Project Title: Generalized Language Abstraction and Specification System (GLASS)

Team Members:
- Tommy Galletta (tgalletta2022@my.fit.edu)
- Alexander Lockard (alockard2022@my.fit.edu)

Faculty Advisor: Dr. Stansifer (ryan@fit.edu)

Client: Dr. Stansifer, Florida Institute of Technology

Meeting Dates for Developing This Plan:
- January 12th
- January 17th

Goals and Motivation:

The goals of this project are as follows:
- To examine and understand the inner workings of parser generators, both from a theory standpoint as well as by examining pre-existing tools.
- To build our own parser generator tool that attempts to resolve some of the issues of pre-existing tools in an elegant manner.
- To implement a GUI system and a macro system around the parser generator that gives the tool additional flexibility and ease of use.

The motivations for this project are as follows:
- We hope to learn about the inner workings of parser generators, as well as the theory that goes along with it.
- We hope to develop a tool that addresses the complexities of other parser generators by having a simpler way to specify syntax and semantics.
- We hope to build a small extension for the parser generator, that being the macro system, that allows for parsed code to be manipulated in a variety of ways.

Approach:

Grammar specification and parser generation

- Users can specify a desired grammar using a new grammar specification language unique to GLASS, but based on the grammar specification languages of other parser generators.
- GLASS can receive a grammar specification along with a source code file, and parsing it into a parse tree that can be outputted as XML.
- If the user's source files contain errors conflicting with the grammar specification they provided, GLASS will provide a series of error messages to the user to assist in debugging the source file.

GUI based grammar specification tool

- Users can build a grammar graphically through an interface.
- Users will interact with a visualization similar to a railroad/syntax diagram.
- Users will export the constructed diagram to a corresponding grammar file for use with GLASS.

Syntax markup language generation

- GLASS will output an XML-like markup language of the input file corresponding to user-defined tokens.
- Users can specify which tokens/grammars to markup (eg. ignore whitespace tokens)
- Users can write the XML to a specified file to use GLASS with other applications.

Querying and macro-based refactoring tool

- Users can execute queries on the syntax markup language to search the input file based on the query.
- Users can call predefined macros on the syntax markup language to transform certain tokens into user-defined tokens.
- Users can specify macros based on basic logical expressions through the query system.

Extensive documentation

- All features within GLASS are well documented on a publicly available, actively maintained documentation website.
- Tutorial-like sections within the documentation allow users to quickly learn new features of GLASS.

Novel Features:

Querying and macro-based refactoring tool

- While tools already exist for markup language-based source code recommendation such as srcML and pyRegurgitator, GLASS is the only tool able to do this for user-specified grammars. GLASS can be used as a tool for source code representation to query any language defined by the user.

Technical Challenges:

Defining the structure of the syntax specification file

- A syntax specification language is something that all parser generators already have, however with the goal of our tool being simplicity, we will have to closely examine the pros and cons of the different specification languages used within these tools to determine which choices we should implement in our language and which we should avoid.

Making the parser generator

- Once we have an idea of what the structure of our syntax specification files will be, we will have to actually construct the parser generator to allow our tool to parse the specified language. This will require a dive into how parsers work, as well as the theory that goes along with it.

Designing the macro-based refactoring tool

- While interpreting the macro files will probably not be a terribly difficult task, we will have to decide which macro functions we should include in our application, and then determine how to implement the functionalities for each macro function.

Milestone 1 (Feb 19):

- Compare and select the best programming language for writing this tool in, taking into consideration scalability, speed, memory usage, our familiarity with the language, and other factors. Possible languages include Java, Python, C++, and Rust.
- Investigate and compare the grammar specification languages in several parser generator tools.
- Compare and select tools, libraries, or packages that may be useful in handling XML.
- Compare and select a tool for building the GUI. GUI framework should have extensive customization and allow for cross-platform execution. (for example not having to install Python to run)
- A “Hello world” example for the chosen programming language, which would consist of controlled speed measurements for graph traversal algorithms implemented within the language. Other benchmarks can be taken into consideration such as memory usage.
- A “Hello World” example for choosing the GUI framework, which would consist of an example app implemented from a mockup to demonstrate the full features of the framework.
- A “Hello World” example for the framework of text ingestion, which would consist of controlled benchmarks of ingestion and an example of handling encoding errors within the file.
- Based on the investigation described above, develop a design pattern for the structure of the defined grammar file.
- Research existing algorithms for parsing a file given grammar eg: LALR, LR, and LL algorithms. Select algorithms from research and provide examples of why it is “better” than the others.
- Develop a design pattern for how the macro-based refactoring tool will work. Provide a Hello world example of the refactoring in action. (not code just a written example)
- Create requirements documents for both the GUI tool and parser generator.
- Create a design document for both the GUI tool and parser generator.
- Create a test document for the parser.

Milestone 2 (Mar 18):

- Reach intermediary checkpoint for the parser. At this point, the overall design should be created in a modular format. All components of the parser should be easily extensible.
- Implement, test, and demo outputting to XML-like format

Milestone 3 (Apr 15):

- Implement, test, and demo final parser generator
- Implement, test, and demo GUI tool for outputting grammar files based on user input.

Task Matrix:

Task	Tommy	Xander
Compare and select technical tools	Grammar specification languages, programming languages	XML, GUI
“Hello world” demos	Grammar specification languages, programming languages	XML, GUI
Resolve technical challenges	Defining syntax specification structure, syntax specification to parse tree algorithm	Designing the macro-based refactoring tool
Compare and select collaboration tools	**N/A (tools already decided)**	
Requirements Document	80%	20%
Design Document	20%	80%
Test Plan	80%	20%

Approval From Faculty Advisor:

I have discussed with the team and approve this project plan. I will evaluate the progress and assign a grade for each of the three milestones.

Signature: _____ Date: _____