Christian Mesina
Hugo Romero
Luis Escobar Urrutia

Lab 3

Introduction:

- For this lab, we wrote a program that reads a grammar and finds its canonical LR(0) sets. This program outputs the list of all productions in G and the canonical LR(0) collection of sets for G. In addition to numbering the sets, it identifies the symbol X that generates the set through the function GOTO(I,X). The test results used for this program are grammar G419 and at least one other grammar.

Source Code:

- **lab3.cpp**

```
/*
* Christian Mesina, Hugo Romero, Luis Escobar Urrutia
* CSE 570 Compilers Lab 3
* 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: ./lab3
<filename>\n";
       return 0;
   }

   std::string grammer = argv[1];

   Parser *parser = new Parser(grammer);

   parser->parse();

   //parser->printFirst();

   //parser->printFollow();

   parser->printCanonicalSet();
```

```
    delete parser;

    return 0;
}
```

- **Parser.cpp**

```
#include "Parser.h"
#include "Utility.h"
#include "Item.h"

#include <iostream>
#include <string>
#include <map>
#include <set>
#include <fstream>
#include <algorithm>

#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]);
           terminals.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();
   findCanonicalSet();
}

// Computes the FIRST set following the algorithm
// given in the CSE 570/670 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 570/670 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('$');
}

std::set<Item> Parser::closure(std::set<Item> items)
{
    std::set<Item> closure;

    // Add evrery item to Closure(i)
    for (auto i : items)
    {
        closure.insert(i);
    }

    do
    {
        changed = false;
        for (auto c : closure)
        {
            std::string currProduction = c.getProduction();
            int dotPos = c.getDotPos();

            char lookAhead = currProduction[dotPos];

            // If A -> Î±BÎ² is in Closure(j)
            if (util.isNonTerminal(lookAhead))
            {
                for (auto p : productions)
                {
                    // If B is a production by using a look ahead
                    if (p[0] == lookAhead)
                    {
                        Item temp(p, 3);

                        // Add B to Closure(j)
                        auto checkInsert = closure.insert(temp);
                        if (checkInsert.second)
                        {
                            changed = true;
                        }
                    }
                }
            }
        }
    } while (changed);

    return closure;
}
```

```
std::set<Item> Parser::getGoto(std::set<Item> items, char symbol)
{
   // The empty set j
   std::set<Item> j;

   for (auto item : items)
   {
       std::string currProduction = item.getProduction();
       int dotPos = item.getDotPos();

       char lookAhead = currProduction[dotPos];

       if (symbol == lookAhead)
       {
           Item temp(currProduction, dotPos + 1);
           j.insert(temp);
       }
   }
   return closure(j);
}

void Parser::findCanonicalSet()
{
   std::set<char> symbols;

   std::string startProduction = "S->E";

   // Add the first production S->E to the canonical set to
   // generate the correct item sets
   auto firstItem = Item(startProduction, 3);
   std::set<Item> tempItem;
   tempItem.insert(firstItem);
   auto firstClosure = closure(tempItem);
   canonicalSet.insert(LRSet(firstClosure, 0, '\0'));

   for (auto p : productions)
   {
       auto currentItem = Item(p, 3);
       std::set<Item> tempItem;
       tempItem.insert(currentItem);
       auto currentClosure = closure(tempItem);

       symbols = util.setUnion(terminals, nonTerminals);

       int id = 1;
       do
       {
           changed = false;
           for (auto item : canonicalSet)
           {
               for (auto symbol : symbols)
               {
                   auto temp = getGoto(item.getClosure(), symbol);

                   // If GOTO(I, X) is not empty && not in the
canonicalSet
                   // then add the GOTO(I, X) to the canonicalSet
                   if (!temp.empty() && !isIn(canonicalSet, temp))
                   {
                       LRSet lrset(temp, id, symbol);
                       canonicalSet.insert(lrset);
                       changed = true;
                       ++id;
                   }
               }
           }
       } while (changed);
   }
}

// Checks if a set of items is in the LRSet
bool Parser::isIn(std::set<LRSet> lrset, std::set<Item> items)
{
   for (auto curr : lrset)
   {
       if (curr.getClosure() == items)
           return true;
   }

   return false;
}

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';
}

void Parser::printCanonicalSet() const
{
   for (auto c : canonicalSet)
   {
       auto temp = c.getClosure();
       std::cout << "Item: " << c.getId() << '\n';
       std::cout << "Symbol: " << c.getSymbol() << '\n';
       for (auto t : temp)
       {
           std::string currProd = t.getProduction();
           currProd.insert(currProd.begin() + t.getDotPos(), '.');
           std::cout << "Production: [" << t.getDotPos() << "] = "
<< currProd << '\n';
       }
       std::cout << '\n';
   }
}
```

