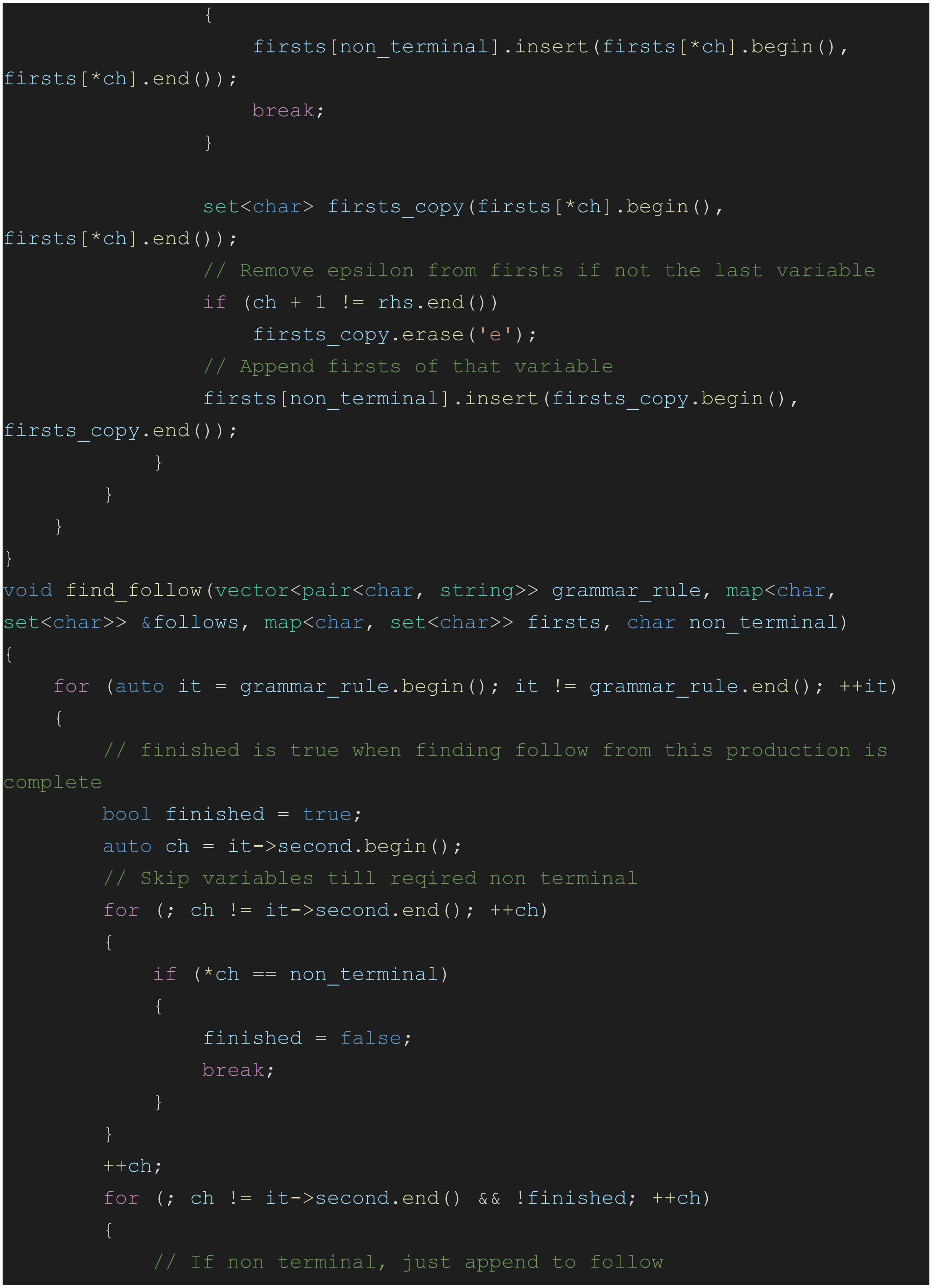
Department of Computer Science and Engineering, SVNIT Surat
System Software
Lab Assignment -5

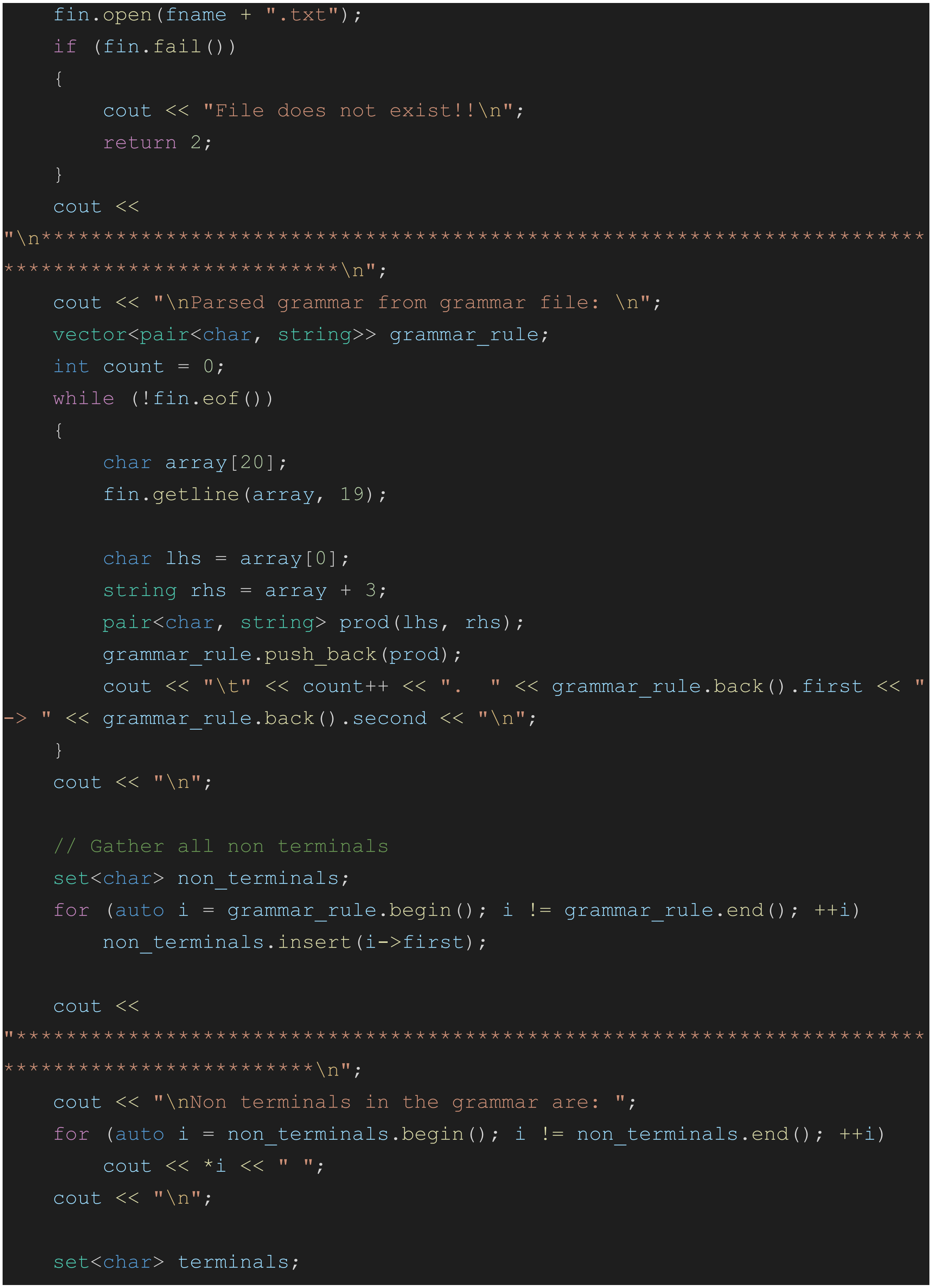
1. Write a program to construct LL (1) parse table for the following grammar and check whether the given input can be accepted or not.

Grammar:
E --> TE'
E' --> +TE' | ε
T --> FT'
T' --> *FT' | ε
F --> id | (E)
***ε denotes epsilon.**

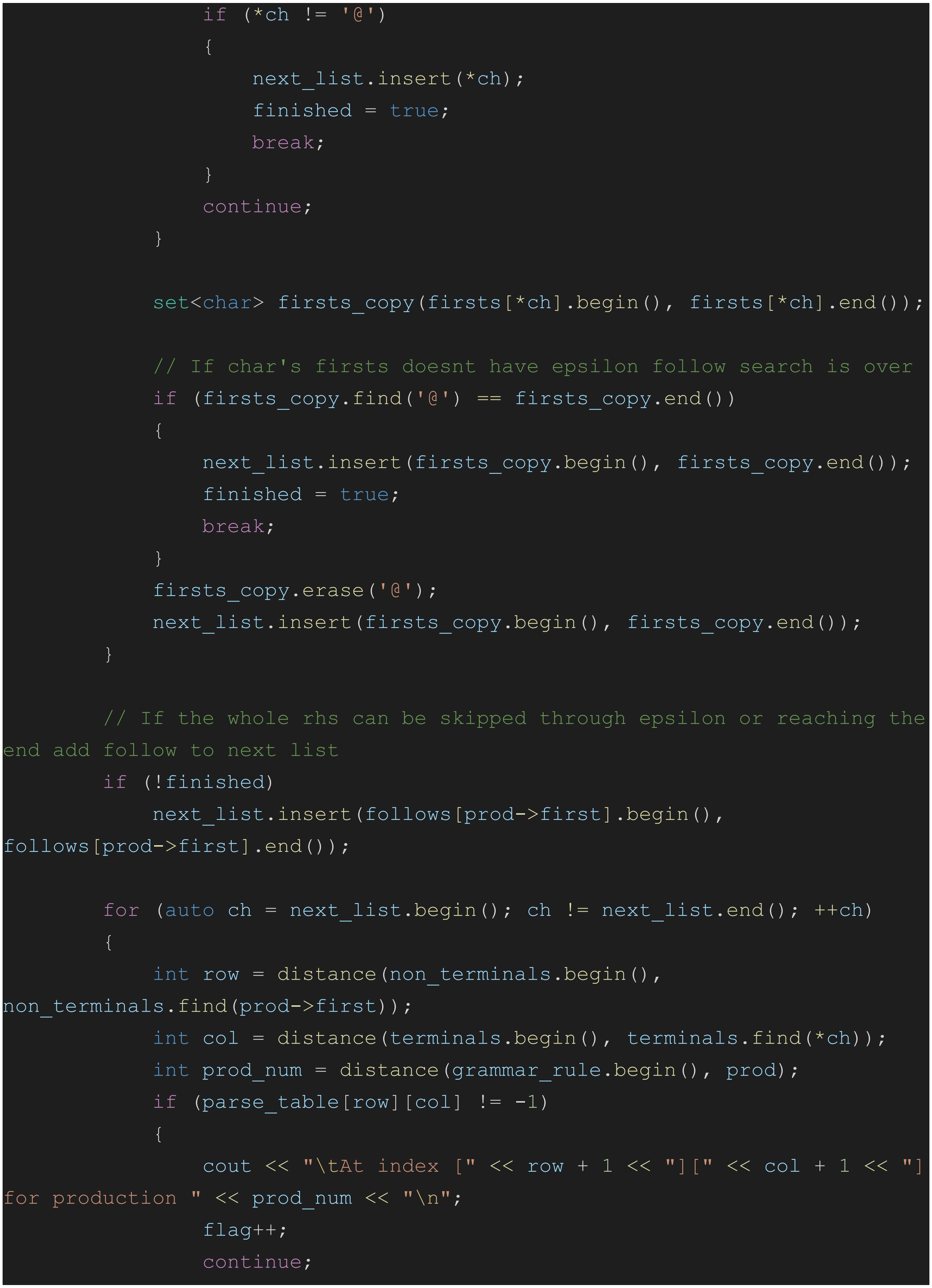
```cpp
#include <bits/stdc++.h>
using namespace std;
void find_first(vector<pair<char, string>> grammar_rule, map<char, set<char>> &firsts, char non_terminal)
{
    for (auto it = grammar_rule.begin(); it != grammar_rule.end(); ++it)
    {
        // Find productions of the non terminal
        if (it->first != non_terminal)
            continue;
        string rhs = it->second;
        // Loop till a non terminal or no epsilon variable found
        for (auto ch = rhs.begin(); ch != rhs.end(); ++ch)
        {
            // If first char in production a non term, add it to firsts
            if (!isupper(*ch))
            {
                firsts[non_terminal].insert(*ch);
                break;
            }
            else
            {
                // If char in prod is non terminal and whose firsts has no yet been found out find first for that non terminal
                if (firsts[*ch].empty())
                    find_first(grammar_rule, firsts, *ch);
                // If variable doesn't have epsilon, stop loop
                if (firsts[*ch].find('@') == firsts[*ch].end())
```



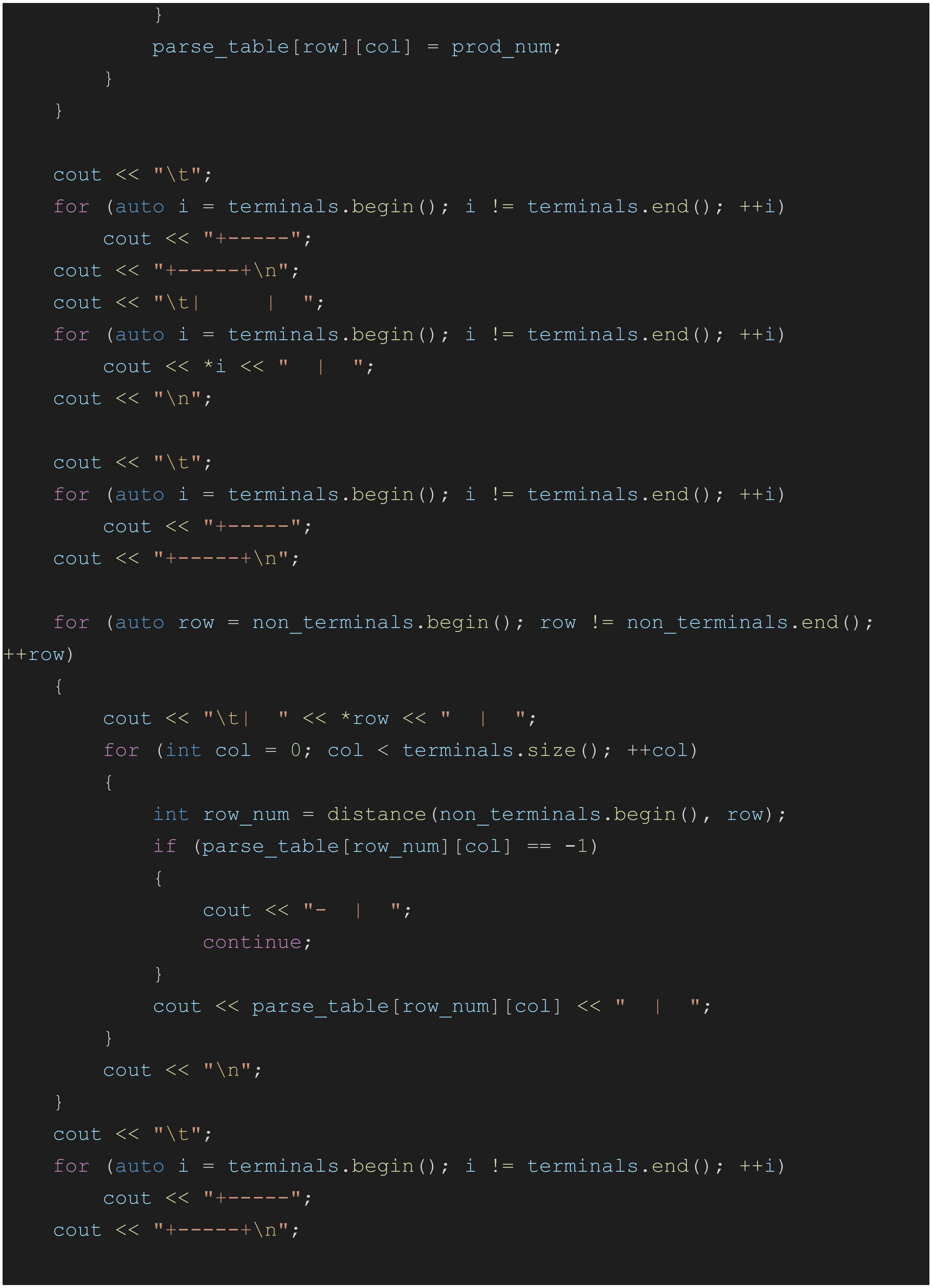
```
                {
                    firsts[non_terminal].insert(firsts[*ch].begin(),
firsts[*ch].end());
                    break;
                }

                set<char> firsts_copy(firsts[*ch].begin(),
firsts[*ch].end());
                // Remove epsilon from firsts if not the last variable
                if (ch + 1 != rhs.end())
