CSE340F23 PROJECT 2 Tasks 4 & 5

IMPLEMENTATION GUIDANCE

Rida Bazzi

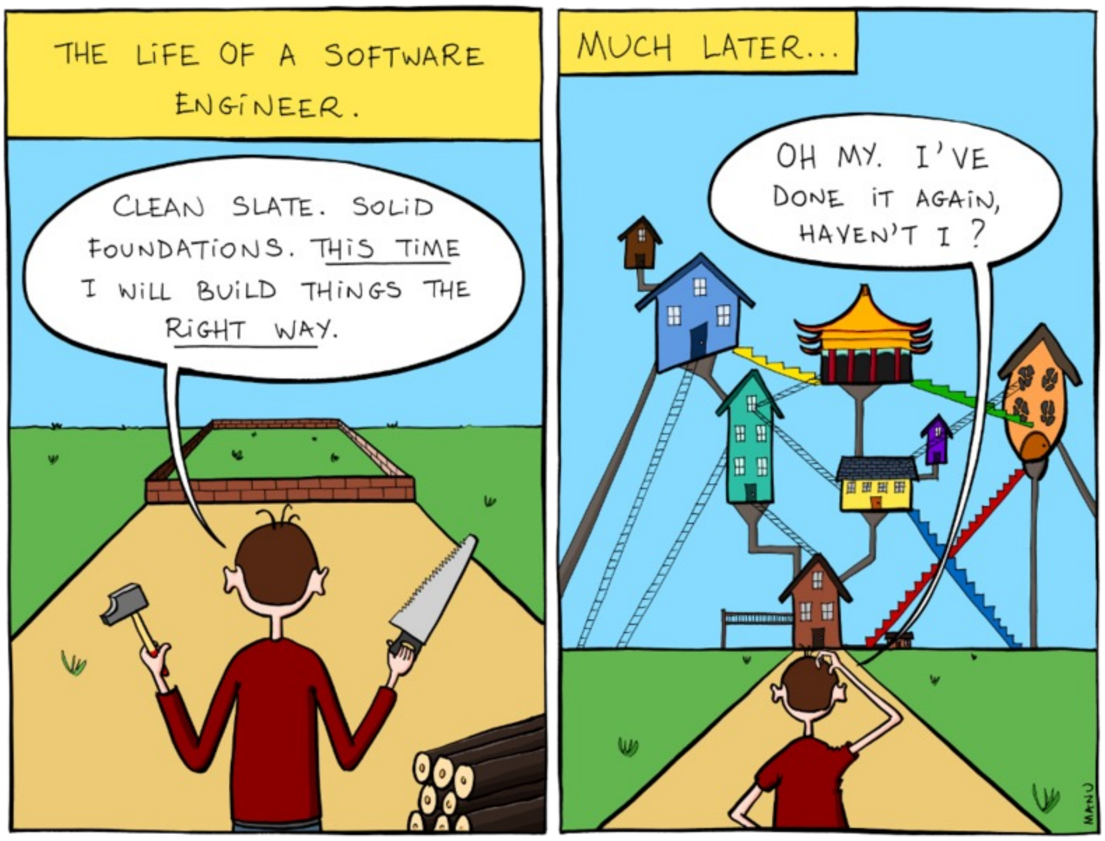
This is not meant to be a detailed implementation guide, but I address major implementation issues that you are likely to encounter.

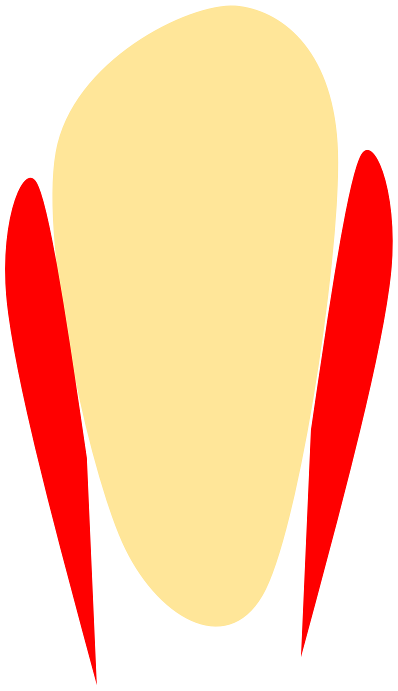
By now you should know that reading everything carefully is essential and can save you a lot of time.

You should have a plan and understand how the various pieces will fit together before you start coding.

Do not delay asking for help.

