Christian Mesina
Hugo Romero
Luis Escobar Urrutia

Lab 2

Introduction:

- For this lab, we wrote a program that reads a grammar, finds all FIRST(X) sets, and finds all FOLLOW(A) sets following the book's algorithm. The output lists all productions in G, the FIRST(X) and FOLLOW(A) sets of G. Also the test results used are for grammar G417 and at least one other grammar.

Source Code:

- **lab2.cpp**

```
/*
* Christian Mesina, Hugo Romero, Luis Escobar
* CSE 570 Compilers Lab 2
* Instructor: Dr. Ernesto Gomez
*
**/

#include "parser.h"
#include <iostream>
int main(int argc, char *argv[])
{
  if (argc < 2)
  {
      std::cout << "Please run the program in this format: ./lab2
<filename>\n";
      return 0;
  }
  std::string grammer = argv[1];
  Parser *parser = new Parser(grammer);
  parser->parse();
  parser->printProductions();
  parser->printFirst();
  std::cout << '\n';
  parser->printFollow();
  delete parser;
  return 0;
}
```

- **parser.cpp**

```
#include "parser.h"
#include "function.h"
```

```cpp
#include <iostream>
#include <string>
#include <map>
#include <set>
#include <fstream>

#define EPLISON 'e'

Parser::Parser(std::string &grammerFile) : grammer(grammerFile){};

void Parser::parse()
{
   std::ifstream inputGrammerFile;

   inputGrammerFile.open(grammer.c_str());

   if (!inputGrammerFile.is_open())
   {
       std::cerr << "Error: " << grammer << " not found." << '\n';
       exit(-1);
   }

   std::string input = "";

   int productionsPositionInTheGrammer = 0;

   while (inputGrammerFile >> input)
   {
       if (input == "$") // End of non-terminals
       {
           productionsPositionInTheGrammer =
inputGrammerFile.tellg();
           break;
       }
       else
       {
           // Add FIRST(X) = {X}
           first[input[0]].insert(input[0]);
       }
   }

   inputGrammerFile.seekg(productionsPositionInTheGrammer);
   input.clear();

   // Now add the productions
   while (inputGrammerFile >> input)
   {
       productions.insert(input);
   }
```

```
    // Do not need the $ in our productions
    // since we just want the productions
    productions.erase(productions.find("$"));

    for (auto p : productions)
    {
        std::string currProd = p;
        char nonTerminal = currProd[0];
        nonTerminals.insert(nonTerminal);

        std::string rhs = currProd.substr(3);
        size_t index = 0;

        // If X -> Îµ is production, add Îµ to FIRST(X)
        while (index < rhs.length())
        {
            if (rhs[index] == EPLISON)
            {
                first[nonTerminal].insert(EPLISON);
            }
            index++;
        }
    }

    computeFirst();
    computeFollow();
}

// Computes the FIRST set following the algorithm
// given in the CSE 570 website
void Parser::computeFirst()
{
    do
    {
        changed = false;

        // Go through the productions
        for (auto p : productions)
        {
            std::string currProduction = p;
            char nonTerminal = currProduction[0];
            std::string rhs = currProduction.substr(3);

            for (size_t i = 0; i < rhs.length();)
            {
                // Retrieve a set from FIRST(X)
                std::set<char> temp = first[rhs[i]];

                // Add everything in FIRST(Y_i) except for Îµ and
```

```
                    // increment i
                    if (util.hasEplison(temp))
                    {
                        temp.erase(temp.find(EPLISON));
                        std::set<char> unionSet =
util.setUnion(first[nonTerminal], temp);

                        for (auto s : unionSet)
                        {
                            auto checkInsert =
first[nonTerminal].insert(s);

                            // Check if a set in FIRST has been changed
after inserting
                            // a new nonTerminal key
                            if (checkInsert.second)
                            {
                                changed = true;
                            }
                        }

                        i++;
                    }
                    // Else FIRST(X) = {X}
                    else
                    {
                        std::set<char> unionSet =
util.setUnion(first[nonTerminal], temp);

                        for (auto s : unionSet)
                        {
                            auto checkInsert =
first[nonTerminal].insert(s);

                            if (checkInsert.second)
                            {
                                changed = true;
