DEVELOPER MANUAL

for

GLASS

Version 0.2.1a

Prepared by: **Tommy Galletta**
Alexander Lockard

Advised by: **Dr. Ryan Stansifer**

Submitted to: **Dr. Philip Chan**
Instructor

November 25th, 2024

Contents

1 Introduction

1.1 Terminology

The following terms are heavily used throughout the documentation, and thus are defined below:

> **Syntax definition file** - A GLASS input file which contains a series of token and production definitions, which are then used to parse a source file.
>
> **Source file** - A file whose contents is parsable by the rules defined in a particular syntax definition file.
>
> **Interpretation script file** - A GLASS input file which contains the instructions on how to interpret the contents of a source file after it has been parsed into a concrete parse tree using the rules defined in a particular syntax definition file.

1.2 Overview and Background

The Generalized Language Abstraction and Scripting System (GLASS) is a tool whose primary purpose is to assist in the parsing and interpretation of structured files. GLASS utilizes the LR1 parsing algorithm to parse the contents of any source file by following the production rules defined in a provided syntax definition file. Once parsed, a source file's contents may then be interpreted by passing the parse tree to an interpretation script, which is executed by GLASS and allows the user to specify operations to be performed at each node of the parse tree.

GLASS was developed as a part of the Florida Institute of Technology CSE 4101/4102 Computer Science Projects courses. It was developed over the course of

six milestone “sprints” spanning two academic semesters. All materials generated for these milestones are publicly available on the *Project Materials* page of the GLASS website (https://www.glass-project.com/project-materials).

1.3 Tools Used

The following tools were used in the development of GLASS:

- Git / GitHub
- Java
- Apache Maven
- Javascript / React
- Launch4j

1.4 “Under the Hood”

The GLASS source code consists of several Java packages, each serving a particular purpose in the project. These packages are as follows:

- **api** - Handles complex interactions with GLASS core systems, allowing for more simple connections to tools such as the GLASS GUI.
- **lr1** - Contains all algorithms and structures necessary to produce parsers using the LR1 parsing algorithm.
- **managers** - Contains Java classes which are used to maintain and manage the overall execution of the GLASS system. This includes managers for things such as command line arguments and logging systems.
- **scripting** - Handles all processes surrounding the interpretation of interpretation script files.

- **syntaxdef** - Handles all processes for reading in and generating the parse tables for languages defined in syntax definition files.
- **utils** - A general package containing miscellaneous utility functions.

The code within several of these packages is described in the sections that follow.

