CSE340S25 PROJECT 2

IMPLEMENTATION GUIDANCE

Rida Bazzi

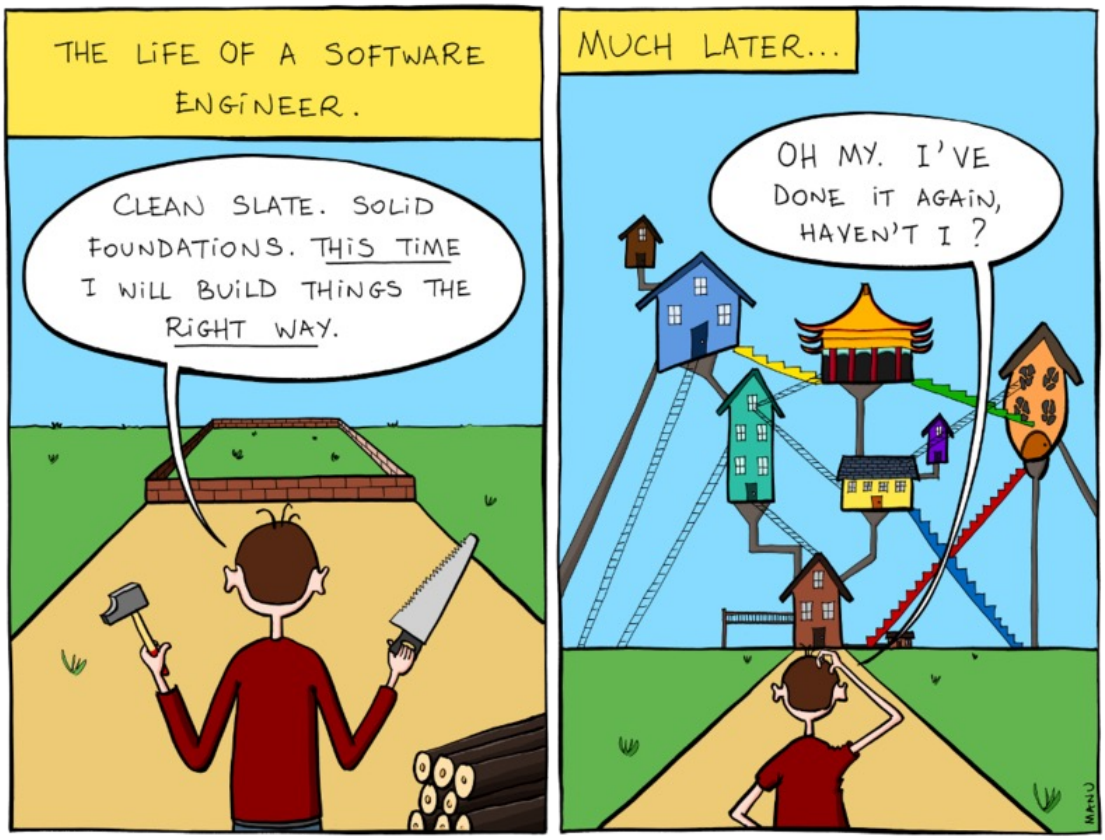
This is not meant to be a detailed implementation guide, but I address major implementation issues that you are likely to encounter.

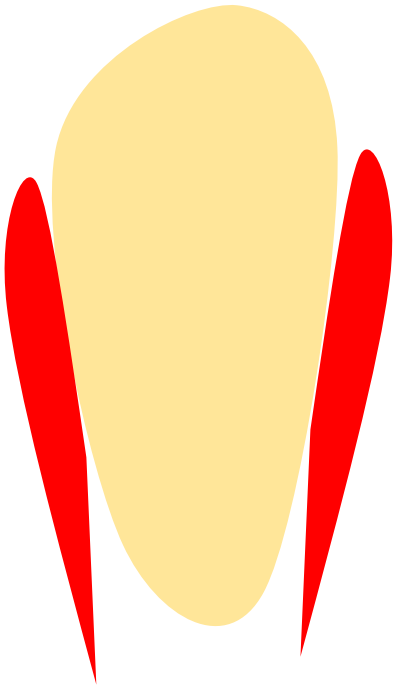
By now you should know that reading everything carefully is essential and can save you a lot of time.

