CSE340 Spring 2015
Project 2: FIRST & FOLLOW

Due: Feb. 18, 2015 11:59 PM

Abstract

Given a context-free grammar G as input, you are asked to write a program that calculates `FIRST` and `FOLLOW` sets for all non-terminals of the grammar and determine if each of the non-terminals in G generate a string of length 1.

1 Introduction

In this assignment, you will write a program that reads a context-free grammar from the standard input and based on the command line argument passed to the program, compute one of the following:

1. For all non-terminals of the input grammar, determine if the non-terminal can generate a string of length 1. For example, in the following grammar:

   $$\begin{aligned} \text{S} &\rightarrow \text{A S} \mid \text{S B} \\ \text{A} &\rightarrow \text{a} \mid \text{A a} \\ \text{B} &\rightarrow \text{A} \mid \text{b b} \end{aligned}$$

   where capital letters S, A, B are non-terminals and small letters a, b are terminals, S does not generate any string, A can generate a string of length 1 ($\text{A} \Rightarrow a$) and B can generate a string of length 1 ($\text{B} \Rightarrow A \Rightarrow a$).

2. Compute `FIRST` sets for all non-terminals of the input grammar.

3. Compute `FOLLOW` sets for all non-terminals of the input grammar. For `FOLLOW` sets you can assume that the grammar does not contain rules of the form $A \rightarrow \epsilon$.

Your program should read the input grammar from standard input (stdin), the input grammar format is described in section 3. Additionally, a task number is passed to your program as a command line argument. It specifies what to do with the input grammar. You need to read the value of the argument in your `main()` function and decide what to do with the input grammar. The tasks are numbered as follows:

1) Determine for each non-terminal of the grammar if it generates a string of length 1

2) Compute `FIRST` sets for all non-terminals of the grammar

3) Compute `FOLLOW` sets for all non-terminals of the grammar

2 Reading Command-line Arguments

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

```
#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 argv[0] is the name of your executable
     * e.g. a.out, and the first argument to your program
     * is stored in argv[1]
     */
    task = atoi(argv[1]);

    switch (task) {
        case 1:
            // Call the function(s) responsible for task 1 here
            break;
        case 2:
            // Call the function(s) responsible for task 2 here
            break;
        case 3:
            // Call the function(s) responsible for task 3 here
            break;
        default:
            printf("Error: unrecognized task number %d\n", task);
            break;
    }
    return 0;
}
```

3 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 defined as follows:

grammar_description → *non_terminal_list* *rule_list* `DOUBLEHASH`
non_terminal_list → *id_list* `HASH`
id_list → `ID` *id_list*
id_list → `ID`
rule_list → *rule rule_list*
rule_list → *rule*
rule → `ID ARROW` *righthand_side* `HASH`
righthand_side → *id_list*
righthand_side → ϵ

The tokens used in the grammar description are:

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

Where

```
digit   = 0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9
letter  = a + b + ... + z + A + B + ... + Z
```

In this project, we assume that there is at least one whitespace character (space, tab or newline) between any two tokens. This can make reading the input easier. Also, tokens are case-sensitive.

First Section: 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. Subsequent sections of the input each represent a grammar rule.

Grammar Rules: A grammar rule starts with a non-terminal symbol (left-hand side of the rule) followed by `->`, 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 of a rule is empty, this represents a rule of the form $A \rightarrow \epsilon$.

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:

$$\text{NonTerminals} = \{\ \texttt{decl},\ \texttt{idList1},\ \texttt{idList}\ \}$$

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

$$\begin{aligned} \text{decl} &\rightarrow \text{idList}\ \ \text{COLON}\ \ \text{ID} \\ \text{idList} &\rightarrow \text{ID}\ \ \text{idList1} \\ \text{idList1} &\rightarrow \epsilon \\ \text{iLlist1} &\rightarrow \text{COMMA}\ \ \text{ID}\ \ \text{idList1} \end{aligned}$$

The list of terminals of the grammar consist of all ID tokens that appear in the grammar rules and that are not non-terminals. In the example:

$$\text{Terminals} = \{\ \texttt{COLON},\ \texttt{ID},\ \texttt{COMMA}\ \}$$

Note: 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. The formal specification given before the example defines the format of the input.

4 String Generation Test

Given a non-terminal A, we say that A can generate a string of length one, if $A \stackrel{*}{\Rightarrow} a$, where a is any terminal symbol of the grammar.