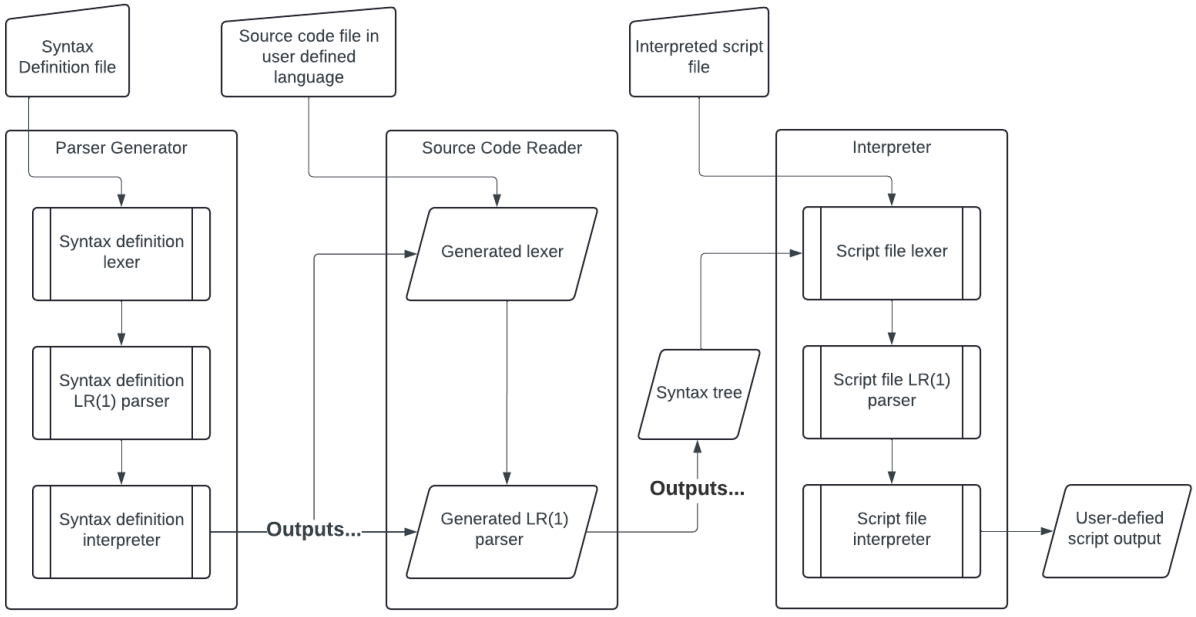

Figure 1. System architecture diagram of the main GLASS system

2 Installation

2.1 Installing from Pre-Packaged Download

To install GLASS as a pre-packaged download, simply navigate to the *Download* page of the GLASS website (https://glass-project.com/download). After clicking *Download*, a `.zip` file will be downloaded to your local system. Which you may unzip and view the contents of. Within the unzipped download, you will see a jar file named `GLASS-[version-number].jar` which can be run using `java -jar GLASS-[version-number].jar`.

The GLASS GUI can be run by simply double clicking the `.exe` file named `GLASS-GUI-[version-number].exe`

2.2 Installing from Source

2.2.1 Installing GLASS Core

Before installing GLASS from source, make sure you have a Java JDK of at least version 20, Apache Maven, and Git installed on your system.

To install GLASS from source you will need to access the GLASS GitHub repository. You may clone this repository by running `git clone https://www.githhub.com/GLASS-group/GLASS` in your terminal. GLASS). GLASS can then be compiled using `mvn clean install` and a `.jar` file will be created in the target directory. The `.jar` file can be run by executing the command `java -jar  target/GLASS-[version-number].jar` within the root directory of the cloned repository.

2.2.2 Installing GLASS GUI

To install GLASS GUI from a source you will first need to build the GLASS core repo using the steps listed in section 2.2.1, after this is complete you can build an uber jar using `mvn package` once complete you will see a jar file output which can be ran using `java -jar  target/GLASS-GUI-[version-number].jar`

3 LR1 Parsing System

3.1 Overview

The GLASS `lr1` subpackage contains all of the code necessary to build a parse table using the LR1 parsing algorithm. This algorithm is a left-to-right, rightmost derivation algorithm with one lookahead terminal. This algorithm allows for a bottom-up parsing approach, and is able to parse most common programming languages (that is, a valid LR1 grammar exists for most common programming languages).

This subpackages contains files such as Token, Production, Parser, Lexer, ParseTable, and more, the details of which are described in the subsections that follow.

3.2 Resources

Since the LR1 parsing algorithm is not our own, we figure it would simply be best to provide links to pre-existing resources for learning about LR1 parsing. In fact, these are the very materials we used to make this tool!

- Canonical LR parser - Wikipedia
- LR Parsing Table Construction - University of Michigan
- The LR(1) Table Construction - University of Delaware

4 Syntax Definition

4.1 Overview

Within GLASS syntax definition files are the medium through which a user conveys the language (i.e. the structure of the file) that they wish to parse. In order to parse syntax definition files, the already established LR1 parsing system is leveraged by generating a lexer and parser with hardcoded rules which define the structure of our syntax definition files.

