Distribute the following questions across the members of your group. You will share your solutions (and most importantly the *method* of your solutions) during the next lecture period. Divide up the questions so that **each** question has at least two solutions from different group members.

1. Refactor the following grammars by consolidating the common prefixes until they are LL(1).

(a)

$$
\begin{array}{rcl}
S & \rightarrow & a\ b\ c\ d\ e\ \$ \\
 & | & a\ b\ c\ q\ y\ z\ \$ \\
 & | & a\ b\ c\ q\ r\ s\ \$ \\
 & | & T\ U\ \$ \\
T & \rightarrow & x\ a\ b \\
 & | & \lambda \\
U & \rightarrow & G\ z \\
G & \rightarrow & d \\
 & | & q
\end{array}
$$

(b)

$$
\begin{array}{rcl}
START & \rightarrow & A\ x\ B\ \$ \\
 & | & C\ \$ \\
 & | & A\ q\ C\ r\ \$ \\
A & \rightarrow & x\ y\ B \\
B & \rightarrow & g\ h\ m \\
 & | & g\ k\ n \\
 & | & g\ h\ q \\
C & \rightarrow & d\ b\ f \\
 & | & \lambda
\end{array}
$$

(c)

$$
\begin{array}{rcl}
START & \rightarrow & S\ \$ \\
S & \rightarrow & a\ S\ e \\
 & | & B \\
B & \rightarrow & b\ a\ e\ B\ e \\
 & | & a\ e\ C \\
 & | & b\ a\ e\ B\ g \\
C & \rightarrow & c\ c\ C\ e \\
 & | & c\ c\ B\ d
\end{array}
$$

2. **("Double coverage" for this question can be one group member doing the coding, and another doing the testing.)** Incorporate the solution to question 3 of `lga-ll1-parsing.pdf` into your group's "grammar code"; test with `parser-test.tok.cfg` and input `parser-test.tok` — see also `show_llparse-parser-test.tok.pdf` .

3. Refactor these grammars' left-recursive rules to make the grammar LL(1) (some grammars may require common prefix refactoring as well).

(a)

$$
\begin{array}{rcl}
S & \rightarrow & Q\ R\ \$ \\
Q & \rightarrow & Q\ x\ Q\ y \\
 & | & \lambda \\
R & \rightarrow & R\ r\ s\ t\ x\ y \\
 & | & R\ r\ s\ t\ y\ y\ y \\
 & | & s\ t
\end{array}
$$

(b)

$$
\begin{array}{rcl}
S & \rightarrow & SUM\ \$ \\
SUM & \rightarrow & SUM\ plus\ PROD \\
 & | & PROD \\
PROD & \rightarrow & PROD\ mult\ POWER \\
 & | & POWER \\
POWER & \rightarrow & val\ exp\ POWER \\
 & | & val
\end{array}
$$

(c)

$$
\begin{array}{rcl}
S & \rightarrow & FUNCTIONS\ \$ \\
FUNCTIONS & \rightarrow & FUNCTIONS\ FUNCTION \\
 & | & FUNCTION \\
FUNCTION & \rightarrow & C \\
 & | & P \\
 & | & H \\
C & \rightarrow & type\ id\ oparen\ CPARAMS\ cparen \\
P & \rightarrow & def\ id\ oparen\ PPARAMS\ cparen \\
H & \rightarrow & id\ dblcln\ type\ HPARAMS \\
CPARAMS & \rightarrow & CPARAMS\ comma\ type\ id \\
 & | & \lambda \\
PPARAMS & \rightarrow & PPARAMS\ comma\ id \\
 & | & \lambda \\
HPARAMS & \rightarrow & HPARAMS\ rarrow\ type \\
 & | & type
\end{array}
$$

4. Write a CFG for a language with two terminals ($\{a, b\}$) that represents all **non-empty** strings that are palindromes. Is your language LL(1)? If not can you refactor it using common prefix or left recursion refactoring so that it is?

Don't be fooled! Last question on next page!

5. In a previous LGA , we've shown that the reverse of any regular language is also a regular language. Now that we have seen LL(1) languages (grammars) and their limitations, it is reasonable to ask if the same property holds true for the class of LL(1) languages and deterministic parsers.

   Show this property **does not hold** by constructing an LL(1) language[1] whose reverse[2] **does have** predict set conflicts; or vice versa.

   Hint: you might call your constuction "the language of brackets dangling."

[1] A language that can be represented by a context free grammar without LL predict set conflicts.

[2] The reverse of a language $Reverse(L)$ permits sentences of terminals in the reverse order as that accepted by L. For example, a palindrome language is its own reverse.