                    firsts_copy.erase('e');
                // Append firsts of that variable
                firsts[non_terminal].insert(firsts_copy.begin(),
firsts_copy.end());
            }
        }
    }
}
void find_follow(vector<pair<char, string>> grammar_rule, map<char,
set<char>> &follows, map<char, set<char>> firsts, char non_terminal)
{
    for (auto it = grammar_rule.begin(); it != grammar_rule.end(); ++it)
    {
        // finished is true when finding follow from this production is
complete
        bool finished = true;
        auto ch = it->second.begin();
        // Skip variables till reqired non terminal
        for (; ch != it->second.end(); ++ch)
        {
            if (*ch == non_terminal)
            {
                finished = false;
                break;
            }
        }
        ++ch;
        for (; ch != it->second.end() && !finished; ++ch)
        {
            // If non terminal, just append to follow
            if (!isupper(*ch))
            {
                follows[non_terminal].insert(*ch);
                finished = true;
                break;
            }
            set<char> firsts_copy(firsts[*ch]);
            // If char's firsts doesnt have epsilon follow search is over
            if (firsts_copy.find('@') == firsts_copy.end())
            {
                follows[non_terminal].insert(firsts_copy.begin(),
firsts_copy.end());
                finished = true;
                break;
            }
            // Else next char has to be checked after appending firsts to
follow
            firsts_copy.erase('@');
            follows[non_terminal].insert(firsts_copy.begin(),
firsts_copy.end());
        }
        // If end of production, follow same as follow of variable
        if (ch == it->second.end() && !finished)
        {
            // Find follow if it doesn't have
            if (follows[it->first].empty())
                find_follow(grammar_rule, follows, firsts, it->first);
