**EXPERIMENT NO.: 01**

**AIM**

Program to count number of characters, words and lines from a file.

**DESCRIPTION**

Given a file, i.e., a text file. Count the total number of characters, words and lines present in that file.

**ALGORITHM**

Step 1: Define c = 0, w = 0, and l = 1

Step 2: Read given file line by line.

Step 3: for each character in line; do

c = c + 1

If character is space; then

w = w + 1

Step 4: l = l + 1

Step 5: Repeat steps 2 to 4, until EOF (End Of File) is reached.

Step 6: Write c, w and l to the output

Step 7: End

**PROGRAM/CODE**

**FILE\_PATH = "./text.txt"**

**def main() -> None:**

**c, l, w = 0, 0, 0**

**with open(FILE\_PATH, "r") as f:**

**for line in f.readlines():**

**c += len(line)**

**w += len(line.split())**

**l += 1**

**print(f"Number of char: {c}, line: {l} and word: {w}")**

**if \_\_name\_\_ == "\_\_main\_\_":**

**exit(main() or 0)**

**OUTPUT**



**CONCLUSION**

Hence, Python script to obtain number of characters, words and lines has been implemented successfully.

**EXPERIMENT NO.: 02**

**AIM**

Program to identify different tokens and token types from a file.

**DESCRIPTION**

Given a file, i.e., a text file. Identify the tokens and there types(i.e., identifier, keyword, etc) present in that file.

**ALGORITHM**

Step 1: Read the given file character by character.

Step 2: Capture the lexeme into buffer, which are separated deliminators.

Step 3: Calculate the line number for each lexeme.

Step 4: Classify each lexeme as token(i.e., String, Constant, etc).

Step 5: Write the discovered tokens to output.

Step 6: Repeat steps 2 to 5, until EOF (End Of File) is reached.

Step 7: Close the file and End the problem.

**PROGRAM/CODE**

***from* enum *import* Enum**

***FILE\_PATH* = "./test.txt"**

***class* TokType(Enum):**

**String = 1**

**Constant = 2**

**Special = 3**

***def* tokens*(data: list)*:**

**toks = []**

**cl = 0**

***for* line *in* data:**

**cl += 1**

***for* tok *in* line.split():**

***try*:**

***int*(tok)**

**toks.append((cl, tok, TokType.Constant))**

***except* *ValueError*:**

***if* tok.isalpha():**

**toks.append((cl, tok, TokType.String))**

***else*:**

**toks.append((cl, tok, TokType.Special))**

***return* toks**

***def* main*()* -> None:**

**data = []**

***with* *open*(*FILE\_PATH*, "r") *as* f:**

**data = f.readlines()**

**toks = tokens(data)**

***print*(*f*"Total token(s): {*len*(toks)}")**

***print*("Tokens:-")**

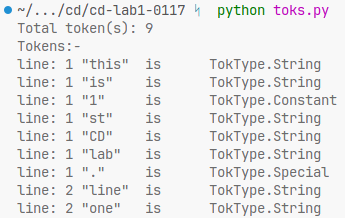
***for* tok *in* toks:**

***print*(*f*"line: {tok[0]} \"{tok[1]}\"\tis\t{tok[2]}")**

***if* *\_\_name\_\_* == "\_\_main\_\_":**

***exit*(main() *or* 0)**

**OUTPUT**



**CONCLUSION**

Hence, Python script to obtain tokens and token types from a file has been implemented successfully.

**EXPERIMENT NO.: 03**

**AIM**

Program to implement lexical analyzer without using lex tool.

**DESCRIPTION**

Given a file, i.e., a text file. Implement Lexical analyzer without using a lex tool present in that file. Lexical analysis, lexing or tokenization is the process of converting a sequence of characters into a sequence of lexical tokens. A program that performs lexical analysis may be termed a lexer, tokenizer, or scanner, although scanner is also a term for the first stage of a lexer.

