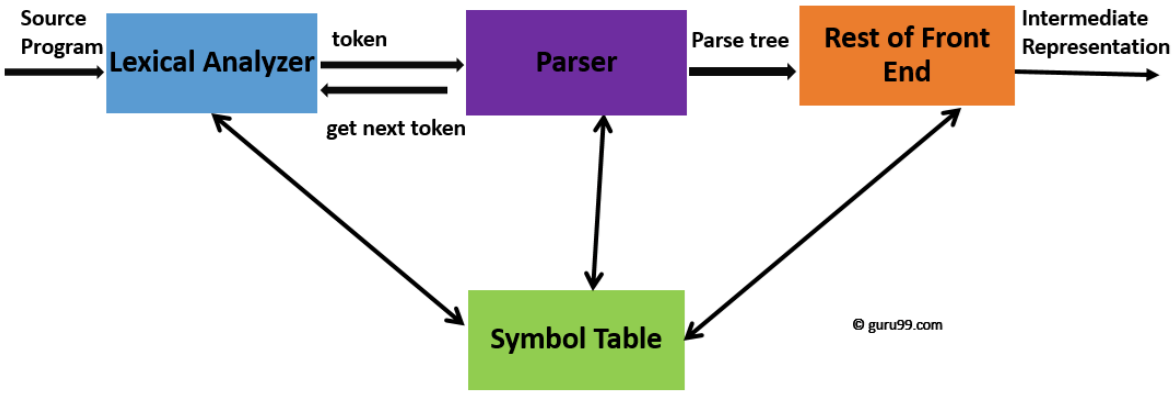
BHARATIYA VIDYA BHAVAN'S
SARDAR PATEL INSTITUTE OF TECHNOLOGY
(Empowered Autonomous Institute Affiliated to University of Mumbai)
[Knowledge is Nectar]

Department of Computer Engineering

Course - System Programming and Compiler Construction (SPCC)

UID	2021300101
Name	Adwait Purao
Class and Batch	TE Computer Engineering - Batch B
Date	15-02-2024
Lab #	3
Aim	Design syntax analyser for various grammars and implement using different parsing techniques* (Top-down and Bottom-up).
Objective	To implement syntax analysis using various parsing techniques.
Theory	**Syntax Analysis: Compiler Top Down & Bottom Up Parsing Types** **What is Syntax Analysis?[1]** Syntax Analysis is a second phase of the compiler design process in which the given input string is checked for the confirmation of rules and structure of the formal grammar. It analyses the syntactical structure and checks if the given input is in the correct syntax of the programming language or not. Syntax Analysis in Compiler Design process comes after the Lexical analysis phase. It is also known as the Parse Tree or Syntax Tree. The Parse Tree is developed with the help of pre-defined grammar of the language. The syntax analyser also checks whether a given program fulfills the rules implied by a context-free grammar. If it satisfies, the parser then creates the parse tree of that source program. Otherwise, it will display error messages.[1]



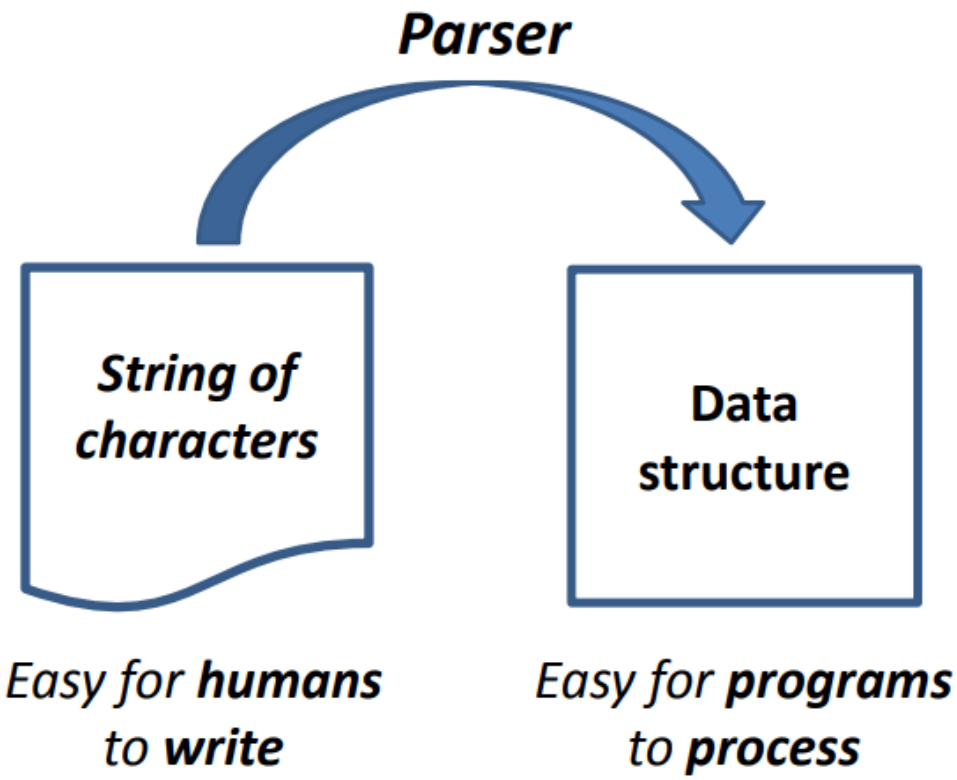
Why do you need Syntax Analyser?