- Item.cpp

```
#include "Item.h"
#include <string>

std::string Item::getProduction()
{
    return production;
}

int Item::getDotPos()
{
   return dotPos;
}

bool Item::operator==(const Item &other) const
{
   return other.production == production;
}

bool Item::operator<(const Item &other) const
{
   return other.production > production || dotPos < other.dotPos;
}

Item &Item::operator=(const Item &other)
{
   production = other.production;
   dotPos = other.dotPos;
   return *this;
}
```

- Utility.cpp

```
#include "Utility.h"

#include <set>
#include <cctype>

#define EPLISON 'e'

// Checks if a set has an eplison
bool Utility::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
std::set<char> Utility::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 Utility::isNonTerminal(char input)
{
   return std::isalpha(input) && std::isupper(input);
}
```

- **LRSet.cpp**

```
#include "LRSet.h"
#include "Item.h"

#include <set>

std::set<Item> LRSet::getClosure() const
{
   return closure;
}

int LRSet::getId() const
{
   return id;
}

char LRSet::getSymbol() const
{
   return symbol;
}

bool LRSet::operator<(const LRSet &other) const
{
   return id < other.id;
}
```

- **Parser.h**

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

#include "Utility.h"
#include "Item.h"
#include "LRSet.h"

#include <string>
#include <map>
#include <set>
#include <fstream>

class Parser
```

```cpp
{
public:
 Parser(std::string &grammerFile);

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

 bool isIn(std::set<LRSet> lrset, std::set<Item> items);

 std::set<Item> closure(std::set<Item> items);
 std::set<Item> getGoto(std::set<Item> items, char symbol);

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

private:
 std::string grammer;

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

 Utility util;

 bool changed;
};

#endif
```

- **Item.h**

```cpp
#pragma once
#ifndef ITEM_H
#define ITEM_H

#include <string>

class Item
{
public:
 Item(std::string &production, int dotPos) : production(production),
dotPos(dotPos){};
```

```
 std::string getProduction();
 int getDotPos();

 bool operator==(const Item &) const;
 bool operator<(const Item &) const;
 Item &operator=(const Item &);

private:
 std::string production;
 int dotPos;
};

#endif
```

- **Utility.h**

```
#pragma once
#ifndef UTILITY_H
#define UTILITY_H

#include <set>

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

};

#endif
```

- **LRSet.h**

```
#pragma once
#ifndef LRSET_H
#define LRSET_H

#include "Item.h"
#include <set>

class LRSet
{
 public:
   LRSet(std::set<Item> closure, int id, char symbol)
       : closure(closure), id(id), symbol(symbol){};

   std::set<Item> getClosure() const;
   int getId() const;
   char getSymbol() const;

   bool operator<(const LRSet &) const;

 private:
   std::set<Item> closure;
   int id;
   char symbol;
};

