Parser Documentation

Table of Contents:

*Note: The pseudocode in these sections is only roughly analogous to the actual code. Reasons for discrepancies include brevity of pseudocode and error checking capabilities of actual code. Pseudocode syntax is loosely in Java.

Section 1 – Usage

Language Parser

Here is the usage for Language Parser:

```
java -jar LanguageParser.jar [languageSpec] [inputFile]
```

`languageSpec`	Token type definitions.
inputFile	This file contains the input to be parsed.

Grammar Parser

Here is the usage for Grammar parser:

```
java -jar GrammarParser.jar [tokenSpec] [grammarSpec] [inputFile]
```

`tokenSpec`	Token type definitions.
`grammarSpec`	Gramar definition.
inputFile	This file contains the input to be parsed.

Section 2 – Architecture

Package Overview

The involved source code files are split up into packages according to the scope of their use. Here is an explanation of all of the packages:

Package Name	Description
generators	The generators package contains classes that only have static methods. These classes provide methods that encapsulate components of this project. Each generator class is independent of each other generator class.
grammar	The grammar package contains classes necessary for the construction of a Grammar object. These classes should not be used outside of this package and the GrammarGenerator class – their immutable equivalents should be used instead.
grammar.defaults	The grammar.defaults package contains classes that provide access to default Grammar objects, which are necessary for the parsing grammar and language specification files.
immutable	The immutable package contains all classes that transmit data between generators. Because these classes represent fully initialized components, there is no need to be able to modify their contents, and furthermore, any modification would be considered corruption. To prevent corruption, these classes are immutable.
interpreter	The interpreter package contains all classes and methods necessary for the MiniRe scripting language.
interpreter.defaults	The interpreter.defaults package contains classes that provide for the functionality of the MiniRe scripting language. These classes are based upon templates found in the interpreter package.
language	The language package contains classes necessary for the construction of token types. These classes are fully mutable, and are consequently only used in the LanguageGenerator class.
language.defaults	The language.defaults package contains classes that provide access to default Language objects, which are necessary for parsing grammar and language specification files.
utilities	The utilities package contains helper classes that standardize and simplify operations that are common throughout the project.

