CSE 340 SPRING 2025 Project 2

Due: **Thursday March 27 2025** by 11:59pm MST on GradeScope

You may delay, but time will not.– Benjamin Franklin

If debugging is the process of removing software bugs, then programming must be the process of putting them in.– Edsger Dijkstra

If I had six hours to chop down a tree, I would spend the first four hours sharpening the axe.– Abraham Lincoln

I had a running compiler and nobody would touch it. They told me computers could only do arithmetic. – Grace Murray Hopper (Inventor of the first compiler)

1 General Advice

You should read this specification document carefully. Multiple readings are recommended. Give yourself time by starting early and taking breaks between readings. You will digest the requirements better.

- The answers to many of your questions can be found in this document.
- Do not start coding until you have a complete understanding of the requirements. At the very least, do not start coding a task, until you have a complete understanding of the task's requirement.
- Ask for help early. I and the TAs can save you a lot of time if you ask for help early. You can get help with how to approach the project to make the solution easier and have an easier time implementing it. That said, when you ask for help, you should be prepared and you should have done your part. The lab and office hours schedule is posted on Canvas!
- Have fun!

2 Overview

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 (see Section 6.3 for more details on how to run the program with command line arguments):

1. print the list of terminals followed by the list of non-terminals in the order in which they appear in the grammar,
2. calculate nullable non-terminals,
3. calculate `FIRST` sets,
4. calculate `FOLLOW` sets ,
5. left-factor the grammar,
6. eliminate left recursion.

All of these tasks are defined in detail in this specification document. We provide you with code to read the command line argument into an integer variable. Depending on the value of the variable, your program will invoke the appropriate functionality. The rest of the document is organized as follows:

1. Section 3 describes the input format (this is just syntax with no meaning).
2. Section 4 describes what the input represents (this is the semantics or meaning of the input).
3. Section 5 describes what the output of your program should be for each task. This is the largest section of the document.
4. Section 6 discusses command line arguments and how you should run and test your program.
5. Section 7 describes the grading scheme.
6. Section 8 addresses some potential submission concerns.

Important Note. For this project, there is a timeout that we enforce when testing submissions.

- Programs that are functionally correct but take an inordinate amount of time can be timed out before finishing execution.
- Write a recursive descent parser for the grammar, but **DO NOT IMPLEMENT YOUR CALCULATIONS RECURSIVELY** . If you try to invent a new recursive algorithm for calculating FIRST and FOLLOW sets, for example, it risks being timed out, and you will not get credit for test cases for which the program is timed out.
- If you follow the algorithms covered in class for *Nullable*, FIRST and FOLLOW and the algorithms that I cover here for left-factoring and elimination of left recursion, you should have no problem with timeout even if your implementation is not particularly efficient.

3 Input Format

The following context-free grammar specifies the input format:

Grammar	→	Rule-list `HASH`
Rule-list	→	Rule Rule-list | Rule
Id-list	→	`ID` Id-list | ϵ
Rule	→	`ID` `ARROW` Right-hand-side `STAR`
Right-hand-side	→	Id-list | Id-list `OR` Right-hand-side

The input consists of a rule list. Each rule has a left-hand side which is an `ID` and a right-hand side which is one or more Id-list's separated with `OR`'s and terminated with the `STAR` token. An Id-list is a list of zero or more `ID`'sThe meaning of the input is explained in the *Semantics* section below.
The tokens used in the above grammar description are defined by the following regular expressions:

```
ID          = letter (letter + digit)*
STAR        = '*'
HASH        = #
OR          = |
ARROW       = ->
```

`digit` is the set of digits from '0' through '9' and `letter` is the upper and lower case letters 'a' through 'z' and 'A' through 'Z'. Tokens are space separated and there is at least one whitespace character between any two successive tokens. We provide a lexer with a getToken() function to recognize these tokens. You should use the provided lexer in you solution. You should not modify the provided lexer.

