CSE340 Spring 2020 Project 2

程序代写代做 CS编程辅导

Due: **March 8. 2020** by 11:59pm MST on GradeScope

1 Note

You should read the description c_____gs are recommended. You will not be able to understand everything on a first reading. Give yourself time_____aking breaks between readings. You will digest the requirements better.

- The answers to many of you_____nd in the description.

- Do no start coding until you have a complete understanding of the requirements.

- Ask for help early.

WeChat: cstutorcs

- Do not forget the instructions given in project 1 and that apply to all programming assignments.

- Have fun!

2 Overview

Assignment Project Exam Help

In this project, you are asked to write a 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:

Email: tutorcs@163.com

1. print the list of non-terminals followed by the list of terminals in the order in which they appear in the grammar rules,

2. find *useless* symbols in the grammar and remove rules with useless symbols,

3. calculate `FIRST` sets,

QQ: 749389476

4. calculate `FOLLOW` sets , or

5. determine if the grammar has a predictive parser.

https://tutorcs.com

We provide you with code to read the command line argument into a integer variable. Depending on the value of the variable, your program will invoke the appropriate functionality.

3 Input Format

The following context-free grammar specifies the input format:

Rule-list	$\rightarrow$	Rule Rule-list | Rule
Id-list	$\rightarrow$	`ID` Id-list | `ID`
Rule	$\rightarrow$	`ID` `ARROW` Right-hand-side `HASH`
Right-hand-side	$\rightarrow$	Id-list | ϵ

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 [...] and there is at least one whitespace character between any two successive tokens. We provide a lexer with a [...] recognize these tokens. You should use the provided lexer in you solution.

4 Semantics

Each grammar `Rule` starts with [...] (the left-hand side of the rule) followed by `ARROW`, then followed by a sequence of zero or more terminals [...] ch represent the right-hand side of the rule. If the sequence of terminals and non-terminals in the right-hand side is empty, then the right hand side represents ϵ.

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. The start symbol of the grammar is the left hand side of the first rule of the grammar.

Note that the convention of using upper-case letters for non-terminals and lower-case letters for terminals that I typically followed in class does not apply in this project.

4.1 Example

Here is an example input:

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

The list of non-terminal symbols in the order in which they appear in the grammar is:

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

The list of terminal symbols in the order in which they appear in the grammar is:

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

The grammar that this input represents is the following:

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

Note that even though the example above 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 following input describes the same grammar as the above example:

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

5   Output Specifications

Your program should read the input grammar from standard input and read the requested task number from the first command line argument (as stated earlier, we provide code to read the task number). Then, your program should 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.

Your parser should properly parse the input and should output SYNTAX ERROR !!! if the input has a syntax error and not output SYNTAX ERROR !!! if the input does not have a syntax error. There are no separate test cases given for syntax checking, but there will be a deduction of 15% if your parser does not correctly parse the input.

5.1   Task 1

Task one simply outputs the list of terminals *in the order in which they appear in the grammar rules* 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 idL
```

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. To be clear, here is the grammar again with the order of each symbol added between parentheses after the first appearance of the symbol.

```
decl (1)    -> idList (2) colon (3) ID (4) #
idList1 (5) -> #
idList1     -> COMMA (6) ID idList1 #
idList      -> ID idList1 #
##
```

5.2 Task 2: Eliminating Useless Symbols

5.2.1 Useless Symbols: Definition

The following is the definition for useless and useful symbols:

Definition: Symbol A is *useful* if there is a derivation starting from S of a string of w of terminals, possibly empty ($w \in T^*$), in which A appears:

$$S \Rightarrow \cdots \Rightarrow xAy \Rightarrow \cdots \Rightarrow w$$

A symbol is *useless* if it is not useful.

The algorithm for determining useless symbols is given in a separate presentation provided with the project documents.

5.2.2 Task Requirements

Determine *useless* symbols in the grammar and remove rules that contain useless symbols. Then output each of the remaining rules 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 the form $A \rightarrow \epsilon$, use `#` to represent the epsilon. Note that this is different from the input format.

The grammar rules should be printed in the same relative order that they originally had. So, if Rule1 and Rule2 are not removed after the elimination of useless symbols, and Rule1 appears before Rule2 in the original grammar, then Rule1 should be printed before Rule2 in the output.

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 idLi
```

Note that none of the symbols of this grammar are 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. Also note that the relative order of the rules is preserved.