#endif
```

- G419.txt

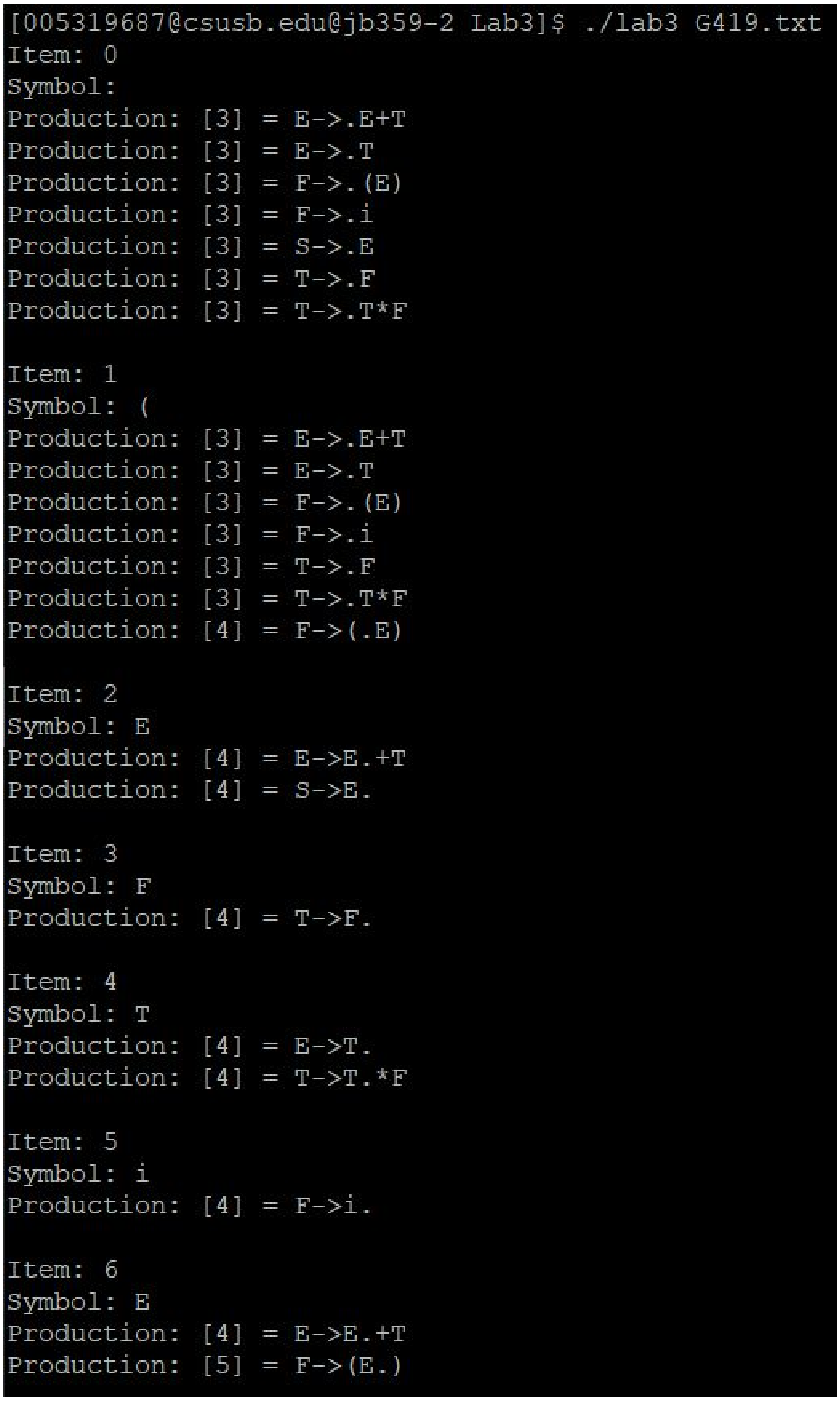
  ```
  i
  +
  *
  (
  )
  $
  S->E
  E->E+T
  E->T
  T->T*F
  T->F