4 Semantics

The input represents a context-free grammar. The ID tokens represent the terminal and non-terminals of the grammar. The lexemes of these ID tokens are the *names* of the terminals and non-terminals of the grammar. Each grammar `Rule` starts with a non-terminal symbol (the left-hand side of the rule) followed by `ARROW`, followed by a right-hand side which is one or more Id-list's separated with `OR`'s. and terminated with the `STAR` token. If an Id-list that appears on the right-hand of a rule is empty, then that Id-list represents ϵ.

The set of non-terminals for the grammar is the set of names that appear to the left of an arrow. Names that do not appear to the left of an arrow are terminal symbols. The start symbol of the grammar is the name of 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 -> COMMA ID idList1 |  *
#
```

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

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

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

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

The grammar that this input represents is the following:

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

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. Your should not confuse ID which is the name of a terminal of the input grammar in this example with ID which is a token The following input describes the same grammar as the above example:

```
decl -> idList colon ID * idList -> ID idList1 *
idList1 ->

 COMMA ID idList1 |  *    #
```

5 Output Specifications: Tasks 1 – 5

Parsing: There is no separate task for parsing the input. Your parser should properly parse the input and should output:

```
SYNTAX ERROR !!!!!!!!!!!!!!
```

if the input has a syntax error, and it should not output:

```
SYNTAX ERROR !!!!!!!!!!!!!!
```

if the input does not have a syntax error. There will be a deduction of 15% if your parser does not parse the input correctly.

Your program should read the input grammar from standard input (which is done by the provided lexer code), 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 should print the results in the specified format for given task to standard output (stdout). The following specifies the exact requirements for each task number.

5.1 Task 1: Printing Terminals and Non-terminals

Task 1 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 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. 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: Calculate the set of nullable non-terminals

Calculate the set of nullable non-terminals, then output the set in the following format

```
Nullable = { <nullable_non_terminals> }
```

where `<nullable_non_terminals>` should be replaced by a comma-separated list of nullable non-terminals. The list should be ordered according to the order in which non-terminals appear in the input grammar.

Example: Given the input grammar:

```
A->  B F C D E F *
C -> E F *
E -> a E  *
B -> a E *
E -> a
F ->  *
D -> *
C -> D F *
#
```

the expected output for task 2 is:

```
Nullable = { F, C, D }
```

5.3 Task 3: Calculate `FIRST` Sets

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

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

where `<symbol>` should be replaced by the non-terminal name and `<set_elements>` should be replaced by a comma-separated list of elements of the set. The elements of the set should be ordered according to 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 2 is:

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

5.4 Task 4: Calculate FOLLOW Sets

Compute the FOLLOW sets for all the non-terminal symbols. Then, for each of the non-terminals of the input grammar, in the order in which it appears in the grammar, output one line in the following format:

```
FOLLOW(<symbol>) = { <set_elements> }
```

where <symbol> should be replaced by the non-terminal and <set_elements> 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 listed 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:

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

5.5 Task 5: Left Factoring

For this task, your program takes as input a grammar G and outputs a new grammar G' that is obtained by left-factoring G. Below, I give a motivation for left factoring, then I present an algorithm for left factoring which is followed by a step by step example and finally, I specify the output format for this task.

5.5.1 Left factoring introduction

The simple expression grammar that we have seen in class does not technically have a predictive parser. For instance the FIRST sets of `T + E` and `T`, which are the right-hand sides of the rules for `E`, are not disjoint.

To write the parser, we looked at the common part (prefix) of the two right hand sides, the `T`, and started with `parse_T()`. Then, we either stopped or we continued parsing `+ E`. What we have done is essentially left factoring. We have two rules

```
E -> T + E
E -> T
```

and we treated them as `E` followed by `+E` or ϵ:

```
E -> T (  + E | epsilon )
```

By *factoring* the `E` out, we have implicitly transformed our input grammar into an equivalent grammar that has a predictive parser. In general, the transformation can be done explicitly and the explicit transformation is called *left factoring*. For the example above, the resulting grammar would be:

```
E -> T E1
E1 -> + E | epsilon
```

In general, we can do left factoring whenever two rules for the same non-terminal have right-hand sides with a common non-trivial (non-empty) prefix. The general algorithm for left factoring is given in the next subsection.

It is important to note that, in general, left-factoring by itself is not sufficient to obtain a grammar that has a predictive recursive descent parser but we need to do left factoring if we hope of obtain an equivalent grammar that might have a predictive recursive descent parser.

5.5.2 Left factoring algorithm

I give the general algorithm for left factoring on the next page, followed by a detailed example. Make an effort to read the algorithm carefully at least twice, once before going through the example and one after going through the example.

In the algorithm, we chose α to be the longest common prefix of two or more rules for A. If there is more than one common prefix that is longest, then, to ensure a unique output of the algorithm, we start with the longest prefix that is first in dictionary order (also, see the discussion below about the output).

Implementation note 1: The algorithm modifies the set of rules R while processing them. If you are implementing the set of rules as a vector that you iterate over, you should not delete items from the vector while you are iterating over it. This will lead to undefined behavior. You should keep that in mind in your implementation.

Implementation note 2: The algorithm takes as input one grammar G and outputs another grammar G'. Your implementation does not need to explicitly produce a full grammar description as output, just the rules of the resulting grammar in an order specified later in the text.

Implementation note 3: The algorithm assume that the grammar is provided as separate rules. In your implementation, you should treat a rule like `A -> B C | B D` as two rules.

Implementation note 4: The algorithms require the output to be sorted in a specified order. To that end, it assumes that strings are compared using C++ comparison. In C++, "ABC" is smaller than "abc" when compared.

5.5.3 Left Factoring Example

In the following example, the rules are numbered so that I can refer to them in the text. At the start of the algorithm, G' is empty and G contains all the rules.

Algorithm 1: Algorithm for left factoring

Input: Grammar $G = (S, R, T, NT)$
Output: Grammar $G' = (S', R', T', NT')$

Initialization:
    $S' = S$;
    $R' = \{\}$;
    $T' = T$;
    $NT' = \{\}$;
    $G' = (S', R', T', NT')$

repeat
    forall $A \in NT$ **do**
        if *A has two different rules with a non-empty common prefix x: $A \to xy$ and $A \to xy'$* **then**
            Let α be the longest common prefix for any two right-hand sides of rules of A (see tie breaking in text)
            We can divide the rules of A into two groups:

$$\begin{array}{ll} A \to \alpha\beta_1 & A \to \gamma_1 \\ A \to \alpha\beta_2 & A \to \gamma_2 \\ \ldots & \ldots \\ A \to \alpha\beta_k & A \to \gamma_m \end{array}$$

            where none of the right-hand sides $\gamma_1, \gamma_2, \ldots, \gamma_m$ have α as a prefix.

            // Remove rules with prefix α and add new rules to replace the removed rules
            for $i = 1$ **to** k **do**
                remove the rules $A \to \alpha\beta_i$ of A from R
            add the rule $A \to \alpha A_{new}$ to R
            for $j = 1$ **to** m **do**
                add the rule $A_{new} \to \beta_j$ to R'
            add A_{new} to NT'
        else
            // *A has no two rules with a non-empty common prefix*
            remove the rules of A from R and add them to R'
            remove A from NT and add it to NT'
until $NT = \emptyset$
return G'

```
G':

G:   S -> C B C D A B C      // 1
     A -> C B C D            // 2
     A -> C B C B            // 3
     A -> C B D              // 4
     A -> C B B              // 5
     B -> b                  // 6
     C -> c                  // 7
     D -> d                  // 8
```

First, say we pick non-terminal S. Non-terminal S does not have two or more rules with a common non-empty prefix, so we add all rules of S to G' and we get:

```
G':  S -> C B C D A B C

G:   A -> C B C D          // 2
     A -> C B C B          // 3
     A -> C B D            // 4
     A -> C B B            // 5
     B -> b                // 6
     C -> c                // 7
     D -> d                // 8
```

Next, we pick non-terminal A. We identify rules 2 and 3 as the two rules with the longest common prefix, which is $C\ B\ C$. So, we get the following.

