```
/* Author: Luis Carbajal
e-mail: luiscarbajal.se@gmail.com

Program that computes the first and follow sets of a grammar.*/


#include <iostream>
#include <cstring>
#include <stdlib.h>
#include <vector>
#include <list>
#include <set>
#include <string>
#include <algorithm>
extern "C"
{
#include "lexer.h"

}
enum type{
      terminal = 1,
      nonterminal,
      epsilon
};
struct symbol{
      type symbol_type;
      std::string string_rep;
};
struct rule{
      symbol lhs;
      std::vector<symbol> rhs;
};
struct Sets{
      std::string setof;
      std::set<std::string> theset;
};

type determine_type(std::string, std::vector<symbol>);
bool is_duplicate(std::string, std::vector<symbol>);
void grammar_info(std::list<rule>, std::vector<symbol>,
std::vector<symbol>);
bool contains_epsilon(std::set<std::string>);
Sets *FirstSets(std::list<rule>, std::vector<symbol>,
std::vector<symbol>);
void FollowSets(std::list<rule>, std::vector<symbol>,
std::vector<symbol>);
void display_sets(int, Sets *arr, std::string);

int main()
{
      int task;

      std::vector<symbol> term;
      std::vector<symbol> nonterm;
```

```
	std::list<rule> CFG;

	symbol var;
	rule subrule;

	while (t_type != HASH)
	{
		getToken();

		if (t_type == ID)
		{
			var.symbol_type = nonterminal;
			var.string_rep = current_token;
			nonterm.push_back(var);
		}
	}

	while (t_type != DOUBLEHASH)
	{
		getToken();

		if (t_type == ID)
		{
			var.symbol_type = determine_type(current_token,
nonterm);
			var.string_rep = current_token;
		}
		else if (t_type == ARROW)
		{
			if (var.symbol_type == nonterminal)
			{
				subrule.lhs.symbol_type = var.symbol_type;
				subrule.lhs.string_rep = var.string_rep;
			}
		}

		if (!subrule.lhs.string_rep.empty() && t_type == ID)
		{

			subrule.rhs.push_back(var);
			if (var.symbol_type == terminal &&
!is_duplicate(var.string_rep, term))
				term.push_back(var);
		}
		else if (t_type == HASH && !subrule.lhs.string_rep.empty())
		{
			if (subrule.rhs.empty())
			{
				var.symbol_type = epsilon;
				var.string_rep = "#";
				subrule.rhs.push_back(var);
			}
			CFG.push_back(subrule);
			subrule.lhs.string_rep.clear();
```

```
                subrule.rhs.clear();
            }
        }

        /*while (t_type != END_OF_FILE)
        {
            getToken();
        } */

        std::cin >> task;

        switch (task)
        {
        case 0:
            grammar_info(CFG, nonterm, term);
            break;
        case 1:
        {
            Sets *arr = FirstSets(CFG, nonterm, term);
            display_sets(nonterm.size(), arr, "FIRST");
        }
            break;
        case 2:
            FollowSets(CFG, nonterm, term);
            break;
        default:
            std::cout << "Error: unrecognized task number " << task <<
::std::endl;
            break;

        }

        return 0;
}

type determine_type(std::string token, std::vector<symbol> vec)
{
        for (size_t i = 0; i < vec.size(); i++)
        {
            if (vec.at(i).string_rep == token)
            {
                return nonterminal;
            }
        }
        return terminal;
}
bool is_duplicate(std::string token, std::vector<symbol> vec)
{
        for (size_t i = 0; i < vec.size(); i++)
        {
            if (vec.at(i).string_rep == token)
            {
                return true;
            }
```

```
	}
	return false;
}
bool contains_epsilon(std::set<std::string> aset)
{
	for (std::set<std::string>::iterator it = aset.begin(); it !=
aset.end(); ++it)
	{
		if (*it == "#")
			return true;
	}
	return false;

}
void display_sets(int size, Sets *arr, std::string typeofset)
{
	for (int i = 0; i < size; i++)
	{
		std::cout << typeofset << "(" << arr[i].setof << ") = { ";
		for (std::set<std::string>::iterator it =
arr[i].theset.begin(); it != arr[i].theset.end(); ++it)
		{
			std::cout << *it;
			if (it != std::max_element(arr[i].theset.begin(),
arr[i].theset.end())) //Find maximum element in set:
http://en.cppreference.com/w/cpp/algorithm/max_element
				std::cout << ", ";
		}
		std::cout << " }" << std::endl;
	}
}
void grammar_info(std::list<rule> CFG, std::vector<symbol> nonterm,
std::vector<symbol> term)
{
	for (size_t i = 0; i < term.size(); i++)
		std::cout << term.at(i).string_rep << " ";
	std::cout << std::endl;

	for (size_t i = 0; i < nonterm.size(); i++)
	{
		int cnt = 0;
		for (std::list<rule>::const_iterator iterator = CFG.begin(),
end = CFG.end(); iterator != end; ++iterator){
			if (iterator->lhs.string_rep ==
nonterm.at(i).string_rep)
				++cnt;
		}
		std::cout << nonterm.at(i).string_rep << ": " << cnt <<
std::endl;
	}
}
Sets *FirstSets(std::list<rule> CFG, std::vector<symbol> nonterm,
std::vector<symbol> term)
{
```

```
	Sets *arr = new Sets[nonterm.size()];
	int j = 0;
	for (size_t i = 0; i < nonterm.size(); i++)
	{
		arr[j].setof = nonterm.at(i).string_rep;
		j++;
	}

	for (unsigned int ivector = 0; ivector < term.size(); ivector++)
	{

		for (size_t i = 0; i < nonterm.size(); i++)
		{
			for (std::list<rule>::const_iterator itCFG =
CFG.begin(), end = CFG.end(); itCFG != end; ++itCFG)
			{
				if (arr[i].setof == itCFG->lhs.string_rep)
				{
					for (size_t iv = 0; iv < itCFG->rhs.size();
iv++)
					{
						if (itCFG->rhs.at(iv).symbol_type ==
terminal || itCFG->rhs.at(iv).symbol_type == epsilon)
						{
							arr[i].theset.insert(itCFG-
>rhs.at(iv).string_rep);
							break;
						}
						else if (itCFG->rhs.at(iv).symbol_type
== nonterminal)
						{

							size_t i2;
							for (i2 = 0; i2 < nonterm.size();
i2++)
							{
								if (arr[i2].setof == itCFG-
>rhs.at(iv).string_rep)
								{
									std::set<std::string>
sub(arr[i2].theset);
									if
(contains_epsilon(sub) && iv != itCFG->rhs.size() - 1)
										sub.erase("#");

	arr[i].theset.insert(sub.begin(), sub.end());
									break;
								}
							}
							if
(!contains_epsilon(arr[i2].theset))
								break;
						}
					}
```

```
                        }
                  }
            }
      }

      return arr;
}
void FollowSets(std::list<rule> CFG, std::vector<symbol> nonterm,
std::vector<symbol> term)
{

      Sets *SetsofFirst = FirstSets(CFG, nonterm, term);