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:





Project 2 Goals

- We have already seen in class predictive parsing with FIRST and FOLLOW sets.
- The goal of this project is to show you how one can automation the calculation of FIRST and FOLLOW sets for a given grammar and how to automatically modify grammars so that they can be handled by top-down parser
- Another important goal of the project is to give you experience in writing a substantial program which is non-trivial conceptually
  - This will make you a better programmer
  - You will have a better understanding of the power of abstraction in building code
  - You will have a better appreciation of the material covered so far

Things to think about

- You should decide on the data structures you will be using. Things you need to represent are
  - initial list of non-terminals
  - initial list of terminals
  - you should think about how these lists will be used for the various tasks and if they need to be combined into a larger list of symbols
  - grammar rules: LHS, RHS
  - Set representation. You should think about the operation you will need to be doing on sets
- Before you start coding, you should have an outline of how you will be using your data structures to implement the various tasks
- Before you start coding, make you you have a correct understanding of the requirements
- When you start coding, we will be happy to look at your code and design to give you feedback. The earlier you ask the better off you will be.

Outline

1. Caution
2. The big picture
3. Advice
4. The foundation: Set Operations
5. Reading the Grammar
6. Task 1: Reading the grammar
7. Task 1: Terminals and non-terminals
8. Grammar representation for the remaining tasks
9. Using Maps
10. Tasks 5 and 6
    1. Tasks 5 and 6: Lexicographic comparison
    2. Task 5: Determining common prefixes
    3. Task 5: Left factoring
    4. Task 6: Elimination of left recursion

Caution

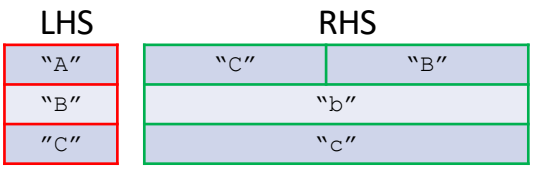
- This document does not replace the project specification document (spec). The project spec is the ultimate reference as far as the requirements of the project are concerned
- This presentation is meant to give you an overview and give some specifics about how to implement some functionality
- For the exact requirements, consult the project specification document which is self-contained as far as the requirements are concerned
- If something is not clear or is vague in the project specification, you should ask for clarification

The Big Picture: Input (1/7)

- In this project, the input is a grammar description.
  - For example, the following is a description of a grammar with three rules, three non-terminal symbols and two terminal symbols (each rule is terminated with *):

```
A -> C B * B -> b * C -> c * #
```

- Your program will read the grammar description and will store it internally in a data structure
  - For example, the grammar above can be represented as



The Big Picture: Task 1 (2/7)

- Task 1. After reading the grammar, your program should determine the terminals and the non-terminals and print them in the order in which they appear in the grammar, terminals first then non-terminals

  For example, for the input:

  ```
  A -> C B * B -> b * C -> c * #
  ```

  Your program will print

  ```
  b c A C B
  ```

  Notice how C is printed before B before it appears before B in the grammar (first rule for A) even though the rule for C appears before the rule for B

The Big picture: Task 2 Nullable set (3/7)

For this task, your program will read the grammar description and prints the Nullable set in the order in which nullable non-terminals appear in the grammar (see Task 1).

For example, for the following grammar

```
A -> C B *
A -> * A -> D C E *
B -> * B -> b *
C -> B D *
D -> d E *
E -> e D * D -> * #
```

Your program will print

```
Nullable = { A , C , B , D }
```

Notice how C is listed before B and D because C appears before B and D in the grammar.

The algorithm for calculating Nullable is given in Notes 4, page 31.

The Big picture: Task 3 FIRST sets (4/7)

For this task, your program will read the grammar description and prints the FIRST sets for every non-terminal of the grammar.

For example, for the following grammar

```
A -> C B *
A -> * A -> D C E *
B -> b * C -> c *
D -> d E * E -> e D * #
```

Your program will print

```
FIRST(A) =  { c , d}
FIRST(B) =  { b }
FIRST(C) =  { c }
FIRST(D) =  { d }
FIRST(E) =  { e }
```

Notice how `FIRST(D)` and `FIRST(E)` are calculated normally even though they are useless symbols.

The calculation is done following the algorithm that we covered in class. I realize that this really doesn't make sense in general, but this is what your program should calculate for this task.

The particular order of the contents of the FIRST sets matters. The order is described in the project specification document.

The algorithm for calculating FIRST sets is given in Notes 4, page 31

The Big picture: Task 4 FOLLOW sets (5/7)

For this task, your program will read the grammar description and prints the FOLLOW sets for every non-terminal of the grammar.

