GLASS Design Document

Syntax Specification Files:
Syntax specification files are one of the two main inputs for GLASS. They are files which contain the definitions for token regexes and grammar productions for a particular user-defined language to be parsed. While the exact format has not yet been decided for these files, an example of what the contents of these files may look like is shown in Figure 4 Users will be able to create these files by hand or can optionally use our GUI tool to assist them in creating and generating the syntax specification file for a language.

Input/Outputs:
The GLASS architecture has a requirement of two inputs, being a syntax specification file and a source code file. GLASS has multiple outputs throughout the process which are chained together to build the process model shown in Figure 5. One concept that we wish to include in our design is that the user may pick and choose which parts of the program they would like to use at any given time, and receive results only from intermediate parts of the architecture pipeline if the user so chooses. These intermediary checkpoints include the following:

- The user can input a syntax specification file and the program will determine if it is properly formatted (can be read without errors). If there are errors, information about each error will be displayed to the user.
- The user can input a syntax specification file along with a source code file and the program will determine if the source code can be parsed without errors using the rules in the syntax specification. Otherwise, information about each error will be displayed to the user.
- The user can input a syntax specification file along with a source code file and, assuming the source code file is written according to the rules of the syntax specification, the program will output a “XML-ized” version of the original source code file.

Lexer Module:
The primary purpose of the lexer module is to break down the input source code file into a token stream to be passed to the parser generator. The input is both token definitions from a syntax specification file and a source code file. The AddTokenRule and AddIgnoreRule are primary methods of the lexer module, adding both “visible” or ignorable token rules to the lexer based on the token definitions within the syntax specification (for example, comments are allowed tokens but are not consumed by the language grammar, and would therefore be ignored tokens.

Keywords such as “if” might be visible tokens). Another primary method of the lexer module is the lex method which drives the entire lexical analysis operation once all token rules have been added to the lexer.

Parser Generator Module:
The primary purpose of the parser generator module is to create a parse tree using the token stream and the grammar productions defined in the syntax specification file. The parser generator uses the LR(1) parsing algorithm to parse the token stream based on the defined grammar productions. The parser generator module outputs a parse tree data structure which is to be passed to the XML Converter.

GUI Module:
The GUI module is an optional user interface that takes user input to generate a syntax specification file used for our parser generator. The base screen shown in Figure 1 is used to show the overview of the current grammar in a top-down form. The two right buttons are used to add new expressions and components. Figure 2 demonstrates what will appear when a new expression is to be added. Once a user is finished defining their grammar they can export it to a file which can be used as input in our parser generator.