**ALGORITHM**

Step 1: Read the given file character by character.

Step 2: Capture the lexeme into buffer, which are separated deliminators.

Step 3: Calculate the line number and column number for each lexeme.

Step 4: Classify each lexeme as token(i.e., Keyword, Identifier, Literal, etc).

Step 5: Write the discovered tokens to output.

Step 6: Repeat steps 2 to 5, until EOF (End Of File) is reached.

Step 7: Close the file and End the problem.

**PROGRAM/CODE**

***from* enum *import* Enum**

***FILE\_PATH* = "./hello.c"**

***K\_WORDS* = ["int", "float", "char", "return", "double", "break", "continue", "if", "else", "for", "while", "do", "include"]**

***class* TkType(Enum):**

**Ident = 0**

**Literal = 1**

**Symbol = 2**

**Const = 3**

**Keyword = 4**

***def* toks*(data: str)*:**

**a = data**

**tks = []**

**buf = []**

**n = *len*(data)**

**i = 0**

**l, c = 1, 1**

***while* i < n:**

***if* a[i] == ' ' *or* a[i] == '\t':**

**c += 1**

**i += 1**

***continue***

***if* a[i] == '\n':**

**l += 1**

**c = 1**

**i += 1**

***continue***

**buf.clear()**

***if* a[i].isalpha():**

***while* i < n *and* a[i].isalpha():**

**buf.append(a[i])**

**i += 1**

**c += 1**

**i -= 1**

***if* "".join(buf) *in* *K\_WORDS*:**

**tks.append(((l, c - *len*(buf)), "".join(buf), TkType.Keyword))**

***else*:**

**tks.append(((l, c - *len*(buf)), "".join(buf), TkType.Ident))**

***elif* a[i].isdigit():**

***while* i < n *and* a[i].isdigit():**

**buf.append(a[i])**

**i += 1**

**c += 1**

**i -= 1**

**tks.append(((l, c - *len*(buf)), "".join(buf), TkType.Const))**

***elif* a[i] == '"':**

**i += 1**

***while* i < n *and* a[i] != '"':**

**buf.append(a[i])**

**i += 1**

**c += 1**

**tks.append(((l, c - *len*(buf)), '"' + "".join(buf) + '"', TkType.Literal))**

***else*:**

**tks.append(((l, c), a[i], TkType.Symbol))**

**i += 1**

**c += 1**

***return* tks**

***def* main*()* -> None:**

**data = ""**

***with* *open*(*FILE\_PATH*, "r") *as* f:**

**data = f.read()**

**tks = toks(data)**

***print*(*f*"TokenType\tline:col:Token")**

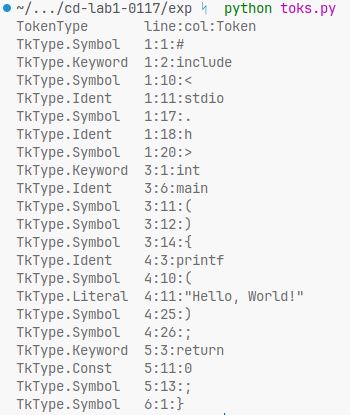
***for* tk *in* tks:**

***print*(*f*"{tk[2]}\t{tk[0][0]}:{tk[0][1]}:{tk[1]}")**

***if* *\_\_name\_\_* == "\_\_main\_\_":**

***exit*(main() *or* 0)**

**OUTPUT**



**CONCLUSION**

Hence, Program to implement lexical analyzer with a lex tool has been implemented successfully.

**EXPERIMENT NO.: 04**

**AIM**

Write a lex program to display the count of vowels and consonants in a word.

**DESCRIPTION**

Given a file, i.e., a text file. Count the total number of vowels and consonants present in that word or line. Ignore all the special character and white spaces.

**ALGORITHM**

Step 1: Read the given file character by character.