```
G':  S -> C B C D A B C
     A1 -> D
     A1 -> B

G:   A -> C B C A1         // 2
     A -> C B D            // 4
     A -> C B B            // 5
     B -> b                // 6
     C -> c                // 7
     D -> d                // 8
```

Note, how the rules for $A1$ are added to G', but the rules for A are not added to G' because they might still have common prefixes with other rules. In the new grammar G, the longest prefix is $C\ B$ which appears in grammar rules 2 , 4 and 5. After the second pass, we obtain:

```
G':     S -> C B C D A B C
        A1 -> D
        A1 -> B
        A2 -> C A1
        A2 -> D
        A2 -> B

G:      A -> C B A2            // 2
        B -> b                 // 6
        C -> c                 // 7
        D -> d                 // 8
```

From this point forward, there will be no two rules for a non-terminal that share a non-empty prefix, so we will end up adding all the remaining rules, one by one, to G' until G is empty. The final grammar that we obtain is:

```
G':     A1 -> D
        A1 -> B
        A2 -> B
        A2 -> C A1
        A2 -> D
        S -> C B C D A B C
        A -> C B A2
```

```
B -> b
C -> c
D -> d
```

This grammar is not sorted, but the actual output of Task 5 should be sorted as explained below.

5.5.4 Task 5 requirements

The algorithm for left factoring as described above is relatively straightforward, but we need to specify the exact format of the output so that the output of your program is uniquely determined by the specified format. There are three sources of possible variation in the output format when the algorithm is followed:

1. The order in which the rules are listed
2. The names given to the new non-terminals that are introduced by the algorithm.
3. If multiple prefixes are longest common prefixes shared by two or more rules, then which one we chose first affects the result because it affects which rules a newly introduced name goes with.

The following output format uniquely defines the expected output:

1. The grammar rules should be printed in lexicographic order (dictionary order) by treating the left-hand side followed by the right-hand side as one sequence. According to this order if we have the two rules `A -> AB C` and `A -> A Z`, then `A -> A Z` is listed first. The lexicographic comparison is done one lexeme at a time. More details are given in the implementation guide.
2. If multiple rules for X are left factored for the first time, then the new introduced name is $X1$
3. If multiple rules for X are left factored and we have already introduced names $X1$, $X2$, ..., Xk, then the new name should be $Xk + 1$. Of course, for this to work, we should assume that these newly introduced names are not already used in the grammar. We will make this assumption and it will be satisfied by all test cases.
4. If in a given iteration, more than one common prefix is longest prefix shared by two or more rules, we chose the longest common prefix that appears first in dictionary order (this situation did not occur in the example above).

The example above already followed the naming convention specified here, so the naming requirement is satisfied. So, we have to sort the grammar lexicographically to obtain a grammar that satisfy the output format requirements. If we do so, we obtain the grammar that should be printed. Note that we only need to print the rules of the resulting grammar. That is why the name G' is omitted from the output. Also, the output format (see below) requires that every rule be terminated with the `#` symbol.

```
A -> C B A2 #
A1 -> B #
A1 -> D #
A2 -> B #
A2 -> C A1 #
A2 -> D #
B -> b #
C -> c #
D -> d #
S -> C B C D A B C #
```

To summarize, the following is what you should do for Task 5:

1. Apply the left factoring algorithm to obtain G'. Make sure to follow the naming convention when new non-terminals are introduced.

2. Sort the resulting grammar lexicographically.

3. Print the rules of the resulting grammar by first printing the left-hand side, followed by space, followed by `->`, followed by space, then followed by the symbols on the right-hand side of the rule separated by space(s) and finally followed by `#`. If there are no symbols on the right-hand side of a particular rule of the resulting sorted grammar, then you simply print the `#` after the `->`.

5.6 Task 6: Eliminating Left Recursion

For this task, your program reads the grammar, which is assumed to have no epsilon rules and to be cycle-free, and prints the resulting grammar after eliminating left recursion as explained below.

5.6.1 Introduction to eliminating left recursion

We say that a rule has immediate left recursion if it is of the form $A \rightarrow A\alpha$ for some α. It is called *left recursion* because the A appears on the left-hand side of the rule and also appears as the leftmost symbol of the right-hand side of the rule. If this is the only rule for A and we try to write a recursive descent parser, we end up with the following which will create an infinite loop.