For example, for the following grammar

```
A -> C B * A -> *
A -> D C E *
B -> b * C -> c *
D -> d E * E -> e D * #
```

Your program will print

```
FOLLOW(A) =  { $ }
FOLLOW(B) =  { $ }
FOLLOW(C) =  { b , e }
FOLLOW(D) =  { c }
FOLLOW(E) =  { $ }
```

The algorithm for calculating FOLLOW sets is given in Notes 4, page 33

The Big picture: Task 5 Left Factoring(6/7)

For this task, your program will take as input a grammar and left factor the grammar by combining rules that have common prefixes as explained in the specification document

For example, if the input is the grammar

```
A -> B c *
A -> B d *
B -> b B *
B -> b * #
```

After left factoring the grammar becomes

```
A -> B A1
A1 -> c
A1 -> d
B -> b B1
B1 -> B
B1 ->
```

which should be printed according to the format specified in the specification document.

The Big picture: Task 6 Elimination of Left Recursion (7/7)

For this task, your program will take as input a grammar and eliminate left recursion from the grammar as described in the specification document.

For example, if the input is the grammar

```
A -> A a | A b * | c #
```

After left factoring the grammar becomes

```
A  -> c A1
A1 -> a A1
A1 -> b A1
A1 ->
```

which should be printed according to the format that is described in the specification document

Advice

- Now that I have described the various tasks at a high level, in the remainder of this document I am going to address some implementation and algorithmic issues
- You are not required to follow exactly what I am describing. Only the output matters for grading.
- I strongly recommend that
  - you think carefully about the project and how the pieces will fit together
  - determine the basic functionality that you will build on
  - think in higher-level terms before getting into the details

The Foundation: Set Operations

- In calculating FIRST and FOLLOW sets, you need to represent theses sets as a data structure in your program and you need to do operations on these sets
- The operations you need are
  - A = A ∪ B: Adding the elements of one set B to another set A and check if the set changed due to the additions
  - printing the elements of a set according to some order
- C++ has a number of libraries and data structures that can allow you to define sets. You can look at those and adopt one of them
- I comment on keeping track of change when adding elements of set a S1 to a set S2. Here is the pseudocode

```
for every element  x in S1
        if x is not in S2
                changed = true
                add x to S2
```

In the pseudocode, changed is a Boolean variable.

I recommend that write and test all the functions for set operations before you attempt to write higher-level functionality. You will end up fighting less with your code

Task 1: Reading the Grammar

One thing that can be confusing about this project is that

- there is a grammar that describes the input format (see project specification document)
- the input itself represents a grammar!

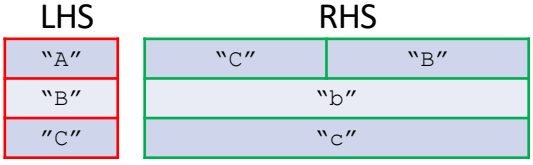
- The grammar that represents the input format is simple. Nevertheless, I strongly suggest that you write a proper recursive descent parser for it. This will result in cleaner code
- When you parse the grammar, you are going to read one rule at a time. Each rule has two parts the `LHS` and the `RHS`. The `LHS` is an `ID` and the `RHS` is a vector of `ID`s, so a rule can be represented as a struct with two fields
  - `LHS: string`
  - `RHS: vector<string>`

- The first step is to simply read all the rules as described in the previous page and store them in a vector of rules.

  for, example, for the grammar :

```
A -> C B * B -> b * C -> c * #
```

  this will look like this:



At this point the grammar is read, and you can start on Task 1

Task 1: Terminals and Non-Terminals

- This task is not really hard, but it has been my experience that sometimes it is made harder than it should be.
- I suggest that the first idea that comes to mind to solve it is not necessarily the easiest to implement!
- It would be useful to give it some thought before implementing it
- A simple approach, which is not the most efficient, is the following:
  - Read all grammar rules, into a vector of rules.
  - Each rule is a struct that consists of two parts
    - LHS: a string, which is the name of the LHS
    - RHS: vector<string> which contains the names of the terminals and non-terminals on the RHS. If the RHS is epsilon, the vector is empty.
  - Do one pass to collect all the names of non-terminals and put the names in a set. This is the set of the names of non-terminals but is not ordered
  - The set of non-terminals allows you to have a simple Boolean function isNonTerminal() that returns whether or not a string is a non-terminal
  - Now, you can do a second pass over the grammar to determine all terminals and non-terminals in the order in which they appear in the grammar

Grammar Representation for the Remaining Tasks (1/2)