5.3 Task 3: Calculate `FIRST` Sets

Compute the `FIRST` sets of all non-terminals. Then, for each of the non-terminals of the input grammar, in the order that it appears in the grammar, output one line in the following format:

```
FIRST(<symbol>) = { <set_items> }
```

where `<symbol>` should be replaced by the non-terminal name and `<set_items>` should be replaced by a comma-separated list of elements of the set ordered in the following manner.

- If ϵ belongs to the set, represent it as `#`.

- If ϵ belongs to the set, it should be listed before any other element of the set.

- All other elements of the set should be sorted in the order in which they appear in the grammar.

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

5.4 Task 4: Calculate FOLLOW Sets

For each of the non-terminals of the input grammar, in the order that they appear in the grammar, 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 to the set, represent it as $.

- If EOF belongs to the set, it should be listed before any other element of the set.

- All other elements of the set should be sorted in the order in which they appear in the grammar.

Example: Given the input grammar:

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

the expected output for task 4 is:

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

5.5 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. If the grammar has useless symbols, your program should output NO.

Example: Given the grammar:

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

the expected output of Task 5 is:

```
YES
```

Note You will not get credit for this task if you output NO for all input or YES for all input. You should get at least 70% of the test cases correct to get credit on this task.

6 Implementation

6.1 Lexer

A lexer that can recognize ID, ARROW, HASH and DOUBLEHASH tokens is provided for this project. You are required to use it.

6.2 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.

```c
/* 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;
}
```

7 Testing

You are provided with a script to run your program on all tasks for each of the test cases. The test cases that we provided for this project are not extensive. **They are meant to serve as example cases and are not meant to test all functionality**. The test cases on the submission ~~~~~~~~~~~ **You are expected to develop your own additional test cases based on the project specifica~~~~~**

To run your program for this p~~~~~~~~~~~~ify the task number through command line arguments. For example, to run task 3:

```
$ ./a.out 3
```

Your program should read the input grammar from standard input. To read the input grammar from a text file, you can redirect standard input:

```
$ ./a.out 3 < test.txt
```

For this project we use 5 expected files per each test case input. For an input file named test.txt , the expected files are test.txt.expected1, test.txt.expected2, test.txt.expected3, test.txt.expected4 and test.txt.expected5 corresponding to tasks 1 through 5. The test script test_p2.sh , provided with the project material, takes one command line argument indicating the task number to use. So for example to test your program against all test cases for task 2, use the following command:

```
$ ./test_p2.sh 2
```

To test your program against all test cases for all tasks, you need to run the test script 5 times:

```
$ ./test_p2.sh 1
$ ./test_p2.sh 2
$ ./test_p2.sh 3
$ ./test_p2.sh 4
$ ./test_p2.sh 5
```

8 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

As mentioned above, if your program does not correctly parse its input, there will be a 15% deduction of the grade.

9 Submission

Submit your code on GradeScope. _____ dents (maybe 4 or so) submitted .zip files that were not accepted by the autograder. The reason is that the _____ by compressing the directory and not compressing the project files. **To avoid any issues, do not subm_____**

The gradescope submission w_____ ate category for syntax checking. There are no provided test cases for that category.

Important Node. There is a _____ e when testing submissions. Programs that are functionally correct but that take an inordinate amount o_____ t before finishing execution. This is typically not an issue because the timeout period is generous, but if _____ **n is very inefficient, it risks being timed out and you will not get credit for test cases for which the program is timed out**.