```
parse_A()
{
        parse_A()
        // parse alpha
}
```

It is clear that if a grammar has rules with left recursion, the grammar cannot have a predictive parser. Left recursion is not restricted to immediate left recursion where the same non-terminal appears on the left side of a rule and as the leftmost symbol of the left-hand side of the rule. Left recursion can be indirect as in the following example:

$$
\begin{array}{lcl}
A & \rightarrow & B\,a \mid b \\
B & \rightarrow & A\,b \mid a
\end{array}
$$

Here, the grammar is left-recursive because $A \Rightarrow B\,a \Rightarrow A\,b\,a$. Also, $B \Rightarrow A\,b \Rightarrow B\,a\,b$. Also, we don't have a predictive parser because $FIRST(B\,a) \cap FIRST(b) \neq \emptyset$. Also, $FIRST(A\,b) \cap FIRST(a) \neq \emptyset$.

In general, we say that a grammar is *left-recursive* if it has a non-terminal A such that $A \stackrel{+}{\Rightarrow} A\,\alpha$ for some string α of terminals and non-terminals (the + above the arrow stands for one or more as opposed to the Kleene star which stands for zero or more). If a grammar is left-recursive, not only it cannot have a predictive top-down parser, but it cannot be handled by a top-down parser! So, if we want to handle a grammar with left recursion in a top-down fashion, we need to transform it to another equivalent grammar that does not have left recursion.

Before giving the general algorithm, we consider the case of immediate left recursion. If A has immediate left recursion, then the rules for A can be divided into two groups:

$$
\begin{array}{lcl}
A & \rightarrow & A\,\alpha_1 \mid A\,\alpha_2 \mid \ldots \mid A\,\alpha_k \\
A & \rightarrow & \beta_1 \mid \beta_2 \mid \ldots \mid \beta_m
\end{array}
$$

where none-of the β_j, $j = 1, \ldots, m$, starts with A. We can rewrite the grammar into the following equivalent grammar.

$$A \rightarrow \beta_1\, A1 \mid \beta_2\, A1 \mid \ldots \mid \beta_m\, A1$$
$$A1 \rightarrow \alpha_1\, A1 \mid \alpha_2\, A1 \mid \ldots \mid \alpha_k\, A1 \mid \epsilon$$

Note that this transformation would work even if all the right-hand sides of the rules of A start with A, in which case A is useless. The transformation works in that case because there are no β_i, so the resulting grammar will have no A rules (rules with A on the left-hand side) and all the resulting rules will be $A1$ rules and $A1$ would also end up being a useless symbol. In the algorithm, we don't concern ourselves with whether or not there are useless symbols in the grammar.

So far, we explained how to eliminate left recursion when we have immediate left recursion (see above). The transformation does not handle the case where we have indirect left recursion. Next we present the general algorithm for eliminating left recursion (direct or indirect).

5.6.2 Algorithm for eliminating left recursion

The algorithm that I will give here for eliminating left recursion assumes that the grammar has no ϵ-rules (rules whose right-hand side is ϵ) and that the grammar has no cyclical derivations, i.e. derivations of the form $A \stackrel{+}{\Rightarrow} A$ (remember that + means one or more). If a grammar has cyclical derivations, we say that the grammar has cycles. For this task, you can assume in your solution that the grammar has no cycles and no ϵ-rules and you don't have to check for that in your solution. I give the general algorithm for eliminating left recursion, followed by a detailed example. Make sure to read the algorithm carefully at least twice, once before reading the example and once after reading the example.

The algorithm starts by initializing the grammar of G' (lines 4–9). The set of non-terminal NT' is initialized to be equal to the set of non-terminals of the given grammar sorted in dictionary order (line 8). For reference in the remainder of the algorithm, the elements of the set NT' are called $A_1, \ldots, A_n$. Then, the algorithm creates a set of *Rules* for each non-terminal (line 11–15). This step groups all the rules for each non-terminal together in a set. Next, each non-terminal A_i is considered in sorted order (line 17) and the algorithm iterates over all non-terminals that appear before A_i in the sorted order (line 18). If any rule r for A_i (line 20) has a right-hand side of the form $A_j\, \gamma$ (line 18), where A_j appears before A_i in the order (note that $j < i$ in line 18), the rule is an *offending* rule (in general, an offending rule is a rule of the form $A \rightarrow B\, \gamma$ such that B appears before A in the sorted order of non-terminals). We get rid of the offending rule by rewriting the rules for A_i. For every offending rule r of A_i of the form $A_i \rightarrow A_j\, \gamma$ (line 18), we remove rule r from $Rules[A_i]$ (line 21) and we add new rules to $Rules[A_i]$. The new rules are obtained by replacing A_j in the offending rule with all possible values of A_j with the right-hand sides of rules for A_j (line 22–24).

This last step might be confusing, so I explain it with an example. Assume we have an offending rule $A \rightarrow B\, \gamma$, where B appears before A in the order of non-terminals. Let $B \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ be the current rules for B (which might be different from the original rules of B in the grammar because B was considered before A and its rules might have been rewritten). We remove $A \rightarrow B\, \gamma$ from the rules of A and we add the rules $B \rightarrow \delta_1\, \gamma \mid \delta_2\, \gamma \mid \ldots \mid \delta_k\, \gamma$ in its place.

After rewriting the rules for A_i, we consider the resulting rules for A_i and eliminate any direct left recursion from them as described earlier. No pseudocode is provided for that.

The resulting grammar G' has the same starting symbol and the same set of terminals as G. The non-terminals of G' are all the original non-terminals plus all the newly added non-terminals when direct left recursion is eliminated. The rules of G' is the union of all the rules of the non-terminals of G'.

5.6.3 Eliminating Left Recursion Example

Consider the grammar

