Program 6:

WAP to Construct the LL(1) Parsing table for a CFG given through a file. This program should call the FIRST-FOLLOW program to generate the First & Follow information for the given CFG which will be used to generate the LL(1) Table.

```
#include <iostream>
#include <stdio.h>
#include <vector>
#include <map>
#include <set>
#include <fstream>
#include <algorithm>

using namespace std;

int Read(char *str)
{
 int i = 0;
 while (1)
 {
  str[i] = getchar();
  if (str[i] == '\n' || str[i] == '\r')
  {
   str[i] = '\0';
   return i;
  }
  i++;
 }
}
template <typename T>
void printSet(set<T> &set1)
{
 for (auto i = set1.begin(); i != set1.end(); ++i)
  cout << *i << " ";
 cout << endl;
}

template <typename T>
void printVector(vector<T> &set1)
{
 for (auto i = set1.begin(); i != set1.end(); ++i)
  cout << *i << " ";
 // cout << endl;
}

set<char>::iterator intersection(set<char> &set1, set<char> &set2)
{
 for (auto i = set1.begin(); i != set1.end(); i++)
  for (auto j = set2.begin(); j != set2.end(); j++)
   if ((*i) == (*j))
    return i;
 return set1.end();
}

int main()
{
 char ch1;
```

```
vector<vector<char>> CFG2D;
vector<char> tempVector;
set<char> nonTerminals;
set<char> terminals;
set<char> tempSet;
map<char, set<char>> firstOf;
map<char, set<char>> followOf;

FILE *filePointer = NULL;
filePointer = fopen("LLParsingTable2.txt", "r");
if (filePointer == NULL)
{
 printf("Unable to open LLParsingTable.txt\n");
 return 1;
}

char startSymbol;
fscanf(filePointer, "%c%c", &startSymbol, &ch1);
printf("Start symbol : %c\n", startSymbol);

while (fscanf(filePointer, "%c", &ch1) ≠ EOF)
{
 if (ch1 == '\n')
 {
  CFG2D.push_back(tempVector);
  tempVector.clear();
  continue;
 }
 tempVector.push_back(ch1);
 if (ch1 ≥ 'A' && ch1 ≤ 'Z')
  nonTerminals.insert(ch1);
 else if (ch1 == '#')
  continue;
 else
  terminals.insert(ch1);
}
CFG2D.push_back(tempVector);

fclose(filePointer);

cout << "Terminals : ";
for (auto terminal : terminals)
 cout << terminal << " ";
cout << endl;
cout << "Non Terminals : ";
for (auto nonTerminal : nonTerminals)
 cout << nonTerminal << " ";
cout << endl;

cout << "CFG : \n";
int index = 0;
```

```
for (auto i = CFG2D.begin(); i ≠ CFG2D.end(); ++i)
{
 auto j = (*i).begin();
 printf("%2d : ", index);
 cout << *j << " ⟶ ";
 ++j;
 for (; j ≠ (*i).end(); ++j)
  cout << *j;
 cout << endl;
 index++;
}
cout << endl;

//* caluclate first
for (auto i = CFG2D.begin(); i ≠ CFG2D.end(); ++i)
{
 auto j = (*i).begin();
 ch1 = *j;
 ++j;
 firstOf[ch1].insert(*j);
}

int flag = 0;

do
{
 flag = 0;
 for (auto i = CFG2D.begin(); i ≠ CFG2D.end(); ++i)
 {
  auto j = (*i).begin();
  char currentTerminal = *j;
  ++j;
  char firstOfNonTerminal = *j;

  // if firstOfNonTerminal is a terminal
  if (nonTerminals.find(firstOfNonTerminal) == nonTerminals.end())
   continue;

  // if first of RHS nonTerminal has notTerminal then recheck and
skip current LHS
  set<char>::iterator it =
intersection(firstOf[firstOfNonTerminal], nonTerminals);
  if (firstOf[firstOfNonTerminal].end() ≠ it)
  {
   flag = 1;
   continue;
  }

  // if first of RHS nonTerminal has no '#' copy first then
continue
  if (firstOf[firstOfNonTerminal].find('#') ==
firstOf[firstOfNonTerminal].end())
```

```
   {
    firstOf[currentTerminal].erase(firstOfNonTerminal);

firstOf[currentTerminal].insert(firstOf[firstOfNonTerminal].begin(),
firstOf[firstOfNonTerminal].end());
    continue;
   }

   // if first of RHS nonTerminal has '#'
   while (++j ≠ (*i).end())
   {