                            }
                        }
                        break;
                    }
                    // If i > k then add Îµ to FIRST(X)
                    if (i >= rhs.length())
                    {
                        auto checkInsert =
first[nonTerminal].insert(EPLISON);

                        if (checkInsert.second)
                        {
                            changed = true;
                        }
                    }
                    i++;
                }
            }
        } while (changed);
     }

     // Computes the FOLLOW set following the algorithm
     // given in the CSE 570 website
     void Parser::computeFollow()
     {
        do
        {
            changed = false;

            // Go through the productions of G
            for (auto p : productions)
            {
                std::string currProduction = p;
                char nonTerminal = currProduction[0];
                std::string rhs = currProduction.substr(3);

                // Go through the characters in the right hand side of
     the production
                // e.g. E -> T + F where T + F is the RHS
                for (size_t i = 0; i < rhs.length();)
                {
                    // Check if the current char in the RHS is a
     non-terminal
                    // and making sure we still have input to process on
     the RHS
                    if (util.isNonTerminal(rhs[i]) && i < rhs.length() -
     1)
                    {
                        // Retrieve a set in FIRST based the non-terminal
                        // we get from the RHS of the production
                        std::set<char> temp = first[rhs[i + 1]];

                        // If A -> Î±BÎ² and Îµ is in FIRST(Î²) then
                        // add everything in FOLLOW(A) to FOLLOW(B)
                        if (util.hasEplison(temp))
                        {

                            std::set<char> nonTerminalFollowSet =
     follow[nonTerminal];
                            for (auto n : nonTerminalFollowSet)
                            {
                                auto checkInsert =
     follow[rhs[i]].insert(n);
```

```
                                if (checkInsert.second)
                                {
                                    changed = true;
                                }
                            }
                        }
                        // Else, just add everything in FIRST(Î²) except
Îµ to FOLLOW(B)
                        for (auto t : temp)
                        {
                            auto checkInsert = follow[rhs[i]].insert(t);
                            if (checkInsert.second)
                            {
                                changed = true;
                            }
                        }
                    }
                    // If we reached the last character in the RHS
                    // then just add everything in FOLLOW(A) to FOLLOW(B)
                    else if (util.isNonTerminal(rhs[i]) && i ==
rhs.length() - 1)
                    {
                        std::set<char> temp = follow[nonTerminal];
                        for (auto t : temp)
                        {
                            auto checkInsert = follow[rhs[i]].insert(t);
                            if (checkInsert.second)
                            {
                                changed = true;
                            }
                        }
                    }
                    i++;
                }
            }
    } while (changed);

    // Add the $ symbol to FOLLOW(S)
    follow['S'].insert('$');
}

void Parser::printFirst() const
{
    std::cout << "FIRST = " << '\n';
    for (auto f : first)
    {
        std::cout << f.first << " -> ";
        std::cout << "{ ";
        for (auto s : f.second)
        {
            std::cout << s << ' ';
        }
        std::cout << "}";
        std::cout << '\n';
    }
}

void Parser::printFollow() const
{
    std::cout << "FOLLOW = " << '\n';
    for (auto f : follow)
    {
        std::cout << f.first << " -> ";
        std::cout << "{ ";
        for (auto s : f.second)
        {
            std::cout << s << ' ';
        }
        std::cout << "}";
        std::cout << '\n';
    }
}

void Parser::printProductions() const
{
    std::cout << "The productions in " << grammer << " are:" << '\n';
    for (auto p : productions)
    {
        std::cout << p << '\n';
    }
    std::cout << '\n';
}
```

