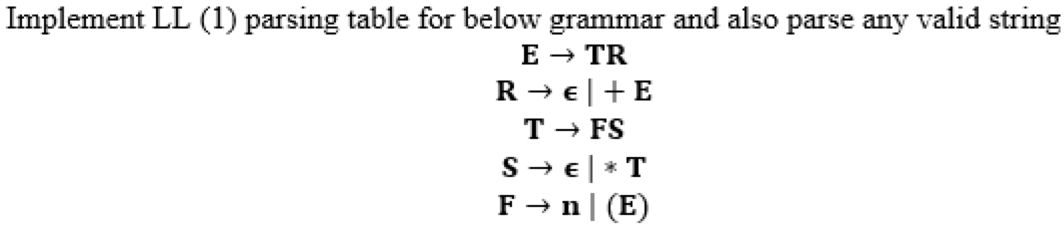
External Practical Exam

Aim:



Implement LL (1) parsing table for below grammar and also parse any valid string

E → TR
R → ϵ | + E
T → FS
S → ϵ | * T
F → n | (E)

Program:

```
/*
    author: mr_bhishm
    created: 29-10-2020 09:15:20
    "Make it work, make it right, make it fast."
                                                - Kent Beck
*/
#include<bits/stdc++.h>
using namespace std;
#define debug(x) cout<<#x<<" "<<x<<endl

map <string,string> first,follow,rules;
set <string> nt,t;
vector <string> calls_for_nt;
map < pair<string,string> , string > parse_table;
vector <string> LL1_stack,cur_str,rule_used;

// convert character to string...
string to_string(char c)
{
      string s="";
      s+=c;
      return s;
}

// map non terminal with its possible terminals...
void set_map(string s)
{
      stringstream ss(s);
      string key,value,temp;
      ss>>key;
      calls_for_nt.push_back(key);
      while(ss>>temp)
      {
            if(temp!="->"&&temp!="|")
                  value+=" "+temp;
```

```
	}
	rules[key]=value;
	return ;
}

// check if non-terminal or not...
bool is_nterminal(string s)
{
	if(find(nt.begin(),nt.end(),s)!=nt.end())
		return true;
	else
		return false;
}

// check if terminal or not...
bool is_terminal(string s)
{
	if(find(t.begin(),t.end(),s)!=t.end())
		return true;
	else
		return false;
}

// find first of all non-terminals...
void set_first(string s)
{
	if(first[s].length()!=0)
		return ;
	string temp=rules[s].substr(1,rules[s].length()-1);
	stringstream ss(temp);
	while(ss>>temp)
	{
		if(is_nterminal(to_string(temp[0])))
			{
				set_first(to_string(temp[0]));
				first[s]=first[to_string(temp[0])];
			}
		else
			first[s]+=temp.substr(0,1)+" ";
	}
	return;
}

// check for epsilon...
bool check_dol(string s)
{
	for(int i=0;i<s.length();i++)
		if(s[i]=='~')
			return true;
```

```
      return false;
}

// find the follow of all non terminals...