            follows[non_terminal].insert(follows[it->first].begin(),
follows[it->first].end());
        }
    }
}

int main()
{
    string fname;
    cout << "Enter file name to read grammar rules(Use @ instead of
epsilon in grammar rules): ";
    cin >> fname;
    ifstream fin;
```



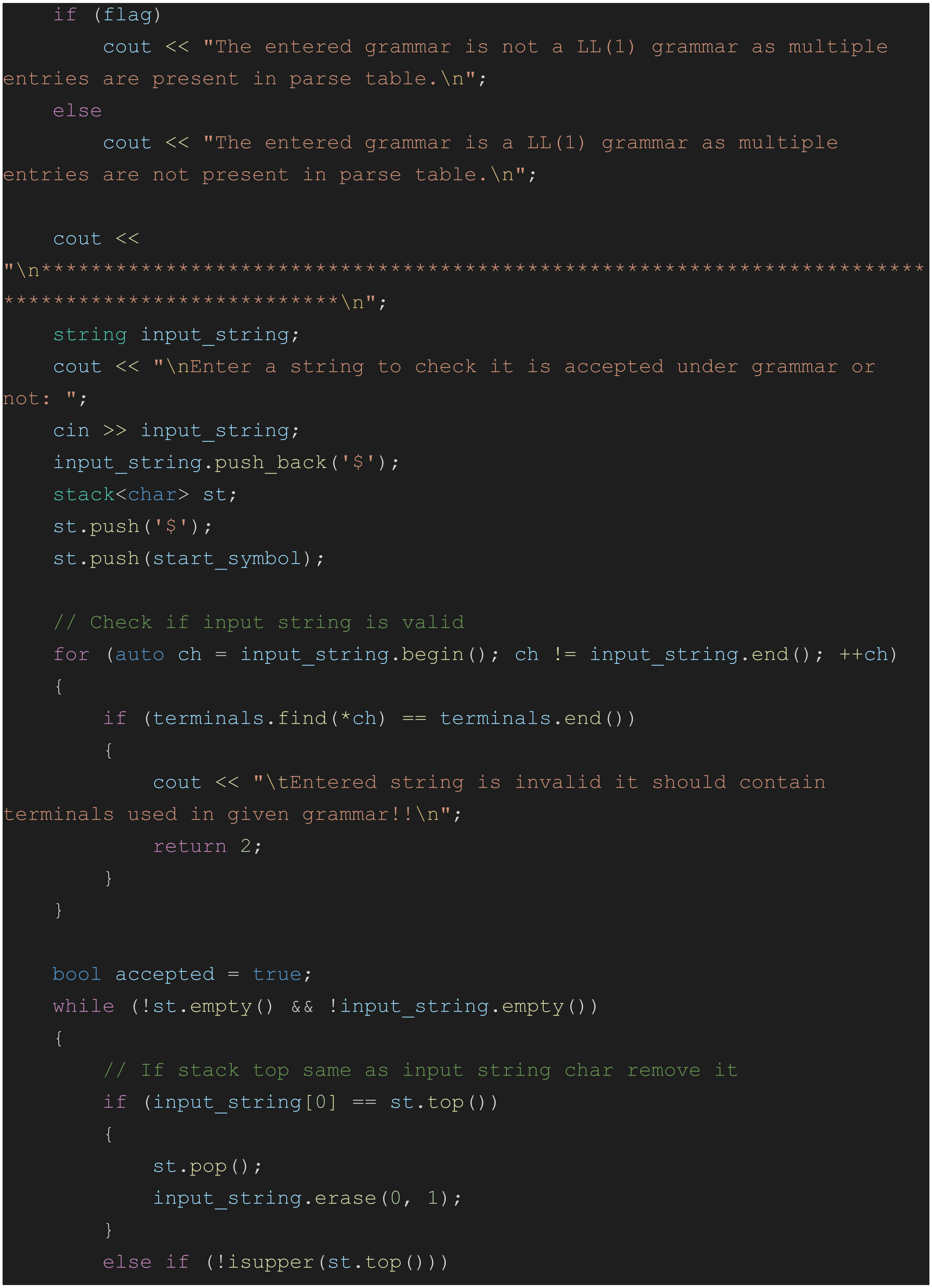
```cpp
    fin.open(fname + ".txt");
    if (fin.fail())
    {
        cout << "File does not exist!!\n";
        return 2;
    }
    cout <<
"\n***********************************************************************
***************************\n";
    cout << "\nParsed grammar from grammar file: \n";
    vector<pair<char, string>> grammar_rule;
    int count = 0;
    while (!fin.eof())
    {
        char array[20];
        fin.getline(array, 19);

        char lhs = array[0];
        string rhs = array + 3;
        pair<char, string> prod(lhs, rhs);
        grammar_rule.push_back(prod);
        cout << "\t" << count++ << ".  " << grammar_rule.back().first << "
-> " << grammar_rule.back().second << "\n";
    }
    cout << "\n";

    // Gather all non terminals
    set<char> non_terminals;
    for (auto i = grammar_rule.begin(); i != grammar_rule.end(); ++i)
        non_terminals.insert(i->first);

    cout <<
"*************************************************************************
*************************\n";
    cout << "\nNon terminals in the grammar are: ";
    for (auto i = non_terminals.begin(); i != non_terminals.end(); ++i)
        cout << *i << " ";
    cout << "\n";

    set<char> terminals;
    for (auto i = grammar_rule.begin(); i != grammar_rule.end(); ++i)
        for (auto ch = i->second.begin(); ch != i->second.end(); ++ch)
            if (!isupper(*ch))
                terminals.insert(*ch);
    terminals.erase('@');
    terminals.insert('$');

    cout <<
"\n***********************************************************************
***************************\n";
    cout << "\nTerminals in the grammar are: ";