- Check if the code is valid grammatically
- The syntactical analyser helps you to apply rules to the code
- Helps you to make sure that each opening brace has a corresponding closing balance
- Each declaration has a type and that the type must be exists

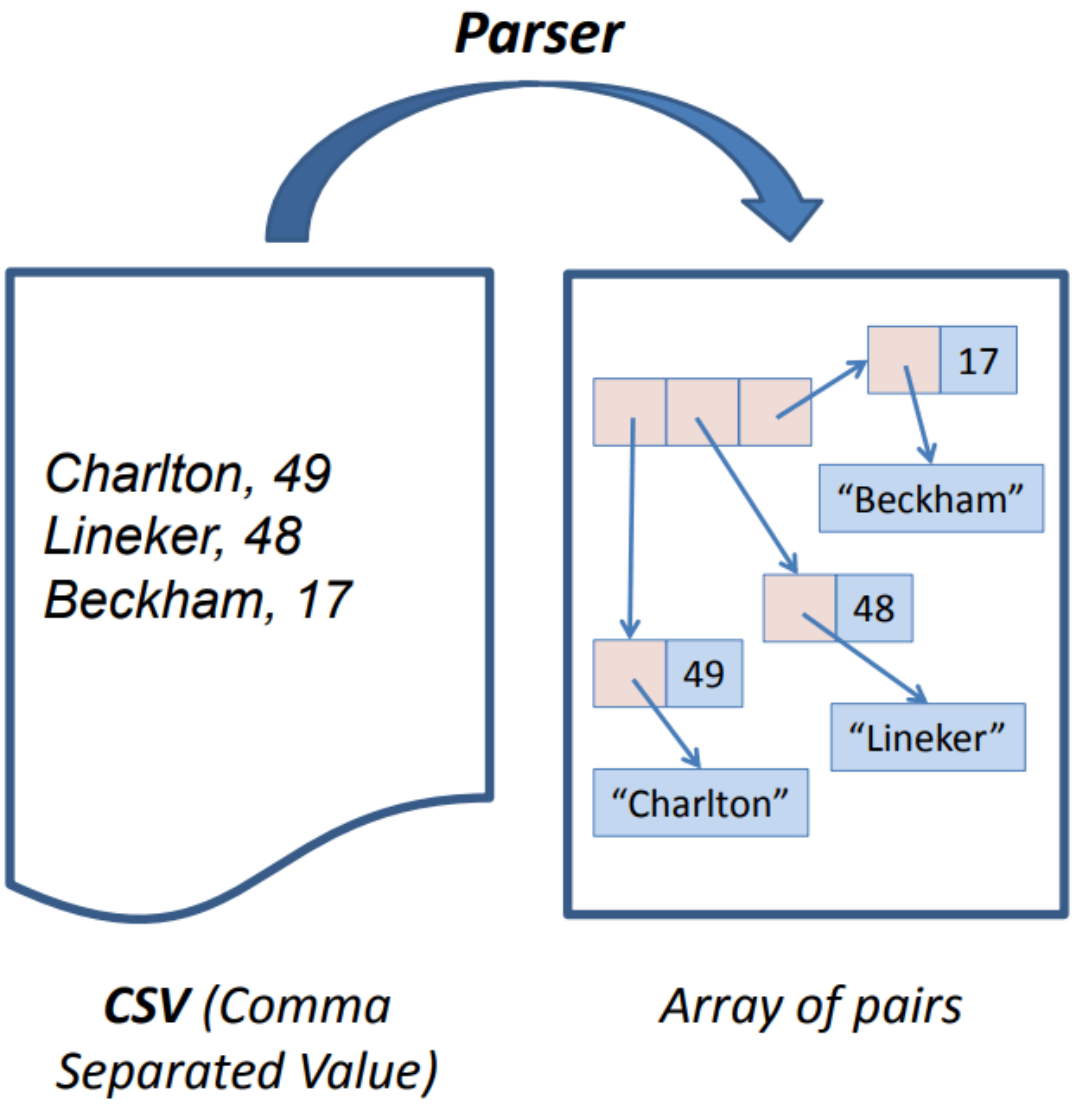
Important Syntax Analyser Terminology
Important terminologies used in syntax analysis process:

Sentence: A sentence is a group of character over some alphabet.
Lexeme: A lexeme is the lowest level syntactic unit of a language (e.g., total, start).
Token: A token is just a category of lexemes.
Keywords and reserved words – It is an identifier which is used as a fixed part of the syntax of a statement. It is a reserved word which you can't use as a variable name or identifier.
Noise words – Noise words are optional which are inserted in a statement to enhance the readability of the sentence.
Comments – It is a very important part of the documentation. It mostly display by, /* */, or//Blank (spaces)
Delimiters – It is a syntactic element which marks the start or end of some syntactic unit. Like a statement or expression, "begin"…"end", or {}.
Character set – ASCII, Unicode
Identifiers – It is a restrictions on the length which helps you to reduce the readability of the sentence.
Operator symbols – + and – performs two basic arithmetic operations.
Syntactic elements of the Language

Why do we need Parsing?



A parse also checks that the input string is well-formed, and if not, reject it.



Following are important tasks perform by the parser in compiler design:

Helps you to detect all types of Syntax errors

- Find the position at which error has occurred
- Clear & accurate description of the error.
- Recovery from an error to continue and find further errors in the code.
- Should not affect compilation of "correct" programs.
- The parse must reject invalid texts by reporting syntax errors

Construction of LL(1) Parsing Table[2]

LL(1) Parsing: Here the 1st L represents that the scanning of the Input will be done from the Left to Right manner and the second L shows that in this parsing technique, we are

going to use the Left most Derivation Tree. And finally, the 1 represents the number of look-ahead, which means how many symbols are you going to see when you want to make a decision.

Essential conditions to check first are as follows:

- The grammar is free from left recursion.
- The grammar should not be ambiguous.
- The grammar has to be left factored in so that the grammar is deterministic grammar.

These conditions are necessary but not sufficient for proving a LL(1) parser.

Algorithm to construct LL(1) Parsing Table:

- Step 1: First check all the essential conditions mentioned above and go to step 2.
- Step 2: Calculate First() and Follow() for all non-terminals.
    - First(): If there is a variable, and from that variable, if we try to drive all the strings then the beginning Terminal Symbol is called the First.
    - Follow(): What is the Terminal Symbol which follows a variable in the process of derivation.
- Step 3: For each production A –> α. (A tends to alpha)

Find First(α) and for each terminal in First(α), make entry A –> α in the table.
If First(α) contains ε (epsilon) as terminal, then find the Follow(A) and for each terminal in Follow(A), make entry A –> ε in the table.
If the First(α) contains ε and Follow(A) contains $ as terminal, then make entry A –> ε in the table for the $.
To construct the parsing table, we have two functions:
In the table, rows will contain the Non-Terminals and the column will contain the Terminal Symbols. All the Null Productions of the Grammars will go under the Follow elements and the remaining productions will lie under the elements of the First set.

Implementation / Code

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

void followfirst(char, int, int);
void findfirst(char, int, int);
void follow(char c);

int count, n = 0;
```

```
char calc_first[10][100];
char calc_follow[10][100];
int m = 0;
char production[10][10], first[10];
char f[10];
int k;
char ck;
int e;

int main(int argc, char **argv)
{
    int jm = 0;
    int km = 0;
    int i, choice;
    char c, ch;
    printf("How many productions ? :");
    scanf("%d", &count);
    printf("\nEnter %d productions in form A=B where A and B are
grammar symbols :\n\n", count);
    for (i = 0; i < count; i++)
    {
        scanf("%s%c", production[i], &ch);
    }
    int kay;
    char done[count];
    int ptr = -1;
    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;
        for (kay = 0; kay <= ptr; kay++)
            if (c == done[kay])
                xxx = 1;
        if (xxx == 1)
            continue;
        findfirst(c, 0, 0);
        ptr += 1;
        done[ptr] = c;
        printf("\n First(%c)= { ", c);
        calc_first[point1][point2++] = c;
        for (i = 0 + jm; 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");
        jm = n;
        point1++;
    }
    printf("\n");
    printf("----------------------------------------------\n\n");
```