    bool containedEpsilonAlready = firstOf[currentTerminal].find('#')
≠ firstOf[currentTerminal].end();
    if (terminals.find(*j) ≠ terminals.end())
    {
     firstOf[currentTerminal].erase(firstOfNonTerminal);

firstOf[currentTerminal].insert(firstOf[firstOfNonTerminal].begin(),
firstOf[firstOfNonTerminal].end());
     firstOf[currentTerminal].insert(*j);
     firstOf[currentTerminal].erase('#');
     if (containedEpsilonAlready)
      firstOf[currentTerminal].insert('#');
     break;
    }
    firstOf[currentTerminal].erase(firstOfNonTerminal);

firstOf[currentTerminal].insert(firstOf[firstOfNonTerminal].begin(),
firstOf[firstOfNonTerminal].end());
    firstOf[currentTerminal].erase('#');
    if (containedEpsilonAlready)
     firstOf[currentTerminal].insert('#');

    firstOfNonTerminal = *j;
    set<char>::iterator it =
intersection(firstOf[firstOfNonTerminal], nonTerminals);
    if (it ≠ firstOf[firstOfNonTerminal].end())
    {
     flag = 1;
     break;
    }

    if (firstOf[firstOfNonTerminal].find('#') ==
firstOf[firstOfNonTerminal].end())
    {
     firstOf[currentTerminal].erase(firstOfNonTerminal);

firstOf[currentTerminal].insert(firstOf[firstOfNonTerminal].begin(),
firstOf[firstOfNonTerminal].end());
     break;
    }
   }
```

```
   if (j == (*i).end())
   {
    firstOf[currentTerminal].erase(firstOfNonTerminal);
    firstOf[currentTerminal]
      .insert(firstOf[firstOfNonTerminal].begin(),
firstOf[firstOfNonTerminal].end());
    firstOf[currentTerminal].insert('#');
    continue;
   }
  }
 } while (flag);

 //* calculating follow
 followOf[startSymbol].insert('$');

 for (auto nonTerminal = nonTerminals.begin(); nonTerminal ≠
nonTerminals.end(); ++nonTerminal)
 {
  for (auto i = CFG2D.begin(); i ≠ CFG2D.end(); ++i)
  {
   auto j = (*i).begin();
   char producingNonTerminal = *j;
   while (++j ≠ (*i).end())
   {
    char firstOfNonTerminal = *j;
    if (firstOfNonTerminal == *nonTerminal)
    {
     auto next = j + 1;
     if (next == (*i).end())
     {
      followOf[*nonTerminal].insert(producingNonTerminal);
      break;
     }
     else if (terminals.find(*next) ≠ terminals.end())
     {
      followOf[*nonTerminal].insert(*next);
      break;
     }
     else
     {
      while (next ≠ (*i).end())
      {
       if (terminals.find(*next) ≠ terminals.end())
       {
        followOf[*nonTerminal].insert(*next);
        break;
       }
       if (firstOf[*next].find('#') ≠ firstOf[*next].end())
       {
        followOf[*nonTerminal].insert(firstOf[*next].begin(),
firstOf[*next].end());
        followOf[*nonTerminal].erase('#');
```

```
        ++next;
       }
       else
       {
        followOf[*nonTerminal].insert(firstOf[*next].begin(),
firstOf[*next].end());
        break;
       }
      }
      if (next == (*i).end())
       followOf[*nonTerminal].insert(producingNonTerminal);
      break;
     }
    }
   }
  }
 }

 flag = 1;
 while (flag)
 {
  flag = 0;
  for (auto nonTerminal = nonTerminals.begin(); nonTerminal ≠
nonTerminals.end(); ++nonTerminal)
  {
   int flag2 = 0;
   std::set<char>::iterator itrToBeErased;
   for (auto followTerminal = followOf[*nonTerminal].begin();
followOf[*nonTerminal].end() ≠ followTerminal; ++followTerminal)
   {
    auto nonTerminalItr = nonTerminals.find(*followTerminal);
    if (nonTerminalItr ≠ nonTerminals.end())
    {
     if (*followTerminal == *nonTerminal)
     {
      flag2 = 1;
      itrToBeErased = followTerminal;
     }
     else
     {
      flag2 = flag = 1;