    for (auto i = terminals.begin(); i != terminals.end(); ++i)
        cout << *i << " ";
    cout << "\n";

    // Start symbol is first non terminal production in grammar
    char start_symbol = grammar_rule.begin()->first;

    map<char, set<char>> firsts;
    for (auto non_terminal = non_terminals.begin(); non_terminal !=
non_terminals.end(); ++non_terminal)
        if (firsts[*non_terminal].empty())
            find_first(grammar_rule, firsts, *non_terminal);

    cout <<
"\n***********************************************************************
***************************\n";
    cout << "\nFIRST of all terminals are: \n";
    for (auto it = firsts.begin(); it != firsts.end(); ++it)
    {
        cout << "\tFIRST(" << it->first << ") is ";
        for (auto firsts_it = it->second.begin(); firsts_it !=
it->second.end(); ++firsts_it)
            cout << *firsts_it << " ";
        cout << "\n";
    }
    cout << "\n";
    map<char, set<char>> follows;

    // Find follow of start variable first
    char start_var = grammar_rule.begin()->first;
    follows[start_var].insert('$');
    find_follow(grammar_rule, follows, firsts, start_var);
    // Find follows for rest of variables
    for (auto it = non_terminals.begin(); it != non_terminals.end(); ++it)
        if (follows[*it].empty())
            find_follow(grammar_rule, follows, firsts, *it);

    cout <<
"*************************************************************************
*************************\n";
    cout << "\nFollows of all non terminals are: \n";
    for (auto it = follows.begin(); it != follows.end(); ++it)
    {
        cout << "\tFOLLOW(" << it->first << ") is ";
        for (auto follows_it = it->second.begin(); follows_it !=
it->second.end(); ++follows_it)
            cout << *follows_it << " ";
        cout << "\n";
    }
    cout << "\n";
    int flag = 0;

    cout <<
"*************************************************************************