```
    char donee[count];
    ptr = -1;
    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;
        for (kay = 0; kay <= ptr; kay++)
            if (ck == donee[kay])
                xxx = 1;
        if (xxx == 1)
            continue;
        land += 1;
        follow(ck);
        ptr += 1;
        donee[ptr] = ck;
        printf(" Follow(%c) = { ", ck);
        calc_follow[point1][point2++] = ck;
        for (i = 0 + km; 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");
        km = m;
        point1++;
    }
    char ter[10];
    for (k = 0; k < 10; k++)
    {
        ter[k] = '!';
    }
    int ap, vp, sid = 0;
    for (k = 0; k < count; k++)
    {
        for (kay = 0; kay < count; kay++)
        {
            if (!isupper(production[k][kay]) && production[k][kay] !=
'#' && production[k][kay] != '=' && production[k][kay] != '\0')
            {
                vp = 0;
                for (ap = 0; ap < sid; ap++)
                {
                    if (production[k][kay] == ter[ap])
                    {
                        vp = 1;
                        break;
                    }
                }
                if (vp == 0)
                {
```

```
                    ter[sid] = production[k][kay];
                    sid++;
                }
            }
        }
    }
    ter[sid] = '$';
    sid++;
    printf("\n\t\t\t\t\t\t\t The LL(1) Parsing Table for the above
grammer :-");

printf("\n\t\t\t\t\t\t^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
^^^^\n");

printf("\n\t\t\t=============================================================
================================================================\n");
    printf("\t\t\t\t|\t");
    for (ap = 0; ap < sid; ap++)
    {
        printf("%c\t\t", ter[ap]);
    }

printf("\n\t\t\t=============================================================
================================================================\n");
    char first_prod[count][sid];
    for (ap = 0; ap < count; ap++)
    {
        int destiny = 0;
        k = 2;
        int ct = 0;
        char tem[100];
        while (production[ap][k] != '\0')
        {
            if (!isupper(production[ap][k]))
            {
                tem[ct++] = production[ap][k];
```

```
                tem[ct++] = '_';
                tem[ct++] = '\0';
                k++;
                break;
            }
            else
            {
                int zap = 0;
                int tuna = 0;
                for (zap = 0; zap < count; zap++)
                {
                    if (calc_first[zap][0] == production[ap][k])
                    {
                        for (tuna = 1; tuna < 100; tuna++)
                        {
                            if (calc_first[zap][tuna] != '!')
                            {
                                tem[ct++] = calc_first[zap][tuna];
                            }
                            else
                                break;
                        }
                        break;
                    }
                }
                tem[ct++] = '_';
            }
            k++;
        }
        int zap = 0, tuna;
        for (tuna = 0; tuna < ct; tuna++)
        {
            if (tem[tuna] == '#')
            {
                zap = 1;
            }
```

```
            else if (tem[tuna] == '_')
            {
                if (zap == 1)
                {
                    zap = 0;
                }
                else
                    break;
            }
            else
            {
                first_prod[ap][destiny++] = tem[tuna];
            }
        }
    }
    char table[land][sid + 1];
    ptr = -1;
    for (ap = 0; ap < land; ap++)
    {
        for (kay = 0; kay < (sid + 1); kay++)
        {
            table[ap][kay] = '!';
        }
    }
    for (ap = 0; ap < count; ap++)
    {
        ck = production[ap][0];
        xxx = 0;
        for (kay = 0; kay <= ptr; kay++)
            if (ck == table[kay][0])
                xxx = 1;
        if (xxx == 1)
            continue;
        else
        {
            ptr = ptr + 1;
```

```
            table[ptr][0] = ck;
        }
    }
    for (ap = 0; ap < count; ap++)
    {
        int tuna = 0;
        while (first_prod[ap][tuna] != '\0')
        {
            int to, ni = 0;
            for (to = 0; to < sid; to++)
            {
                if (first_prod[ap][tuna] == ter[to])
                {
                    ni = 1;
                }
            }
            if (ni == 1)
            {
                char xz = production[ap][0];
                int cz = 0;
                while (table[cz][0] != xz)
                {
                    cz = cz + 1;
                }
                int vz = 0;
                while (ter[vz] != first_prod[ap][tuna])
                {
                    vz = vz + 1;
                }
                table[cz][vz + 1] = (char)(ap + 65);
            }
            tuna++;
        }
    }
    for (k = 0; k < sid; k++)
    {
```