4.2 SyntaxDefinitionLexer

The SyntaxDefinition contains the regex patterns for all of the possible token types (or non-terminals) that may appear within a syntax definition file. These include:

- The keywords `tokens`, `productions`, `name`, `active`, and `ignored`
- Braces and brackets, thin arrows (`->`), thick arrows (`=>`), and periods
- Comments
    - Regex pattern : `#.*\r\n`
- Identifiers
    - Regex pattern: `[a-zA-Z][a-zA-Z0-9_]*[a-zA-Z0-9]*`
- Regex definitions
    - Regex pattern: `/.*[^\\]/`

4.3 SyntaxDefinitionParser

The SyntaxDefinitionParser contains all of the grammar productions necessary in order to parse the contents of an arbitrary syntax definition file. The current grammar productions used are as follows:

- S -> NAME_DECLARATION TOKEN_BLOCK PRODUCTION_BLOCK
- S -> TOKEN_BLOCK PRODUCTION_BLOCK
- NAME_DECLARATION -> “name” <identifier>
- TOKEN_BLOCK -> “tokens” “{“ TOKEN_DECLARATION “}”
- TOKEN_DECLARATION -> ACTIVE_TOKENS TOKEN_DECLARATION
- TOKEN_DECLARATION -> IGNORED_TOKENS TOKEN_DECLARATION
- TOKEN_DECLARATION -> ACTIVE_TOKEN TOKEN_DECLARATION
- TOKEN_DECLARATION -> IGNORED_TOKEN TOKEN_DECLARATION
- TOKEN_DECLARATION -> ε
- ACTIVE_TOKENS -> “active” “{“ TOKEN_LIST “}”
- IGNORED_TOKENS -> “ignored” “{“ TOKEN_LIST “}”
- ACTIVE_TOKEN -> “active” TOKEN_DATA
- IGNORED_TOKEN -> “ignored” TOKEN_DATA
- TOKEN_LIST -> TOKEN_DATA TOKEN_LIST
- TOKEN_LIST -> ε
- TOKEN_DATA -> <identifier> “->” <regex pattern>
- PRODUCTION_BLOCK -> “productions” “{“ PRODUCTION_DECLARATION “}”
- PRODUCTION_DECLARATION -> PARENT_PRODUCTION CHILD_PRODUCTIONS PRODUCTION_DECLARATION
- PRODUCTION_DECLARATION -> ε
- PARENT_PRODUCTION -> <identifier> “->” TERM_NONTERM_LIST
- PARENT_PRODUCTION -> “[“ <identifier> “]” <identifier> “->” TERM_NONTERM_LIST
- CHILD_PRODUCTION -> “=>” TERM_NONTERM_LIST
- CHILD_PRODUCTION -> “[“ <identifier> “]” “=>” TERM_NONTERM_LIST
- TERM_NONTERM_LIST -> TERM_NONTERM_LIST <identifier>
- TERM_NONTERM_LIST -> <identifier>

4.4 SyntaxDefinitionReader

The SyntaxDefinitionReader is the class responsible for taking the parsed contents of a syntax definition file, and interpreting the contents of the file to build a parse table.

This process has two main steps, each of which contain multiple substeps:

- Interpret the syntax definition file contains
    - Determine defined tokens (terminals)
    - Determine defined productions
    - Determine defined non-terminals (from productions)
- Send found information to lexer and parser to be built
    - Add all found tokens to the new lexer
    - Allow the new parser to access the new lexer and its tokens
    - Add all found productions to the new parser
    - Begin the build process for the new parser

5 Interpretation Script

5.1 Overview

Once a source file is parsed using the parser generated from the SyntaxDefinitionReader, the contents of the constructed parse tree need to be interpreted. This is where the interpretation script system comes in. Interpretation scripts allow the user to define what operations should be performed as the parse trees for source files in their defined language are traversed.

To learn more about how interpretation scripts are written, read sections 4 and 5 of the documentation available on our project website (https://www.glass-project.com/documentation). The purpose of this manual is to explain how the scripts themselves are interpreted.