Step 2: Define vowel = 0, consonant = 0

Step 3: Apply Regular expressions for identify Vowels or Consonants.

Step 4: If Step 2 found a match for Vowels; then

vowel = vowel + 1

Else if Step 2 found a match for Real; then

consonant = consonant + 1

Step 5: Write vowel and consonant to the output

Step 6: End

**PROGRAM/CODE**

**%{**

***int* vowel = 0, cons = 0;**

**%}**

**%%**

**[aeiouAEIOU] { vowel++; }**

**[A-z] { cons++; }**

**\n { *return* 0; }**

**. {}**

**%%**

***int* yywrap() {}**

***int* main() {**

**yylex();**

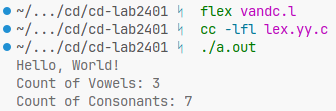
**printf("Count of Vowels: *%d*\n", vowel);**

**printf("Count of Consonants: *%d*\n", cons);**

***return* 0;**

**}**

**OUTPUT**



**CONCLUSION**

Hence, lex program to count vowels and consonants has been implemented successfully.

**EXPERIMENT NO.: 05**

**AIM**

Write a lex program to find given number is integer or a real number.

**DESCRIPTION**

Given a file, i.e., a text file. Find Integer or Real number present in that word or line. Ignore all the special character and white spaces.

**ALGORITHM**

Step 1: Read the given file character by character.

Step 2: Apply Regular expressions to identify Integer or Real Number.

Step 3: If Step 2 found a match for Integer; then

Write ‘Given String is Integer’

Else if Step 2 found a match for Real; then

Write ‘Given String is Real Number’

Else

Write ‘Invalid String’

Step 4: End

**PROGRAM/CODE**

**%{**

***#include*<stdio.h>**

**%}**

**%%**

**[0-9]\*(\.|\.e|e)[0-9]+ { printf("\"*%s*\" is a Real Number\n", yytext); }**

**[0-9]+ { printf("\"*%s*\" is an Integer\n", yytext); }**

**\n { *return* 0; }**

**. {}**

**%%**

***int* yywrap() {}**

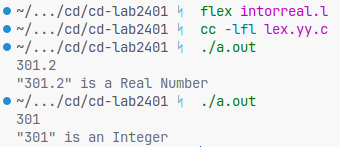
***int* main() {**

**yylex();**

***return* 0;**

**}**

**OUTPUT**



**CONCLUSION**

Hence, lex program to find given number is integer or a real number has been implemented successfully.

**EXPERIMENT NO.: 06**

**AIM**

Write a lex program to capitalize the characters.

**DESCRIPTION**

Given a file, i.e., a text file. To capitalize characters present in that word or line.

**ALGORITHM**

Step 1: Read the given file character by character.

Step 2: Apply Regular expressions to identify Lowercase letters.

Step 3: if the current encoding format is UTF-8, then subtract 32 from the character to obtain the lower case character. else use the mapping take to obtain the lower case character.

Step 4: End

**PROGRAM/CODE**

**%{**

***#include*<stdio.h>**

**%}**

**%%**

**[a-z] { printf("*%c*", yytext[0] - 32); }**

**\n { printf("\n"); *return* 0; }**

**%%**

***int* yywrap() {}**

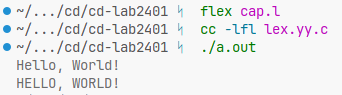
***int* main() {**

**yylex();**

***return* 0;**

**}**

**OUTPUT**



**CONCLUSION**

Hence, lex program to capitalize characters has been implemented successfully.

**EXPERIMENT NO.: 07**

**AIM**

Write a lex program to accept words starting with a or A letter.

**DESCRIPTION**

Given a file, i.e., a text file. To accept words starting with a or A present in that word or line. In Reg Ex a word is generally defined as \b\w, where \b denotes the white space and \w denotes a word.

**ALGORITHM**

Step 1: Read the given file character by character.