Figure 1: GUI PyQT Mock Up

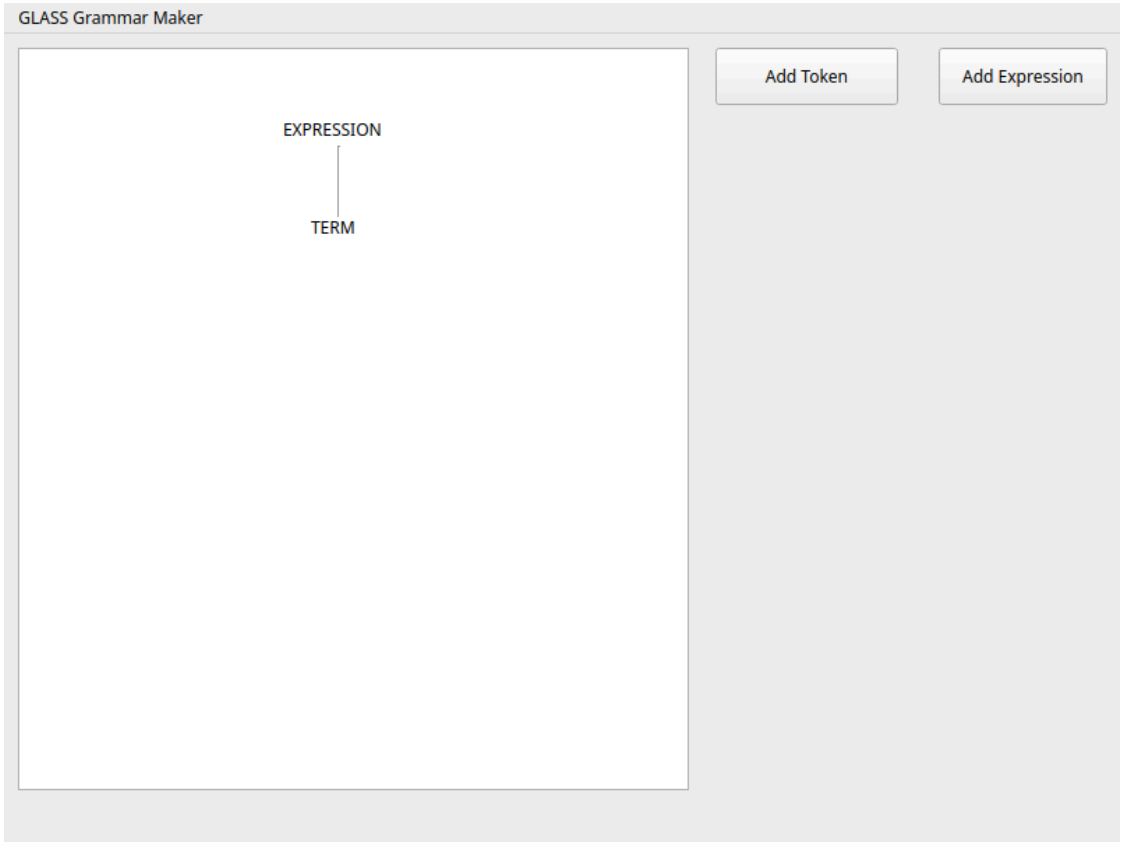
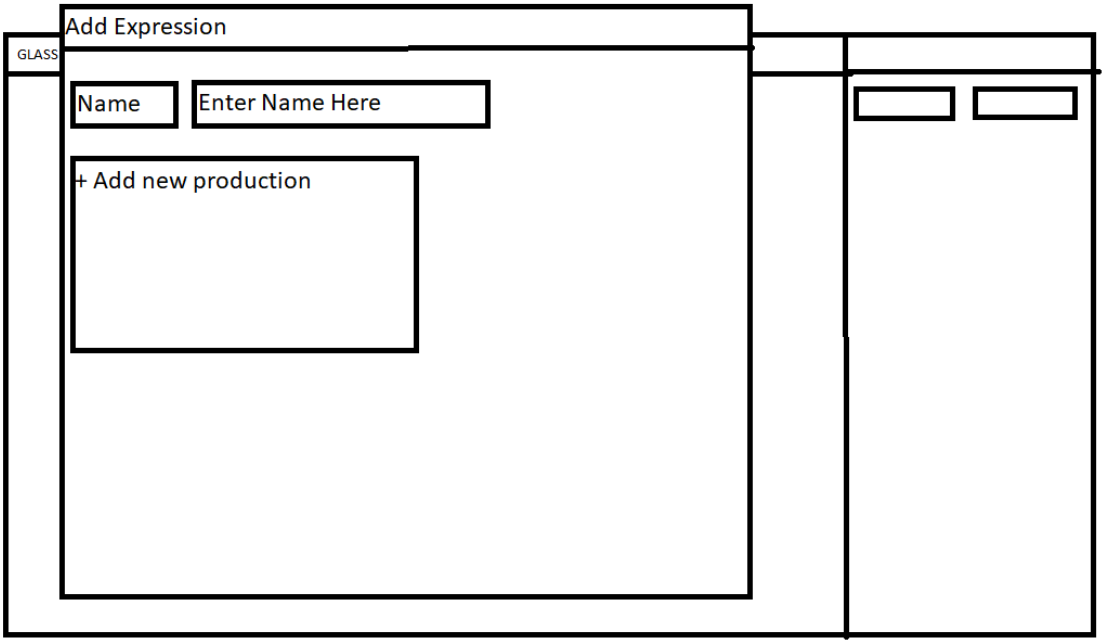


Figure 2: Add Expression Menu Mock Up



Parse Tree XML Converter:
The primary purpose of this module is to create an XML file which is a copy of the source input file with optional tags placed based on the parse tree. The input for this module will be the parse tree generated by the parser module. In standard use, the XML converter will be called immediately following the parser completing the parsing of a particular source code file (see Figure 5). The module follows a node visitor design pattern with a primary driving visit function which performs a depth-first search of the parse tree checking to see each node within the tree contains information about the original source code file, lexing, or parsing. All of this information will be written to an XML tag. The module writes all of the generated XML to a newly generated XML file, which is written to the system locally. The idea is that the XML file with all tags removed will be identical to the source file, which means the source code is fully preserved.