  F->(E)
  F->i
  $
  ```

- samplegrammar.txt

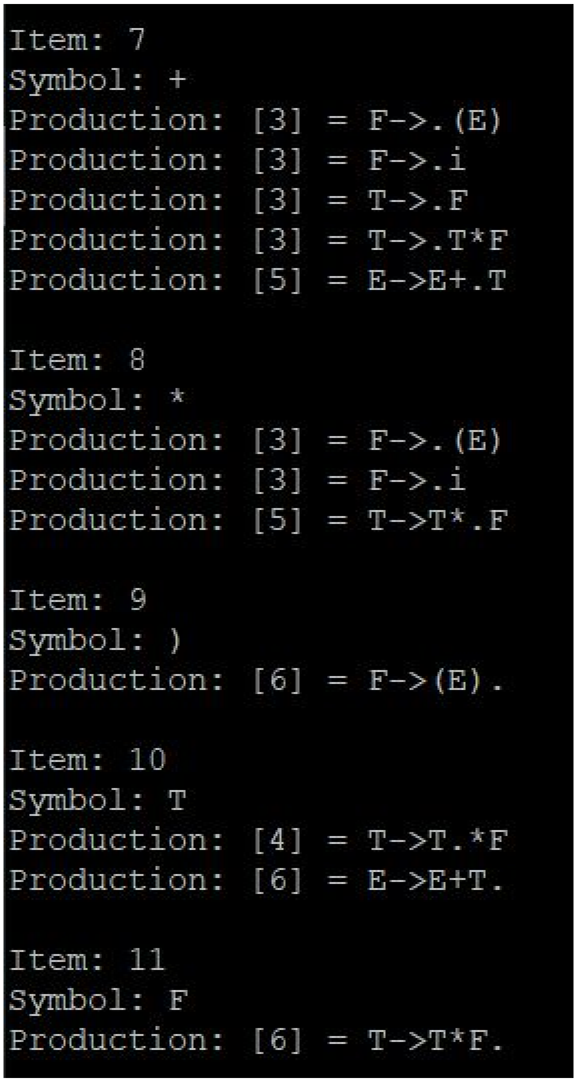
  ```
  (
  )
  +
  *
  e
  i
  $
  E->TX
  X->+E
