

198
199 void followfirst(char c, int c1, int c2) {
200     int k;
201     // The case where we encounter a Terminal
202     if (!(isupper(c)))
203         f[m++] = c;
204     else {
205         int i = 0, j = 1;
206         for (i = 0; i < count; i++) {
207             if (calc_first[i][0] == c)
208                 break;
209         }
210         // Including the First set of the Non-Terminal in the Follow of the original query
211         while (calc_first[i][j] != '!') {
212             if (calc_first[i][j] != '#') {
213                 f[m++] = calc_first[i][j];
214             } else {
215                 if (production[c1][c2] == '\\0') {
216                     // Case where we reach the end of a production
217                     follow(production[c1][0]);
218                 } else {
219                     // Recursion to the next symbol in case we encounter a "#"
220                     followfirst(production[c1][c2], c1, c2 + 1);
221                 }
222             }
223             j++;
224         }
225     }
226 }

```

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.
█
```