```
G:   S -> D B C D A B C
     A -> D B C D
     B -> A B C B
     C -> B D
     A -> C B B
     B -> b
     C -> c
     D -> d
```

Algorithm 2: Algorithm Eliminating Left Recursion

Input: Grammar $G = (S, R, T, NT)$ with no ϵ-rules and no cycles
Output: Grammar $G' = (S', R', T', NT')$ with the same language as G but with no left recursion

Initialization:
    $S' = S$
    $R' = \{\}$
    $T' = T$
    $NT' = NT$ sorted lexicographically (dictionary order). Say $NT' = \{A_1, A_2, \ldots, A_n\}$
    $G' = (S', R', T', NT')$

forall *non-terminal* $A \in NT$ **do**
    $Rules[A] = \{\}$

// Group rules with the same left-hand side together
forall *rule* $r \in R$ **do**
    $Rules[r.LHS] = Rules[r.LHS] \cup \{r\}$

for $i = 1$ **to** n **do**
    for $j = 1$ **to** $i - 1$ **do**
        forall $r \in Rules[A_i]$ **do**
            if r *has the form* $A_i \to A_j\,\gamma$ **then**
                $Rules[A_i] = Rules[A_i] - \{r\}$
                forall $r' \in Rules[A_j]$ **do**
                    if r' *has the form* $A_j \to \delta$ **then**
                        $Rules[A_i] = Rules[A_i] \cup \{(A_i \to \delta\,\gamma)\}$
    Eliminate immediate left recursion from the rules of A_i (see text)

forall $A \in NT'$ **do**
    forall $r \in Rules[A]$ **do**
        $R' = R' \cup \{r\}$
$G' = (S, R', T, NT')$

First, we sort the non-terminals and we group the rules for each non-terminal together. We get the following grammar.