If your implementation seems to be getting too complicated, especially conceptually, you should step back and simplify. Do not end up like this guy:


THE LIFE OF A SOFTWARE ENGINEER.
CLEAN SLATE. SOLID FOUNDATIONS. THIS TIME I WILL BUILD THINGS THE RIGHT WAY.
MUCH LATER...
OH MY. I'VE DONE IT AGAIN, HAVEN'T I ?
MANU



Lexicographic Comparison

The output for Tasks 5 & 6 needs to be sorted lexicographically (dictionary order). I explain what that means with examples

Example 1

A -> C A rule 1
B -> A B C rule 2

We compare the two sequences $\boxed{A}\boxed{C}\boxed{A}$ and

^

$\boxed{B}\boxed{A}\boxed{B}\boxed{C}$

so, rule 1 appears before rule 2 in dictionary order

Example 2

AB -> A C rule 1
A -> B B C rule 2

We compare the two sequences $\boxed{AB}\boxed{A}\boxed{C}$

>

$\boxed{A}\boxed{B}\boxed{B}\boxed{C}$

So, rule 2 appears before rule 1 in dictionary order

Example 3

A -> B C rule 1
A -> B C C rule 2

We compare the two sequences $\boxed{A}\boxed{B}\boxed{C}$

= = =

$\boxed{A}\boxed{B}\boxed{C}\boxed{C}$

Since all comparisons are equal, the shorter rule appears before the longer rules in dictionary order.

Example 4

A -> B BB A Z Z rule 1
A -> B C C rule 2

We compare the two sequences $\boxed{A}\boxed{B}\boxed{BB}\boxed{A}\boxed{Z}\boxed{Z}$

= = <

$\boxed{A}\boxed{B}\boxed{C}\boxed{C}$

Rule 1 appears before 2 in dictionary order.

It will help if you write a function that takes two rules rule1 and rule2 as parameters and that returns true if rule1 appears before rule2 in dictionary order and returns false otherwise. You don't have to worry about two rules being the same because all initial rules are different and all rules that your program generates inTask 5 and Task 6 will also be different

Finding the length of the common prefix of the righthand sides of the rules of a non-terminal

In order to do left factoring, you need to identify the common prefixes of righthand sides of the rules of a given non-terminal. I will explain how to do that later, but here I emphasize one aspect of the process.

Given two rules for A, we would like to determine if the righthand sides of the two rules have a common prefix and, if they do, we would like to determine the length of the common prefix.

It will help if you write a function that returns the length of the common prefix that two righthand sides have.

This function will be useful in implementing a relatively simple solution for left factoring and that I will explain later in this document. Here are examples of what this function will return for various pairs of rules

Example 1

```
A -> A B C    rule1
A -> B C A    rule2
```

length_common_prefix(rule1, rule2) = 0

Example 2

```
A ->  A   B C rule1
A -> AB C     rule2
```

length_common_prefix(rule1, rule2) = 0 because rule1 starts with A and rule 2 starts with AB

Example 3

```
A ->  A  B C  rule1
B ->  A  B C  rule2
```

length_common_prefix(rule1, rule2) = 0 because the lefthand sides are not the same

Example 4

```
A ->  A  BB  CC             rule1
A ->  A  BB    C  C         rule2
```

length_common_prefix(rule1, rule2) = 2 because the righthand sides only match for the first two symbols. The third symbols are not the same (CC and C)

Finding the lengths of all common refixes

It would be helpful to determine for every grammar rule the length of the longest common prefix that the rule has with any other rule. For example, for the following grammar (the numbers are added to refer to the rules):

G

1. A -> A B C D
2. B -> A B C D
3. A -> A B D E
4. A -> C D E
5. B -> A C D
6. B -> A C E
7. A -> E C E
8. A -> F G E
9. A -> F G F

We obtain

G'

1.	A -> A B C D	longest_match = 2	// with rule 3
2.	B -> A B C D	longest_match = 1	// with rules 5 and 6
3.	A -> A B D E	longest_match = 2	// with rule 1
4.	A -> C D E	longest_match = 0	
5.	B -> A C D	longest_match = 2	// with rule 6
6.	B -> A C E	longest_match = 2	// with rule 5
7.	A -> E C E	longest_match = 0	
8.	A -> F G E	longest_match = 2	// with rule 9
9.	A -> F G F	longest_match = 2	// with rule 8

The longest match for each rule can be calculate by checking the rule with every other rule for longest_matching prefix and choosing the longest amongst them. Notice how this is only calculating the length of the longest match and is not making an attempt to keep track of which rules have the longest match.

