



Experiment – 3

3.1)

Aim: Simulate First and Follow of a Grammar.

Description: In compiler design, the **First** and **Follow** functions are crucial for syntax analysis, particularly when constructing parsing tables for **LL(1) parsers**. These functions help in predicting the next possible symbols during parsing, enabling a more efficient analysis of the input stream.

1. First Function:

The **First** function determines the set of terminals that can appear at the beginning of a string derived from a given non-terminal in a grammar. Essentially, it helps in predicting which terminal symbol can appear first in the derivation of any string that can be generated from that non-terminal.

Key Points:

- The **First** of a terminal symbol is the terminal itself (i.e., $\text{First}(a) = \{a\}$ for any terminal a).
- If a non-terminal can produce a terminal symbol directly, that terminal is included in the **First** set of the non-terminal.
- If a non-terminal can produce ϵ (the empty string), then ϵ is included in its **First** set.
- If a non-terminal A produces a sequence of symbols like $A \rightarrow X_1 X_2 \dots X_n$, and X_1 is a terminal, then the terminal X_1 will be added to $\text{First}(A)$. If X_1 is a non-terminal, $\text{First}(X_1)$ is added (after excluding ϵ).

Example:

For the grammar:

$S \rightarrow AB$
 $A \rightarrow b \mid \epsilon$
 $B \rightarrow c$

- **First(S)** would include **First(A)** (since S starts with A), and since A can derive b and ϵ , **First(S)** will be $\{b, \epsilon\}$.
- **First(A)** will be $\{b, \epsilon\}$ because A can derive b and also ϵ .
- **First(B)** will be $\{c\}$ because B starts with the terminal c .

2. Follow Function:

The **Follow** function determines the set of terminal symbols that can appear immediately after a non-terminal in any derivation derived from the start symbol. The **Follow** set helps identify the possible next symbols in a parse and is crucial for constructing the **LL(1) parsing table**.

Date:

Key Points:

- The **Follow** of the start symbol always includes the end-of-input symbol ($\$$), because the start symbol can eventually derive the entire input string.
- If a non-terminal A appears in a production rule like $A \rightarrow \alpha B \beta$, then the terminals in $\text{First}(\beta)$ (excluding ϵ) are added to **Follow**(B).
- If B can derive ϵ (i.e., $B \rightarrow \epsilon$), then everything in **Follow**(A) is added to **Follow**(B).
- If A is the last symbol in a production or if A can derive ϵ , then **Follow**(A) will include **Follow**(B).

Example:

For the grammar:

$S \rightarrow AB$
 $A \rightarrow b \mid \epsilon$
 $B \rightarrow c$

- **Follow**(S) will be $\{ \$ \}$ because S is the start symbol.
- **Follow**(A) will include everything in **Follow**(S) (which is $\{ \$ \}$) because A is followed by B , and B can derive a terminal (c). Hence, **Follow**(A) = $\{ c, \$ \}$.
- **Follow**(B) will include **Follow**(S) because B is at the end of the production $S \rightarrow AB$. Hence, **Follow**(B) = $\{ \$ \}$.

Program:

```

#include <stdio.h>
#include <string.h>
#include <ctype.h>
int n, m = 0, p, i = 0, j = 0;
char a[10][10], f[10];
void follow(char c);
void first(char c);
int main() {
    int i, z;
    char c, ch;
    printf("Enter number of productions: ");
    scanf("%d", &n);
    printf("Enter the productions (epsilon = $):\n");
    for(i = 0; i < n; i++) {

```



Date:

```

    scanf("%s", a[i]);
}
do {
    m = 0;
    printf("Enter element whose FIRST & FOLLOW is to be found: ");
    scanf(" %c", &c);
    printf("FIRST(%c) = {", c);
    first(c);
    for(i = 0; i < m; i++) {
        printf("%c", f[i]);
    }
    printf("}\n");
    m = 0;
    follow(c);
    printf("FOLLOW(%c) = {", c);
    for(i = 0; i < m; i++) {
        printf("%c", f[i]);
    }
    printf("}\n");
    printf("Do you want to continue (0/1)? ");
    scanf("%d", &z);
} while(z == 1);
return 0;
}

void follow(char c) {
    int i, j;
    if(a[0][0] == c) {
        f[m++] = '$';
    }
}

```



Date:

```

    }
    for(i = 0; i < n; i++) {
        for(j = 2; j < strlen(a[i]); j++) {
            if(a[i][j] == c) {
                if(a[i][j + 1] != '\0') {
                    first(a[i][j + 1]);
                }
                if(a[i][j + 1] == '\0' && c != a[i][0]) {
                    follow(a[i][0]);
                }
            }
        }
    }
}

void first(char c) {
    int k;
    if(!(isupper(c)) && c != '$') {
        f[m++] = c;
    }
    for(k = 0; k < n; k++) {
        if(a[k][0] == c) {
            if(a[k][2] == '$') {
                follow(a[k][0]);
            } else if(islower(a[k][2])) {
                f[m++] = a[k][2];
            } else {
                first(a[k][2]);
            }
        }
    }
}