Step 2: Apply Regular expressions to identify a | A is present in the start of a word.

Step 3: If Step 2 found a match; then

Write ‘Given String is Accepted’

Else

Write ‘Given String is not Accepted’

Step 4: End

**PROGRAM/CODE**

**%{**

***#include*<stdio.h>**

***int* c = 0;**

**%}**

**%%**

**^[aA][a-zA-Z]+ { c++; }**

**\n { *return* 0; }**

**. {}**

**%%**

***int* yywrap() {}**

***int* main() {**

**yylex();**

***if* (c > 0) {**

**printf("Accepted!\n");**

**} *else* {**

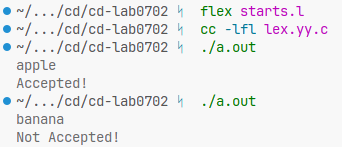
**printf("Not Accepted!\n");**

**}**

***return* 0;**

**}**

**OUTPUT**



**CONCLUSION**

Hence, lex program to accept word starting with a or A has been implemented successfully.

**EXPERIMENT NO.: 08**

**AIM**

Write a lex program to design token separator.

**DESCRIPTION**

Given a file, i.e., a text file. To separate tokens present in word or line(s). A separator is one or two tokens that separate some language features from other language features.

**ALGORITHM**

Step 1: Read the given file character by character.

Step 2: Apply Regular expressions for the different set tokens.

Step 3: Write the discovered tokens to output.

Step 4: Repeat steps 2 and 3, until EOF (End Of File) is reached.

Step 5: Close the file and End the problem.

**PROGRAM/CODE**

**%{**

***#include*<stdio.h>**

***int* l = 1;**

**%}**

**delim [ \t\b]**

**ws {delim}\***

**ident [A-Za-z][A-Za-z0-9]\***

**op [\+\-\\*/%=]**

**special [;\{\}\[\]\(\)<>]**

**%%**

**{ws}(*int*|*return*|include) { printf("*%d*\t\"*%s*\"\t\t\tKeyword\n", l, yytext); }**

**{ident} { printf("*%d*\t\"*%s*\"\t\t\tIdentifier\n", l, yytext); }**

**{op} { printf("*%d*\t\"*%s*\"\t\t\tOperator\n", l, yytext); }**

**{special} { printf("*%d*\t\"*%s*\"\t\t\tSpecial\n", l, yytext); }**

**\n { l++; }**

**. {}**

**%%**

***int* main() {**

***extern* FILE \*yyin;**

**printf("LineNumber\tLexme\t\tToken\n");**

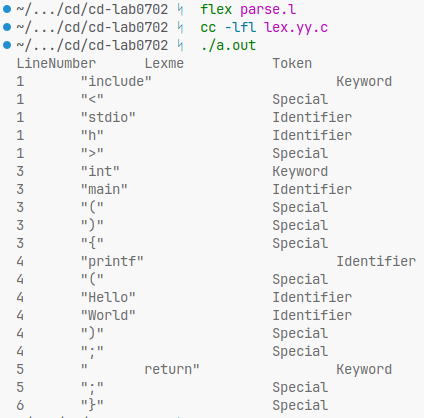
**yyin = fopen("hello.c", "r");**

**yylex();**

***return* 0;**

**}**

**OUTPUT**



**CONCLUSION**

Hence, lex program to design token separator has been implemented successfully.

**EXPERIMENT NO.: 09**

**AIM**

Write a program to implement FIRST function.

**DESCRIPTION**

.Let α be a string of tokens and nonterminals. FIRST(α) is the set of all tokens t so that α ⇒\* tβ for some string β. Additionally, FIRST(α) contains ε if α ⇒\* ε. That is, FIRST(α) contains all tokens that can begin α, plus ε if α can be erased.

**ALGORITHM**

For each nonterminal N, start with FIRST(N) = {} and add members as necessary until no more members can be added.

