CSE340 Fall 2017 Project 2

Due: **October 6th, 2017** by 11:59pm MST

1 Introduction

In this project, you will write a C or C++ program that reads a description of a context free grammar, then, depending on the command line argument passed to the program, performs one of the following tasks: 1) for each terminal and non-terminal in the grammar determine the number of grammar rules in which the symbol appears, 2) determine *useless* symbols in the grammar and remove them, 3) calculate `FIRST` sets, 4) calculate `FOLLOW` sets and 5) determine if the grammar has a predictive parser.

2 Input Format

We specify the input format using a context-free grammar:

$$\begin{aligned}\text{Rule-list} &\to \text{Rule}\ \ \text{Rule-list} \mid \text{Rule}\\ \text{Id-list} &\to \texttt{ID}\ \ \text{Id-list} \mid \texttt{ID}\\ \text{Rule} &\to \texttt{ID}\ \ \texttt{ARROW}\ \ \text{Right-hand-side}\ \ \texttt{HASH}\\ \text{Right-hand-side} &\to \text{Id-list} \mid \epsilon\end{aligned}$$

The tokens used in the above grammar description are defined by the following regular expressions:

```
ID         = letter (letter + digit)*
HASH       = #
DOUBLEHASH = ##
ARROW      = ->
```

Where `digit` is the digits from 0 through 9 and `letter` is the upper and lower case letters a through z and A through Z. Tokens are case-sensitive. Tokens are space separated and there is at least one whitespace character between any two successive tokens.

3 Semantics

Each grammar rule starts with a non-terminal symbol (the left-hand side of the rule) followed by `ARROW`, then followed by a sequence of zero or 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 is empty, then it represents a rule of the form A $\to \epsilon$.

The set of non-terminals for the grammar is the set of symbols that appear to the left of an arrow. Grammar symbols that do not appear to the left of an arrow are terminal symbols.

Note that the convention of using upper-case letters for non-terminals and lower-case letters for terminals is not used here.

3.1 Example

Here is an example input:

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

The non-terminals of the grammar are:

$$\text{Non-Terminals} = \{\text{decl}, \text{idList1}, \text{idList}\}$$

The terminals of the grammar are:

$$\text{Terminals} = \{\text{ID}, \text{COMMA}, \text{colon}\}$$

and the grammar rules are:

$$\begin{aligned}
\text{decl} &\rightarrow \text{idList} \;\; \text{colon} \;\; \text{ID} \\
\text{idList} &\rightarrow \text{ID} \;\; \text{idList1} \\
\text{idList1} &\rightarrow \epsilon \\
\text{idList1} &\rightarrow \text{COMMA} \;\; \text{ID} \;\; \text{idList1}
\end{aligned}$$

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. The following input describes the same grammar as the above example:

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

4 Output Specifications

Your program should read the input grammar from standard input, and read the requested task number from the first command line argument (we provide code to read the task number) then calculate the requested output based on the task number and print the results in the specified format for each task to standard output (stdout). The following specifies the exact requirements for each task number.

4.1 Task 1

Task one simply outputs the list of terminals followed by the list of non-terminals in the order in which they appear in the grammar rules.

Example: For the input grammar

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

the expected output for task 1 is:

```
colon ID COMMA decl idList idList1
```

Example: Given the input grammar:

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

the expected output for task 1 is:

```
colon ID COMMA decl idList idList1
```

Note that in this example, even though the rule for `idList1` is before the rule for `idList`, `idList` appears before `idList1` in the grammar rules.

4.2 Task 2: Find useless symbols

Determine *useless* symbols in the grammar and remove them. Then output each rule of the modified grammar on a single line in the following format:

`<LHS> -> <RHS>`

Where `<LHS>` should be replaced by the left-hand side of the grammar rule and `<RHS>` should be replaced by the right-hand side of the grammar rule. If the grammar rule is of form $A \to \epsilon$, use `#` to represent the epsilon. Note that this is different from the input format. Also note that the order of grammar rules that are not removed from the original input grammar must be preserved.

Definition: Symbol A is *not* useless if there is a derivation starting from S (the start symbol) in which A appears and the derivation leads to a string of terminals w or the empty string ($w \in T^*$):

$$S \overset{*}{\Rightarrow} ...A... \overset{*}{\Rightarrow} w$$

Example 1: Given the following input grammar :

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

the expected output for task 2 is:

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

Note that none of the symbols were useless.

Example 2: Given the following input grammar:

```
S -> A B #
S -> C #
C -> c #
S -> a #
A -> a A #
B -> b #
##
```

the expected output for task 2 is:

```
S -> C
C -> c
S -> a
```

Note that A and B are useless symbols and the modified grammar has only three rules.

4.3 Task 3: Calculate `FIRST` Sets

For each of the non-terminals of the input grammar, in the order that they appear in grammar, 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 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 the order in which they appear in the terminal list (first section of the input)

Example: Given the input grammar:

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

the expected output for task 3 is:

```
FIRST(decl) = { ID }
FIRST(idList1) = { #, COMMA }
FIRST(iDList) = { ID }
```

4.4 Task 4: Calculate `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 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 the order in which they appear in the terminal list (first section of the input)

Example: Given the input grammar:

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

the expected output for task 3 is:

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

```
FOLLOW(decl) = { $ }
FOLLOW(idList1) = { colon }
FOLLOW(idList) = { colon }
```

4.5 Task 5: Determine if the grammar has a predictive parser

Determine if the grammar has a predictive parser and output either `YES` or `NO` accordingly.

Example: The expected output for the example input grammar given in section 3.1 for task 5 is:

```
YES
```

5 Implementation

5.1 Lexer

A lexer that can recognize `ID`, `ARROW`, `HASH` and `DOUBLEHASH` tokens is provided for this project. You are free to use it if you like.

5.2 Reading command-line argument

As mentioned in the introduction, your program must read the grammar from stdin and the task number from command line arguments. The following piece of code shows how to read the first command line argument and perform a task based on the value of that argument. Use this code as a starting point for your main function.

```
/* NOTE: You should get the full version of this code as part of the project
   material, do not copy/paste from this document. */

#include <stdio.h>
#include <stdlib.h>

int main (int argc, char* argv[])
{
    int task;

    if (argc < 2) {
        printf("Error: missing argument\n");
        return 1;
    }

    task = atoi(argv[1]);

    switch (task) {
        case 1:
            // TODO: perform task 1.
            break;

        // ...

        default:
            printf("Error: unrecognized task number %d\n", task);
            break;
    }
    return 0;
}
```

6 Evaluation

Your submission will be graded on passing the automated test cases. The test cases (there will be multiple test cases in each category, each with equal weight) will be broken down in the following way (out of 100 points):

- Task 1: 10 points
- Task 2: 30 points
- Task 3: 30 points
- Task 4: 25 points
- Task 5: 5 points

Submit your code on the course submission website: `http://cse340.fulton.asu.edu/cse340/`