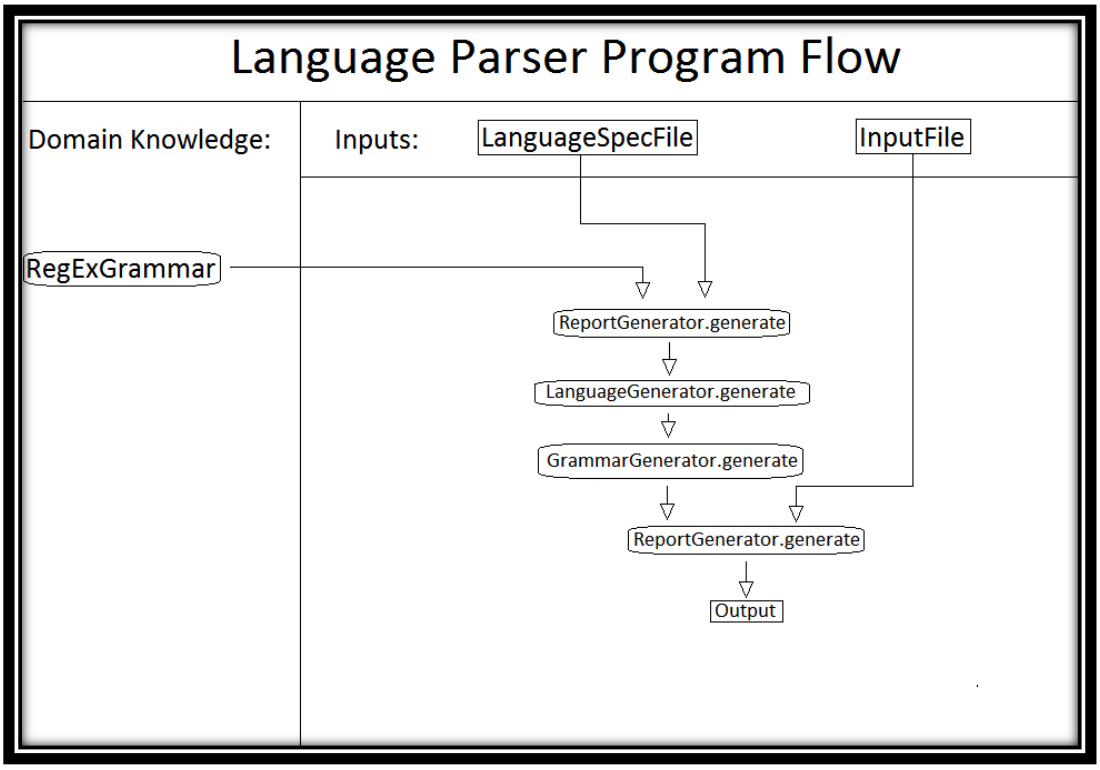
Class Structure

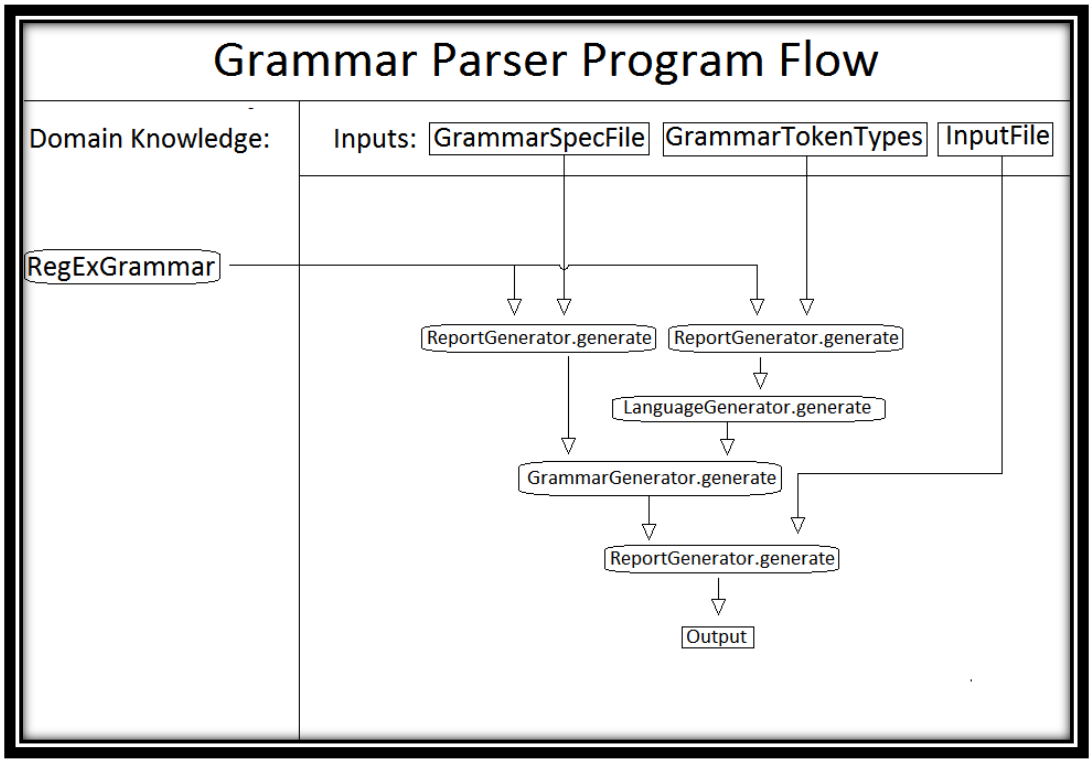
There are 8 classes that are essential to understanding this project. Here's an explanation of those classes:

Class	Fields	Description
State		
	String	The name associated with this State.
	Map<Character, Set<State>>	The mapping of characters to the destination States.
	Map<State, Set<Character>>	The mapping of destination States to characters that transition to them.
	Set<State>	The set of States to which this State can epsilon transition.
StateMachine		
	State	The initial state of this StateMachine.
	State	The final state of this StateMachine.
	List<State>	The states involved with this StateMachine.
	Set<Character>	The characters that could possibly start a string matching this StateMachine.
Token		
	String	The contents of this Token.
	TokenType	The token type of this Token.
TokenType		
	String	The name associated with this TokenType.
	StateMachine	The state machine that is used to extract Tokens of this TokenType.
	Language	The language to which this TokenType belongs.
Language		
	Alphabet	The Alphabet that defines valid characters of this Language.
	Map<String, TokenType>	The TokenTypes of this Language, indexed by their names.
Rule		
	String	The name associated with this Rule.
	List<List<Object>>	The possible children (Rules or TokenTypes) of this Rule.
	Set<TokenType>	The set of TokenTypes that can start this Rule.
Grammar		
	String	The name associated with this Grammar.
	Rule	The root Rule of this Grammar.
	Set<Rule>	The set of Rules that belong to this Grammar.
Report		
	Grammar	The Grammar that constructed this Report.
	List<Token>	The Tokens extracted according to the Grammar.