  T->iY
  Y->*T
  X->e
  T->(E)
  Y->e
  $
  ```

Outputs:

- Test results for grammar G419:

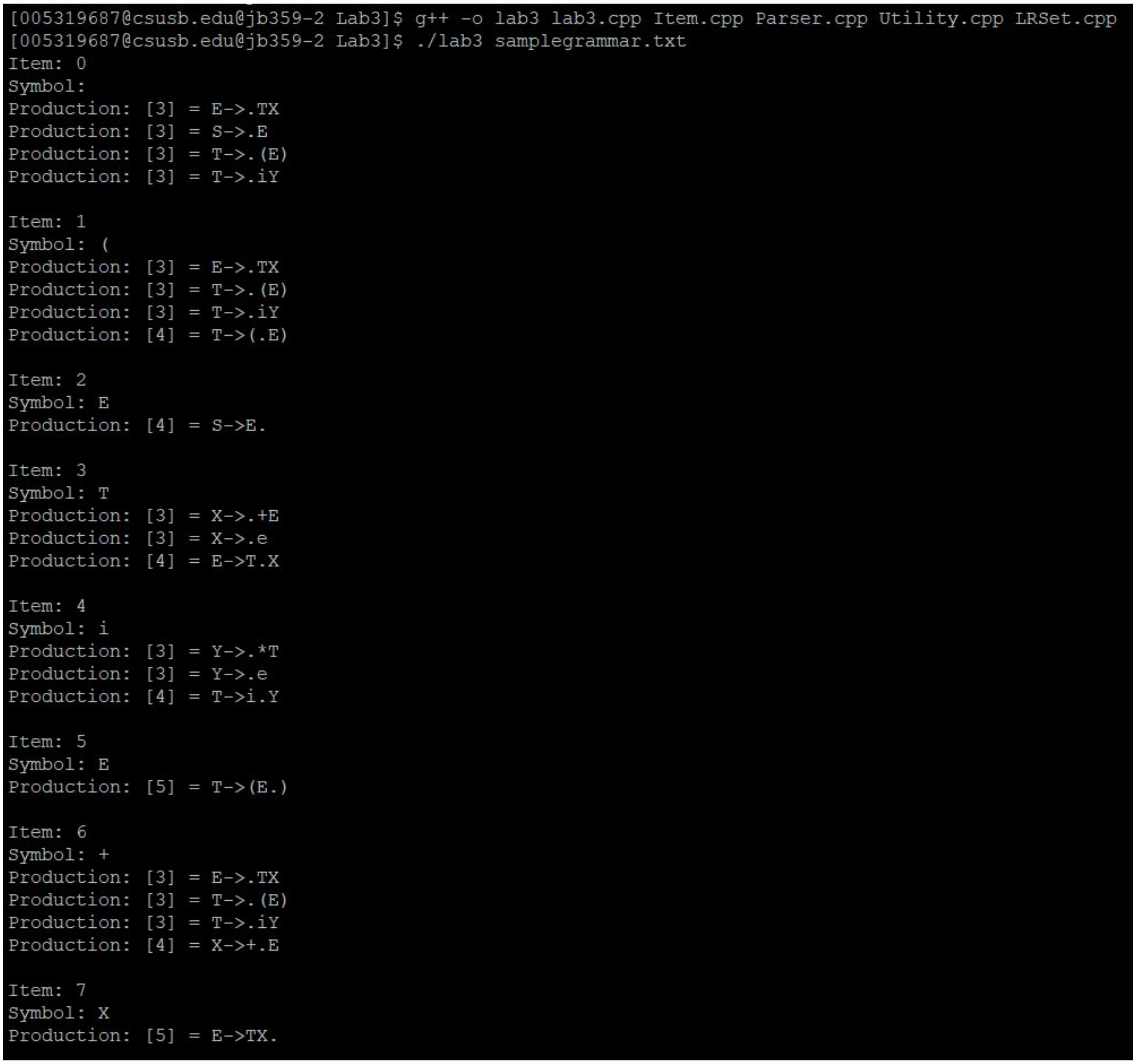