5.2 ScriptLexer

The ScriptLexer contains the regex patterns for all of the possible token types (or non-terminals) that may appear within an interpretation script file. These include:

- The keywords `production`, `function`, `return`, `if`, `else`, `true`, `false`, `while`, and `new`
- The following symbols: `(`, `)`, `[`, `]`, `{`, `}`, `,`, `.`, `;`, `+`, `-`, `*`, `/`, `//`, `%`, `=`, `->`, `==`, `!=`, `<`, `>`, `<=`, `>=`, `||`, `&&`
- Floating point numbers
    - Regex pattern: `(\.[0-9]+)|([0-9]+\.[0-9]*)`
- Integer values:
    - Regex pattern: `0|[1-9][0-9]*`

- Identifiers:
    - Regex pattern: `[a-zA-Z][a-zA-Z0-9_]*[a-zA-Z0-9]*`
- String literals:
    - Regex pattern: `"(?:[^"\\\n\r]|\\[^\n\r])*"`
- Comments:
    - Regex pattern: `#.*(\r\n)?`

5.3 ScriptParser

The ScriptParser contains all of the grammar productions necessary in order to parse the contents of an arbitrary interpretation script file. The current grammar productions used are as follows:

- `S -> STATEMENTS  S`
- `S -> PRODUCTION_FUNCTION_DECLARATION S`
- `S -> USER_FUNCTION_DECLARTATION S`
- `S -> ε`
- `STATEMENTS -> STATEMENT STATEMENTS`
- `STATEMENTS -> STATEMENT`
- `STATEMENT -> FUNCTION_CALL SEMICOLON`
- `STATEMENT -> IF_BLOCK`
- `STATEMENT -> WHILE_BLOCK`
- `STATEMENT -> VARIABLE_ASSIGNMENT SEMICOLON`
- `STATEMENT -> “return” EXPRESSION SEMICOLON`
- `STATEMENT -> OBJECT_ACCESS SEMICOLON`
- `OBJECT_ACCESS -> OBJECT_INDICATOR ACCESSOR_LIST`
- `EXPRESSION -> EXPRESSION LOGICAL_OR OR_CONDITION`
- `EXPRESSION -> OR_CONDITION`
- `OR_CONDITION -> OR_CONDITION LOGICAL_AND AND_CONDITION`
- `OR_CONDITION -> AND_CONTIDION`
- `AND_CONDITION -> AND_CONDITION RELATION_OP NUMERICAL_EXPRESSION`
- `AND_CONDITION -> NUMERICAL_EXPRESSION`
- `RELATION_OP -> “==”`

- RELATION_OP -> “!-”
- RELATION_OP -> “>”
- RELATION_OP -> “<”
- RELATION_OP -> “>=”
- RELATION_OP -> “<=”
- NUMERICAL_EXPRESSION -> NUMERICAL_EXPRESSION ADDOP TERM
- NUMERICAL_EXPRESSION -> TERM
- ADDOP -> “+”
- ADDOP -> “-”
- TERM -> TERM MULOP FACTOR
- TERM -> FACTOR
- MULOP -> “*”
- MULOP -> “/”
- MULOP -> “//”
- MULOP -> “%”
- FACTOR -> OPT_NEGATIVE <integer>
- FACTOR -> OPT_NETATIVE <float>
- FACTOR -> OBJECT_INDICATOR ACCESSOR_LIST
- FACTOR -> “true”
- FACTOR -> “false”
- FACTOR -> FUNCTION_CALL
- FACTOR -> “(“ EXPRESSION “)”
- OPT_NEGATIVE -> “-”
- OPT_NEGATIVE -> ε
- OBJECT_INDICATOR -> <identifier>
- OBJECT_INDICATOR -> <string literal>
- OBJECT_INDICATOR -> ARRAY_LIT
- OBJECT_INDICATOR -> NEW_OBJECT_DECLARATION
- NEW_OBJECT_DECLARATION -> “new” <identifier> “(“ PARAMETER_LIST “)”
- ARRAY_LIT -> “[“ PARAMETER_LIST “]”
- ACCESSOR_LIST -> “.” FUNCTION_CALL ACCESSOR_LIST
- ACCESSOR_LIST -> “.” <identifier> ACCESSOR_LIST
- ACCESSOR_LIST -> “[“ EXPRESSION “]” ACCESSOR_LIST
- ACCESSOR_LIST -> ε
- FUNCTION_CALL -> <identifier> “(“ PARAMETER_LIST “)”

