GLASS Requirements Document

Functional Requirements:

Syntax Specification Input

- As input, the program should receive a file containing specifications for tokens types that may appear in a given source code file, as well as a series of productions on how the source code file should be structured.
- This syntax specification file will have to follow a particular format, and will be read by the program for token definitions and grammar productions, ensuring that all parts of the syntax specification file are formatted correctly.
- In the event that an error occurs when attempting to read the syntax specification file, the program will output to the console a series of error messages indicating where within the input file the error occurred, and what the error was.
- In the event that no errors occur, a message should be printed to the console confirming that the syntax specification was successfully loaded.

Lexing

- As input, the program should receive a source code file that adheres to the syntax defined in some syntax specification file, which we will assume has already been inputted to the program (described above).
- The program will be able to take in a source code file and pass it to the lexer, which will use the token definitions provided in the syntax specification file to break the program down into a list of tokens.
- Each token will contain information about the type of token it is as defined in the syntax definition file, the row and column of the first character of the token within the source code file, as well as the original string from the source code file that the token represents.
- In the event that an error occurs while parsing the source code file, the lexer will attempt to recover and continue lexing with the goal of potentially finding other errors. If at least one error occurs, a series of error messages will be printed to the console indicating where within the source code file the error occurred, and what the error was.
- Assuming no errors are encountered, the lexing component will output a list of token objects which will be passed to the parser as input.

Parsing

- As input, the parser will receive a list of tokens from the lexer, along with productions read from the syntax definition file that was read by the main program itself.
- Assuming the lexer completes its tasks without issue, a list of tokens and a set of grammar productions will be passed to the parser.
- The parser will utilize LR(1) parsing, which requires the computation of first sets, parser item closures, transition states, and of the canonical collection of states for the provided grammar.
- From these, the parser will generate a parsing table which will be used to parse the list of tokens received from the lexer.
- In the event that an error occurs while parsing the list of tokens, the parser will attempt to recover and continue parsing with the goal of potentially finding other errors. If at least one error occurs, a series of error messages will be printed to the console indicating where within the source code file the error occurred, and what the error was.
- Assuming no errors are encountered, the parsing component will output a parse tree structure which will be passed to the XML converter as input.

XML *Conversion*

- As input, the XML converter will receive a parse tree from the parser.
- Assuming the parser completes its tasks without issue, a parsed representation of the inputted source code file will be passed to the XML converter.
- The XML converter will take this parse tree and convert it into an string in XML format acting as a marked up version of the original source code. Each part of the parse tree will be included in the XML as a tag, including the productions used to generate the tree during parse, the token type of each element within the tree, the original text from the source file that was identified as the token, and the row and column of the first character in the token.
- Presumably, as long as a valid parse tree is passed to the XML converter, there should be no issues converting the parse tree into an XML representation.
- Once the parse tree has been converted into an XML representation, the string will be written to a text file as output.

Macro-Based Refactoring Tool

- As an extension of the project's pipeline, as well as a way to show an application of this "XML-ized" source code representation, the project will come with a macro system which can be used to receive and manipulate the XML-representation of source code files.
- As input, the macro system will take in the XML representation of a source code file that has been passed through the main program, as well as a file containing a list of macro commands to be executed.
- The macro system will attempt to interpret the commands present within the macro file, and execute them accordingly.
- If an error occurs while processing the macro file, the interpreter will halt and an error will be printed to the console providing information about where and why the error occurred.
- If no errors occur, the macro system will output a new file containing the modified contents of the inputted "XML-ized" source code.

Interface Requirements:

Command Line Input and Output

- The user will be able to interact with the tool via the command line using only keyboard inputs.
- Input will consist of numerical inputs for navigating menus, and string input for inputting things such as file paths.
- During use, the tool will output text to the command line that will prompt the user to input certain information or to navigate a menu. Depending on the user's actions, different things may be outputted, including error or warning messages printed to the command line, an XML representation of an inputted source code file, and a modified version of said original source code file.

Command Line Automation

- The tool's command line command will be able to run with a variety of arguments that allow for a certain degree of automation.
- Input will be a command line command that accepts a set of arguments to optionally pre-specify an input syntax specification file, source code file, and macro file.
- Output will be dependent on the arguments specified by the user, but may include error or warning messages printed to the command line, an XML representation of an inputted source code file, and a modified version of said original source code file.

GUI Input and Output

- The tool will have the option to run in a GUI mode, where the user will be presented with a drag and drop style, node-based interface which can be used for the creation of syntax specification files. The user will then also have options to immediately apply their syntax specification to a source code file to mark it up with XML.
- Input will be done via keyboard for things such as text field, and via mouse for buttons, drag and drop selection, and connecting nodes.
- Output will be chosen by the user by interacting with the interface, but may include a syntax specification file and an XML representation of inputted source code.

Performance Requirements:

Speed

- We do not anticipate our tool taking an incredible amount of time to perform its required tasks. However, efforts will be made to ensure that in simple use cases (10-20 tokens and productions in the syntax specification, 10-100 lines of code in the source code file) the program performs its tasks in under a second. Ideally, the program can execute in under a second even for more extreme use cases.

Memory

- As with speed, we do not anticipate that our tool will require extreme amounts of memory. Will will do our best to try and ensure our tool does use more than 1GB of memory, excluding the size of inputted files.

Other Requirements:

Compatibility

- The program should be compatible with both Windows and Linux operating systems.

Documentation

- Our tool will come with extensive documentation that allows new users to be able to pick up our tool quickly, while also being a useful reference material for experienced users
- Documentation will be available on the project website, where the user will be able to view a full overview of the documentation in a table of contents. There will also be an option to view pages by topic alphabetically or filter pages based on a search keyword. Navigation of the documentation will be achieved via mouse input.