void set_follow(string s)
{
      int flag=0;
      for(auto itr=rules.begin();itr!=rules.end();itr++)
      {
            stringstream ss(itr->second);
            string temp;
            while(ss>>temp)
            {
                  for(int i=0;i<temp.length();i++)
                  {
                        flag=0;
                        if(temp[i]==s[0])
                        {
                              if(i+1<temp.length())
                              {
                                    if(is_terminal(to_string(temp[i+1])))
                                          {
                                          follow[s]+=""+to_string(temp[i+1])+"";
                                          }
                                    else
                                          {

      if(check_dol(rules[to_string(temp[i+1])]))
                                                      {
                                                       for(int
j=0;j<first[to_string(temp[i+1])].length();j++)

      if(first[to_string(temp[i+1])][j]!='~')

      follow[s]+=""+to_string(first[to_string(temp[i+1])][j]);
                                                      follow[s]+=" ";
                                                      }

      follow[s]+=""+follow[to_string(temp[i+1])];
                                          }
                              }
                              else
                              {
                                    follow[s]+=""+follow[itr->first]+" ";
                              }
                        flag=1;
                        break;
                        }
                  }
```

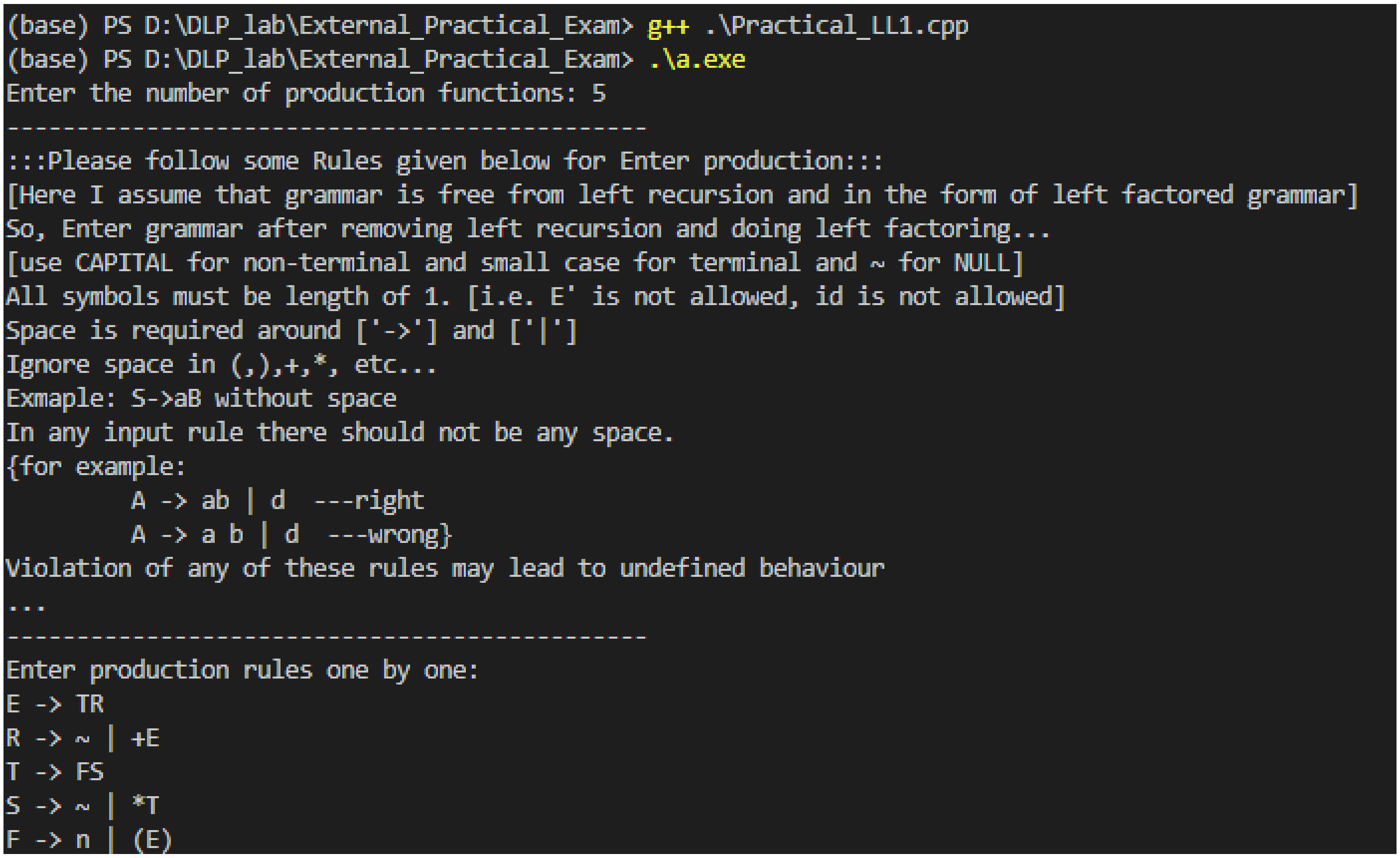
```
                if(flag==1)
                    break;
            }
        }
}

//check particular production is in given rules or not (for printing production in
parse table )...
