

<b>Name</b>	Shubham Golwal
<b>UID no.</b>	2020300015
<b>Experiment No.</b>	04-b

<b>AIM:</b>	To Implement a LL(1) parser for the given grammar
-------------	---

### Program 1

<b>PROBLEM STATEMENT:</b>	Implement a LL(1) parser for the given grammar
---------------------------	--

<b>ALGORITHM:</b>	<p>A top-down parser builds the parse tree from the top down, starting with the start non-terminal. There are two types of Top-Down Parsers:</p> <ol style="list-style-type: none"> <li>1. Top-Down Parser with Backtracking</li> <li>2. Top-Down Parsers without Backtracking</li> </ol> <p>Prerequisite - Classification of top-down parsers, FIRST Set, FOLLOW Set Top-Down Parsers without backtracking can further be divided into two parts:</p> <div style="text-align: center; margin: 20px 0;"> <pre> graph TD     A[Top Down Parsers] --&gt; B[TDP with full Backtracking]     A --&gt; C[TDP without Backtracking]     B --&gt; D[Bruteforce Method]     C --&gt; E[Recursive Descent]     C --&gt; F[Non-Recursive Descent (LL(1))] </pre> </div> <p>In this article, we are going to discuss Non-Recursive Descent which is also known as LL(1) Parser.</p> <p><b>LL(1) Parsing:</b> Here the 1st <b>L</b> represents that the scanning of the Input will be done from Left to Right manner and the second <b>L</b> shows that in this parsing technique we are going to use Left most Derivation Tree. And finally, the <b>1</b> represents the number of look-ahead, which means how many symbols are you going to see when you want to make a decision.</p>
-------------------	---

**Essential conditions to check first are as follows:**

1. The grammar is free from left recursion.
2. The grammar should not be ambiguous.
3. 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.

1. **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.
2. **Follow()**: What is the Terminal Symbol which follows a variable in the process of derivation.

**Step 3:** For each production  $A \rightarrow \alpha$ . ( $\alpha$  tends to alpha)

1. Find First( $\alpha$ ) and for each terminal in First( $\alpha$ ), make entry  $A \rightarrow \alpha$  in the table.
2. If First( $\alpha$ ) contains  $\epsilon$  (epsilon) as terminal, then find the Follow(A) and for each terminal in Follow(A), make entry  $A \rightarrow \epsilon$  in the table.
3. If the First( $\alpha$ ) contains  $\epsilon$  and Follow(A) contains \$ as terminal, then make entry  $A \rightarrow \epsilon$  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.

Now, let's understand with an example.

**Example-1:** Consider the Grammar:

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \epsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \epsilon$

$F \rightarrow id \mid (E)$

\* $\epsilon$  denotes epsilon

**Step1** – The grammar satisfies all properties in step 1

**Step2** – calculating first() and follow() Find their First and Follow sets:

	First	Follow
<b>E</b> $\rightarrow$ <b>TE'</b>	{ id, ( }	{ \$, ) }
<b>E'</b> $\rightarrow$ <b>+TE'/<math>\epsilon</math></b>	{ +, $\epsilon$ }	{ \$, ) }
<b>T</b> $\rightarrow$ <b>FT'</b>	{ id, ( }	{ +, \$, ) }
<b>T'</b> $\rightarrow$ <b>*FT'/<math>\epsilon</math></b>	{ *, $\epsilon$ }	{ +, \$, ) }
<b>F</b> $\rightarrow$ <b>id/(E)</b>	{ id, ( }	{ *, +, \$, ) }

**Step 3** – making parser table Now, the LL(1) Parsing Table is:

	id	+	*	(	)	\$
<b>E</b>	$E \rightarrow TE'$			$E \rightarrow TE'$		
<b>E'</b>		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
<b>T</b>	$T \rightarrow FT'$			$T \rightarrow FT'$		
<b>T'</b>		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
<b>F</b>	$F \rightarrow id$			$F \rightarrow (E)$		

As you can see that all the null productions are put under the Follow set of that symbol and all the remaining productions lie under the First of that symbol.

**Note:** Every grammar is not feasible for LL(1) Parsing table. It may be possible that one cell may contain more than one production.

Let's see an example.

**Example 2:** Consider the Grammar

$S \rightarrow (L) \mid a$

$L \rightarrow SL'$

$L' \rightarrow )SL' \mid \varepsilon$

**Step1** – The grammar satisfies all properties in step 1

**Step 2** – calculating first() and follow()

	First	Follow
<b>S</b>	{ ( , a }	{ \$, ) }
<b>L</b>	{ ( , a }	{ ) }
<b>L'</b>	{ ) , $\varepsilon$ }	{ ) }

**Step 3** – making parser table Parsing Table:

	(	)	a	\$
<b>S</b>	$S \rightarrow (L)$		$S \rightarrow a$	
<b>L</b>	$L \rightarrow SL'$		$L \rightarrow SL'$	
<b>L'</b>	$L' \rightarrow )SL'$	$L' \rightarrow \varepsilon$		

Here, we can see that there are two productions in the same cell. Hence, this grammar is not feasible for LL(1) Parser. Although the grammar satisfies all the essential conditions in step 1, it is still not feasible for LL(1) Parser. We saw in example 2 that we must have these essential conditions and in example 3 we saw that those conditions are insufficient to be a LL(1) parser.