*************************\n";
    cout << "\nParsing Table of given grammar is: \n";
    int parse_table[non_terminals.size()][terminals.size()];
    fill(&parse_table[0][0], &parse_table[0][0] + sizeof(parse_table) /
sizeof(parse_table[0][0]), -1);
    for (auto prod = grammar_rule.begin(); prod != grammar_rule.end();
++prod)
    {
        string rhs = prod->second;
        set<char> next_list;
        bool finished = false;
        for (auto ch = rhs.begin(); ch != rhs.end(); ++ch)
        {
            if (!isupper(*ch))
            {
```



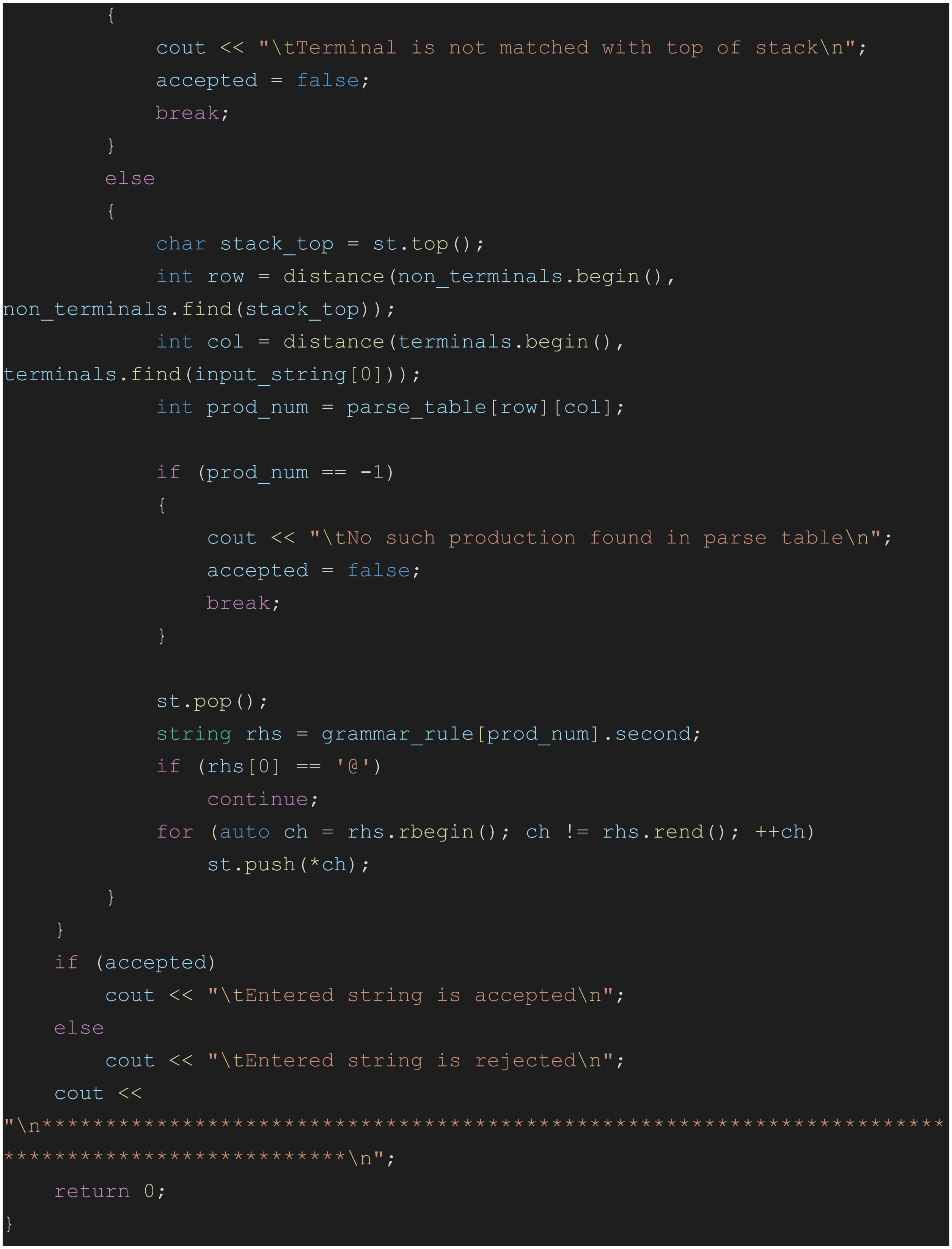
```
                if (*ch != '@')
                {
                    next_list.insert(*ch);
                    finished = true;
                    break;
                }
                continue;
            }

            set<char> firsts_copy(firsts[*ch].begin(), firsts[*ch].end());

            // If char's firsts doesnt have epsilon follow search is over
            if (firsts_copy.find('@') == firsts_copy.end())
            {
                next_list.insert(firsts_copy.begin(), firsts_copy.end());
                finished = true;
                break;
            }
            firsts_copy.erase('@');
            next_list.insert(firsts_copy.begin(), firsts_copy.end());
        }

        // If the whole rhs can be skipped through epsilon or reaching the
end add follow to next list
        if (!finished)
            next_list.insert(follows[prod->first].begin(),
follows[prod->first].end());

        for (auto ch = next_list.begin(); ch != next_list.end(); ++ch)
        {
            int row = distance(non_terminals.begin(),
non_terminals.find(prod->first));
            int col = distance(terminals.begin(), terminals.find(*ch));
            int prod_num = distance(grammar_rule.begin(), prod);
            if (parse_table[row][col] != -1)
            {
                cout << "\tAt index [" << row + 1 << "][" << col + 1 << "]
for production " << prod_num << "\n";
