Software Re-Engineering Refactoring Project

Name	Roll Number	Section
Syed Farhan Jafri	21L-6074	BSE-8A
Haider Khan	21L-6067	BSE-8A
Muhammad Abdullah	21L-6101	BSE-8B

Chosen Application

Jison / Documentation
Jison takes a context-free grammar as input and outputs a JavaScript file capable of parsing the language described by that grammar. You can then use the generated script to parse inputs and accept, reject, or perform actions based on the input. If you're familiar with Bison or Yacc, or other
https://gerhobbelt.github.io/jison/docs/

A program to convert a context-free grammar to a Java-script based parser generator on the same pattern as Bison. Technology: Javascript / Node

Introduction

This document outlines the comprehensive refactoring of the Jison parser generator codebase. The original codebase contained approximately 2,500 lines of code across a small number of files, with significant portions of intertwined functionality. Our refactoring aimed to improve code organization, maintainability, and readability while preserving all existing functionality.

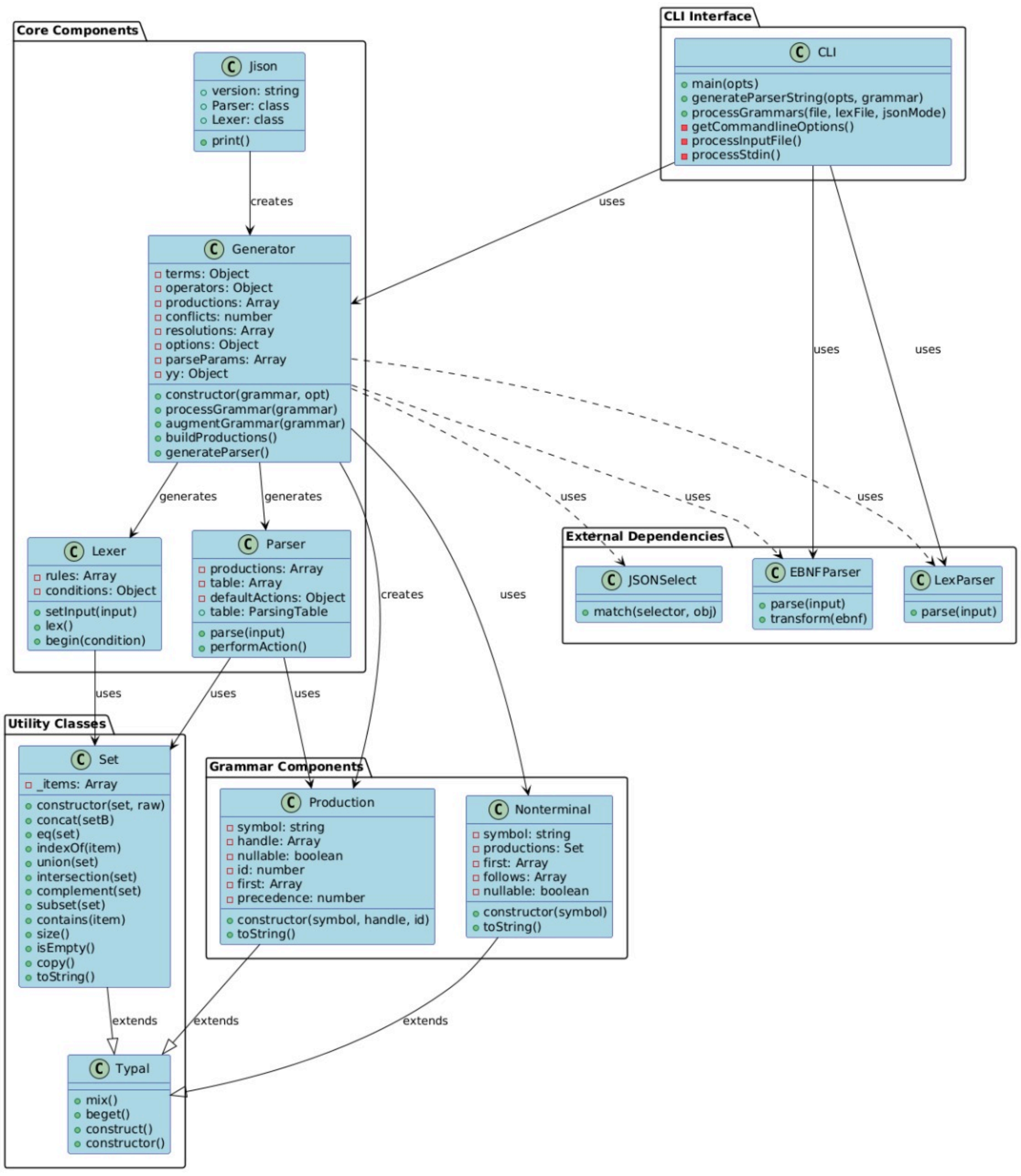
Directory Structure

Before Refactoring

The original structure was quite flat, with most of the code concentrated in a few large files:

```
lib/
├── cli.js
├── jison.js
└── util/
    ├── set.js
    └── typal.js
```