```
        for (kay = 0; kay < 100; kay++)
        {
            if (calc_first[k][kay] == '!')
            {
                break;
            }
            else if (calc_first[k][kay] == '#')
            {
                int fz = 1;
                while (calc_follow[k][fz] != '!')
                {
                    char xz = production[k][0];
                    int cz = 0;
                    while (table[cz][0] != xz)
                    {
                        cz = cz + 1;
                    }
                    int vz = 0;
                    while (ter[vz] != calc_follow[k][fz])
                    {
                        vz = vz + 1;
                    }
                    table[k][vz + 1] = '#';
                    fz++;
                }
                break;
            }
        }
    }
    for (ap = 0; ap < land; ap++)
    {
        printf("\t\t\t %c\t|\t", table[ap][0]);
        for (kay = 1; kay < (sid + 1); kay++)
        {
            if (table[ap][kay] == '!')
                printf("\t\t");
```

```
            else if (table[ap][kay] == '#')
                printf("%c=#\t\t", table[ap][0]);
            else
            {
                int mum = (int)(table[ap][kay]);
                mum -= 65;
                printf("%s\t\t", production[mum]);
            }
        }
        printf("\n");

printf("\t\t\t-------------------------------------------------------------
-------------------------------------------------------------");
        printf("\n");
    }
    int j;
    printf("\n\nPlease enter the desired INPUT STRING = ");
    char input[100];
    scanf("%s%c", input, &ch);
    //
printf("\n\t\t\t\t\t==================================================
==========================\n");
    printf("\t\t\t\t\tStack\t\t\tInput\t\t\tAction");
    //
printf("\n\t\t\t\t\t==================================================
==========================\n");
    int i_ptr = 0, s_ptr = 1;
    char stack[100];
    stack[0] = '$';
    stack[1] = table[0][0];
    while (s_ptr != -1)
    {
        printf("\t\t\t\t\t\t");
        int vamp = 0;
        for (vamp = 0; vamp <= s_ptr; vamp++)
        {
```

```
            printf("%c", stack[vamp]);
        }
        printf("\t\t\t");
        vamp = i_ptr;
        while (input[vamp] != '\0')
        {
            printf("%c", input[vamp]);
            vamp++;
        }
        printf("\t\t\t");
        char her = input[i_ptr];
        char him = stack[s_ptr];
        s_ptr--;
        if (!isupper(him))
        {
            if (her == him)
            {
                i_ptr++;
                printf("POP ACTION\n");
            }
        }
        else
        {
            for (i = 0; i < sid; i++)
            {
                if (ter[i] == her)
                    break;
            }
            char produ[100];
            for (j = 0; j < land; j++)
            {
                if (him == table[j][0])
                {
                    if (table[j][i + 1] == '#')
                    {
                        printf("%c=#\n", table[j][0]);
```

```
                        produ[0] = '#';
                        produ[1] = '\0';
                    }
                    else if (table[j][i + 1] != '!')
                    {
                        int mum = (int)(table[j][i + 1]);
                        mum -= 65;
                        strcpy(produ, production[mum]);
                        printf("%s\n", produ);
                    }
                    else
                    {
                        printf("\nString Not Accepted by LL(1) Parser
!!\n");
                    }
                }
            }
            int le = strlen(produ);
            le = le - 1;
            if (le == 0)
            {
                continue;
            }
            for (j = le; j >= 2; j--)
            {
                s_ptr++;
                stack[s_ptr] = produ[j];
            }
        }
    }

printf("\n\t\t\t==========================================================
==========================================================\n");
    if (input[i_ptr] == '\0')
    {
        printf("\t\t\t\t\t\t\t\tYOUR STRING HAS BEEN ACCEPTED !!\n");
```

