Name – Bhargav Shamuvel Gurav	**PRN** – 2041009
Class – L.Y. B-Tech (Computer)	**Batch** – B1
Course Code – CO406U	**Course Name -** CDL

Practical no. 6

Aim: Write a program for constructing LL(1) parsing.

Theory :

LL(1) PARSER

If an LL parser uses k look-ahead tokens to parse a sentence, it is called an LL(k) parser. Because LL grammars, especially LL(1) grammars, are simple to build as parsers, many computer languages are created to be LL(1). The 1 denotes using a single input symbol for forward-looking decisions during each phase of the parsing process.

When LL(k) parsers encounter a non-terminal, they must anticipate which production it will be replaced with. The fundamental LL algorithm begins with a stack comprising [S, $] (top to bottom) and performs the relevant action till finished:

1. If a non-terminal is at the top of the stack, choose one of its productions as the top by utilizing the following k input symbols (without changing the input cursor), and proceed.

2. Read the following input token if a terminal is at the top of the stack. Pop the stack and carry on if it's the same terminal. Otherwise, the procedure ends because the parse was unsuccessful.

3. The procedure is complete if the stack is empty because the parse was successful. (We presume the input's final $ EOF-marker is unique.)

Prime Requirement of LL(1)

The following are the prime requirements for the LL(1) parser:

- The grammar must have no left factoring and no left recursion.

- FIRST() & FOLLOW()

- Parsing Table

- Stack Implementation
- Parse Tree

Algorithm to Construct LL(1) Parsing Table
Step 1: Verify the prime requirement of LL Parser before moving on to step 2.

Step 2: Perform First() and Follow() calculations on each non-terminal. Let us see these two calculations.

- First(): The first terminal symbol is referred to as the First if there is a variable, and we attempt to derive all the strings from that variable.
- Follow(): The terminal symbol that follows a variable throughout the derivation process.

Step 3. For each production, like A –> α.

- Locate First(α) and enter A -> α for each terminal in First() in the table.
- Locate Follow(A) and put an item in the table for each terminal in Follow(A) if First(α) contains epsilon as a terminal.
- Make entry A-> ε in the table for the $ if First(ε) includes and Follow(A) contains $ as terminal.

The Non-Terminals will be in the table's rows, and the Terminal Symbols will be in the table's column. The Follow elements will cover all the Grammars' Null Productions, while the First set of elements will cover the rest of the productions.

Let us see an example of an LL(1) parser.
Example
The grammar is given below:

```
G --> SG'
G' --> +SG' | ε
S --> FS'
S' --> *FS' | ε
F --> id | (G)
```

Step 1: Each of the properties in step 1 is met by the grammar.

	First	Follow
G --> SG'	{ id, (}	{ $,) }
G' --> +SG' | ε	{ +, ε }	{ $,) }
S --> FS'	{ id, (}	{ +, $,) }
S' --> *FS' | ε	{ *, ε }	{ +, $,) }
F --> id | (G)	{ id, (}	{ *, +, $,) }

Step 2: Determine first() and follow().

Step 3: The parsing table for the above grammar will be:

	id	+	*	(	)	$
G	G -> SG’			G -> SG’		
G’		G’ -> +SG’			G’ -> ε	G’ -> ε
S	S -> FS’			S -> FS’		
S’		S’ -> ε	S’ -> *FS’		S’ -> ε	S’ -> ε
F	F -> id			F -> (G)		