Class Diagram Before Refactoring

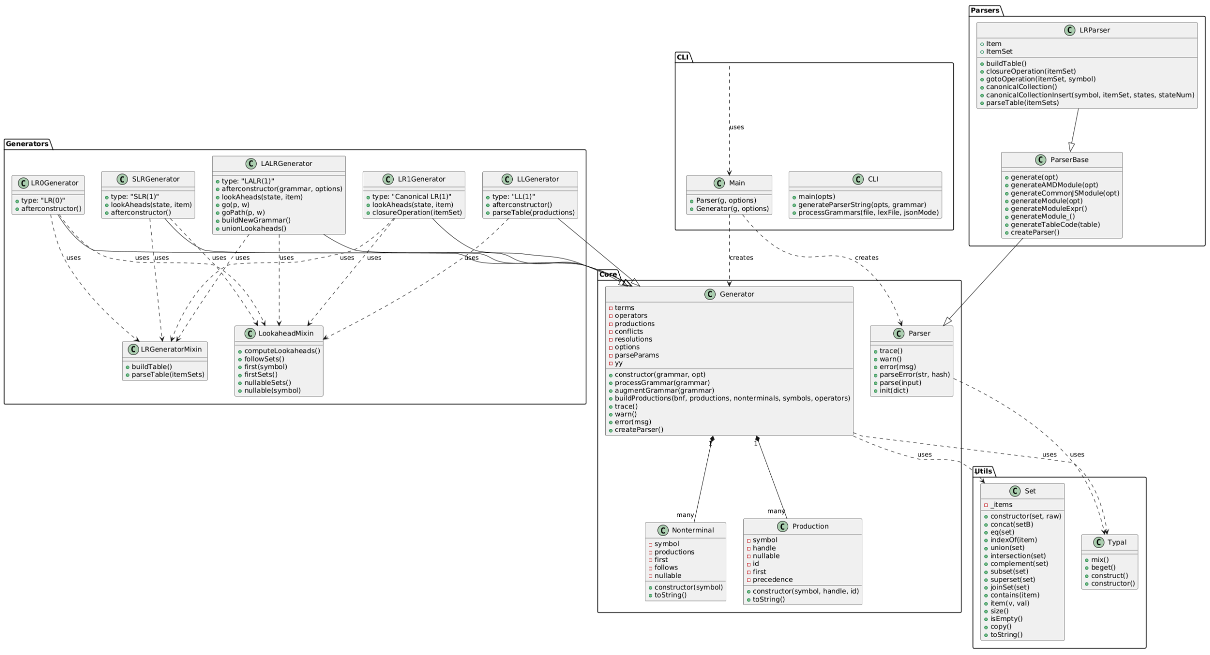


After Refactoring

The refactored structure is more modular and follows better separation of concerns:

```
lib/
├── cli.js              # Command-line interface
├── index.js            # Main entry point
├── core/
│   ├── Generator.js    # Base generator functionality
│   └── Parser.js       # Core parser implementation
├── generators/
│   ├── LR0Generator.js   # LR(0) Parser Generator
│   ├── SLRGenerator.js   # SLR Parser Generator
│   ├── LALRGenerator.js  # LALR Parser Generator
│   ├── LR1Generator.js   # LR(1) Parser Generator
│   └── LLGenerator.js    # LL Parser Generator
├── parsers/
│   ├── base.js         # Common parser methods
│   └── LRParser.js     # LR parser implementation
└── utils/
    ├── Set.js          # Set utility class
    └── Typal.js        # Object inheritance utilities
```

Class Diagram After Refactoring



Refactoring Strategies Applied

We have a separate file called “Jison Test Cases Report” which contains details of each and every initial test case run and checked.

1. Composing Methods

Extract Method

- **Before**: Large methods in jison.js handling multiple responsibilities.
- **After**: Extracted smaller, focused methods in appropriate modules.
- **Example**: Separated table generation logic from grammar processing in the Generator class.
- **Benefit**: Improved readability and easier maintenance of individual methods.