      Sets *SetofFollows = new Sets[nonterm.size()];
      int j = 0;
      for (size_t i = 0; i < nonterm.size(); i++)
      {
            SetofFollows[j].setof = nonterm.at(i).string_rep;
            j++;
      }

      SetofFollows[0].theset.insert("$");

      for (size_t it1 = 0; it1 < term.size(); it1++)
      {
            for (size_t i = 0; i < nonterm.size(); i++)
            {
                  for (std::list<rule>::const_iterator itCFG =
CFG.begin(), end = CFG.end(); itCFG != end; ++itCFG)
                  {
                        for (size_t iv = 0; iv < itCFG->rhs.size(); iv++)
                        {
                              if (itCFG->rhs.at(iv).string_rep ==
SetofFollows[i].setof)
                              {
                                    if (iv == itCFG->rhs.size() - 1)
                                    {
                                          for (size_t j = 0; j <
nonterm.size(); j++)
                                          {
                                                if (SetofFollows[j].setof
== itCFG->lhs.string_rep)
                                                {

      SetofFollows[i].theset.insert(SetofFollows[j].theset.begin(),
SetofFollows[j].theset.end());
                                                      break;
                                                }
                                          }
                                    }
                                    else if (itCFG->rhs.at(iv +
1).symbol_type == terminal)
                                    {
```

```
      SetofFollows[i].theset.insert(itCFG->rhs.at(iv + 1).string_rep);
                                    //break;
                              }
                              else if (itCFG->rhs.at(iv +
1).symbol_type == nonterminal)
                              {
                                    size_t j;
                                    for (j = 0; j < nonterm.size();
j++)
                                    {
                                          if (SetsofFirst[j].setof ==
itCFG->rhs.at(iv + 1).string_rep)
                                          {
                                                std::set<std::string>
sub(SetsofFirst[j].theset);
                                                if
(contains_epsilon(sub))
                                                      sub.erase("#");

      SetofFollows[i].theset.insert(sub.begin(), sub.end());
                                                break;
                                          }
                                    }
                                    if
(contains_epsilon(SetsofFirst[j].theset) && (iv + 1) == itCFG->rhs.size()
- 1)
                                    {
                                          for (size_t x = 0; x <
nonterm.size(); x++)
                                          {
                                                if
(SetofFollows[x].setof == itCFG->lhs.string_rep)
                                                {

      SetofFollows[i].theset.insert(SetofFollows[x].theset.begin(),
SetofFollows[x].theset.end());
                                                      break;
                                                }
                                          }
                                    }
                                    else if
(contains_epsilon(SetsofFirst[j].theset))
                                    {
                                          unsigned int iv2 = iv + 2;
                                          size_t itset;
                                          do
                                          {
                                                if (itCFG-
>rhs.at(iv2).symbol_type == terminal)
                                                {

      SetofFollows[i].theset.insert(itCFG->rhs.at(iv2).string_rep);
                                                      break;
```

```
                                                        }
                                                        for (itset = 0; itset
< nonterm.size(); itset++)
                                                        {
                                                              if
(SetsofFirst[itset].setof == itCFG->rhs.at(iv2).string_rep)
                                                              {

      std::set<std::string> sub(SetsofFirst[itset].theset);
                                                                    if
(contains_epsilon(sub))

      sub.erase("#");

      SetofFollows[i].theset.insert(sub.begin(), sub.end());
                                                                    break;
                                                              }
                                                        }
                                                        if
(contains_epsilon(SetsofFirst[itset].theset) && iv2 == itCFG->rhs.size()
- 1)
                                                        {
                                                              for (size_t x =
0; x < nonterm.size(); x++)
                                                              {
                                                                    if
(SetofFollows[x].setof == itCFG->lhs.string_rep)
                                                                    {

      SetofFollows[i].theset.insert(SetofFollows[x].theset.begin(),
SetofFollows[x].theset.end());

      break;
                                                                    }
                                                              }
                                                              break;
                                                        }
                                                        ++iv2;
                                                  } while
(contains_epsilon(SetsofFirst[itset].theset));
                                             }
                                       }

                                 }
                           }

                     }

               }
      }
      display_sets(nonterm.size(), SetofFollows, "FOLLOW");
}
```