- PARAMETER_LIST -> EXPRESSION “,” PARAMETER_LIST
- PARAMETER_LIST -> EXPRESSION
- PARAMETER_LIST -> ε
- PRODUCTION_FUNCTION_DECLARATION -> “production” <identifier> “{“ BLOCK_CONTENTS “}”
- PRODUCTION_FUNCTION_DECLARATION -> “production” PRODUCTION_DEFINITION “{“ BLOCK_CONTENTS “}”
- USER_FUNCTION_DECLARATION -> “function” <identifier> “(“ PARAMETER_NAMES “)” “{“ BLOCK_CONTENTS “}”
- PARAMETER_NAMES -> <identifier> “,”, PARAMETER_NAMES
- PARAMETER_NAMES -> <identifier>
- PARAMETER_NAMES -> ε
- PRODUCTION_DEFINITION -> <identifier> “->” TERM_NONTERM_LIST
- TERM_NONTERM_LIST -> <identifier> TERM_NONTERM_LIST
- TERM_NONTERM_LIST -> <identifier>
- VARIABLE_ASSIGNMENT -> <identifier> “=” EXPRESSION
- IF_BLOCK -> “if” “(“ EXPRESSION “)” “{“ BLOCK_CONTENTS “}” ELSE_BLOCK
- ELSE_BLOCK -> “else” “{“ BLOCK “}”
- ELSE_BLOCK -> ε
- WHILE_BLOCK -> “while” “(“ EXPRESSION “)” “{“ BLOCK_CONTENTS “}”
- BLOCK_CONTENTS -> STATEMENTS
- BLOCK_CONTENTS -> ε

5.4 ScriptReader

The ScriptReaderis the class responsible for taking the parsed contents of an interpretation script file, and performing the actions specified by the user in the script.

These actions include:

- Calling global or user-defined function, handled by the FunctionCall class
- Executing statements conditionally, handled by the IfStatement class

- Accessing object values or functions in an isolated statement, handled by the ObjectAccessStatement class
- Returning values, handled by the ReturnStatement class
- Assigning a value to a variable, handled by the VariableAssignmentStatement class
- Executing statements in a loop as long as a condition is true, handled by the WhileStatement class

Any data values that are used within the interpretation script while interpreting its contents are managed using the ScriptValue class, which is the class responsible for handling the type of the data on the Java end so that the user does not have to specify data types in their interpretation script.

In order to make the interpretation script code easily extensible, various parts of the interpretation process are built from abstract classes or interfaces:

- All statement types used by the interpretation script system implement the ScriptStatement interface
- All functions that can be executed globally within the interpretation script system extend the ScriptFunction abstract class
- All object data types that can be used within the interpretation script system (arrays, strings, etc.) extend the ScriptObject abstract class.

6 GUI

6.1 Executing GLASS

To execute GLASS from the GUI you must navigate to the Execution Window. Afterwards you will first select a

6.2 Creating a Grammar

To create a grammar you will first need to navigate to the grammar creation window. Upon entering this window you will see various options such as “Create Terminal” and “Create Non Terminal’. Terminals here coincide with tokens created in the Syntax Definition files. You will define a Terminal using a name and a regular expression on the other hand NonTerminals will only need to be defined with a name. After defining all Terminal and NonTerminals, you will create your productions by clicking the “Create Production” button. To create a production select a NonTerminal from the left dropdown menu and select one or more Terminal or NonTerminal symbols on the right-hand side. After selecting “Create” you will see your defined production inside the Grammar Tree viewer in the previous window. When you’re happy with the grammar you have created you can save the syntax file to your computer using the “Write Grammar to File” option.