- **function.cpp**

```
#include "function.h"

#include <set>
#include <cctype>

#define EPLISON 'e'

// Checks if a set has an eplison
bool Function::hasEplison(std::set<char> &s)
{
    return s.find(EPLISON) != s.end();
}

// Combines two sets which one set will contain
// all of the elements from the two sets
```

```cpp
std::set<char> Function::setUnion(std::set<char> &s1, std::set<char>
&s2)
{
   std::set<char> result;
   result.insert(s1.begin(), s1.end());
   result.insert(s2.begin(), s2.end());
   return result;
}

// Checks if a character is an non-terminal
// where a non-terminal is uppercase and alphanumeric
bool Function::isNonTerminal(char input)
{
   return std::isalpha(input) && std::isupper(input);
}
```

- parser.h

```cpp
#pragma once
#ifndef PARSER_H
#define PARSER_H

#include "function.h"
#include <string>
#include <map>
#include <set>
#include <fstream>

class Parser
{
 public:
   Parser(std::string &grammerFile);

   void parse();
   void computeFirst();
   void computeFollow();

   void printFirst() const;
   void printFollow() const;
   void printProductions() const;

 private:
   std::string grammer;

   std::set<std::string> productions;
   std::set<char> nonTerminals;
   std::map<char, std::set<char> > first;
   std::map<char, std::set<char> > follow;

   Function util;
```

```
    bool changed;
};

#endif
```

- **function.h**

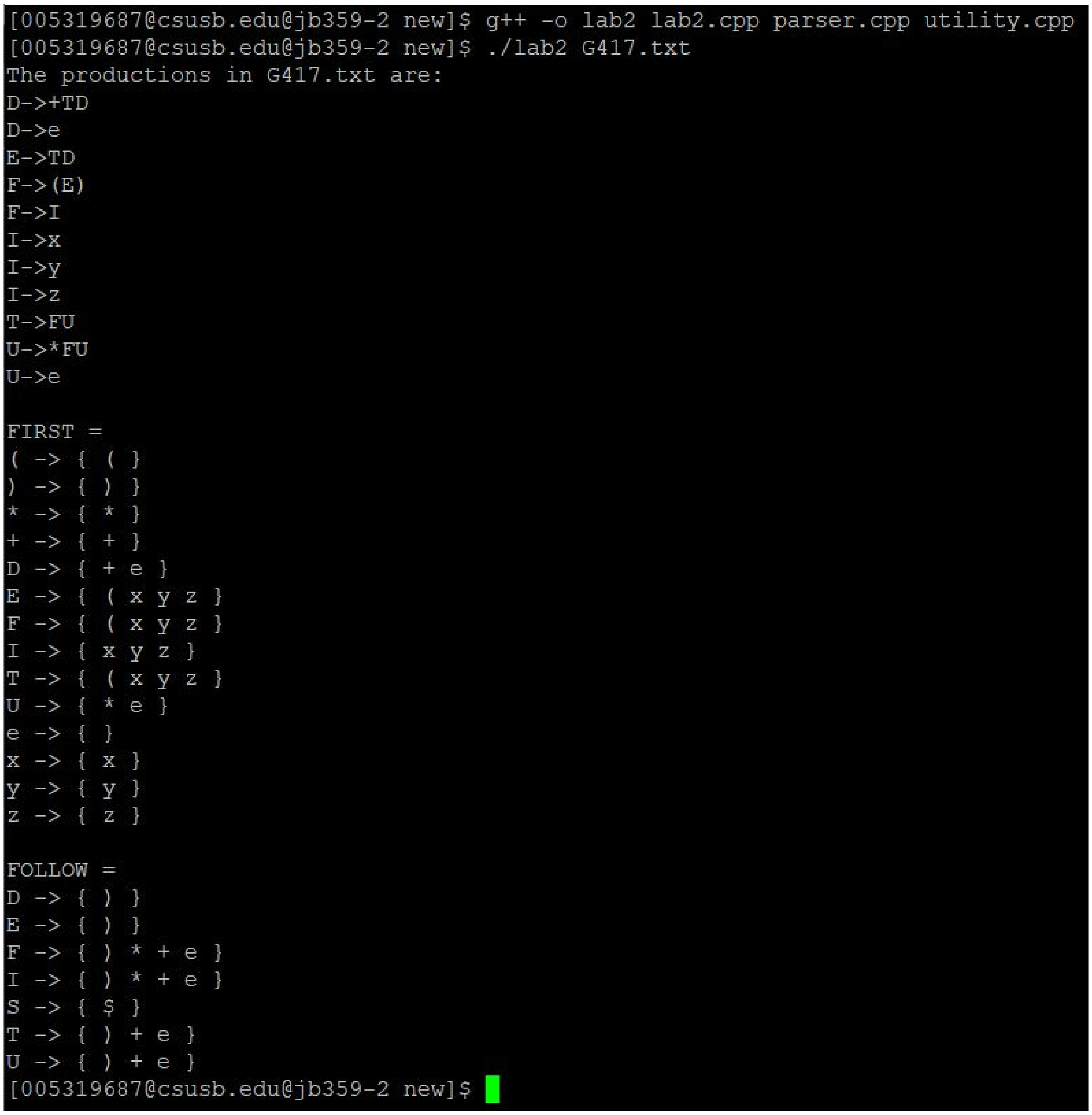
```
#pragma once
#ifndef FUNCTION_H
#define FUNCTION_H

#include <set>

class Function
{
 public:
    Function(){};
    bool hasEplison(std::set<char> &s);
    std::set<char> setUnion(std::set<char> &s1, std::set<char> &s2);
    bool isNonTerminal(char input);
};

#endif
```

