**Experiment No:-08**

**Aim: -**To study and implement LL1 parser.

**Theory: -**

In [computer science](https://en.wikipedia.org/wiki/Computer_science), an **LL parser** is a [top-down parser](https://en.wikipedia.org/wiki/Top-down_parsing) for a subset of [context-free languages](https://en.wikipedia.org/wiki/Context-free_languages). It parses the input from **L**eft to right, performing [**leftmost**](https://en.wikipedia.org/wiki/Context-free_grammar#Derivations_and_syntax_trees) derivation of the sentence. An LL parser is called an LL(*k*) parser if it uses *k* [tokens](https://en.wikipedia.org/wiki/Token_(parser)) of [look ahead](https://en.wikipedia.org/wiki/Parsing#Lookahead) when parsing a sentence. If such a parser exists for a certain grammar and it can parse sentences of this grammar without [backtracking](https://en.wikipedia.org/wiki/Backtracking) then it is called an LL(*k*) grammar. LL(*k*) grammars can generate more languages the higher the number *k* of look ahead tokens.[[1]](https://en.wikipedia.org/wiki/LL_parser#cite_note-1) A corollary of this is that not all context-free languages can be recognized by an LL(k) parser. An LL parser is called an LL(*\**) parser (an LL-regular parser[[2]](https://en.wikipedia.org/wiki/LL_parser#cite_note-GruneJacobs2007-2)) if it is not restricted to a finite *k*tokens of look ahead, but can make parsing decisions by recognizing whether the following tokens belong to a [regular language](https://en.wikipedia.org/wiki/Regular_language) (for example by means of a [Deterministic Finite Automaton](https://en.wikipedia.org/wiki/Deterministic_Finite_Automaton)). LL grammars, particularly LL(1) grammars, are of great practical interest, as parsers for these grammars are easy to construct, and many [computer languages](https://en.wikipedia.org/wiki/Computer_language) are designed to be LL(1) for this reason. LL parsers are table-based parsers, similar to LR parsers. LL grammars can also be parsed by [recursive descent parsers](https://en.wikipedia.org/wiki/Recursive_descent_parser).

**Parser:**

The LL(k) parser is a [deterministic pushdown automaton](https://en.wikipedia.org/wiki/Deterministic_pushdown_automaton) with the ability to peek on the next k input symbols without reading. This capability can be emulated by storing the lookahead buffer contents in the finite state space, since both buffer and input alphabet are finite in size. As a result, this does not make the automaton more powerful, but is a convenient abstraction.

The stack alphabet \Gamma = N \cup \Sigma, where:

* Nis the set of non-terminals;
* \Sigmathe set of terminal (input) symbols with a special end-of-input (EOI) symbol $.

The parser stack initially contains the starting symbol above the EOI: [\ S\ $\ ]. During operation, the parser repeatedly replaces the symbol X on top of the stack:

* with some \alpha, if X \in N and there is a rule X \to \alpha;
* with \epsilon (in some notations \lambda), i.e. X is popped off the stack, if X \in \Sigma. In this case, an input symbol x is read and if x \neq X, the parser rejects the input.

If the last symbol to be removed from the stack is the EOI, the parsing is successful; the automaton accepts via an empty stack.

The states and the transition function are not explicitly given; they are specified (generated) using a more convenient *parse table* instead. The table provides the following mapping:

* row: top-of-stack symbol X
* column: |w| \le k lookahead buffer contents
* cell: rule number for X \to \alpha or \epsilon

If the parser cannot perform a valid transition, the input is rejected (empty cells). To make the table more compact, only the non-terminal rows are commonly displayed, since the action is the same for terminals.

### Setup:

To explain an LL(1) parser's workings we will consider the following small LL(1) grammar:

1. S → F
2. S → ( S + F )
3. F → a

and parse the following input:

**( a + a )**

We construct a parsing table for this grammar by expanding all the terminals by column and all non terminals by row. Later, the expressions are numbered by the position where the columns and rows cross. For example, the terminal '(' and non-terminal 'S' match for expression number 2. The table is as follows:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | ( | ) | a | + | $ |
| S | 2 | - | 1 | - | - |
| **F** | - | - | 3 | - | - |

(Note that there is also a column for the special terminal, represented here as **$**, that is used to indicate the end of the input stream.)