string in_rules(string s, string t)
{
        if(find(rules[s].begin(),rules[s].end(),t[0])!=rules[s].end())
        {
            stringstream ss(rules[s]);
            string temp;
            while(ss>>temp)
                if(find(temp.begin(),temp.end(),t[0])!=temp.end())
                    return temp;
        }
        else
        {
            stringstream ss(rules[s]);
            string temp;
            ss>>temp;
            return temp;
        }
        return t;
}

// generate parse table...
void set_parse_table(string s)
{
        string temp=first[s];
        if(find(temp.begin(),temp.end(),'~')!=temp.end())
            {
                string for_dol;
                stringstream ss(follow[s]);
                while(ss>>for_dol)
                {
                    parse_table[{s,for_dol}]=s+"-> ~";
                }
            }
        stringstream ss(temp);
        while(ss>>temp)
        {
            if(temp==to_string('~'))
                continue;
            parse_table[{s,temp}]=s+"-> "+in_rules(s,temp);
        }
        return ;
```

```
}

// check the given input string is accept by parser or not...
void check_str()
{
	if(LL1_stack[LL1_stack.size()-1][0]=='$' && cur_str[cur_str.size()-1][0]=='$')
		return ;
	if(LL1_stack[LL1_stack.size()-1][0]==cur_str[cur_str.size()-1][0])
	{
		LL1_stack.push_back(LL1_stack[LL1_stack.size()-
1].substr(1,LL1_stack[LL1_stack.size()-1].length()-1));
		cur_str.push_back(cur_str[cur_str.size()-
1].substr(1,cur_str[cur_str.size()-1].length()-1));
		rule_used.push_back(" ");
		check_str();
	}
	else if(parse_table[{to_string(LL1_stack[LL1_stack.size()-
1][0]),to_string(cur_str[cur_str.size()-1][0])}].length()!=0)
	{
		stringstream ss(parse_table[{to_string(LL1_stack[LL1_stack.size()-
1][0]),to_string(cur_str[cur_str.size()-1][0])}]);
		string temp;
		ss>>temp;ss>>temp;
		if(temp=="~")
		{
		LL1_stack.push_back(LL1_stack[LL1_stack.size()-
1].substr(1,LL1_stack[LL1_stack.size()-1].length()-1));
		cur_str.push_back(cur_str[cur_str.size()-1]);
		rule_used.push_back(parse_table[{to_string(LL1_stack[LL1_stack.size()-
1][0]),to_string(cur_str[cur_str.size()-1][0])}]);