At this point, if we sort all the rules first by longest match and then lexicographically, we obtain

G''

1.	A -> A B C D	longest_match = 2
2.	A -> A B D E	longest_match = 2
3.	A -> F G E	longest_match = 2
4.	A -> F G F	longest_match = 2
5.	B -> A C D	longest_match = 2
6.	B -> A C E	longest_match = 2
7.	B -> A B C D	longest_match = 1
8.	A -> C D E	longest_match = 0
9.	A -> E C E	longest_match = 0

Notice how the the two rules that should be processed next are at the top of the list, so breaking ties between the yellow highlighted rules and the blue highlighted rules is relatively straightforward as I describe next.

To determine the rules that need to be processed next, we start with the first rule (new rule 1) and find all rules whose righthand sides match (new rule 1) in the first two symbols (the length of the longest match). This will yield rules 1 and 2 as the answer and these are the rules that need to be left-factored before the other rules (in particular rules 3 and 4) with a longest_match = 2, but that appear after rules 1 and 2 in dictionary order.

It would be helpful to write a function that checks if two rules match up to a certain number of positions.

For example, rule 1 of G'' matches rules 2 of G'' to two positions but does not match rule 2 of G'' to 3 positions.

It would be helpful to write a function that takes a grammar as input and return a new grammar in which rules are sorted by longest_match of the righthand side then by dictionary order

Finding the lengths of all common refixes

It would be helpful to determine for every grammar rule the length of the longest common prefix that the rule has with any other rule. For example, for the following grammar (the numbers are added to refer to the rules):

G

1. A -> A B C D
2. B -> A B C D
3. A -> A B D E
4. A -> C D E
5. B -> A C D
6. B -> A C E
7. A -> E C E
8. A -> F G E
9. A -> F G F

We obtain

G'

```
1.  A -> A B C D        longest_match = 2      // with rule 3
2.  B -> A B C D        longest_match = 1      // with rules 5 and 6
3.  A -> A B D E        longest_match = 2      // with rule 1
4.  A -> C D E          longest_match = 0
5.  B -> A C D          longest_match = 2      // with rule 6
6.  B -> A C E          longest_match = 2      // with rule 5
7.  A -> E C E          longest_match = 0
8.  A -> F G E          longest_match = 2      // with rule 9
9.  A -> F G F          longest_match = 2      // with rule 8
```

The longest match for each rule can be calculate by checking the rule with every other rule for longest_matching prefix and choosing the longest amongst them. Notice how this is only calculating the length of the longest match and is not making an attempt to keep track of which rules have the longest match.

At this point, if we sort all the rules first by longest match and then lexicographically, we obtain

G''

```
1.  A -> A B C D        longest_match = 2
2.  A -> A B D E        longest_match = 2
3.  A -> F G E          longest_match = 2
4.  A -> F G F          longest_match = 2
5.  B -> A C D          longest_match = 2
6.  B -> A C E          longest_match = 2
7.  B -> A B C D        longest_match = 1
8.  A -> C D E          longest_match = 0
9.  A -> E C E          longest_match = 0
```

Notice how the the two rules that should be processed next are at the top of the list, so breaking ties between the yellow highlighted rules and the blue highlighted rules is relatively straightforward as I describe next.

To determine the rules that need to be processed next, we start with the first rule (new rule 1) and find all rules whose righthand sides match (new rule 1) in the first two symbols (the length of the longest match). This will yield rules 1 and 2 as the answer and these are the rules that need to be left-factored before the other rules (in particular rules 3 and 4) with a longest_match = 2, but that appear after rules 1 and 2 in dictionary order.

It would be helpful to write a function that compares checks if two rules match up to a certain number of positions.

For example, rule 1 of G'' matches rules 2 of G'' to two positions but does not match rule 2 of G'' to 3 positions.

It would be helpful to write a function that takes a grammar as input and return a new grammar in which rules are sorted by longest_match of the righthand side then by dictionary order

Left factoring

Now, that we have the tools to identify the rules that need to be left factored (rules 1 and 2 of G'' on page 6), we need to actually do the left factoring.

The process involves:

1. determining a new name for newly introduced non-terminal.
2. Introduce new rules for the newly introduced non-terminal.
3. Introduce one left-factored rule

For the example

1. A -> A B C D	longest_match = 2
2. A -> A B D E	longest_match = 2

We need to introduce a new non-terminal A1 (given that this is the first left-factoring of a rule for A) and the rules

1. A -> A B A1
2. A1 -> C D
3. A1 -> D E

To obtain these rules, we need to construct the left-factored rule for A. This rule would look like

A -> longest_common_match A1

To construct this rule, you need to make a copy of the first longest_match symbols on the RHS then push A1 after that.