```
Rules[A] = { A -> D B C D , A -> C B B }
Rules[B] = { B -> A B C B , B -> b }
Rules[C] = { C -> B D  , C -> c }
Rules[D] = { D -> d }
Rules[S] = { S -> D B C D A B C }
```

Now, we consider the non-terminals one by one in the sorted order. We start with A. A has no offending rules, so we don't need to rewrite any of the rules of A. Also, A does not have any direct left recursion, so we are done with A without having changed any rules. The resulting rules are the same as the original rules. I highlight the right-hand sides of the rules of A in red because that will be used in explaining how the rules of B are rewritten.

```
Rules[A] =  A -> D B C D , A -> C B B   // done
Rules[B] =  B -> A B C B , B -> b
```

```
Rules[C] =  C -> B D  , C -> c
Rules[D] =  D -> d
Rules[S] =  S -> D B C D A B C
```

Next, we consider B. B has one offending rule which is $B \rightarrow A\ B\ C\ B$. This rule is offending because A appears before B in the sorted list of non-terminals. We replace the offending rule for B with the rules $B \rightarrow D\ B\ C\ D\ B\ C\ B$ and $B \rightarrow C\ B\ B\ B\ C\ B$ which are obtained by replacing A with all possible right-hand sides of rules of A. After the replacement, the resulting rules for B have no direct left recursion. We obtain the following grammar

```
Rules[A] =  A -> D B C D , A -> C B B                        // done
Rules[B] =  B -> D B C D B C B , B -> C B B B C B,  B -> b   // done
Rules[C] =  C -> B D  , C -> c
Rules[D] =  D -> d
Rules[S] =  S -> D B C D A B C
```

Now, we consider the rules for C. C has one offending rule $C \rightarrow B\ D$. We eliminate the offending rule by replacing B with the *current* right-hand sides of rules for B and we obtain the following

```
Rules[A] = { A -> D B C D , A -> C B B }                                    // done
Rules[B] = { B -> D B C D B C B , B -> C B B B C B,  B -> b }               // done
Rules[C] = { C -> D B C D B C B D , C -> C B B B C B D , C -> b D , C -> c }
Rules[D] = { D -> d }
Rules[S] = { S -> D B C D A B C }
```

The resulting rules for C now have direct left recursion, so we need to eliminate it as required in line 15 of the algorithm. We follow the approach presented in Section 5.5.1. The rules for C can be divided into two parts

```
C -> C B B B C B D.    // direct left recursion
C -> D B C D B C B D   // no direct left recursion
C -> b D              // no direct left recursion
C -> c                // no direct left recursion
```

We rewrite these rules as we explained in Section 5.5.1 by introducing a new non-terminal $C1$ and the rules become

```
C -> D B C D B C B D C1
C -> b D C1
C -> c C1
C1 -> B B B C B D C1
C1 ->
```

The resulting grammar is now

```
Rules[A]  = { A -> D B C D , A -> C B B }                          // done
Rules[B]  = { B -> D B C D B C B , B -> C B B B C B,  B -> b }     // done
Rules[C]  = { C -> D B C D B C B D C1 , C -> b D C1 , C -> c C1 }  // done
Rules[C1] = { C1 -> B B B C B D C1, C1 ->   }                      // done
Rules[D]  = { D -> d }
Rules[S]  = { S -> D B C D A B C }
```

Next, we consider non-terminal D which has no offending rules and no direct left recursion. We get:

```
Rules[A]  = { A -> D B C D , A -> C B B }                              // done
Rules[B]  = { B -> D B C D B C B , B -> C B B B C B,  B -> b }         // done
Rules[C]  = { C -> D B C D B C B D C1 , C -> b D C1 , C -> c C1 }      // done
Rules[C1] = { C1 -> B B B C B D C1, C1 ->   }                         // done
Rules[D]  = { D -> d }                                                // done
Rules[S]  = { S -> D B C D A B C }
```

Finally, we consider non-terminal S which has one offending rule, so we rewrite it by replacing D with all possible right-hand sides of the rules for D (which is one rule only) and we get