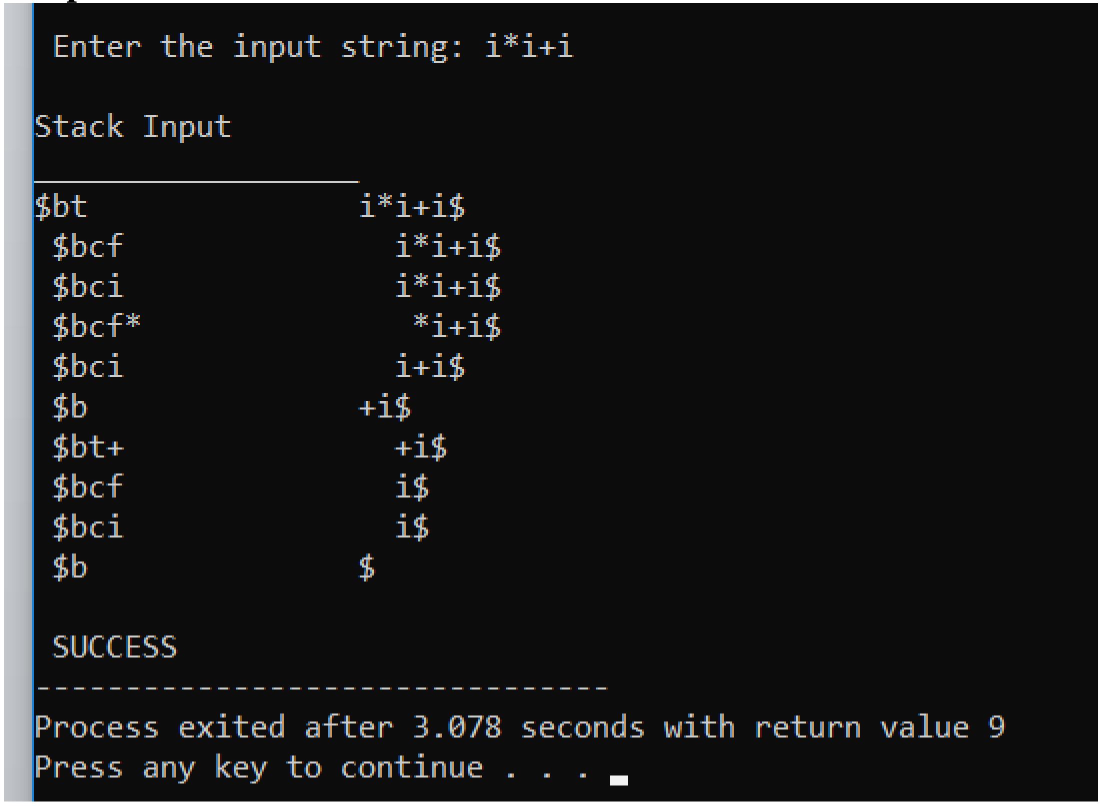
As you can see, all of the null productions are grouped under that symbol's Follow set, while the remaining creations are grouped under its First.

Program Code:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
char s[20],stack[20];
void main()
{
char m[5][6][3]={"tb"," "," ","tb"," "," "," ","+tb"," "," ","n","n","fc"," "," ","fc","
"," "," ","n","*fc"," a","n","n","i"," "," ","(e)"," "," "};

int size[5][6]={2,0,0,2,0,0,0,3,0,0,1,1,2,0,0,2,0,0,0,1,3,0,1,1,1,0,0,3,0,0};
int i,j,k,n,str1,str2;

printf("\n Enter the input string: ");
scanf("%s",s);
strcat(s,"$");
n=strlen(s);
stack[0]='$';
stack[1]='e';
i=1;
j=0;
printf("\nStack Input\n");
printf("__________________\n");
while((stack[i]!='$')&&(s[j]!='$'))
{
if(stack[i]==s[j])
{
i--;
j++;

}
switch(stack[i])
{
case 'e': str1=0;
break;
case 'b': str1=1;
break;
case 't': str1=2;
break;
case 'c': str1=3;
break;
case 'f': str1=4;
break;

}
switch(s[j])
{
case 'i': str2=0;
break;
case '+': str2=1;
break;
case '*': str2=2;
break;
case '(': str2=3;
break;
case ')': str2=4;
break;
case '$': str2=5;
break;

}
if(m[str1][str2][0]=='\0')
{
printf("\nERROR");
exit(0);
}
else if(m[str1][str2][0]=='n')
i--;
else if(m[str1][str2][0]=='i')

stack[i]='i';
else
{
for(k=size[str1][str2]-1;k>=0;k--)
{
stack[i]=m[str1][str2][k];
i++;
}
i--;
}
```

```
for(k=0;k<=i;k++)
printf("%c",stack[k]);
printf("%15s", " ");
for(k=j;k<=n;k++)
printf("%c",s[k]);
printf(" \n ");
}
printf("\n SUCCESS");
}
```

Output:



```
 Enter the input string: i*i+i

Stack Input
__________________
$bt                  i*i+i$
 $bcf                  i*i+i$
 $bci                  i*i+i$
 $bcf*                  *i+i$
 $bci                  i+i$
 $b                  +i$
 $bt+                  +i$
 $bcf                  i$
 $bci                  i$
 $b                  $

 SUCCESS
--------------------------------
Process exited after 3.078 seconds with return value 9
Press any key to continue . . .
```

Conclusion : In this practical we implemented predictive parser.