- G417.txt
  ```
  x
  y
  z
  +
  *
  (
  )
  $
  E->TD
  D->+TD
  D->e
  T->FU
  U->*FU
  U->e
  F->(E)
  F->I
  I->x
  I->y
  I->z
  $
  ```

- samplegrammar.txt
  ```
  (
  )
  +
  *
  e
  i
  $
  E->TX
  X->+E
  T->iY
  Y->*T
  X->e
  T->(E)
  Y->e
  $
  ```

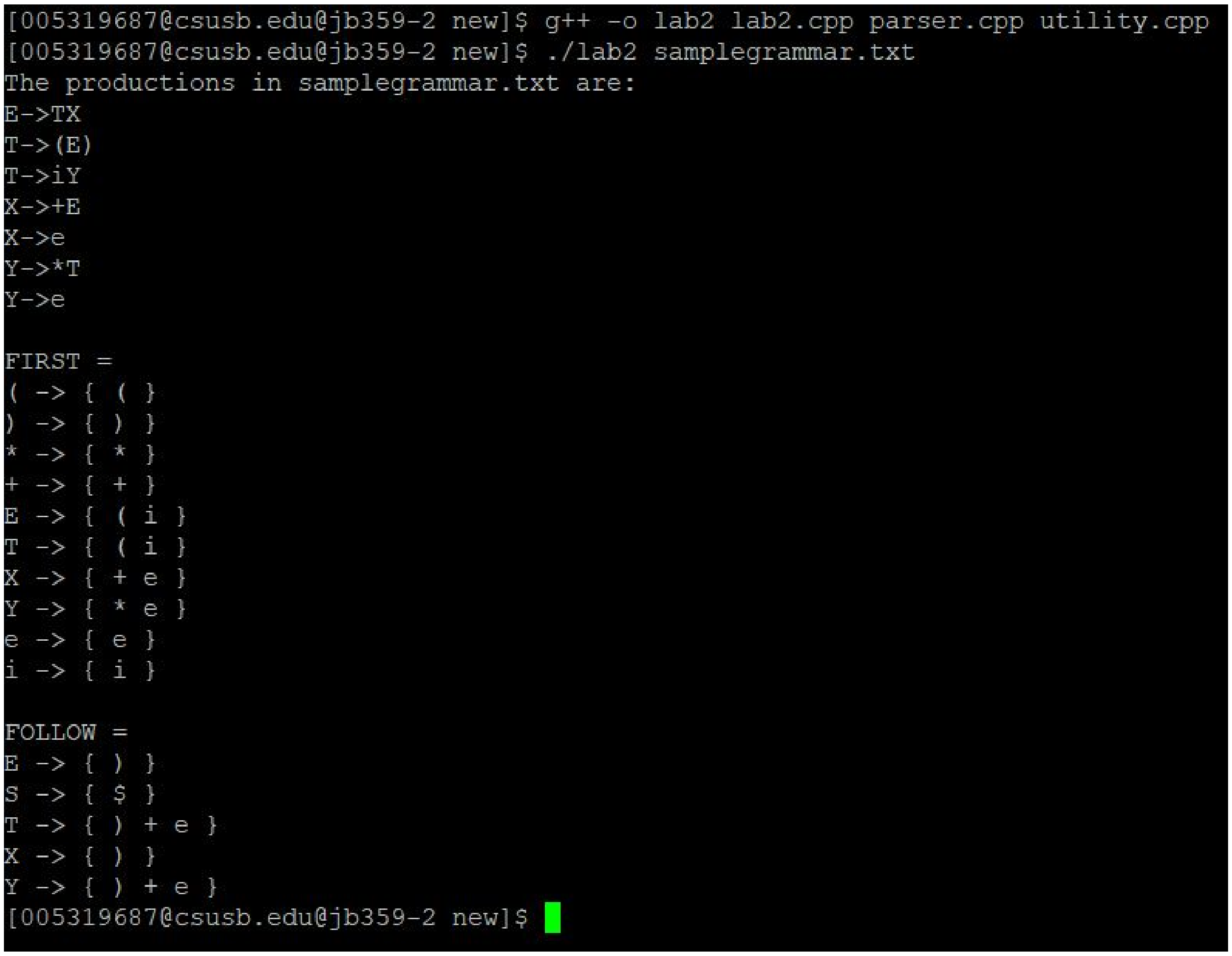
Outputs:

- Test results for grammar G417:



```
[005319687@csusb.edu@jb359-2 new]$ g++ -o lab2 lab2.cpp parser.cpp utility.cpp
[005319687@csusb.edu@jb359-2 new]$ ./lab2 G417.txt
The productions in G417.txt are:
D->+TD
D->e
E->TD
F->(E)
F->I
I->x
I->y
I->z
T->FU
U->*FU
U->e

FIRST =
( -> { ( }
) -> { ) }
* -> { * }
+ -> { + }
D -> { + e }
E -> { ( x y z }
F -> { ( x y z }
I -> { x y z }
T -> { ( x y z }
U -> { * e }
e -> { }
x -> { x }
y -> { y }
z -> { z }

FOLLOW =
D -> { ) }
E -> { ) }
F -> { ) * + e }
I -> { ) * + e }
S -> { $ }
T -> { ) + e }
U -> { ) + e }
[005319687@csusb.edu@jb359-2 new]$
```

- Test results for one other sample grammar:



```
[005319687@csusb.edu@jb359-2 new]$ g++ -o lab2 lab2.cpp parser.cpp utility.cpp
[005319687@csusb.edu@jb359-2 new]$ ./lab2 samplegrammar.txt
The productions in samplegrammar.txt are:
E->TX
T->(E)
T->iY
X->+E
X->e
Y->*T
Y->e

FIRST =
( -> { ( }
) -> { ) }
* -> { * }
+ -> { + }
E -> { ( i }
T -> { ( i }
X -> { + e }
Y -> { * e }
e -> { e }
i -> { i }

FOLLOW =
E -> { ) }
S -> { $ }
T -> { ) + e }
X -> { ) }
Y -> { ) + e }
[005319687@csusb.edu@jb359-2 new]$
```