Program Flows

There are 4 functions that handle the generation of Tokens, Reports, Languages, and Grammars. These same generator classes handle the reading in and processing of specification files using a default set of Languages and Grammars. The following diagrams illustrate the flow of information for each of the portions of this submission.





Section 3 – Token Generation

The TokenGenerator handles the construction of all Token objects from given TokenType objects and string contents. It employs a greedy matching algorithm to return the longest possible string that will match the specified TokenType. It does this by stepping through the TokenType's StateMachine, and tracking visited states and removing characters from the contents until it can't progress anymore. Then it removes visited states and puts characters back into the contents until it lands on a final state. At that point it has the contents of the longest possible Token of that TokenType, so it constructs the Token and then returns. The following high-level pseudocode explains the algorithm more precisely.

```
TokenGenerator.generate(TokenType tokentype, String contents) {
      visitedStates = [];
      currentState = tokenType.initialState;
      currentContents = “”;
      while (true) {
            if (contents.isEmpty()) {
                  break;
            }
            nextChar = contents.getFront()
            if (currentState.canTransition(nextChar) == false) {
                  break;
            }
            currentContents.append(nextChar);
            contents.deleteFront();
            visitedStates.append(currentState);
            currentState = currentState.transition(nextChar);
      }

      while (tokenType.isFinalState(currentState) == false && visitedStates.size() > 0) {
            visited.deleteBack();
            currentState = visited.getBack();
            contents.pushFront(currentContents.getBack());
            currentContents.deleteBack();
      }

      return new Token(tokenType, currentContents);
}
```

Preconditions:

- The given contents must start with the given TokenType.

Section 4 – Report Generation

The ReportGenerator handles the construction of all Report objects from given Grammar objects and string contents, leveraging the TokenGenerator's functionality. It employs an LL1 parsing algorithm to traverse the Grammar and queries the TokenGenerator repeatedly. The ReportGenerator is able to resolve ambiguities in constant time by leveraging the starting TokenType of each rule, which is computed during the construction of the Grammar object, and by giving precedence toward reserved words. The following high-level pseudocode explains the algorithm more precisely.

```
Report ReportGenerator.generate(Grammar grammar, String contents) {
	return new Report(recursiveDescent(grammar.rootRule, contents));
}

List<Token> recursiveDescent(Rule rule, String contents) {
	result = new List<Token>();
	contents.removeLeadingWhitespace();
	possibleChains = new Set<List<Object>>;
	foreach(List<Object> chain : rule.chains) {
		if (chain.front().canStartWith(contents.front())) {
			possibleChains.add(chain);
		}
	}
	nextChain = null;
	if (possibleChains.size() > 1) {
		foreach(List<Object> chain : possibleChains) {
			if (chain.front().startsWithReservedWord()) {
				nextChain = chain;
			}
		}
		if (nextChain != null)
			break;
	} else {
		nextChain = possibleChains.getFront();
	}
	foreach(Object item : chain) {
		if (item is TokenType) {
			result.add(TokenGenerator.generate(item, contents);
		} else if (item is Rule) {
			Result.addAll(recursiveDescent(item, contents);
		}
	}
	return result;
}
```

Preconditions:

- The given Grammar must be unambiguous.
- The given contents must be valid in the given Grammar.

Section 5 – Language Generation

The LanguageGenerator handles the construction of all Language objects from given Report objects. It uses domain knowledge of the format of the given Report object to construct TokenType objects and their associated StateMachine objects. The LanguageGenerator does have to ensure that all TokenType objects that that a particular TokenType depends on have been constructed before constructing it. The following high-level pseudocode explains the algorithm more precisely.

```
Language LanguageGenerator.generate(Report report) {
      tokenTypeMap = report.constructNameToTokenListMap();
      tokenTypes = new Map<String, TokenType>();
      foreach(String tokenTypeName : tokenTypeMap.keys()) {
            tokenTypes = processTokenType(tokenTypeName, tokenTypeMap, tokenTypes));
      }
      return new Language(tokenTypes);
}

Map<String, TokenType> processTokenType(String name, Map<String, List<Token>> map,
            Map<String, TokenType> tokenTypes) {
      if (tokenTypesMap.contains(name)) {
            return tokenTypes;
      }
      foreach(Token token : map.get(name)) {
            if (token.tokenType == RegExLanguage.DEFINED_CLASS_TOKEN) {
                  tokenTypes = processTokenType(token.contents, map, tokenTypes);
            }
      }
      stateMachine = constructStateMachine(map.get(name));
      tokenTypes.add(new TokenType(name, stateMachine));
      return tokenTypes;
}
```

Preconditions:

- The given Report must follow the format of TokenType specification files.
- There are no cyclic dependencies in TokenType definitions.