- Before continuing with the other tasks, you need to decide on a representation of the grammar that you can use with the various algorithms
- A common approach I saw students use is to continue with the vector of rules in which all symbols are string.
- This approach works very well in terms of ease of implementation (it is not efficient, though). You need to know how to make it work.
- In particular, one functionality you need in calculating FIRST and FOLLOW sets is the ability to refer to something like FIRST(A) or FOLLOW(B).
- Summary: in your program, you will need to
  - represent terminals and non-terminals
  - refer to the FIRST and FOLLOW sets of particular terminals and non-terminals

Grammar Representation for the Remaining Tasks (2/2)

- You should read all terminals and non-terminals as strings and store them in a list that I will call universe or (symbols). The universe will include representations EOF (“$”)
- In order to be able to refer to FIRST(A), you can use the index of A in the list, so you can say FIRST[Index(A)], where index (A) is a function that takes a string as a parameter and returns its index in the list. This is not efficient, but it work!
- Alternatively, you can use an unordered map for FIRST sets and another one for FOLLOW sets and refer to FIRST[A] and FOLLOW[A], where A is a string. You should lookup how to use unordered maps if you want to follow this approach. This is a clear improvement and probably the easiest to work with! I recommend using it.
- Alternatively, you can have a more efficient implementation in terms of space and performance. You can store the indices and not the strings when representing grammar rules. This will effectively replace every symbol with an integer index which allows you to use FIRST[A_{index}] where A_{index} is the index for A.
- Let us see how this can be done and then we get back to the various tasks

Using Maps

- Once you have a vector of rules, you can easily iterate over all the rules
- Also, for a given rule, you can easily iterate over the RHS
- For calculating FIRST sets, for example, you can refer to FIRST[rule.LHS] or FIRST[rule.RHS[j]], which is easily supported with unordered maps
- Sample code is given on the next slide
- This code should also be helpful for other tasks

```
#include <unordered_map>
#include <set>
using namespace std;

set<string> NT = {"A","B","C"};
set<string> T = {"a","b","c"};

// universe in sorted order
vector<string> universe = {"$", "c", "a", "b"};

// FIRST sets
unordered_map<string, set<string>> FIRST;

void printSetUniverseOrder(set<string> s)
{
        // print the elements of FIRST["test"] in the order
        // in which they appear in the universe
        for (string t : universe)
                if (s.find(t) != s.end())
                                cout << t << endl;
}

void initializeFIRST()
{
 FIRST["a"] = {"a"}; FIRST["b"] = {"b"}; FIRST["c"] = {"c"}; FIRST["A"] = {};
 FIRST["B"] = {}; FIRST["C"] = {};
 FIRST["testset"] = {"b", "c"};

 cout << "printing testset" << endl;
 printSetUniverseOrder(FIRST["testset"]);

 }
```

Tasks 5 and 6

This part will address implementation issues for Tasks 5 and 6.

I will start by presenting common functionalities needed for theses tasks:

1. Lexicographic Comparison
2. Finding the length of a common prefix of two rules
3. Finding the length of common prefixes of all rules

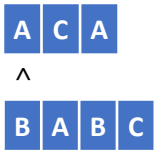
Then I will give suggestions for Tasks 5 and Task 6.

Tasks 5 and 6: 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

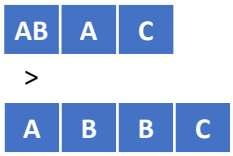
We compare the two sequences 

and

so, rule 1 appears before rule 2 in dictionary order

Example 2

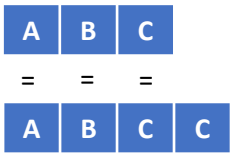
AB -> A C rule 1
A -> B B C rule 2

We compare the two sequences 

So, rule 2 appears before rule 1 in dictionary order

Example 3

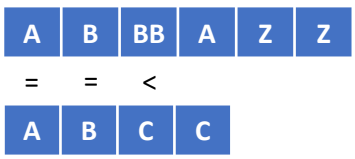
A -> B C rule 1
A -> B C C rule 2

We compare the two sequences 

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 

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

Tasks 5: 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)

Tasks 5: 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

Task 5: 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

Task 5: 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.

Task5: 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.

Task5: 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 Eliminating Left Recursion

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

Things to think about

- You should decide on the data structures you will be using. Things you need to represent are
  - initial list of non-terminals
  - initial list of terminals
  - you should think about how these lists will be used for the various tasks and if they need to be combined into a larger list of symbols
  - grammar rules: LHS, RHS
  - Set representation. You should think about the operation you will need to be doing on sets
- Before you start coding, you should have an outline of how you will be using your data structures to implement the various tasks
- Before you start coding, make you you have a correct understanding of the requirements
- When you start coding, we will be happy to look at your code and design to give you feedback. The earlier you ask the better off you will be.