                flag++;
                continue;
```



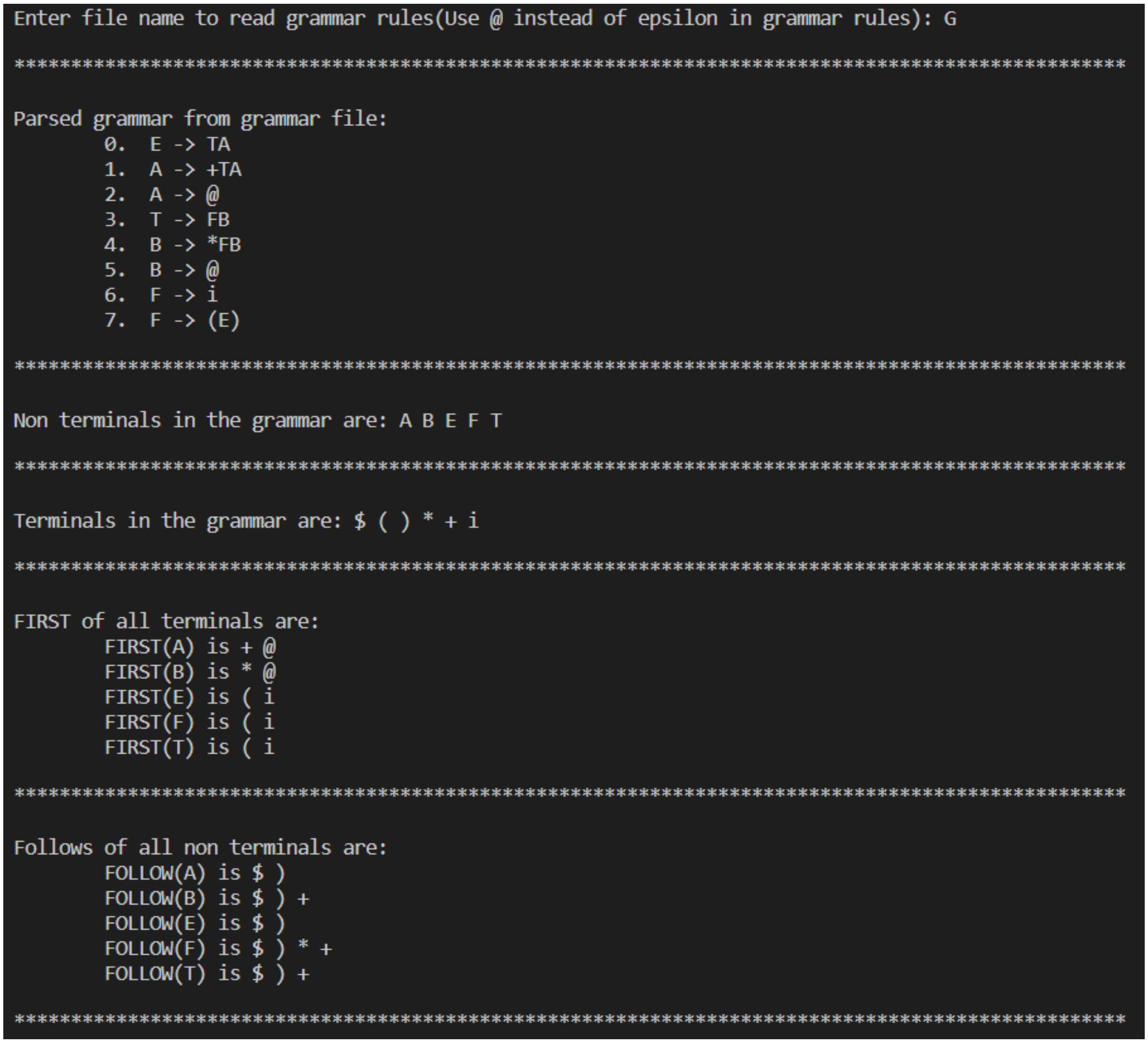
```
            }
            parse_table[row][col] = prod_num;
        }
    }

    cout << "\t";
    for (auto i = terminals.begin(); i != terminals.end(); ++i)
        cout << "+-----";
    cout << "+-----+\n";
    cout << "\t|     |  ";
    for (auto i = terminals.begin(); i != terminals.end(); ++i)
        cout << *i << "  |  ";
    cout << "\n";

    cout << "\t";
    for (auto i = terminals.begin(); i != terminals.end(); ++i)
        cout << "+-----";
    cout << "+-----+\n";

    for (auto row = non_terminals.begin(); row != non_terminals.end();
++row)
    {
        cout << "\t|  " << *row << "  |  ";
        for (int col = 0; col < terminals.size(); ++col)
        {
            int row_num = distance(non_terminals.begin(), row);
            if (parse_table[row_num][col] == -1)
            {
                cout << "-  |  ";
                continue;
            }
            cout << parse_table[row_num][col] << "  |  ";
        }
        cout << "\n";
    }
    cout << "\t";
    for (auto i = terminals.begin(); i != terminals.end(); ++i)
        cout << "+-----";
    cout << "+-----+\n";
```



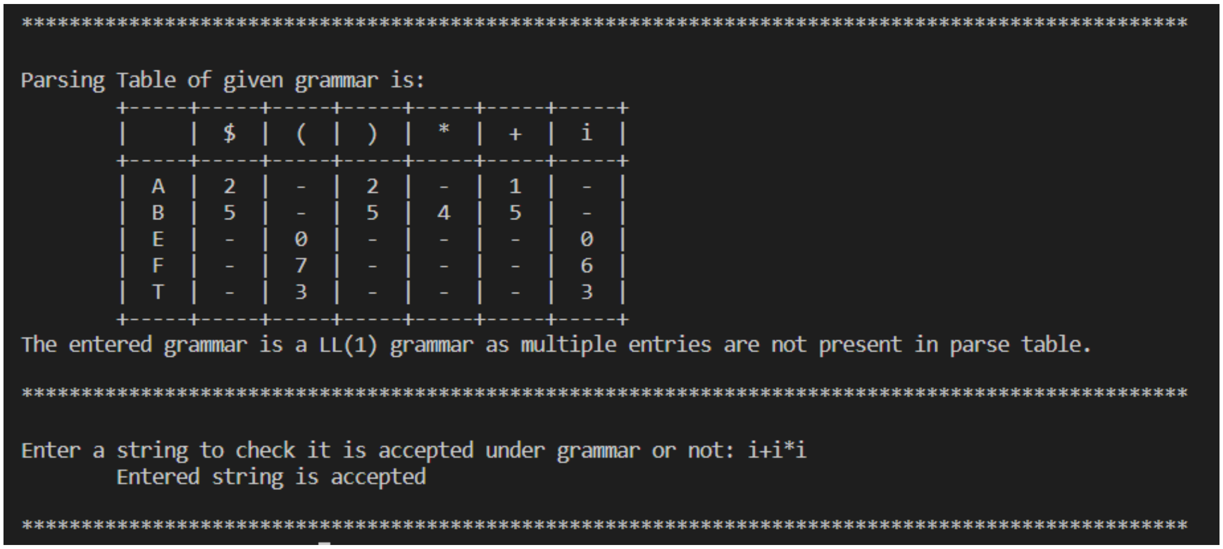
```cpp
    if (flag)
        cout << "The entered grammar is not a LL(1) grammar as multiple
entries are present in parse table.\n";
    else
        cout << "The entered grammar is a LL(1) grammar as multiple
entries are not present in parse table.\n";

    cout <<
"\n***********************************************************************
***************************\n";
    string input_string;
    cout << "\nEnter a string to check it is accepted under grammar or
not: ";
    cin >> input_string;
    input_string.push_back('$');
    stack<char> st;
    st.push('$');
    st.push(start_symbol);

    // Check if input string is valid
    for (auto ch = input_string.begin(); ch != input_string.end(); ++ch)
