System Software (CS306)

Assignment - 4

U20CS135

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.**

CODE

```
#include<bits/stdc++.h>

 using namespace std;
```

```
void find_first(vector< pair<char, string> > gram,
           map< char, set<char> > &firsts,
                   char non_term);

void find_follow(vector< pair<char, string> > gram,
           map< char, set<char> > &follows,
            map< char, set<char> > firsts,
                   char non_term);



      int main(int argc, char const *argv[])
                        {
                   if(argc != 3) {
        cout<<"Arguments should be <grammar file>
                <input string>\n";
                        return 1;
                          }
```

```
        fstream grammar_file;
    grammar_file.open(argv[1], ios::in);
        if(grammar_file.fail()) {
    cout<<"Error in opening grammar file\n";
                return 2;
                  }

cout<<"Grammar parsed from grammar file: \n";
     vector< pair<char, string> > gram;
              int count = 0;
        while(!grammar_file.eof()) {
               char buffer[20];
       grammar_file.getline(buffer, 19);

              char lhs = buffer[0];
             string rhs = buffer+3;
```

```
        pair <char, string> prod (lhs, rhs);

                gram.push_back(prod);

    cout<<count++<<".  "<<gram.back().first<<" ->
         "<<gram.back().second<<"\n";

                        }

                    cout<<"\n";









               set<char> non_terms;

for(auto i = gram.begin(); i != gram.end(); ++i)
                        {

              non_terms.insert(i->first);

                        }

 cout<<"The non terminals in the grammar are: ";

      for(auto i = non_terms.begin(); i !=
           non_terms.end(); ++i) {

                    cout<<*i<<" ";

                        }

                    cout<<"\n";
```

```
                set<char> terms;

for(auto i = gram.begin(); i != gram.end(); ++i)
                        {

        for(auto ch = i->second.begin(); ch !=
            i->second.end(); ++ch) {

                    if(!isupper(*ch)) {

                       terms.insert(*ch);

                             }

                          }

                        }



                terms.erase('e');

               terms.insert('$');

  cout<<"The terminals in the grammar are: ";

 for(auto i = terms.begin(); i != terms.end();
                    ++i) {

                   cout<<*i<<" ";

                        }

                 cout<<"\n\n";
```

```
      char start_sym = gram.begin()->first;




         map< char, set<char> > firsts;

 for(auto non_term = non_terms.begin(); non_term
      != non_terms.end(); ++non_term) {

           if(firsts[*non_term].empty()){

          find_first(gram, firsts, *non_term);

                        }

                      }



            cout<<"Firsts list: \n";

for(auto it = firsts.begin(); it != firsts.end();
                  ++it) {

              cout<<it->first<<" : ";

      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;



     char start_var = gram.begin()->first;

        follows[start_var].insert('$');

find_follow(gram, follows, firsts, start_var);



    for(auto it = non_terms.begin(); it !=
          non_terms.end(); ++it) {

            if(follows[*it].empty()) {

       find_follow(gram, follows, firsts, *it);

                    }

                    }
```

```
            cout<<"Follows list: \n";

        for(auto it = follows.begin(); it !=
              follows.end(); ++it) {

                cout<<it->first<<" : ";

        for(auto follows_it = it->second.begin();
  follows_it != it->second.end(); ++follows_it) {

                  cout<<*follows_it<<" ";

                        }

                    cout<<"\n";

                       }

                  cout<<"\n";




   int parse_table[non_terms.size()][terms.size()];

    fill(&parse_table[0][0], &parse_table[0][0] +
sizeof(parse_table)/sizeof(parse_table[0][0]), -1);

  for(auto prod = gram.begin(); prod != gram.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 != 'e') {

                       next_list.insert(*ch);

                          finished = true;

                              break;

                               }

                            continue;

                              }



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

                    if(firsts_copy.find('e') ==
                  firsts_copy.end()) {

                next_list.insert(firsts_copy.begin(),
                  firsts_copy.end());

                          finished = true;

                              break;
```

```
                        }

                firsts_copy.erase('e');

            next_list.insert(firsts_copy.begin(),
                firsts_copy.end());

                        }



                    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_terms.begin(),
           non_terms.find(prod->first));

              int col = distance(terms.begin(),
                 terms.find(*ch));

            int prod_num = distance(gram.begin(),
                      prod);

              if(parse_table[row][col] != -1) {
```

```
                    cout<<"Collision at
    ["<<row<<"]["<<col<<"] for production
              "<<prod_num<<"\n";

                          continue;

                            }

             parse_table[row][col] = prod_num;

                          }



                        }



             cout<<"Parsing Table: \n";

                    cout<<"    ";

    for(auto i = terms.begin(); i != terms.end();
                     ++i) {

                    cout<<*i<<" ";

                         }

                   cout<<"\n";

     for(auto row = non_terms.begin(); row !=
           non_terms.end(); ++row) {

                  cout<<*row<<"   ";

      for(int col = 0; col < terms.size(); ++col) {
```

```
int row_num = distance(non_terms.begin(),
            row);

  if(parse_table[row_num][col] == -1) {

                cout<<"- ";

                 continue;

                   }

  cout<<parse_table[row_num][col]<<" ";

                 }

              cout<<"\n";

                }

            cout<<"\n";




  string input_string(argv[2]);

   input_string.push_back('$');

          stack<char> st;

           st.push('$');

           st.push('S');
```

```
    for(auto ch = input_string.begin(); ch !=
          input_string.end(); ++ch) {

         if(terms.find(*ch) == terms.end()) {

            cout<<"Input string is invalid\n";

                        return 2;

                          }

                        }




              bool accepted = true;

    while(!st.empty() && !input_string.empty()) {




           if(input_string[0] == st.top()) {

                        st.pop();

                 input_string.erase(0, 1);

                          }

            else if(!isupper(st.top())) {
```

```
            cout<<"Unmatched terminal found\n";

                    accepted = false;

                        break;

                        }

                    else {

                char stack_top = st.top();

             int row = distance(non_terms.begin(),
            non_terms.find(stack_top));

               int col = distance(terms.begin(),
           terms.find(input_string[0]));

             int prod_num = parse_table[row][col];



                    if(prod_num == -1) {

               cout<<"No production found in parse
                     table\n";

                       accepted = false;

                           break;

                           }



                        st.pop();
```

```
            string rhs = gram[prod_num].second;

                    if(rhs[0] == 'e') {

                            continue;

                              }

                for(auto ch = rhs.rbegin(); ch !=
                  rhs.rend(); ++ch) {

                            st.push(*ch);

                              }

                            }

                    if(accepted) {

            cout<<"Input string is accepted\n";

                            }

                        else {

            cout<<"Input string is rejected\n";

                            }

                        return 0;
```

```
                        }


void find_first(vector< pair<char, string> > gram,

           map< char, set<char> > &firsts,

                   char non_term) {









    for(auto it = gram.begin(); it != gram.end();
                      ++it) {



              if(it->first != non_term) {

                          continue;

                            }









                string rhs = it->second;
```

```
    for(auto ch = rhs.begin(); ch != rhs.end();
                    ++ch) {



                    if(!isupper(*ch)) {

                firsts[non_term].insert(*ch);

                            break;

                             }

                           else {



                     if(firsts[*ch].empty()) {

                     find_first(gram, firsts, *ch);

                                 }



                    if(firsts[*ch].find('e') ==
                firsts[*ch].end()) {


  firsts[non_term].insert(firsts[*ch].begin(),
                firsts[*ch].end());

                                 break;

                                 }
```

