CSE 351

Programming Languages

Term Project

In this project, you are expected to write a tool for eliminating left recursion from any given BNF grammar. A grammar, which is free from any left recursion, is suitable for top-down parser implementations.

Your program should expect any input BNF grammar in the following example text file format.

```
<A> -> <A> a

<A> -> b
```

Here, nonterminals are represented by uppercase letters written between “<” and “> symbols. Terminals are represented by lowercase letters. Each row indicates a single rule of the BNF grammar. Each rule consists of three parts:

1) left-hand side (LHS), which consists of a single nonterminal symbol,
2) rule operator, which is represented by consecutive “-“ and “>” operators, and, finally,
3) right-hand side (RHS), which consists of a mixture of nonterminal and terminal symbols, each separated by whitespace tokens. Whitespace can contain a mixture of space and tab characters.

You are expected to implement a very simple grammar transformation algorithm. The algorithm is given as follows.

```
for each rule i in the grammar
do
    if i is not left recursive then
    copy rule i to the output
    else
      apply left recursion elimination on i
    fi
done
```

For example, when we examine the first rule of the input BNF grammar given above, we find that the nonterminal of the LHS and the leftmost symbol of the RHS are identical (i.e. both are <A>). That means the first rule has a left recursion that is needed to be eliminated. When we consider the second rule of <A>, we discover that those two rules are used to generate strings that start with a "b" letter followed by either zero, one or many "a" letters.

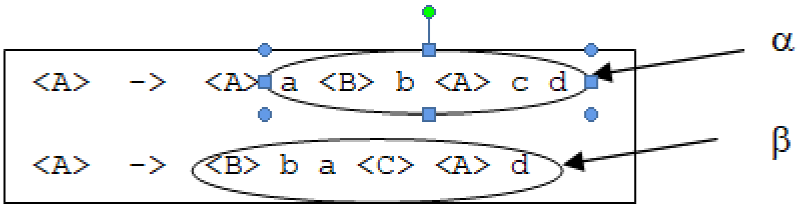
To eliminate such left recursion, we carry out a transformation as described below. First, we generate a new nonterminal, which is not in the grammar, say 'Z'. Then, we output.

```
<A> -> b <Z>

<Z> -> epsilon

<Z> -> a <Z>
```

Actually, the same transformation is valid for any number of rsh symbols in the place of "a" and "b" terminal symbols. Let's apply the same transformation to the following example grammar.



```
<A>  ->  <A> a <B> b <A> c d
<A>  ->  <B> b a <C> <A> d
```

Now, assume that "a <B> b <A> c d" string in the rsh of the first rule is alpha, and "<B> b a <C> <A> d" in the right hand side of the second rule is beta. The resulting grammar, which is given below, will still be correct and also free from any left recursion.

```
<A> -> <B> b a <C> <A> d <Z>

<Z> -> epsilon

<Z> -> a <B> b <A> c d <Z>
```

ASSUMPTIONS AND CONSTRAINTS FOR THE PROJECT

- You can assume that the nonterminals start with a single uppercase letter following any alphabetical characters in the input grammar. Such as <Aa>, <BabC> etc…
- You can assume that the terminals start with a single lowercase letter following any alphabetical characters. Such as a, b, etc...
- For the removal of a left recursion, you need to generate new nonterminals, and you can add "2" after the nonterminal name which has left recursion for this purpose. Such as <Aa2>, <BabC2> etc…
- In your project, you should consider three cases for grammar inputs.

    a) The first one simply includes grammar rules which do not need left recursion.
    b) In the second one, there will be a combination of non-terminals with only one left recursion.
    c) In the third, you should consider the case with multiple lines of rules which can have one left recursion in each.

Output Format For the Project

The output of the project should satisfy the properties below strictly:

1. There must be 1 space character between each nonterminal, terminal and "->" symbol.
2. The order of rules should not change.

Input	Output
<S> -> a <X> -> b	<S> -> a <X> -> b

3. The created new nonterminal "E2" and their branches should come after the nonterminal "E" where E is the nonterminal that E2 is produced from.

Input	Output
<E> -> <E> t <T> <E> -> <T> <X> -> b	<E> -> <T> <E2> <E2> -> epsilon <E2> -> t <T> <E2> <X> -> b

4. The order of branches in each rule should not change.

Input	Output
<E> -> <E> t <T> <E> -> <T> <E> -> a	<E> -> <T> <E2> <E> -> a <E2> <E2> -> epsilon <E2> -> t <T> <E2>

5. The epsilon branch of the newly created rule should come before the other new branches.

Input	Output
<E> -> <E> t <T> <E> -> <T>	<E> -> <T> <E2> <E2> -> epsilon <E2> -> t <T> <E2>

6. If there is more than one left recursion in a rule, then order of left recursion should be preserved in the output grammar.

Input	Output
<S> -> <S> a <S> -> <S> b <S> -> c	<S> -> c <S2> <S2> -> epsilon <S2> -> a <S2> <S2> -> b <S2>

Error Detection

In the input file, there can be faulty inputs. You should detect and print the appropriate message for each error.

1. In the input file, there can be nonterminals that have a missing < or > symbol. In that case, print one of the message below with the correct line number.
   “missing < symbol in line x“
   “missing > symbol in line y“

2. In the input grammar file, rules for a specific nonterminal has to be defined in consequent lines. if not, print an error message.

Example Input	Example Output
<E> -> <E> t <T>; <T> -> <T> f <F>; <E> -> <T>;	In line 3, nonterminal <E> is at the wrong place

Testing Your Code and Grading Policy

- Example input and output files are provided to you. Be careful, those are examples so do not forget to test your code with different inputs.
- Your output should be exactly the same as the output provided to you.
- Your code should be compiled with the Makefile given to you.
- Do not change the Makefile.
- run your code redirect output to a file
  - ./project exampleInput1.txt > output1.txt
- Check whether your output and our output file is the same or not by using the diff command on terminal
  - diff exampleOutput1.txt output1.txt
  - If your output and our output is the same there will be no message on the screen
  - Otherwise, the difference will be shown on the screen.

How to submit your code

- Zip and upload only your lex and yacc file which has the name:
  - project.l
  - project.y