### Parsing procedure:

In each step, the parser reads the next-available symbol from the input stream, and the top-most symbol from the stack. If the input symbol and the stack-top symbol match, the parser discards them both, leaving only the unmatched symbols in the input stream and on the stack. Thus, in its first step, the parser reads the input symbol '**(**' and the stack-top symbol 'S'. The parsing table instruction comes from the column headed by the input symbol '**(**' and the row headed by the stack-top symbol 'S'; this cell contains '2', which instructs the parser to apply rule (2). The parser has to rewrite 'S' to '**(** S **+** F **)**' on the stack by removing 'S' from stack and pushing '(', 'S', '+', 'F', ')' onto the stack and this writes the rule number 2 to the output. The stack then becomes:

[ **(**, S, **+**, F, **)**, **$** ]

Since the '**(**' from the input stream did not match the top-most symbol, 'S', from the stack, it was not removed, and remains the next-available input symbol for the following step. In the second step, the parser removes the '**(**' from its input stream and from its stack, since they now match. The stack now becomes:

[ S, **+**, F, **)**, **$** ]

Now the parser has an '**a'** on its input stream and an 'S' as its stack top. The parsing table instructs it to apply rule (1) from the grammar and write the rule number 1 to the output stream. The stack becomes:

[ F, **+**, F, **)**, **$** ]

The parser now has an '**a'** on its input stream and an 'F' as its stack top. The parsing table instructs it to apply rule (3) from the grammar and write the rule number 3 to the output stream. The stack becomes:

[ **a**, **+**, F, **)**, **$** ]

In the next two steps the parser reads the '**a'** and '**+'** from the input stream and, since they match the next two items on the stack, also removes them from the stack. This results in:

[ F, **)**, **$** ]

In the next three steps the parser will replace '**F'** on the stack by '**a'**, write the rule number 3 to the output stream and remove the '**a'** and '**)'** from both the stack and the input stream. The parser thus ends with '**$'** on both its stack and its input stream. In this case the parser will report that it has accepted the input string and write the following list of rule numbers to the output stream:

[ 2, 1, 3, 3 ]

This is indeed a list of rules for a [leftmost derivation](https://en.wikipedia.org/wiki/Context-free_grammar#Derivations_and_syntax_trees) of the input string, which is:

S → **(** S **+** F **)** → **(** F **+** F **)** → **( a +** F **)** → **( a + a )**

***Code: -***

#include <iostream.h>

#include <conio.h>

#include <string.h>

#include <stdio.h>

#include <stdlib.h>

void main()

{

clrscr();

int i=0,j=0,k=0,m=0,n=0,o=0,o1=0,var=0,l=0,f=0,c=0,f1=0;

char str[30],str1[40]="E",temp[20],temp1[20],temp2[20],tt[20],t3[20];

strcpy(temp1,'\0');

strcpy(temp2,'\0');

char t[10];

char array[6][5][10] = {

"NT", "<id>","+","\*",";",

"E", "Te","Error","Error","Error",

"e", "Error","+Te","Error","\0",

"T", "Vt","Error","Error","Error",

"t", "Error","\0","\*Vt","\0",

"V", "<id>","Error","Error","Error"

};

cout << "\n\tLL(1) PARSER TABLE \n";

for(i=0;i<6;i++)

{

for(j=0;j<5;j++)

{

cout.setf(ios::right);

cout.width(10);

cout<<array[i][j];

}

cout<<endl;

}

cout << endl;

cout << "\n\tENTER THE STRING :";

gets(str);

if(str[strlen(str)-1] != ';')

{

cout << "END OF STRING MARKER SHOULD BE ';'";

getch();

exit(1);

}

cout << "\n\tCHECKING VALIDATION OF THE STRING ";

cout <<"\n\t" << str1;

i=0;

while(i<strlen(str))

{

again:

if(str[i] == ' ' && i<strlen(str))

{

cout << "\n\tSPACES IS NOT ALLOWED IN SOURSE STRING ";

getch();

exit(1);

}

temp[k]=str[i];

temp[k+1]='\0';

f1=0;

again1:

if(i>=strlen(str))

{

getch();

exit(1);

}

for(int l=1;l<=4;l++)

{

if(strcmp(temp,array[0][l])==0)

{

f1=1;

m=0,o=0,var=0,o1=0;

strcpy(temp1,'\0');

strcpy(temp2,'\0');

int len=strlen(str1);

while(m<strlen(str1) && m<strlen(str))

{

if(str1[m]==str[m])

{

var=m+1;

temp2[o1]=str1[m];

m++;

o1++;

}

else

{

if((m+1)<strlen(str1))

{

m++;

temp1[o]=str1[m];

o++;

}

else

m++;}

}

temp2[o1] = '\0';

temp1[o] = '\0';

t[0] = str1[var];

t[1] = '\0';

for(n=1;n<=5;n++)

{

if(strcmp(array[n][0],t)==0)

break;

}

strcpy(str1,temp2);

strcat(str1,array[n][l]);

strcat(str1,temp1);

cout << "\n\t" <<str1;

getch();

if(strcmp(array[n][l],'\0')==0)

{

if(i==(strlen(str)-1))

{

int len=strlen(str1);

str1[len-1]='\0';

cout << "\n\t"<<str1;

cout << "\n\n\tENTERED STRING IS VALID";

getch();

exit(1);

}

strcpy(temp1,'\0');

strcpy(temp2,'\0');

strcpy(t,'\0');

goto again1;