Non-terminal A can generate a string of length 1 if any of the following holds:

- $A \rightarrow a$ where a is a terminal
- $A \rightarrow B$ where B is a non-terminal and B can generate a string of length one
- $A \rightarrow A_1\ A_2\ ...\ A_n$ and there exists an i such that A_i is either a terminal or it can generate a string of length one and all other symbols in the right-hand-side can generate ϵ i.e. $\forall j \in \{1, 2, ..., n\} \setminus \{i\},\ A_j \stackrel{*}{\Rightarrow} \epsilon$

Non-terminal A can generate ϵ if any of the following holds:

- $A \rightarrow \epsilon$
- $A \rightarrow A_1\ A_2\ ...\ A_n$ and all the right-hand-side symbols can generate ϵ

To figure out which non-terminals in the grammar can generate a string of length 1, we need to start by figuring out which non-terminals can generate ϵ. One way to determine if a non-terminal generates ϵ is to calculate the `FIRST` sets: $A \stackrel{*}{\Rightarrow} \epsilon$ if and only if $\epsilon \in$ `FIRST(A)`. Here is an alternative method that does not depend on calculating FIRST sets:

```
01      int num_symbols;
02      int num_non_terminals;
03      int num_terminals;
04      int num_rules;
05
06      char *symbols[MAX_SYMBOLS];
07      // in C++ you should use string vectors to store the symbols.
08      // Using MAX_SYMBOLS is convenient but can lead to programming
09      // errors and is not efficient memory-wise. In the code below,
10      // I assume that the symbols array has as first element a
11      // representation of epsilon followed by a representation of EOF,
12      // followed by all non-terminals and finally followed by all
13      // terminals sorted according to dictionary order.
14
15      struct rule {
16          int rhs_length;
17          int LHS;                // is the index of the non-terminal LHS in the symbols array
18          int RHS[MAX_RHS_SIZE];  // RHS is an array of indices of RHS symbols
19      }
20
21      gen_epsilon[0] = true;      // the first symbol is epsilon and it has index 0
22      for (i = 1; i < num_symbols; i++)
23          gen_epsilon[i] = false;
24
25      // Find out which non-terminals can generate epsilon
26      changed = true;
27      while (changed)
28      {
29          changed = false;  // if we change something, we will set changed to true
30          for (i = 0; i < num_rules; i++)
31          {
32              if ( gen_epsilon[rule[i].LHS] )
33                  continue;
34              else if ( rule[i].rhs_length == 0 ) // A -> epsilon
35              {
36                  gen_epsilon[rule[i].LHS] = true;
37                  changed = true;
38              }
39              else   // A -> A1 A2 ... An
40              {
41                  some_does_not_gen_epsilon = false;
42                  for (j = 0; j < rule[i].rhs_length; j++)
43                      some_does_not_gen_epsilon |= !gen_epsilon[rule[i].RHS[j]);
44
45                  if (!some_does_not_gen_epsilon)
46                  {
47                      gen_epsilon[rule[i].LHS] = true;
48                      changed = true;
49                  }
50              }
51          }
52      }
```

A similar approach can be used to determine which non-terminals generate strings of length 1. It is important in your work to have a representation of the symbols and the rules that can support all parts of the project. The representation I have here is one possible representation if you are programming in C.

5 Requirements

Your program should read the input grammar from standard input and print the requested output to standard output. You should not open any files for reading or writing in your code. The test case files are fed to your program with standard I/O redirection as explained in the document `cse340_S15_programming_projects.pdf`.

For each task (specified with the command line argument passed to your program) you need to output the result of your computations in a very specific manner described in this section. Your program should only generate the requested information (no debugging messages, no extra information) in the exact format specified here.

5.1 Task 1: String Generation Test

For each of the non-terminals of the input grammar, in the order they appear in the non-terminals section of the input, determine if the non-terminal generates a string of length 1 consisting solely of terminals and output one line in the following format:

```
<symbol>: <result>
```

Where `<symbol>` should be replaced by the non-terminal and `<result>` should be either `YES` or `NO`. For example, for our sample grammar in section 3, we will have the following output:

```
decl: NO
idList1: NO
idList: YES
```

5.2 Task 2: `FIRST` Sets

For each of the non-terminals of the input grammar, in the order they appear in the non-terminals section of the input, compute `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 (represented by `#` in your output) belongs to the set, it should be listed before any other elements
- All other elements of the set should be sorted in dictionary order

For our sample grammar from section 3, the output would be:

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

5.3 Task 3: FOLLOW Sets

For each of the non-terminals of the input grammar, in the order they appear in the non-terminals section of the input, compute 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* (represented by $ in your output) belongs to the set, it should be listed before any other elements
- All other elements of the set should be sorted in dictionary order