<b>Program</b>	<pre>#include &lt;iostream&gt; #include &lt;vector&gt; #include &lt;stack&gt; #include &lt;unordered_map&gt;  using namespace std;  // global variables // parsing table from which string to be parsed unordered_map&lt;char, string&gt; table;  /// @brief function to create parse string using stack /// @param str string to be parsed /// @return true if parsed successfully else false bool parseString(string str);  bool parseString(string str) { // initialize stack stack&lt;char&gt; st;  // push '\$' and start symbol 'S' onto the stack st.push('\$'); st.push('S');  // initialize variable i to zero int i = 0; int i = 0;  // run while loop until stack is not empty and i is less than size of string while (!st.empty() &amp;&amp; i &lt; str.size()) { // if stack top symbol is equal to current string character then // pop the topmost character from the stack and increment  if (str[i] == st.top()) { st.pop(); i++; } }</pre>
----------------	---

```

// else if stack top symbol is 'S' then pop from the stack and perform parsing algorithm
else if (st.top() == 'S')
{
st.pop();
for (int j = table[str[i]].size() - 1; j >= 0; j--)

if (table[str[i]][j] != '#')
st.push(table[str[i]][j]);
}
// else break as the stack top symbol is not same as c haracter and also not the non-
terminal
else
break;
}
// return if stack is empty
return st.empty();

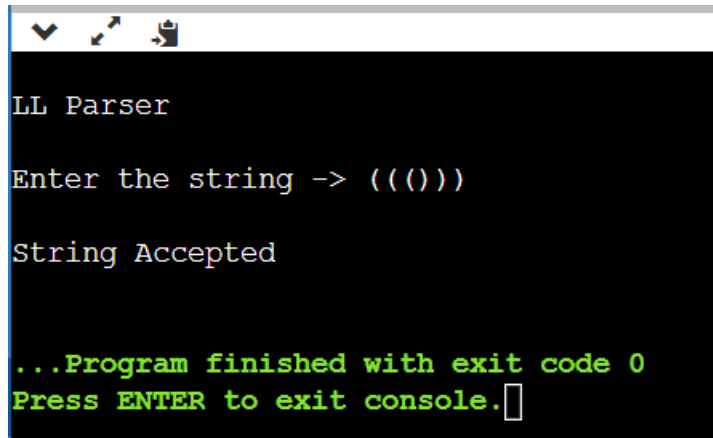
}

int main()
{
// string to be parsed
string str;
cout << "\nLL Parser\n\n";
// take string input from user
cout << "Enter the string -> "; cin >> str;
// if string is # then delete the string
if (str == "#")
str = "";
// append '$' at the end of the string
str += "$";
// create parsing table
table['('] = "(S)";
table[')'] = "#";
table['$'] = "#";
cout << (parseString(str) ? "\nString Accepted" : "\nString Rejected") << endl;
return 0;
}

```

**RESULT:**

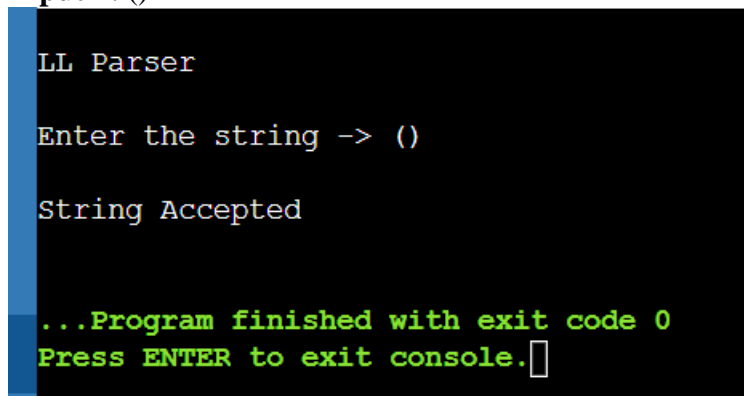
**Input 1: (())**



```
LL Parser
Enter the string -> (())
String Accepted

...Program finished with exit code 0
Press ENTER to exit console.
```

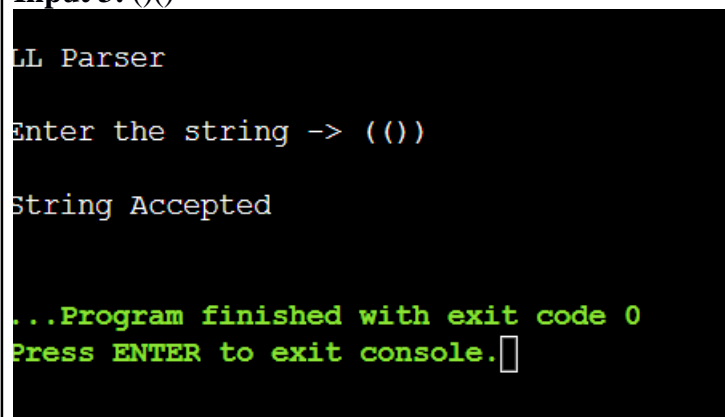
**Input 2: ()**



```
LL Parser
Enter the string -> ()
String Accepted

...Program finished with exit code 0
Press ENTER to exit console.
```

**Input 3: (())**



```
LL Parser
Enter the string -> (())
String Accepted

...Program finished with exit code 0
Press ENTER to exit console.
```

<b>CONCLUSION:</b>	<ul style="list-style-type: none"><li>• We have started with an introduction to LL(1) parser in compiler design, then gave a brief description about LL(1) parser with examples.</li><li>• We also discussed rules to calculate parsing table and solved examples on it.</li><li>• We need to find parsing table for a given grammar so that the parser can properly apply the needed rule at the correct position and can check if the string is parsable or not.</li></ul>
--------------------	--