		}
		else
		{
		LL1_stack.push_back(temp+LL1_stack[LL1_stack.size()-
1].substr(1,LL1_stack[LL1_stack.size()-1].length()-1));
		cur_str.push_back(cur_str[cur_str.size()-1]);
		rule_used.push_back(parse_table[{to_string(LL1_stack[LL1_stack.size()-
1][0]),to_string(cur_str[cur_str.size()-1][0])}]);
		}
		check_str();
	}
	else
	{
		return;
	}
}

int main(){
    int n;
```

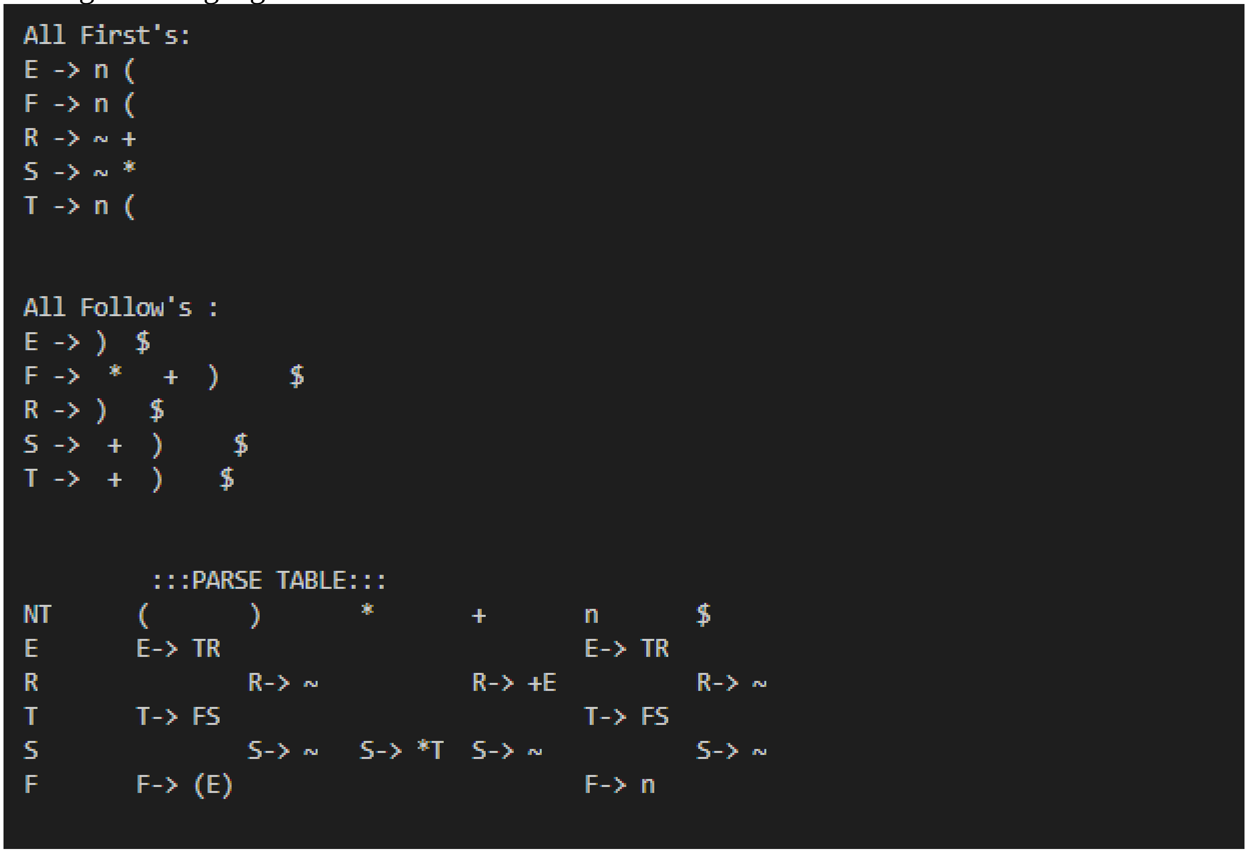
```
      cout<<"Enter the number of production functions: ";
      cin>>n;
    cout<<"----------------------------------------------"<<endl;
    cout<<":::Please follow some Rules given below for Enter production:::"<<endl;
    cout<<"[Here I assume that grammar is free from left recursion and in the form of
left factored grammar]"<<endl;
    cout<<"So, Enter grammar after removing left recursion and doing left
factoring..."<<endl;
    cout<<"[use CAPITAL for non-terminal and small case for terminal and ~ for
NULL]"<<endl;
    cout<<"All symbols must be length of 1. [i.e. E' is not allowed, id is not
allowed]"<<endl;
    cout<<"Space is required around ['->'] and ['|']"<<endl;
    cout<<"Ignore space in (,),+,*, etc..."<<endl;
    cout<<"Exmaple: S->aB without space"<<endl;
    cout<<"In any input rule there should not be any space."<<endl;
    cout<<"{for example:\n\t A -> ab | d  ---right\n\t A -> a b | d  ---wrong}"<<endl;
    cout<<"Violation of any of these rules may lead to undefined behaviour"<<endl;
    cout<<"..."<<endl;
    cout<<"----------------------------------------------"<<endl;
    cout<<"Enter production rules one by one: "<<endl;
      string s;
      getline(cin,s);
      for(int i=0;i<n;i++)
      {
            getline(cin,s);
            set_map(s);
      }
    cout<<endl;
      cout<<"Non-terminal- Terminal Map is : \n";
      for(auto itr=rules.begin();itr!=rules.end();itr++)
            cout<<itr->first<<" "<<itr->second<<endl;
    cout<<endl;
      for(auto itr=rules.begin();itr!=rules.end();itr++)
            {
                  nt.insert(itr->first);
            }
      for(auto itr=rules.begin();itr!=rules.end();itr++)
    {
        stringstream ss(itr->second);