For our sample grammar from section 3, the output would be:

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

6 Implementation

It is very important that you plan your implementation before you start coding. Read the specifications a couple of times to make sure you understand what is required, then come up with a design for your solution. At a high-level, your program will do the following:

1. Read in the set of non-terminals and store them preserving their order.
2. Read grammar rules and store them in appropriate data structures. While reading grammar rules, store new symbols (those that are not found in non-terminals) as terminals in the symbol table.
3. Read the command line argument to determine which computation is requested
4. Based on the command line argument, do one of the following:
   - Determine for each of the non-terminals if they can generate a string of length 1 and output the results as specified in section 5.1.
   - Calculate FIRST sets for all non-terminals of the grammar and output the sets as specified in section 5.2.
   - Calculate FIRST sets for all non-terminals of the grammar, then calculate FOLLOW sets for all non-terminals of the grammar. Then output only the FOLLOW sets as specified in section 5.3. You need to compute FIRST sets because they are needed for calculating the FOLLOW sets.

To calculate `FIRST` and `FOLLOW` sets you need to do a lot of set operations like intersection and union. So you need a data structure to store sets and some functions to operate on set data structures. A simple and efficient way of implementing a set data structure is to use fixed-length binary arrays. For example, let's say we want to represent the `FIRST` set of some symbol for the example grammar given in section 3. A `FIRST` set can contain any one of the terminals of the grammar and/or ϵ. So we consider a *reference set* that includes all terminals plus ϵ:

$$\text{U} = \{\ \epsilon,\ \texttt{COLON},\ \texttt{COMMA},\ \texttt{ID}\}$$

Now if we want to store `FIRST(idList)`, we can use the following binary array of length 4:

$$\texttt{FIRST(idList)} = \{0, 0, 0, 1\}$$

This binary array specifies which elements of the reference set are present in a subset of that reference set. In this example, the only element present in `FIRST(idList)` is `ID`. This data structure is very easy to use and convenient for implementing set operations. For example if you want to implement set union operation, you can simply calculate the logical OR of the elements of the operands and if you want the intersection of two sets, you can simply use AND. Here is an example:

$$\begin{aligned} A &= \{0, 1, 0, 1\} \\ B &= \{0, 0, 1, 1\} \\ A \cup B &= \{0, 1, 1, 1\} \end{aligned}$$

Be careful when using this technique, the operands should be subsets of *the same* reference set, otherwise the result is not meaningful. You will need a reference set for this project that contains all terminals plus ϵ and `eof` symbols (to represent both `FIRST` and `FOLLOW` sets). Note that the order of elements in the reference set is important and should not change during the execution. To conform to the order requirements given in section 5.2 and 5.3, you can first add `#` and `$` to your reference set and then add all terminals in alphabetic order.

To run your program with a command line argument and redirect standard input to a test file use the following command:

```
$ ./a.out  1  < test01.txt
```

This assumes that your executable is named `a.out` and you want to perform task 1 and the test file `test01.txt` is located in the current working directory. Adjust the values to your needs.

7 Grading

1. Correctly doing the string generation test: 30 points
2. Correctly calculating `FIRST` sets:
   (a) `FIRST` sets for grammars without ϵ: 25 points
   (b) `FIRST` sets for grammars with ϵ: 20 points
3. Correctly calculating `FOLLOW` sets:
   (a) `FOLLOW` sets for grammars without ϵ: 25 points
   (b) **(Bonus)** `FOLLOW` sets for grammars with ϵ: 10 points

The distribution of points is not necessarily proportional to the difficulty.

Your program will be executed 3 times for each test case with command line arguments 1, 2 and 3. In each execution, it should generate the appropriate output based on the command line argument. The generated output will be compared to the expected output for each task and it should match that expected output exactly, in order to get credit for the test case. A modified version of the test script `test1_p2.sh` is provided on Blackboard and it can be used to test your code. If you have not used the test script in Project 1, you should read about test scripts in the document `cse340_S15_programming_projects.pdf`.

To pass a test case, the output of the program should **match** the expected output. It is not enough that the right information be in the output that your program generates. The format and order are essential.

If you are unsure about the requirements, it is your responsibility to ask for clarification.

8 Submission

1. Only C or C++ can be used for this project
2. You should submit all your code on the course website by 11:59:59 pm on due date.
3. Make sure your submission has no compile problems. If after submission you get a compiler error on the website, fix the problem and submit again. A submission that does not compile will not be given any credit
4. You can submit any number of times you need, but remember that we only grade your last submission
5. Don't include test scripts or test cases with your submission
6. Don't use any whitespace characters in your file names