      // followOf[*nonTerminal].erase(*followTerminal);
      itrToBeErased = followTerminal;

followOf[*nonTerminal].insert(followOf[*followTerminal].begin(),
followOf[*followTerminal].end());
     }
    }
   }
   if (flag2 == 1)
    followOf[*nonTerminal].erase(itrToBeErased);
  }
```

```
}
//* Print first of
cout << "First of : \n";
for (auto i = firstOf.begin(); i != firstOf.end(); ++i)
{
 ch1 = (*i).first;
 printf("%c : ", ch1);
 auto &firstOfCh = firstOf[ch1];
 printSet(firstOfCh);
}
cout << endl;

//* Print first of
cout << "Follow of : \n";
for (auto i = followOf.begin(); i != followOf.end(); ++i)
{
 ch1 = (*i).first;
 printf("%c : ", ch1);
 auto &followOfCh = followOf[ch1];
 printSet(followOfCh);
}
cout << endl;

// Calculate LL Parsing Table
terminals.insert('$');
vector<vector<vector<int>>> LLParsingTable;
int i = 0, j = 0;
for (auto notTerminal = nonTerminals.begin(); notTerminal !=
nonTerminals.end(); notTerminal++, i++)
{
 j = 0;
 vector<vector<int>> tempVV;
 LLParsingTable.push_back(tempVV);
 for (auto terminal = terminals.begin(); terminal !=
terminals.end(); terminal++, j++)
 {
  vector<int> tempV;
  LLParsingTable[i].push_back(tempV);
  int k = 0;
  for (auto CFG2DCol = CFG2D.begin(); CFG2DCol != CFG2D.end();
CFG2DCol++, k++)
  {
   if (*notTerminal != (*CFG2DCol)[0])
    continue;
   if (firstOf[*notTerminal].find('#') !=
firstOf[*notTerminal].end())
   {
    if (followOf[*notTerminal].find(*terminal) !=
followOf[*notTerminal].end())
    {
     if ('#' == (*CFG2DCol)[1])
      LLParsingTable[i][j].push_back(k);
```

```
    }
   }
   for (auto CFG2DRow = (*CFG2DCol).begin() + 1; CFG2DRow ≠
(*CFG2DCol).end(); CFG2DRow++)
   {
    if (*CFG2DRow == *terminal)
    {
     LLParsingTable[i][j].push_back(k);
     break;
    }
    else if (terminals.find(*CFG2DRow) ≠ terminals.end())
     break;
    else if (nonTerminals.find(*CFG2DRow) ≠ nonTerminals.end())
    {
     if (*notTerminal == *CFG2DRow)
      break;
     if (firstOf[*CFG2DRow].find(*terminal) ≠
firstOf[*CFG2DRow].end())
     {
      LLParsingTable[i][j].push_back(k);
      break;
     }

     if (firstOf[*CFG2DRow].find('#') == firstOf[*CFG2DRow].end())
      break;
    }
   }
  }
 }
}

// Printing LL Parsing table
i = 0;
cout << "Non-Terminal Terminal     Productions\n";
for (auto notTerminal = nonTerminals.begin(); notTerminal ≠
nonTerminals.end(); notTerminal++, i++)
{
 j = 0;
 for (auto terminal = terminals.begin(); terminal ≠
terminals.end(); terminal++, j++)
 {
  printf("    %c            %c          ", *notTerminal, *terminal);
  if (LLParsingTable[i][j].size() == 0)
  {
   cout << "--\n";
   continue;
  }
  for (int len = 0; len < LLParsingTable[i][j].size(); len++)
  {
   int index = LLParsingTable[i][j][len];
   cout << CFG2D[index][0] << " ⟶ ";
   for (int k = 1; k ≠ CFG2D[index].size(); ++k)
```

```
    cout << CFG2D[index][k];
   printf(" ; ");
  }
  cout << endl;
 }
 cout << endl;
}

return 0;
}
```

LLParsingTable2.txt

```
S
SAB
AaA
Aa
BbB
B#
```

Output:

```
Start symbol : S
Terminals : a b
Non Terminals : A B S
CFG :
 0 : S --> AB
 1 : A --> aA
 2 : A --> a
 3 : B --> bB
 4 : B --> #

First of :
A : a
B : # b
S : a

Follow of :
A : $ b
B : $
S : $

Non-Terminal Terminal    Productions
    A          $         --
    A          a         A --> aA ; A --> a ;
    A          b         --

    B          $         B --> # ;
    B          a         --
    B          b         B --> bB ;

    S          $         --
    S          a         S --> AB ;
    S          b         --
```