Section 6 – Grammar Generation

Context Independent

The GrammarGenerator handles the construction of all Grammar objects. In the first case, it generates a completely context independent grammar from a given Language. It does this through a repetitive pattern of matching the longest first Token, which it discerns through TokenGenerator.getTokenLength. The following high-level pseudocode explains the algorithm more precisely.

```
Grammar GrammarGenerator.generate(Language language) {
	root = new Rule(“root”);
	foreach(TokenType tokenType : language.tokenTypes) {
		chain = new List<Object>();
		chain.add(tokenType);
		chain.add(root);
		root.possibleChildren.add(chain);
	}
	finalChain = new List<Object>();
	finalChain.add(RegExLanguage.EPSILON_TOKEN);
	root.possibleChildren.add(finalChain);
}
```

Preconditions:

- The given Language is valid.

File Specified

The GrammarGenerator also handles the construction of Grammar objects from specification strings. First it calls the LanguageGenerator on the first specification string to construct TokenType objects, which are necessary for interpreting the Grammar. Then it constructs the Rule objects based on the second string's specification. The following high-level pseudocode explains the algorithm more precisely.

```
Grammar GrammarGenerator.generate(String tokenTypeSpec, String grammarSpec) {
	tokenReport = ReportGenerator.generate(RegExGrammar, tokenTypeSpec);
	language = LanguageGenerator.generate(tokenReport);
	grammarReport = ReportGenerator.generate(GrammarSpecGrammar, grammarSpec);
	grammarLines = List<List<Token>>();
	grammarLines.add(new List<Token>());
	foreach(Token token : grammarReport.tokens) {
		if (token.tokenType == GrammarSpecGrammar.NEW_LINE_TOKEN) {
			grammarLines.add(new List<Token>());
		}
	}
	ruleTokenMap = new Map<String, List<Token>>();
	ruleMap = new Map<String, Rule>();
	foreach(List<Token> tokens : grammarLines) {
		ruleName = tokens.getFront().contents
		tokens.deleteFront();
		tokens.deleteFront();
		if (ruleMap.containsKey(ruleName) == false) {
			ruleTokenMap.put(ruleName, tokens);
			ruleMap.put(ruleName, new Rule(ruleName);
		} else if (ruleMape.containsKey(ruleName)) {
			ruleMap.get(ruleName).add(GrammarSpecGrammar.OR_TOKEN);
			ruleMap.get(ruleName).addAll(tokens);
		}
	}
	foreach(String ruleName : ruleTokenMap.keys()) {
		rule = ruleTokenMap.get(ruleName);
		ruleTokenMap.get(ruleName).chains.add(new List<Object>());
		foreach(Token token : ruleTokenMap.get(ruleName)) {
			contents = token.contents;
			if (token.tokenType == GrammarSpecGrammar.RULE_TOKEN) {
				rule.chains.getBack().add(ruleTokenMap.get(contents));
			} else if (token.tokenType == GrammarSpecGrammar.TOKEN_TYPE_TOKEN) {
				rule.chains.getBack().add(language.tokenTypes.get(contents));
			} else if (token.tokenType == GrammarSpecGrammar.OR_TOKEN) {
				ruleTokenMap.get(ruleName).chains.add(new List<Object>());
			}
		}
	}
	return new Grammar(ruleMap);
}
```

Preconditions:

- The given Report objects must follow the formats of TokenType specification files and Grammar specification files, respectively.

Appendix A - Class Overview

Here is a list of all of the classes used in this project, organized by package. For a more in depth breakdown of these classes, please see Section 1 – Class Design.