        string test;
        while(ss>>test)
        {
            for(int i=0;i<test.length();i++)
                if(!is_nterminal(test.substr(i,1))&&(test.substr(i,1))!="~")
                    t.insert(test.substr(i,1));
        }
    }
      cout<<"Non-Terminals: "<<endl;
      for(string s:nt)
            cout<<s<<" ";
    cout<<endl;
      cout<<"Terminals: "<<endl;
      for(string s:t)
            cout<<s<<" ";
    cout<<endl;
      for(string s:nt)
            set_first(s);
      follow.clear();
      for(string s:calls_for_nt)
            set_follow(s);
      for(auto itr=follow.begin();itr!=follow.end();itr++)
            itr->second+=" $";
    cout<<endl;
      cout<<"\nAll First's:"<<endl;
      for(auto itr=first.begin();itr!=first.end();itr++)
            cout<<itr->first<<" -> "<<itr->second<<endl;
    cout<<endl;
      cout<<"\nAll Follow's : "<<endl;
      for(auto itr=follow.begin();itr!=follow.end();itr++)
            cout<<itr->first<<" -> "<<itr->second<<endl;
      cout<<endl;
    for(string s:calls_for_nt)
            set_parse_table(s);
      cout<<"\n     \t :::PARSE TABLE:::\t"<<endl;
      cout<<"NT\t";
      for(string s:t)
      {
            cout<<s<<"\t";
      }
      cout<<"$\t";
      cout<<endl;
      for(int i=0;i<calls_for_nt.size();i++)
      {
            cout<<calls_for_nt[i]<<"\t";
            for(string ts:t)
            {
                  cout<<parse_table[{calls_for_nt[i],ts}]<<"\t";
            }
            cout<<parse_table[{calls_for_nt[i],"$"}]<<"\t";
            cout<<endl;
      }
      LL1_stack.push_back(calls_for_nt[0]+"$");
      cout<<"\n\nEnter string to check if it is accepted by parser or not: ";
      getline(cin,s);
      cur_str.push_back(s+"$");
      rule_used.push_back(" ");
      check_str();
```

```
	cout<<"\n\nStack\t\tInput\t\tProductions\n";
	for(int i=0;i<LL1_stack.size();i++)
		cout<<LL1_stack[i]<<"\t\t"<<cur_str[i]<<"\t\t"<<rule_used[i]<<endl;
	if(LL1_stack[LL1_stack.size()-1][0]=='$' && cur_str[cur_str.size()-1][0]=='$')
		cout<<"\n\nString is ACCEPTED!!!"<<endl;
	else
		cout<<"\nString is NOT-ACCEPTED !!!"<<endl;
	return 0;
}

/*