For the rules of A1, we need to copy, for each rule of A, the symbols after the longest_common_match and make them the righthand side of a new rule for A1.

I suggest that you write functions to extractPrefix() of a certain length from the RHS. The function will take a vector as input and returns a vector as output.

If we call the function extractPrefix(2) on rules 1 or 2 above, we should get a vector of size 2 who entries contain "A" and "B". That vector can be used to construct the new left factored rule for A.

I suggest that you write a function to extractAllButPrefixOfSize() of a certain length from RHS. This function will be useful to construct the righthand sides of the rules for A1 above.

For example, if we call extractAllButPrefixOfSize(2) on rule 1 above, we get a vector of size 2 containing "C" and D".

If we call if we call extractAllButPrefixOfSize(1) on rule 1 above, we get a vector of size 3 containing "B", "C" and D". Obviously, we don't need to call the function with argument 1, but I am giving the example to emphasize that the argument is the size of the prefix to be excluded and not of the suffix to be retained.

What to do after left factoring a group of rules

After left-factoring a group of rules, we continue left factoring the rules until there are no common prefixes. But how do we do that?

We first note that if we have done the identification of the longest match correctly, the rules for A1 (on the previous page) should have a longest match = 0, for otherwise, we would have a longer longest match for rules of A (think about it).

This means that the rules of A1 cannot be involved in further left-factoring. So, we can go ahead and push them not on the original grammar, but on the new grammar (initially empty) that will be the output of the whole left-factoring task.

The new rule for A might still need to be left-factored with other rules for A, so we need to push it back on the original grammar. At this point we have

New grammar:

1. A1 -> C D
2. A1 -> D E

Modified old grammar:

1.	A -> F G E	longest_match = 2
2.	A -> F G F	longest_match = 2
3.	B -> A C D	longest_match = 2
4.	B -> A C E	longest_match = 2
5.	B -> A B C D	longest_match = 1
6.	A -> C D E	longest_match = 0
7.	A -> E C E	longest_match = 0
8.	A -> A B A1	

So, we need to repeat the whole process again and keep adding to the new grammar until all the longest matches remaining are = 0 at which point we can push all the remaining rules to the new grammar

One last point has to do with recalculating the longest_match for the modified old grammar. We note that the longest match of the existing rules

1. either don't involve the left-factored rules, in which case the longest matches of these rules will note change, or
2. involve the left_factored rule, but here, also, the longest match will not change because the newly added rule for A has the longest common match as a prefix

This means that we only need to recalculate the longest match for the new rule of A and then proceed as before.

Sorting the rules lexicographically

The output format requires that the rules be sorted lexicographically. Sorting rules (for a given non-terminal) is also helpful in breaking ties when determining the longest common prefix for two rules of a non-given terminal as we have seen earlier.

I strongly suggest that you write functions to

- compare two rules lexicographically (explained on page 4)

- compare two rules lexicographically but taking longest_match into consideration:

```
if rule1.longest match  < rule2.longest_match  then
      1.   rule 1 < rule 2
else if rule2.longest match < rule1.longest_match  then
      1.   rule 2 < rule 1
else if rule2.longest match = rule1.longest_match  then
            return the result of lexicographic comparison of ulre1 and rule2
```

Task 6

For Task 6, you need the following functionality

1. Given a non-terminal, determine all the rules for the non-terminals

2. Given a rule A -> B R, where B is a non-terminal and R is a sequence of terminals and non-terminals (a vector) and rules R1, R2, … Rm of B with righthand sides R1, R2, … Rm, created new rules

```
A -> R1 R
A -> R2 R
…
A -> Rm R
```

Given an implementation of rules as a

```
struct rule {
        string LHS;
        vector<string> RHS
};
```

This functionality is relatively easy to implement as follows: for the i'th rule of B with RHS Ri, create a new rule whose LHS = A and whose RHS is obtained by pushing all of Ri followed by pushing all of R on an empty RHS.

3. Given rules for A, divide the rules of A into two groups

those whose righthand sides start with A
those whose righthand sides do not start with A

such a function should take one rule as input and produces two sets as output, maybe as a pair

4. Given two sets obtained as part of functionality number 3, generate two sets of rules by eliminating of immediate left recursion