Inline Method

- **Before**: Some tiny helper methods adding unnecessary indirection.
- **After**: Inlined these methods where they were only used once.
- **Benefit**: Reduced unnecessary complexity without sacrificing readability.

Replace Temp with Query

- **Before**: Many temporary variables used for intermediate calculations.

- **After**: Replaced with direct query methods where appropriate.
- **Example**: In LRParser, direct function calls replaced some temporary state variables.
- **Benefit**: Reduced variable scope and improved encapsulation.

2. Moving Features Between Objects

Move Method

- **Before**: All parser functionality resided in jison.js.
- **After**: Moved methods to appropriate classes based on responsibility.
- **Example**: Moved parser-related methods to Parser.js and generator-related methods to Generator.js.
- **Benefit**: Better organization and separation of concerns.

Extract Class

- **Before**: All generator types were defined in a single file.
- **After**: Created separate classes for each generator type (LR0, SLR, LALR, LR1, LL).
- **Example**: Created LALRGenerator.js for LALR-specific functionality.
- **Benefit**: Better maintainability and possibility for specialized enhancements.

3. Organizing Data

Replace Array with Object

- **Before**: Some data structures used arrays with special indexing semantics.
- **After**: Used objects with clear property names.
- **Benefit**: Improved code clarity and reduced the chance of index-related bugs.

4. Simplifying Conditional Expressions

Decompose Conditional

- **Before**: Complex if-else chains for parser type selection.

- **After**: Simple switch statement with clear cases in index.js.
- **Benefit**: Easier to understand and modify.

Consolidate Conditional Expressions

- **Before**: Repeated similar conditions checking parser types.
- **After**: Unified conditions in a single place.
- **Example**: Parser type selection in the Generator factory method.
- **Benefit**: Reduced code duplication and centralized logic.

Replace Conditional with Polymorphism

- **Before**: Large conditional blocks to handle different parsing strategies.
- **After**: Each parser type implements its own strategy in its class.
- **Example**: Each generator class defines its own lookAheads method.
- **Benefit**: New parser types can be added without modifying existing code.

5. Making Method Calls Simpler

Replace Constructor with Factory Method

- **Before**: Direct constructor calls with complex parameters.
- **After**: Factory method in index.js that handles the complexity.
- **Example**: Jison.Generator factory method that creates the right generator type.
- **Benefit**: Simplified client code and centralized creation logic.

6. Dealing with Generalization

Form Template Method

- **Before**: Similar process repeated in different parser generators.
- **After**: Common template method in base class with specializations in subclasses.
- **Example**: The base process for building parse tables is now shared.
- **Benefit**: Reduced duplication while allowing specialization.

Extract Superclass/Interface

- **Before**: Duplicate code across generator implementations.
- **After**: Common functionality moved to base generator class.
- **Benefit**: Better code sharing and consistent interface.

Class Relationships

The refactored design establishes a clearer inheritance hierarchy:

1. **Generator Classes**:
   - Base Generator provides common functionality
   - Specialized generators (LR0, SLR, LALR, LR1, LL) inherit and extend
2. **Parser Classes**:
   - Base Parser provides parse functionality
   - Specialized parsers implement specific algorithms
3. **Utility Classes**:
   - Set provides collection utilities
   - Typal provides object inheritance tools

Challenges Encountered

During refactoring, we encountered several challenges:

1. **Circular Dependencies**: Resolving interdependencies between modules required careful restructuring.
2. **Mixed Responsibilities**: The original code had significant intertwining of responsibilities, making clean separation difficult.
3. **Maintaining Compatibility**: Ensuring the refactored code produces identical parsers was challenging.

Benefits of Refactoring

1. **Improved Maintainability**: Smaller, focused modules are easier to update.
2. **Better Extensibility**: New parser types can be added with minimal changes to existing code.

3. **Enhanced Readability**: Clearer structure makes the code easier to understand.
4. **Reduced Coupling**: Modules have clearer responsibilities and less interdependence.
5. **Preserved Functionality**: All original capabilities remain intact.

Task Distribution List

Task	Assigned To
Class Diagrams	Farhan
Analysis of Source Code	Farhan/Haider
Report Compilation	Farhan
Initial Test Cases	Farhan/Haider
Complete Refactoring	Haider/Abdullah
Revised Test Cases	Haider/Abdullah

Conclusion

The refactoring of the Jison codebase has transformed a monolithic structure into a modular, maintainable system while preserving all functionality. The new architecture follows sound software engineering principles and will be easier to maintain and extend in the future.