```
Rules[A]  = { A -> D B C D , A -> C B B }                              // done
Rules[B]  = { B -> D B C D B C B , B -> C B B B C B,  B -> b }         // done
Rules[C]  = { C -> D B C D B C B D C1 , C -> b D C1 , C -> c C1 }      // done
Rules[C1] = { C1 -> B B B C B D C1, C1 ->   }                         // done
Rules[D]  = { D -> d }                                                // done
Rules[S]  = { S -> d B C D A B C }                                    // done
```

This is the end of the example. Next we state the requirements for Task 6.

5.6.4 Task 6 requirements

As we have discussed for left factoring, we need to ensure that the output of the algorithm for eliminating left recursion is unique. A large part of ensuring uniqueness is already handled by sorting the non-terminals in dictionary order. In order to ensure uniqueness of the final output, we require that the resulting rules be printed in dictionary order as I explained for the left factoring task. Also, when introducing a new name to eliminate direct left recursion for non-terminal X, we call the newly introduced name $X1$ as is explained in the general case and I did in the example. You can assume that none of the original non-terminal names will be the same as the the newly introduced name.

The example above already followed the naming convention specified here, so the naming requirement is satisfied. To obtain the final output, we have to sort the grammar lexicographically to obtain a grammar that satisfy the output format requirements. If we do so, we obtain the grammar that should be printed. Note that you are only asked to print the rules of the resulting grammar. Also, the output format (see below) requires that every rule be terminated with the `#` symbol. The final output for the example above will be the following.

```
A -> C B B #
A -> D B C D #
B -> C B B B C B #
B -> D B C D B C B #
B -> b  #
C -> b D C1 #
C -> c C1 #
C -> D B C D B C B D C1 #
C1 ->  #
C1 -> B B B C B D C1 #
D -> d #
S -> d B C D A B C #
```

To summarize, the following is what you should do for Task 5:

1. Apply the elimination of left recursion algorithm to obtain a new grammar whose rules don't have direct or indirect left recursion.
2. Sort the resulting grammar lexicographically.
3. Print the rules of the resulting grammar by first printing the left-hand side, followed by space, followed by `->`, followed by space, the followed by the symbols on the right-hand side separated by space and finally followed by `#`. If there are no symbols on the right-hand side of a particular rule of the resulting sorted grammar, then you simply print the `#` after the `->`.

6 Implementation

6.1 Lexer

A lexer that can recognize `ID`, `ARROW`, `STAR`, `OR` and `HASH` tokens is provided for this project. You are required to use it and you should not modify it.

6.2 Reading command-line argument

As mentioned in the introduction, your program must read the grammar from `stdin` (standard input) 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 in the
// provided code. 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;
}
```

6.3 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 site will be extensive. **You are expected to develop your own additional test cases based on the project specification**.

To run your program for this project, you need to specify 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 6 times (you can also write a script to do that):

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

7 Evaluation

Your submission will be graded on passing the test cases on Gradescope. 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):

- Parsing: No points if correct. If not correct, there will be a 15% deduction from the grade.
- Task 1: 10 points
- Task 2: 10 points
- Task 3: 20 points
- Task 4: 20 points
- Task 5: 20 points
- Task 6: 20 points

The points are not assigned based on the difficulty. Tasks 5 and 6 are more involved than tasks 3 and 4 but are assigned the same points. As mentioned above, if your program does not correctly parse its input, there will be a 15% deduction from the grade.

8 Submission

Submit your individual code files on GradeScope. **Do not submit .zip files**.

The gradescope submission will be tested on a separate category for syntax checking. There are no provided test cases for that category.

Important Note. For this project, there is a timeout that we enforce when testing submissions. Programs that are functionally correct but that take an inordinate amount of time can be timed out before finishing execution. This is typically not an issue because the timeout period is generous, but if **your implementation is very inefficient, it risks being timed out and you will not get credit for test cases for which the program is timed out**.