Package	Class	Purpose
immutable	Grammar	Contains a set of Rule objects and a root Rule.
immutable	Language	Contains a map of names to TokenType objects and an Alphabet object.
immutable	Rule	Contains a name and a set of chains of Token Type and Rule objects.
immutable	State	Contains mappings of characters to sets of State objects and Character objects.
immutable	StateMachine	Contains an initial State object, a final State object, and a list of involved State objects.
immutable	TokenType	Contains a StateMachine and the Language object to which this TokenType belongs.
immutable	Report	Contains a list of Token objects produced from a document.
immutable	Token	Contains a string and its token type.
grammar	_Grammar	A mutable version of immutable.Grammar.
grammar	_Rule	A mutable version of immutable.Rule.
grammar.defaults	RegExGrammar	Initializes and statically exposes a Grammar used for parsing regular expressions.
grammar.defaults	SimpleGrammar	Constructs a simple grammar from a language.
language	_Language	A mutable version of immutable.Language.
language	_State	A mutable version of immutable.State.
language	_StateMachine	A mutable version of immutable.StateMachine.
language	_TokenType	A mutable version of immutable.TokenType.
language	Alphabet	A container that manages the range of characters available for any particular language.
language.defaults	RegExAlphabet	Statically exposes the default Alphabet for regular expressions.
language.defaults	RegExLanguage	Initializes and statically exposes a Language used for parsing regular expressions.
utilities	CharacterUtilities	Contains common character sets.
utilities	ConversionUtilities	Contains methods for converting from mutable to immutable versions of objects.
utilities	ErrorUtilities	Contains methods helpful for halting program execution in the event of an error.
utilities	FileUtilities	Contains methods common to file manipulation.
utilities	LogUtilities	Allows for toggling logging functionality.
utilities	StringUtilities	Contains common methods for string formatting.
generators	GrammarGenerator	Contains methods for the generation of Grammar objects.
generators	ReportGenerator	Contains methods for the generation of Report objects.
generators	LanguageGenerator	Contains methods for the generation of Language objects.
generators	TokenGenerator	Contains methods for the generation of Token objects.
	LangaugeParser	Contains the main method for running the Language Parser.
	GrammarParser	Contains the main method for running the Grammar Parser.

Appendix B - Functional Overview

Here is a list of all of the classes used in this project, organized by package. For a more in depth breakdown of these classes, please see Section 2 – Functional Design.

Package	Class	Method	Input	Output	Description
generators	TokenGenerator	generate	TokenType, StringBuffer	Token	The next token
generators	TokenGenerator	getTokenLength	TokenType, StringBuffer	int	The length of the next token
generators	GrammarGenerator	generate	Language	Grammar	A simple default grammar
generators	GrammarGenerator	generate	String, String	Grammar	A specified grammar
generators	ReportGenerator	generate	Grammar, String	Report	The tokens of the given string
generators	LanguageGenerator	generate	Report	Language	A specified language

Appendix C – Specification Formats

Regular Expression

The format for Regular Expression specifications must conform to the following grammar.

```
<reg-ex> -> <rexp>
<rexp> -> <rexp1> <rexp’>
<rexp’> -> $UNION <rexp1> <rexp’> | $EPSILON
<rexp1> -> <rexp2> <rexp1’>
<rexp1’> -> <rexp2> <rexp1’> | $NEW_LINE <rexp1’> | $EPSILON
<rexp2> -> $OPEN_PARENS <rexp> $CLOSE_PARENS <rexp2-tail> | $RE_CHAR <rexp2-tail> | <rexp3>
<rexp2-tail> -> $STAR | $PLUS | $EPSILON
<rexp3> -> <char-class> | $EPSILON
<char-class> -> $DOT | $OPEN_BRACKET <char-class1> | <defined-class>
<char-class1> -> <char-set-list> | <exclude-set>
<char-set-list> -> <char-set> <char-set-list> | $CLOSE_BRACKET | $EPSILON
<char-set> -> $CLS_CHAR <char-set-tail>
<char-set-tail> -> $DASH $CLS_CHAR | $EPSILON
<exclude-set> -> $CARROT <char-set> $CLOSE_BRACKET $IN <exclude-set-tail>
<exclude-set-tail> -> $OPEN_BRACKET <char-set> $CLOSE_BRACKET | $DEFINED_CLASS
```

Grammar

The format for Grammar specifications must conform to the following grammar.

```
<root> -> <rule-def-list>
<rule-def-list> -> $RULE $RULE_ASSIGN <rule-term> <rule-term-tail> <rule-def-list-tail>
<rule-def-list-tail> -> $NEW_LINE <rule-def-list> | $EPSILON
<rule-term-tail> -> <rule-term> <rule-term-tail> | $EPSILON
<rule-term> -> $RULE | $TOKEN_TYPE
```