**To compute first(t) and first (N)**

1. If t is a token then FIRST(t) = {t}.

2. If N is a nonterminal, find all productions with N on the left-hand side. For each such production N → Y1Y2…Yn (n ≥ 0), do the following.

a. Add all tokens in FIRST(Y1) to FIRST(N).

b. For k = 2, … n−1

if ε is in every one of FIRST(Yj), for j = 1, …, k−1,

then add all tokens in FIRST(Yj) to FIRST(N).

c. If ε is in FIRST(Yj) for j = 1, …, n, then add ε to FIRST(N).

The next step is to define FIRST(α) for every string of tokens and nonterminals α.

**To compute FIRST(α) for strings α**

1. FIRST(ε) = {ε}

2. If α begins with token t then FIRST(α) = {t}.

3. If α = Nβ starts with nonterminal N, then

a. If FIRST(N) does not contain ε then FIRST(α) = FIRST(N).

b. If FIRST(N) contains ε then FIRST(α) = FIRST(N) ∪ FIRST(β).

**PROGRAM/CODE**

**E = 'Є'**

**is\_terminal = *lambda* x: *not* x.isupper()**

***def* first*(head: str, prods: dict)*:**

***if* is\_terminal(prods[head][0][0]):**

**ret = [p[0][0] *for* p *in* prods[head] *if* is\_terminal(p[0][0])]**

***for* p *in* prods[head]:**

***if* E *in* p: ret.append(E)**

***return* ret**

***return* first(prods[head][0][0], prods)**

***def* main*()*:**

**prods = {}**

**p = *int*(*input*("Enter no.of prods: "))**

***for* i *in* *range*(1, p+1):**

**buf = *input*(*f*"{i}: ")**

**head, body = [x.strip() *for* x *in* buf.split("->")]**

**body = [[y *for* y *in* x.strip().split(" ")] *for* x *in* body.split("|")]**

**prods[head] = body**

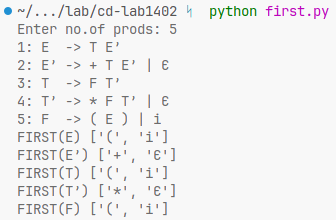
***for* (k, \_) *in* prods.items():**

***print*(*f*'FIRST({k})', first(k, prods))**

***if* *\_\_name\_\_* == "\_\_main\_\_":**

***exit*(main() *or* 0)**

**OUTPUT**



**CONCLUSION**

Hence, the program to implement FIRST function has been executed successfully.

**EXPERIMENT NO.: 10**

**AIM**

Write a program to implement FOLLOW function.

**DESCRIPTION**

Let N be a nonterminal of G. FOLLOW(N) is a set of tokens, possibly additionally including special symbol $ indicating end-of-input. FOLLOW(N) contains all tokens t so that there exists a sentential form that contains substring N t. That is, t is in FOLLOW(N) provided t can follow N in a derivation. Additionally, FOLLOW(N) contains $ if N there exists a sentential form that ends on N.

**ALGORITHM**

Start with FOLLOW(N) = {} for every nonterminal N. Then perform the following steps until none of the FOLLOW sets can be enlarged any more.

1. Add $ to FOLLOW(S), where S is the start nonterminal.

2. If there is a production A → αBβ, then add every token that is in FIRST(β) to FOLLOW(B). (Do not add ε to FOLLOW(B).

3. If there is a production A → αB, then add all members of FOLLOW(A) to FOLLOW(B). (If t can follow A, then there must be a sentential form βAtγ Using production A → αB gives sentential form βαBtγ, where B is followed by t.

4. If there is a production A → αBβ where FIRST(β) contains ε, then add all members of FOLLOW(A) to FOLLOW(B). (Reasoning is like rule 3. Just erase β.)