    {
        if (terminals.find(*ch) == terminals.end())
        {
            cout << "\tEntered string is invalid it should contain
terminals used in given grammar!!\n";
            return 2;
        }
    }

    bool accepted = true;
    while (!st.empty() && !input_string.empty())
    {
        // If stack top same as input string char remove it
        if (input_string[0] == st.top())
        {
            st.pop();
            input_string.erase(0, 1);
        }
        else if (!isupper(st.top()))
```



```
        {
            cout << "\tTerminal is not matched with top of stack\n";
            accepted = false;
            break;
        }
        else
        {
            char stack_top = st.top();
            int row = distance(non_terminals.begin(),
non_terminals.find(stack_top));
            int col = distance(terminals.begin(),
terminals.find(input_string[0]));
            int prod_num = parse_table[row][col];

            if (prod_num == -1)
            {
                cout << "\tNo such production found in parse table\n";
                accepted = false;
                break;
            }

            st.pop();
            string rhs = grammar_rule[prod_num].second;
            if (rhs[0] == '@')
                continue;
            for (auto ch = rhs.rbegin(); ch != rhs.rend(); ++ch)
                st.push(*ch);
        }
    }
    if (accepted)
        cout << "\tEntered string is accepted\n";
    else
        cout << "\tEntered string is rejected\n";
    cout <<
"\n***********************************************************************
***************************\n";
    return 0;
}
```



```
Enter file name to read grammar rules(Use @ instead of epsilon in grammar rules): G

**************************************************************************************************

Parsed grammar from grammar file:
        0.  E -> TA
        1.  A -> +TA
        2.  A -> @
        3.  T -> FB
        4.  B -> *FB