```

Date:

```

    }
}
}

```

Output:

```

C:\Users\Aditya\Desktop\CD& x + v
Enter number of productions: 3
Enter the productions (epsilon = $):
S=A
A=a
S=a
Enter element whose FIRST & FOLLOW is to be found: S
FIRST(S) = {aa}
FOLLOW(S) = {$}
Do you want to continue (0/1)? 1
Enter element whose FIRST & FOLLOW is to be found: A
FIRST(A) = {a}
FOLLOW(A) = {$}
Do you want to continue (0/1)? |

```

```

C:\Users\Aditya\Desktop\CD& x + v
Enter number of productions: 2
Enter the productions (epsilon = $):
S=A/a
A=a
Enter element whose FIRST & FOLLOW is to be found: S
FIRST(S) = {a}
FOLLOW(S) = {$}
Do you want to continue (0/1)? 1
Enter element whose FIRST & FOLLOW is to be found: A
FIRST(A) = {a}
FOLLOW(A) = {/}
Do you want to continue (0/1)? 0

-----
Process exited after 57.05 seconds with return value 0
Press any key to continue . . . |

```

Experiment 4

4.1)

Aim: Develop an operator precedence parser for a given language.

Description:

A grammar that is used to define mathematical operators is called an operator grammar or operator precedence grammar.

Operator precedence grammar is kinds of shift reduce parsing method. It is applied to a small class of operator grammars.

A grammar is said to be operator precedence grammar if it has two properties:

- No R.H.S. of any production has $a \in$.
- No two non-terminals are adjacent.

Operator precedence can only established between the terminals of the grammar. It ignores the non-terminal.

There are the three operator precedence relations:

- $a > b$ means that terminal "a" has the higher precedence than terminal "b".
- $a < b$ means that terminal "a" has the lower precedence than terminal "b".
- $a = b$ means that the terminal "a" and "b" both have same precedence.

Program:

```
#include<stdio.h>
#include<string.h>
char stack[20];
int top = -1;
void push(char item) {
    if (top >= 19) {
        printf("STACK OVERFLOW\n");
        return;
    }
    stack[++top] = item;
}
```



Date:

```
char pop() {
    if (top <= -1) {
        printf("STACK UNDERFLOW\n");
        return '\0';
    }
    char c = stack[top--];
    printf("Popped element: %c\n", c);
    return c;
}

char TOS() {
    if (top <= -1) {
        printf("STACK EMPTY\n");
        return '\0';
    }
    return stack[top];
}

// Operator precedence values

int convert(char item) {
    switch (item) {
        case 'i': return 0;
        case '+': return 1;
        case '*': return 2;
        case '(': return 3;
        case ')': return 4;
        case '$': return 5;
        default: return -1;
    }
}
```

Date:

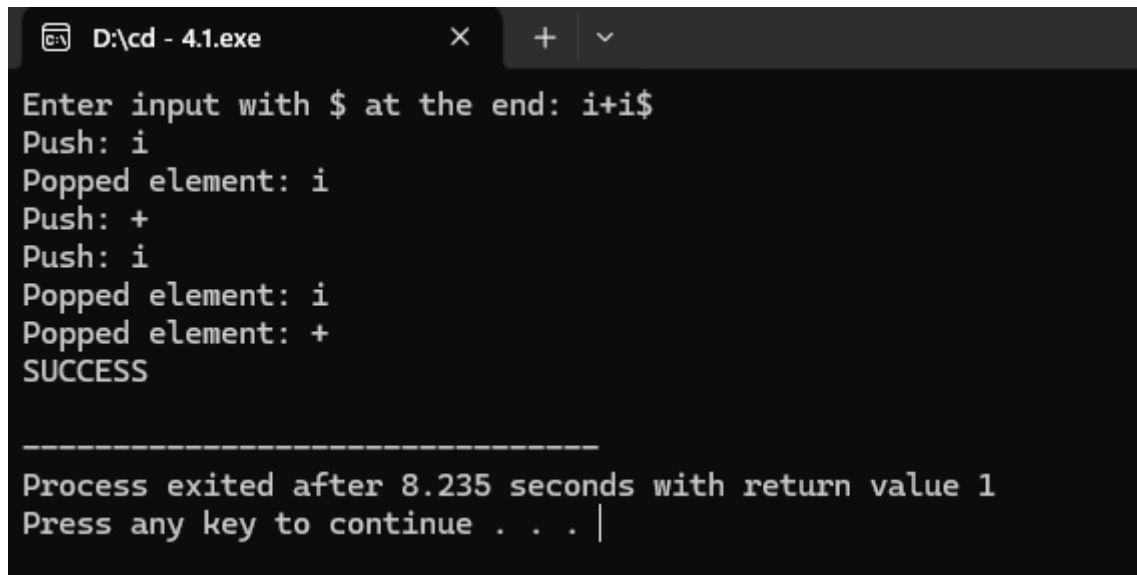
```

int main() {
    // Operator precedence table
    char pt[6][6] = {
        {'-', '>', '>', '<', '>', '>'},
        {'<', '>', '>', '<', '>', '>'},
        {'<', '>', '>', '<', '>', '>'},
        {'<', '<', '<', '<', '=', '>'},
        {'<', '<', '<', '<', '=', '>'},
        {'<', '<', '<', '<', '<', '=' }
    };
    char input[20];
    int lkh = 0;
    printf("Enter input with $ at the end: ");
    scanf("%s", input);
    push('$');
    while (lkh < strlen(input)) {
        if (TOS() == '$' && input[lkh] == '$') {
            printf("SUCCESS\n");
            return 1;
        }
        else if (convert(TOS()) != -1 && convert(input[lkh]) != -1 &&
pt[convert(TOS())][convert(input[lkh])] == '<') {
            push(input[lkh]);
            printf("Push: %c\n", input[lkh]);
            lkh++;
        } else if (convert(TOS()) != -1 && convert(input[lkh]) != -1 &&
pt[convert(TOS())][convert(input[lkh])] == '>') {
            pop();