**PROGRAM/CODE**

**E = 'Є'**

**is\_terminal = *lambda* x: *not* x.isupper()**

**prods = {}**

**first\_set = {}**

**follow\_set = {}**

***def* first*(head: str, prods: dict)*:**

***if* is\_terminal(prods[head][0][0]):**

**ret = [p[0][0] *for* p *in* prods[head] *if* is\_terminal(p[0][0])]**

***for* p *in* prods[head]:**

***if* E *in* p: ret.append(E)**

***return* ret**

***return* first(prods[head][0][0], prods)**

***def* follow*(rel: dict)*:**

***global* E, prods, first\_set, follow\_set**

***for* cur *in* rel.keys():**

***if* cur == *list*(rel.keys())[0]:**

**follow\_set[*list*(rel.keys())[0]] = ['$']**

***for* h *in* rel[cur]:**

***for* prod *in* prods[h]:**

***if* cur *in* prod:**

**idx = prod.index(cur)**

***if* idx+1 == *len*(prod):**

***if* h != cur:**

**follow\_set[cur].extend(follow\_set[h])**

***elif* is\_terminal(prod[idx+1]):**

**follow\_set[cur].append(prod[idx+1])**

***else*:**

**f = first\_set[prod[idx+1]]**

***if* E *in* f:**

**f.remove(E)**

**follow\_set[cur].extend(f)**

***if* cur != h:**

**follow\_set[cur].extend(follow\_set[h])**

***def* main*()*:**

***global* prods, first\_set, follow\_set**

**p = *int*(*input*("Enter no.of prods: "))**

***for* i *in* *range*(1, p+1):**

**buf = *input*(*f*"{i}: ")**

**head, body = [x.strip() *for* x *in* buf.split("->")]**

**body = [[y *for* y *in* x.strip().split(" ")] *for* x *in* body.split("|")]**

**prods[head] = body**

***for* (k, \_) *in* prods.items():**

**first\_set[k] = first(k, prods)**

**rel = {k:[x[0] *for* x *in* prods.items()**

***if* *any*([k *in* y *for* y *in* x[1]])]**

***for* k *in* prods.keys()}**

**follow\_set = {k: [] *for* k *in* prods.keys()}**

**follow(rel)**

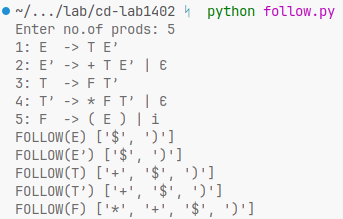
***for* (head, body) *in* follow\_set.items():**

***print*(*f*"FOLLOW({head})", body)**

***if* *\_\_name\_\_* == "\_\_main\_\_":**

***exit*(main() *or* 0)**

**OUTPUT**



**CONCLUSION**

Hence, the program to implement FOLLOW function has been executed successfully.

**EXPERIMENT NO.: 11**

**AIM**

Write a program to implement LL(1) parsing table.

**DESCRIPTION**

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

**ALGORITHM**

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.

Step 3: For each production A –> α. (A tends to alpha)

1. Find First(α) and for each terminal in First(α), make entry A –> α in the table.

2. 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.

3. If the First(α) contains ε and Follow(A) contains $ as terminal, then make entry

A –> ε in the table for the $.