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




                    if(ch + 1 != rhs.end()) {

                       firsts_copy.erase('e');

                              }




    firsts[non_term].insert(firsts_copy.begin(),
                firsts_copy.end());

                               }

                             }


                           }

                         }


void find_follow(vector< pair<char, string> > gram,

           map< char, set<char> > &follows,
```

```
        map< char, set<char> > firsts,

                char non_term) {









    for(auto it = gram.begin(); it != gram.end();
                    ++it) {




                bool finished = true;

            auto ch = it->second.begin();




        for(;ch != it->second.end() ; ++ch) {

                  if(*ch == non_term) {

                      finished = false;

                           break;

                            }

                         }
```

```
                    ++ch;


    for(;ch != it->second.end() && !finished;
                ++ch) {


                if(!isupper(*ch)) {

            follows[non_term].insert(*ch);

                    finished = true;

                        break;

                        }


        set<char> firsts_copy(firsts[*ch]);


            if(firsts_copy.find('e') ==
          firsts_copy.end()) {


follows[non_term].insert(firsts_copy.begin(),
            firsts_copy.end());

                    finished = true;

                        break;

                        }
```

```
                firsts_copy.erase('e');

    follows[non_term].insert(firsts_copy.begin(),
                firsts_copy.end());


                            }









        if(ch == it->second.end() && !finished) {


              if(follows[it->first].empty()) {

               find_follow(gram, follows, firsts,
                   it->first);

                              }


follows[non_term].insert(follows[it->first].begin(),
              follows[it->first].end());

                            }
```

```
                    }


                }
```

OUTPUT

```
node_sm@temple:~/Desktop/CourseWork/SS/Practicals/Assignment 5$ ./ll grammer1.txt ab
Grammar parsed from grammar file:
0.  S -> AB
1.  A -> a
2.  A -> e
3.  B -> b
4.  B -> e

The non terminals in the grammar are: A B S
The terminals in the grammar are: $ a b

Firsts list:
A : a e
B : b e
S : a b e

Follows list:
A : $ b
B : $
S : $

Parsing Table:
   $ a b
A  2 1 2
B  4 - 3
S  0 0 0

Input string is accepted
node_sm@temple:~/Desktop/CourseWork/SS/Practicals/Assignment 5$ ./ll grammer2.txt ab
Grammar parsed from grammar file:
0.  S -> F
1.  S -> (S+F)
2.  F -> a

The non terminals in the grammar are: F S
The terminals in the grammar are: $ ( ) + a

Firsts list:
F : a
S : ( a

Follows list:
F : $ ) +
S : $ +

Parsing Table:
   $ ( ) + a
F  - - - - 2
S  - 1 - - 0

Input string is invalid
```

SUBMITTED BY:

U20CS135

Shivam Mishra