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.

You can use the posted lecture resources here `ll-tables.zip` that provides the token input files and grammars used in lecture to test and develop your code.

A substantial portion of this assignment is continued effort on your first compiler. As previously recommended, while these functional steps appear dependent, stubbed out routines, pre-agreed upon data structures, and a small amount of hard-coded values for initial development allows these functions to be developed simultaniously.

1. Add **Predict Sets** to your group's coding project. Hint: separate your logic into these primitives:
   a. a function that returns the predict set for one production rule,
   b. a function that tests for pairwise disjoint sets within grammar non-terminals.
2. Add logic for building an **LL(1) Parsing Table** for a grammar.
3. Implement the logic for building a parse tree from a set of productions P, an LL(1) parsing table, and of course a token stream. Arrange your token stream to simply be an object that reads lines from a file that contain either a `TOKENTYPE` or a `TOKENTYPE srcValue` per line. You will hand craft these input files for now — eventually they will be generated by a proper lexer. Don't make the parse tree too complicated, trees are just lists of lists `;)`.
4. (a) Write a grammar for the basic regular expression notation we are familiar with in class. This means open and close paranthesis, the pipe symbol and asterisk (the Kleene operator) are the operator symbols. Otherwise any other character is acceptable. The order of operations (in decreasing precedence) are Kleene, sequence, and then alternation. So `a|bc*` with "grouping parenthesis" would be equivilently written as `(a|(b(c*)))`.

   (b) When you have the basic LL(1) grammar complete from part a, add the minus sign to represent character ranges and the plus symbol for "one or more" (these are the same interprettations as used in the book). An example valid regex would be:

   ```
   ab(a|e|i|o|u)+(xyz|0-9|)
   ```

   which is equivilent to

   ```
   ab(a|e|i|o|u)+(xyz|0-9)*
   ```

   (Yes, we've omitted the square brackets from common character range notation of most popular RE engines, and inside the `|)` is where the book *would have written* λ.)