        5.  B -> @
        6.  F -> i
        7.  F -> (E)

**************************************************************************************************

Non terminals in the grammar are: A B E F T

**************************************************************************************************

Terminals in the grammar are: $ ( ) * + i

**************************************************************************************************

FIRST of all terminals are:
        FIRST(A) is + @
        FIRST(B) is * @
        FIRST(E) is ( i
        FIRST(F) is ( i
        FIRST(T) is ( i

**************************************************************************************************

Follows of all non terminals are:
        FOLLOW(A) is $ )
        FOLLOW(B) is $ ) +
        FOLLOW(E) is $ )
        FOLLOW(F) is $ ) * +
        FOLLOW(T) is $ ) +

**************************************************************************************************
```



```
*************************************************************************************************

Parsing Table of given grammar is:
        +-----+-----+-----+-----+-----+-----+-----+
        |     |  $  |  (  |  )  |  *  |  +  |  i  |
        +-----+-----+-----+-----+-----+-----+-----+
        |  A  |  2  |  -  |  2  |  -  |  1  |  -  |
        |  B  |  5  |  -  |  5  |  4  |  5  |  -  |
        |  E  |  -  |  0  |  -  |  -  |  -  |  0  |
        |  F  |  -  |  7  |  -  |  -  |  -  |  6  |
        |  T  |  -  |  3  |  -  |  -  |  -  |  3  |
        +-----+-----+-----+-----+-----+-----+-----+
The entered grammar is a LL(1) grammar as multiple entries are not present in parse table.

*************************************************************************************************

Enter a string to check it is accepted under grammar or not: i+i*i
        Entered string is accepted

*************************************************************************************************
```