This is not a proper **learning group assignment** in that you won't be able to earn participation points for it. You don't have to show any progress on it at the next lecture. Instead it's a list of utility features that you'll either **have to** or want to implement for future assignments (`ZOBOS`, possibly `WRECK`). There is no deadline or due date for these tasks — except for the project deadlines you'll want (need) to use them in.

It was probably mentioned in lecture, but you are encouraged to work on these two future assignments in two-person partner groups. `ZOBOS` and `WRECK` may well be two of the more complex projects you've encountered in the curriculum. It would probably be best to tackle them with more than one pair of eyes and as much coding background as possible. (Over the semesters I've tried to refine these projects to reduce their difficulty level, but compilers are non-trivial and I'm still told by many students they're tough nuts to crack.) It would of course be best to tackle these little side-coding tasks with your partner!

Finally, the intention in the course is that everyone has access to one or two code bases already developed by your current or previous learning group. Partnered projects can use the code base of any of the "repos" either had contributed to. If one of these has a solid SDT architecture, you probably want to lean on it for future work.

1. First and foremost, test the LL(1) parsing engine on some simple grammars and "programs," here are some candidates to use:

   i. `fischer-4-1.cfg` and `fischer-4-1_src.tok`

   ii. `fischer-p-140-10.cfg` and `fischer-p-140-10_src.tok`

   iii. `llre.cfg` , `reprogram_src.tok` and `reprogram2_src.tok`

   Make sure your LL(1) parser is working and generating "raw" parse trees that are easy to understand and work with.

   Warning: One of the examples has a syntax error — did your code find it?

2. Hopefully question 1 isn't too big a lift and you have a working data structure for representing arbitrary parse trees (`lga-ll1-parsing.pdf`). But you will need a "rich tree structure" **that can actually hold data or have a value**, not just a plain vanilla grammar construction symbol. For instance: not only a leaf node of grammar terminal type `variable`, but also the name associated with that variable. Another example would be a grammar terminal of type `float` or `integer` but also the literal value scanned for this terminal (`3.14159`, or `-10`). In general you want to be able to associate with each node multiple generic **key-value** pairings that cover at least the basic program types: integers, floating point values, strings. Of course you want a clear and easily maintained API for adding, retrieving and testing these key-value pairs.

3. Now that you have an RTS (rich tree structure), make sure it has a construction and manipulation API that makes coding up SDT (syntax directed translation, aka **semantic actions** and "production procedures") easy–peasy–lemon–squeezy. Some of the actions you'll want to support include

   i. rename or retype a node

   ii. is a leaf node? how many children does a node have?

   iii. access the first child, last child of a node

   iv. drop or remove a particular child from a parent node

   v. prune or replace an entire subtree of a tree (with a leaf node or another tree)

   vi. easily move the children (or subset of children) between two nodes — the typical case is appending the new child list at the end of the destination node's current list of children.

   You'll want (need?) to write SDT procedures for both `WRECK` and `ZOBOS`, and if you do the final project `CZAR` you will definitely be traversing and manipulating the attributes in these trees **a lot**. So you really want a good solid interface to your RTS objects.

4. It will be immensely useful in the future to be able to dump a parse tree (or some evolution of an abstract syntax tree) to disk in an easily visualized form. I personally don't call console output of bracket heavy DFS expressions as "easily visualization."[1]

   On Unix boxes graphvis and the `dot(1)` utility are excellent. The graphvis utilities have a Windows port, and I'm sure there are several FOSS packages that make it easy to write `dot(1)` files in many different languages.

   Implement a routine in your code base so that your "rich" parse tree data structure can be easily visualized.

   Hint: the schedule page row for `lga-ll1-parsing.pdf` has links to a `tree-to-graphvis` script of my own creation that permits a greatly simplified input for `dot(1)`. You are welcome to incorporate this script into your visualization solution — and using this simplified input format **will be compatible with future ZOBOS project requirements**.

[1] You may have a different opinion, but this will be a **requirement** for ZOBOS.