Input:
E -> TR
R -> ~ | +E
T -> FS
S -> ~ | *T
F -> n | (E)

*/
```

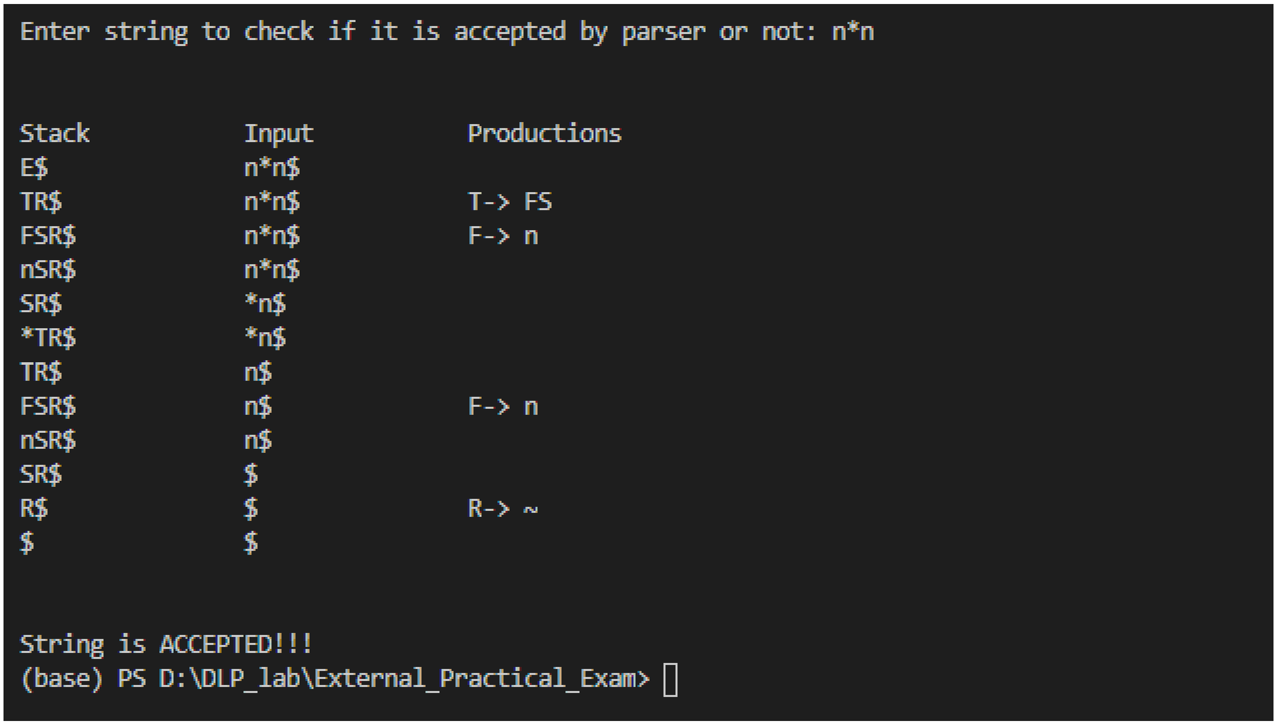
Output:



```
(base) PS D:\DLP_lab\External_Practical_Exam> g++ .\Practical_LL1.cpp
(base) PS D:\DLP_lab\External_Practical_Exam> .\a.exe
Enter the number of production functions: 5
---------------------------------------------
:::Please follow some Rules given below for Enter production:::
[Here I assume that grammar is free from left recursion and in the form of left factored grammar]
So, Enter grammar after removing left recursion and doing left factoring...
[use CAPITAL for non-terminal and small case for terminal and ~ for NULL]
All symbols must be length of 1. [i.e. E' is not allowed, id is not allowed]
Space is required around ['->'] and ['|']
Ignore space in (,),+,*, etc...
Exmaple: S->aB without space
In any input rule there should not be any space.
{for example:
         A -> ab | d  ---right
         A -> a b | d  ---wrong}
Violation of any of these rules may lead to undefined behaviour
...
---------------------------------------------
Enter production rules one by one:
E -> TR
R -> ~ | +E
T -> FS
S -> ~ | *T
F -> n | (E)
```



```
All First's:
E -> n (
F -> n (
R -> ~ +
S -> ~ *
T -> n (


All Follow's :
E -> )  $
F ->  *   +  )    $
R -> )  $
S ->  +  )    $
T ->  +  )   $


         :::PARSE TABLE:::
NT      (       )       *       +       n       $
E       E-> TR                          E-> TR
R               R-> ~           R-> +E          R-> ~
T       T-> FS                          T-> FS
S               S-> ~   S-> *T  S-> ~           S-> ~
F       F-> (E)                         F-> n
```



```
Enter string to check if it is accepted by parser or not: n*n


Stack           Input           Productions
E$              n*n$
TR$             n*n$            T-> FS
FSR$            n*n$            F-> n
nSR$            n*n$
SR$             *n$
*TR$            *n$
TR$             n$
FSR$            n$              F-> n
nSR$            n$
SR$             $
R$              $               R-> ~
$               $


String is ACCEPTED!!!
(base) PS D:\DLP_lab\External_Practical_Exam>
```