}

if(strcmp(array[n][l],"Error")==0)

{

cout << "\n\tERROR IN YOUR SOURCE STRING";

getch();

exit(1);

}

strcpy(tt,'\0');

strcpy(tt,array[n][l]);

strcpy(t3,'\0');

f=0;

for(c=0;c<strlen(tt);c++)

{

t3[c]=tt[c];

t3[c+1]='\0';

if(strcmp(t3,temp)==0)

{

f=0;

break;

}

else

f=1;

}

if(f==0)

{

strcpy(temp,'\0');

strcpy(temp1,'\0');

strcpy(temp2,'\0');

strcpy(t,'\0');

i++;

k=0;

goto again;

}

else

{

strcpy(temp1,'\0');

strcpy(temp2,'\0');

strcpy(t,'\0');

goto again1;

}}}

i++;

k++; }

if(f1==0)

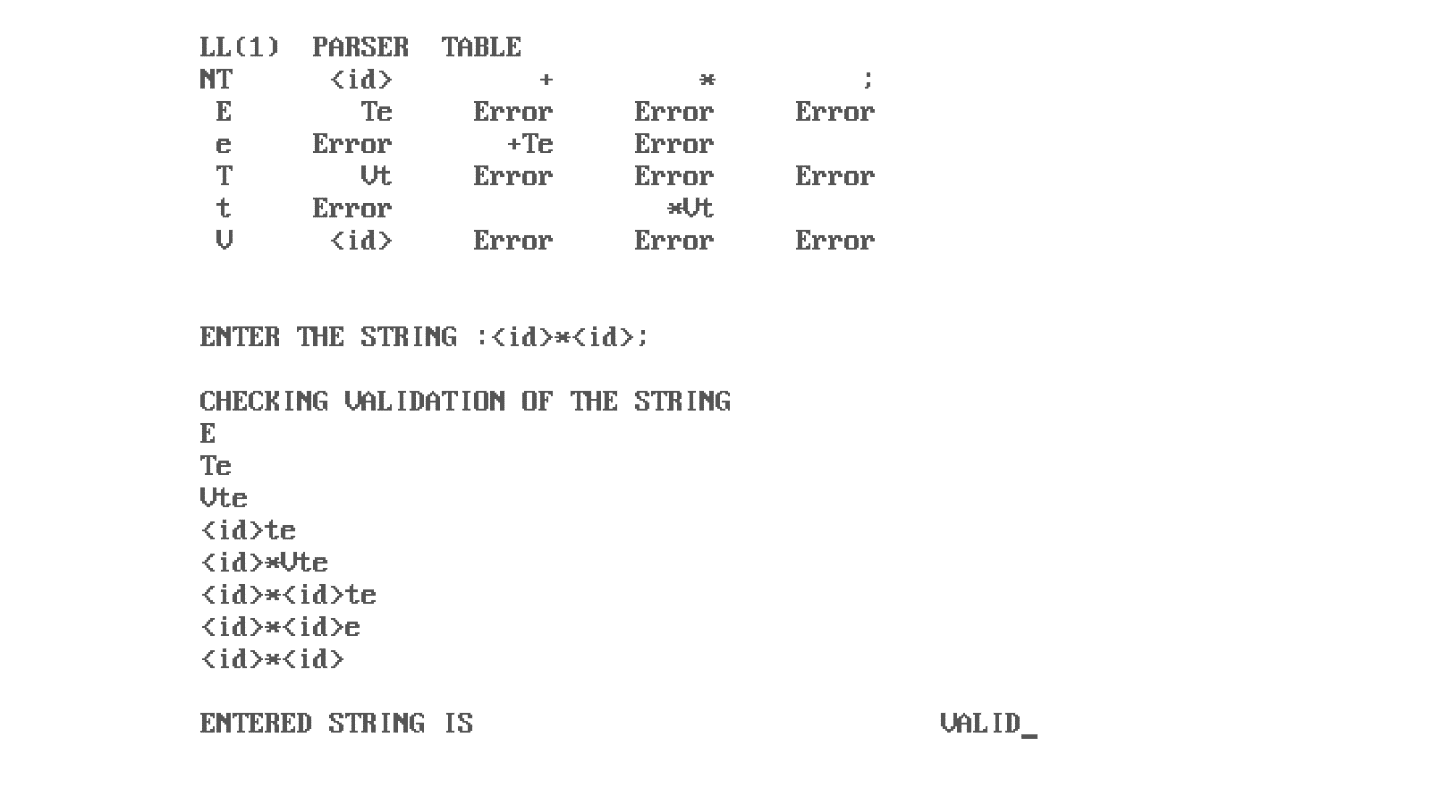
cout << "\nENTERED STRING IS INVALID";

else

cout << "\n\n\tENTERED STRING IS VALID";

getch();}

***OUTPUT:***



**Conclusion:** Hence understood and implemented LL1 parser.