**PROGRAM/CODE**

**E = 'Є'**

**is\_terminal = *lambda* x: *not* x.isupper()**

**prods = {}**

**first\_set = {}**

**follow\_set = {}**

**ll1\_table = {}**

***def* first*(head: str, prods: dict)*:**

***if* is\_terminal(prods[head][0][0]):**

**ret = [p[0][0] *for* p *in* prods[head] *if* is\_terminal(p[0][0])]**

***for* p *in* prods[head]:**

***if* E *in* p: ret.append(E)**

***return* ret**

***return* first(prods[head][0][0], prods)**

***def* follow*(rel: dict)*:**

***global* E, prods, first\_set, follow\_set**

***for* cur *in* rel.keys():**

***if* cur == *list*(rel.keys())[0]:**

**follow\_set[*list*(rel.keys())[0]] = ['$']**

***for* h *in* rel[cur]:**

***for* prod *in* prods[h]:**

***if* cur *in* prod:**

**idx = prod.index(cur)**

***if* idx+1 == *len*(prod):**

***if* h != cur:**

**follow\_set[cur].extend(follow\_set[h])**

***elif* is\_terminal(prod[idx+1]):**

**follow\_set[cur].append(prod[idx+1])**

***else*:**

**f = first\_set[prod[idx+1]].copy()**

***if* E *in* f:**

**f.remove(E)**

**follow\_set[cur].extend(f)**

***if* cur != h:**

**follow\_set[cur].extend(follow\_set[h])**

***def* ll1*()*:**

***global* E, prods, first\_set, follow\_set, ll1\_table**

***for* (k, v) *in* prods.items():**

***for* t *in* first\_set[k]:**

***if* t == E:**

***for* b *in* follow\_set[k]:**

**ll1\_table[k][b] = {k: E}**

***else*:**

**is\_t\_in\_body = [pd *for* pd *in* *range*(*len*(v)) *if* t *in* v[pd]]**

***if* *not* is\_t\_in\_body:**

**ll1\_table[k][t] = {k: v}**

***else*:**

**ll1\_table[k][t] = {k: v[is\_t\_in\_body[0]]}**

***def* main*()*:**

***global* prods, first\_set, follow\_set, ll1\_table**

**p = *int*(*input*("Enter no.of prods: "))**

***for* i *in* *range*(1, p+1):**

**buf = *input*(*f*"{i}: ")**

**head, body = [x.strip() *for* x *in* buf.split("->")]**

**body = [[y *for* y *in* x.strip().split(" ")] *for* x *in* body.split("|")]**

**prods[head] = body**

***for* (k, \_) *in* prods.items():**

**first\_set[k] = first(k, prods)**

**rel = {k:[x[0] *for* x *in* prods.items()**

***if* *any*([k *in* y *for* y *in* x[1]])]**

***for* k *in* prods.keys()}**

**follow\_set = {k: [] *for* k *in* prods.keys()}**

**follow(rel)**

**terms = *set*(['$'])**

***for* prod *in* prods.values():**

***for* l *in* prod:**

***for* el *in* l:**

***if* is\_terminal(el):**

**terms.add(el)**

**ll1\_table = {r: {c: None *for* c *in* terms} *for* r *in* first\_set.keys()}**

**ll1()**

***print*("LL(1) Parsing Table:-")**

***for* (r, c) *in* ll1\_table.items():**

***for* (h, b) *in* c.items():**

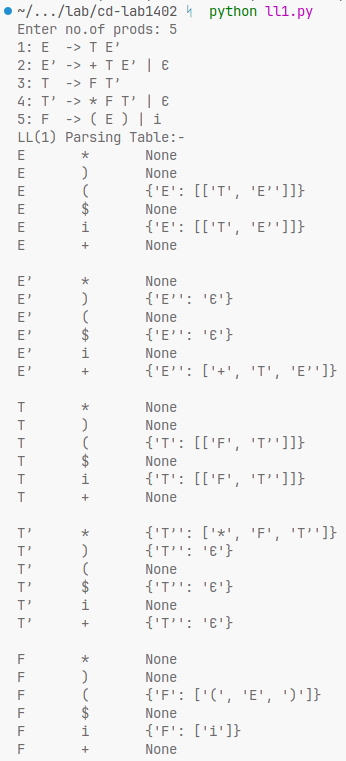
***print*(r, h, b, sep="\t")**

***print*()**

***if* *\_\_name\_\_* == "\_\_main\_\_":**

***exit*(main() *or* 0)**

**OUTPUT**



**CONCLUSION**

Hence, the program to implement LL(1) parsing table has been executed successfully.