```
[005319687@csusb.edu@jb359-2 Lab3]$ ./lab3 G419.txt
Item: 0
Symbol:
Production: [3] = E->.E+T
Production: [3] = E->.T
Production: [3] = F->.(E)
Production: [3] = F->.i
Production: [3] = S->.E
Production: [3] = T->.F
Production: [3] = T->.T*F

Item: 1
Symbol: (
Production: [3] = E->.E+T
Production: [3] = E->.T
Production: [3] = F->.(E)
Production: [3] = F->.i
Production: [3] = T->.F
Production: [3] = T->.T*F
Production: [4] = F->(.E)

Item: 2
Symbol: E
Production: [4] = E->E.+T
Production: [4] = S->E.

Item: 3
Symbol: F
Production: [4] = T->F.

Item: 4
Symbol: T
Production: [4] = E->T.
Production: [4] = T->T.*F

Item: 5
Symbol: i
Production: [4] = F->i.

Item: 6
Symbol: E
Production: [4] = E->E.+T
Production: [5] = F->(E.)
```

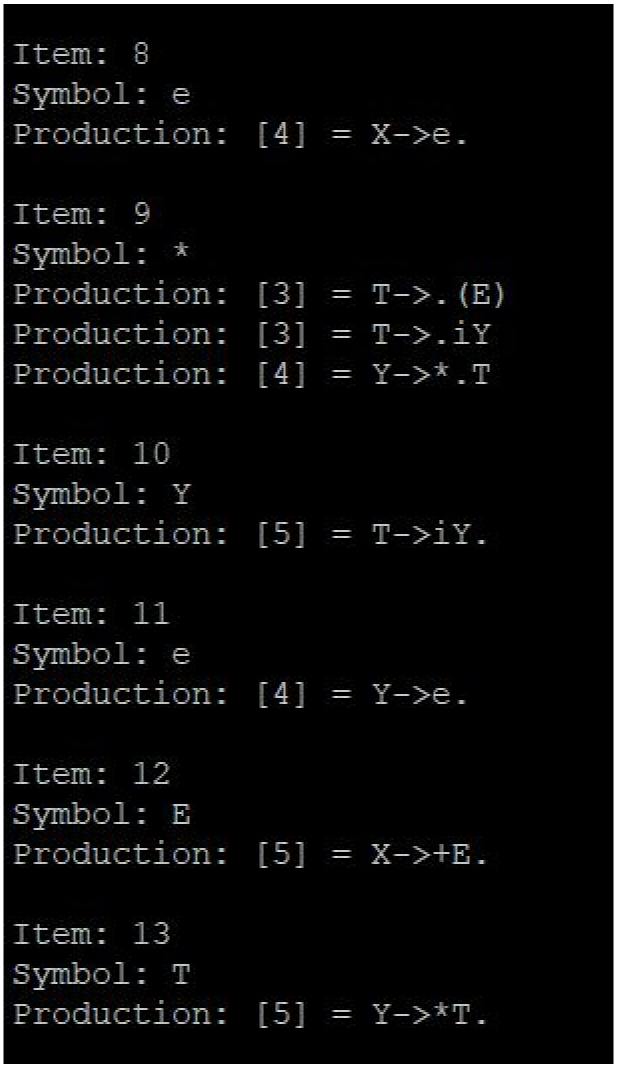


```
Item: 7
Symbol: +
Production: [3] = F->.(E)
Production: [3] = F->.i
Production: [3] = T->.F
Production: [3] = T->.T*F
Production: [5] = E->E+.T

Item: 8
Symbol: *
Production: [3] = F->.(E)
Production: [3] = F->.i
Production: [5] = T->T*.F

Item: 9
Symbol: )
Production: [6] = F->(E).

Item: 10
Symbol: T
Production: [4] = T->T.*F
Production: [6] = E->E+T.

Item: 11
Symbol: F
Production: [6] = T->T*F.
```

- Test results for one other sample grammar:



```
[005319687@csusb.edu@jb359-2 Lab3]$ g++ -o lab3 lab3.cpp Item.cpp Parser.cpp Utility.cpp LRSet.cpp
[005319687@csusb.edu@jb359-2 Lab3]$ ./lab3 samplegrammar.txt
Item: 0
Symbol:
Production: [3] = E->.TX
Production: [3] = S->.E
Production: [3] = T->.(E)
Production: [3] = T->.iY

Item: 1
Symbol: (
Production: [3] = E->.TX
Production: [3] = T->.(E)
Production: [3] = T->.iY
Production: [4] = T->(.E)

Item: 2
Symbol: E
Production: [4] = S->E.

Item: 3
Symbol: T
Production: [3] = X->.+E
Production: [3] = X->.e
Production: [4] = E->T.X

Item: 4
Symbol: i
Production: [3] = Y->.*T
Production: [3] = Y->.e
Production: [4] = T->i.Y

Item: 5
Symbol: E
Production: [5] = T->(E.)

Item: 6
Symbol: +
Production: [3] = E->.TX
Production: [3] = T->.(E)
Production: [3] = T->.iY
Production: [4] = X->+.E

Item: 7
Symbol: X
Production: [5] = E->TX.
```



```
Item: 8
Symbol: e
Production: [4] = X->e.

Item: 9
Symbol: *
Production: [3] = T->.(E)
Production: [3] = T->.iY
Production: [4] = Y->*.T

Item: 10
Symbol: Y
Production: [5] = T->iY.

Item: 11
Symbol: e
Production: [4] = Y->e.

Item: 12
Symbol: E
Production: [5] = X->+E.

Item: 13
Symbol: T
Production: [5] = Y->*T.
```