**Introduction**

Your goal is to write, in C or C++, a program that reads in a description of a context free grammar, then, depending on the command line argument passed to the program, outputs either the FIRST sets for all non-terminals in the grammar or FOLLOW sets for all non-terminals in the grammar or other information about the grammar (read on for details). The input grammar will be given on standard input (stdin), and your program should output to standard output (stdout). Specifications for the input grammar and expected output follow, and they must be followed precisely.

**Reading Command-line Arguments**

The following piece of C code shows how to read the first command line argument and call a function based on the argument value:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54 | #include <stdio.h>  #include <stdlib.h>  int main (int argc, char\* argv[])  {  int task;  if (argc < 2)  {  printf("Error: missing argument\n");  return 1;  }  /\*  Note that by convention argv[0] is the name of your executable,  and the first argument to your program is stored in argv[1]  \*/  task = atoi(argv[1]);  // TODO: Read the input grammar at this point  /\*  Hint: You can modify and use the lexer from previous project  to read the input. Note that there are only 4 token types needed  for reading the input in this project.  WARNING: You will need to modify lexer.c and lexer.h to support  the project 3 input language.  \*/  switch (task) {  case 0:  // TODO: Output information about the input grammar  break;  case 1:  // TODO: Calculate FIRST sets for the input grammar  // Hint: You better do the calculation in a function and call it here!  // TODO: Output the FIRST sets in the exact order and format required  break;  case 2:  // TODO: Calculate FIRST sets for the input grammar  // TODO: Calculate FOLLOW sets for the input grammar  // TODO: Output the FOLLOW sets in the exact order and format required  break;  default:  printf("Error: unrecognized task number %d\n", task);  break;  }  return 0;  } |

**NOTE:** You can download this code as part of the project material, no need to copy/paste from this document.

**Grammar Description**

The grammar specification has multiple *sections* separated by the # symbol. The grammar specification is terminated with ##. If there are any symbols after the ##, they are ignored. All grammar symbols, as well as the # and ## symbols, are whitespace-separated. The grammar description is a context-free grammar as follows:

S → Non-terminal-list Rule-list DOUBLEHASH

Non-terminal-list → ID-list HASH

ID-list → ID ID-list | ID

Rule-list → Rule Rule-list | Rule

Rule → ID ARROW Right-hand-side HASH

Right-hand-side → ID-list | ε

The tokens used in the grammar description are:

ID = letter(letter|digit)\*

HASH = #

DOUBLEHASH = ##

ARROW = ->

where (note that digit is 0-9 and letter is all uppercase and lowercase letters, same as we have been using in class, however we define them here to be precise):

digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

letter = a | b | c | d | e | f | g | h | i | j | k | l | m | n | o

| p | q | r | s | t | u | v | w | x | y | z | A | B | C | D | E

| F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V

| W | X | Y | Z

In this project, we assume that there is at least one whitespace character (where whitespace is defined as an input that causes the isspace function in ctype.h to return 1) between any two tokens. Tokens are case-sensitive.

**Semantics**

The first section of the input lists all the non-terminals of the grammar. The first non-terminal in this list is the start symbol of the grammar. The following sections each represent a grammar rule.

A grammar rule starts with a non-terminal symbol (the left-hand side of the rule) followed by ->, then followed by a sequence of zero of more terminals and non-terminals, which represent the right-hand side of the rule. If the sequence of terminals and non-terminals in the right-hand side of a rule is empty, and if the left-hand side of the rule is the non-terminal A, then this represents a rule of the form A → ε.

**Example**

Here is an example input grammar:

decl idList1 idList #

decl -> idList colon ID #

idList -> ID idList1 #

idList1 -> #

idList1 -> COMMA ID idList1 # ##

The first section is the non-terminals list:

Non-Terminals = { decl, idList1, idList }

The rest of the input (terminated by the double-hash) specifies the grammar rules:

decl → idList colon ID

idList → ID idList1

idList1 → ε

idList1 → COMMA ID idList1

The set of terminals of the grammar consist of all ID tokens that appear in the grammar rules that are not non-terminals (once again, case sensitive). In the example:

Terminals = { colon, ID, COMMA }

Note that even though the example shows that each rule is on a line by itself, a rule can be split into multiple lines, or even multiple rules can be on the same line, according to the formal specification. This input is an equivalent grammar:

decl idList1 idList # decl -> idList colon ID # idList -> ID idList1 #

idList1 -> # idList1 -> COMMA ID idList1 #

##

**Implementation**

You should test your code on CentOS 7 as before. You should submit *all* your source files and there are no automatically-added source code this time. You might use a Makefile for this project if needed. You are welcome to use code from lexer.c and lexer.h from the previous project, however be warned that lexer.c and lexer.h are **for a different grammar**, so you **must** modify them to accomidate the project 3 grammar.

**Requirements**

Your program should read in the input grammar from standard input, and read the requested task number from the command line arguments as described, then calculate the requested output according to the task number and print the results in the exact format for each task to standard output (stdout). The following specifies the exact requirements for each task number.

**Task 0: Information about the grammar**

When a 0 is passed in as an argument to the program it will print out a summary of the CFG input. The first line of the summary will list all the non-terminals, which will be delimited by a space. The non-terminals will be listed in the order they are found in the input. After the last non-terminal, the line will be ended with a new line character.

The subsequent lines of the summary will create a line for each unique terminal. On the terminal lines, a terminal will be followed by colon :; a space; an integer, which represents the number of rules in which that terminal appears on the right-side; and a new line character. The terminal lines must be in dictionary order (specifically, the C library function strcmp from string.h).

**Output**

The output for the example grammar given previously when run with the command line argument of 0 like ./a.out 0:

|  |  |
| --- | --- |
| 1  2  3  4 | decl idList1 idList  COMMA: 1  ID: 3  colon: 1 |

**Task 1: FIRST Sets**

For each of the non-terminals of the input grammar, in the order that they appear in the non-terminal section of the input, compute the FIRST set for that non-terminal and output one line in the following format:

FIRST(<symbol>) = { <set\_items> }

Where <symbol> should be replaced by the non-terminal and <set\_items> should be replaced by the comma-separated list of elements of the FIRST set ordered in the following manner:

* If ε belongs in the set, represent it as #
* If ε belongs in the set, it should be listed before any other elements
* All other elements of the set should be sorted in dictionary order (specifically, the C function strcmp from string.h defines the sorting order)

**Output**

The output for the example grammar given previously when run with the command line argument of 1 like ./a.out 1:

FIRST(decl) = { ID }

FIRST(idList1) = { #, COMMA }

FIRST(iDList) = { ID }

**Task 2: FOLLOW Sets**

For each of the non-terminals of the input grammar, in the order that they appear in the non-terminals section of the input, compute the FOLLOW set for that non-terminal and output one line in the following format:

FOLLOW(<symbol>) = { <set\_items> }

Where <symbol> should be replaced by the non-terminal and <set\_items> should be replaced by the comma-separated list of elements of the FOLLOW set ordered in the following manner:

* If EOF belongs in the set, represent it as $
* If EOF belongs in the set, it should be listed before any other elements
* All other elements of the set should be sorted in dictionary order (specifically, the C function strcmp from string.h defines the sorting order)

**Output**

The output for the example grammar given previously when run with the command line argument of 2 like ./a.out 2:

FOLLOW(decl) = { $ }

FOLLOW(idList1) = { colon }

FOLLOW(idList) = { colon }