```


Date:

```
    } else {  
        printf("FAILURE\n");  
        return 0;  
    }  
}  
  
printf("FAILURE\n");  
return 0;  
}
```

Output:A screenshot of a Windows command prompt window titled 'D:\cd - 4.1.exe'. The window shows the execution of a program that simulates a stack. The user enters 'i+i\$' as input. The program outputs: 'Push: i', 'Popped element: i', 'Push: +', 'Push: i', 'Popped element: i', 'Popped element: +', and 'SUCCESS'. Below this, a separator line is shown, followed by the message 'Process exited after 8.235 seconds with return value 1' and 'Press any key to continue . . . |'.

```
D:\cd - 4.1.exe  
Enter input with $ at the end: i+i$  
Push: i  
Popped element: i  
Push: +  
Push: i  
Popped element: i  
Popped element: +  
SUCCESS  
  
-----  
Process exited after 8.235 seconds with return value 1  
Press any key to continue . . . |
```



Date:

4.2)

Aim: Construct a recursive descent parser for an expression

Description:

Parsing is the process to determine whether the start symbol can derive the program or not. If the Parsing is successful then the program is a valid program otherwise the program is invalid.

There are generally two types of Parsers: Top-Down Parsers and Bottom-Up Parsers

Top-Down Parsers:

- In this Parsing technique we expand the start symbol to the whole program.
- Recursive Descent and LL parsers are the Top-Down parsers.

Recursive Descent Parser:

- The process of parsing is frequently employed in language processing and compiler design. It is founded on the idea that a difficult problem can be broken down into simpler problems and then solved recursively. Starting with a top-level nonterminal symbol, the parsing process proceeds by recursively expanding non terminals until it reaches terminal symbols.
- It is a kind of Top-Down Parser. A top-down parser builds the parse tree from the top to down, starting with the start non-terminal. A Predictive Parser is a special case of Recursive Descent Parser, where no Back Tracking is required.
- By carefully writing a grammar means eliminating left recursion and left factoring from it, the resulting grammar will be a grammar that can be parsed by a recursive descent parser.

Program:

```
#include<stdio.h>
#include<ctype.h>
#include<string.h>
void Tp();
void Ep();
void E();
void T();
void check();
int count, flag;
char expr[10];
int main() {
```



Date:

```
count = 0;
flag = 0;
printf("\nEnter an Algebraic Expression:\t");
scanf("%s", expr);
E();
if ((strlen(expr) == count) && (flag == 0))
printf("\nThe expression %s is valid\n", expr);
else
printf("\nThe expression %s is invalid\n", expr);
return 0;
}
void E() {
T();
Ep();
}
void T() {
check();
Tp();
}
void Tp() {
if (expr[count] == '*') {
count++;
check();
Tp();
}
}
void check() {
if (isalnum(expr[count]))
```



Date:

```
        count++;  
    else if (expr[count] == '(') {  
        count++;  
    E();  
        if (expr[count] == ')')  
            count++;  
        else  
            flag = 1;  
    }  
    else  
        flag = 1;  
}  
void Ep() {  
    if (expr[count] == '+') {  
        count++;  
    T();  
    Ep();  
    }  
}
```

Output:

Date:

```
C:\Users\Aditya\Desktop\CD8 X + v
Enter an Algebraic Expression: (8+9)*6
The expression (8+9)*6 is valid
-----
Process exited after 11.3 seconds with return value 0
Press any key to continue . . . |
```

```
C:\Users\Aditya\Desktop\CD8 X + v
Enter an Algebraic Expression: 4*7)+%
The expression 4*7)+% is invalid
-----
Process exited after 20.88 seconds with return value 0
Press any key to continue . . . |
```