Macro System:
This module is an optional application of our GLASS tool used to modify the XML produced by our process flow (see Figure 5). The macro system consists of a base "AbstractMacro" class in which all macros are derived (see Figure 3). The reason this is done is that we would like for advanced users to extend this to write their macros. The primary method here is accomplished using the query function which uses a user-defined XPath query to find a certain tag within the XML tree. In our UML diagram you can see a FindReplaceMacro has already been defined which is

used to replace text within a tag specified this is a simple example implementation of the base AbstractMacro class and is used by a FindReplaceMacroController which is responsible for the execution of the entire macro sequence. The architecture is defined in a way to make this system easily user-extensible. The idea is that GLASS will ship with several more predefined macros.

Figure 3: Macro System UML Diagram

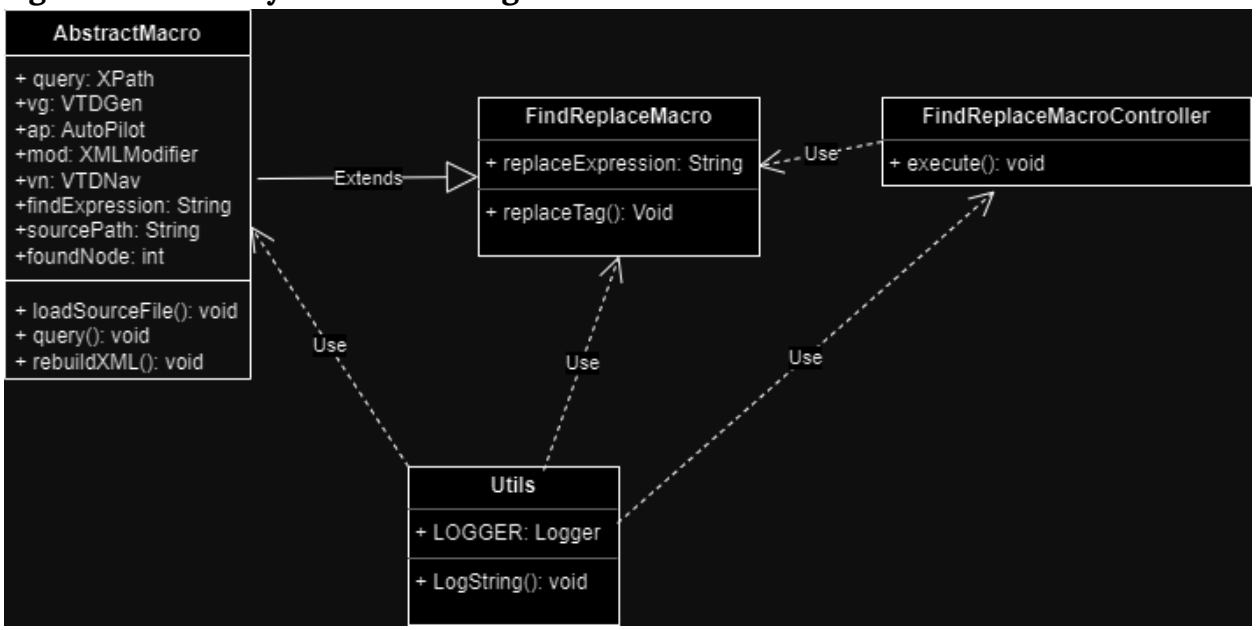


Figure 4: Example Syntax Specification Contents

```
tokens {
    visible IDENTIFIER : [a-zA-Z][a-zA-Z_]*
    visible PLUS : \+
    visible STAR : \*
    visible LPAREN : \(
    visible RPAREN : \)
    ignore WHITESPACE : \s+
    ignore COMMENT : \/\/.*\r\n
}

// comments may be added to
// syntax specification files

productions {
    E -> T E'
    E' -> PLUS T E'
    E' -> EPSILON
    T -> F T'
    T' -> STAR F T'
    T' -> EPSILON
    F -> LPAREN E RPAREN
    F -> IDENTIFIER
}
```

Figure 5: System Architecture Diagram