```
    }
    else
        printf("\n\t\t\t\t\t\t\t\tYOUR STRING HAS BEEN REJECTED
!!\n");

printf("\t\t\t==========================================================
============================================================\n");

    return 0;
}

void follow(char c)
{
    int i, j;
    if (production[0][0] == c)
    {
        f[m++] = '$';
    }
    for (i = 0; i < 10; i++)
    {
        for (j = 2; j < 10; j++)
        {
            if (production[i][j] == c)
            {
                if (production[i][j + 1] != '\0')
                {
                    followfirst(production[i][j + 1], i, (j + 2));
                }
                if (production[i][j + 1] == '\0' && c !=
production[i][0])
                {
                    follow(production[i][0]);
                }
            }
        }
    }
```

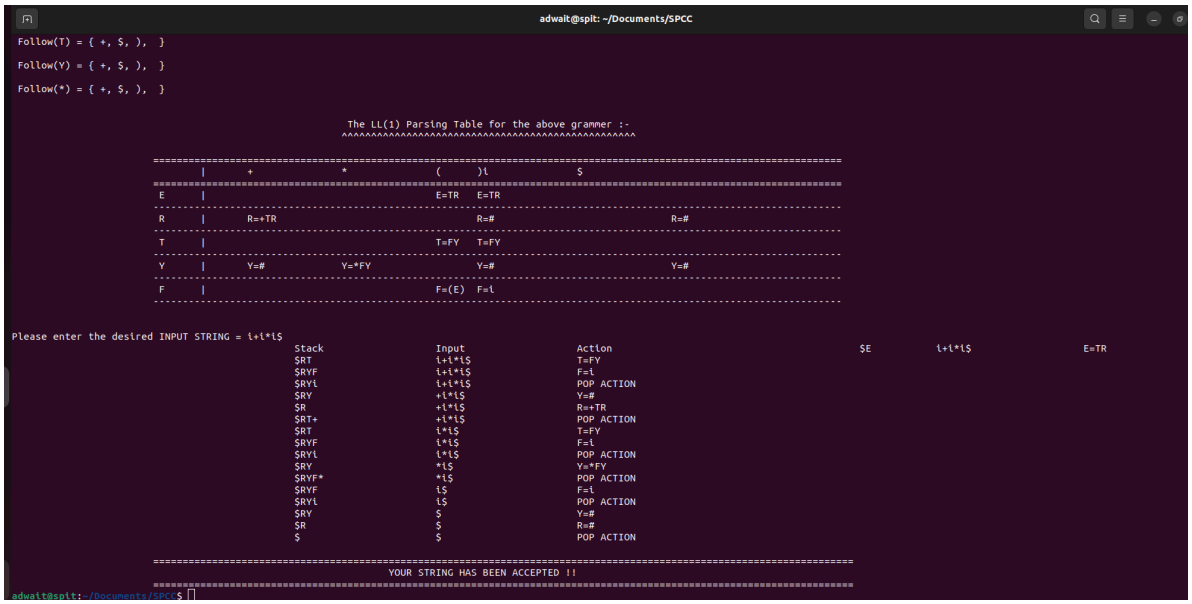
```
}

void findfirst(char c, int q1, int q2)
{
    int j;
    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))
                {
                    findfirst(production[q1][q2], q1, (q2 + 1));
                }
                else
                    first[n++] = '#';
            }
            else if (!isupper(production[j][2]))
            {
                first[n++] = production[j][2];
            }
            else
            {
                findfirst(production[j][2], j, 3);
            }
        }
    }
}
```

```
void followfirst(char c, int c1, int c2)
{
    int k;
    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;
        }
        while (calc_first[i][j] != '!')
        {
            if (calc_first[i][j] != '#')
            {
                f[m++] = calc_first[i][j];
            }
            else
            {
                if (production[c1][c2] == '\0')
                {
                    follow(production[c1][0]);
                }
                else
                {
                    followfirst(production[c1][c2], c1, c2 + 1);
                }
            }
            j++;
        }
    }
}
```

BHARATIYA VIDYA BHAVAN'S
SARDAR PATEL INSTITUTE OF TECHNOLOGY
(Empowered Autonomous Institute Affiliated to University of Mumbai)
[Knowledge is Nectar]

Department of Computer Engineering

Output	 
Conclusion	Through this experiment, I gained hands-on experience in understanding and implementing the functionality of an LL(1) parser. I successfully applied the concepts of computing the First and Follow sets for a given production, subsequently constructing a parse tree. Additionally, I successfully parsed a string to validate its correctness, and I created a corresponding stack during the parsing process.
References	[1] Syntax Analysis: Compiler Top Down & Bottom Up Parsing Types from https://www.guru99.com/syntax-analysis-parsing-types.html

	[2] Construction of LL(1) Parsing Table from https://www.geeksforgeeks.org/construction-of-ll1-parsing-table/