CS 2112 | Assignment 4 Overview

Parsing and Fault Injection

Charles Qian {cq38}

Kelly Yu {kly24}

1 Our Main Class

Our Main class is named Parse. Included in the compressed and submitted file is Parse.jar, which can be run from the command line.

2 Summary

We ran into multiple challenges in this assignment, mostly with Fault Injection, and some minor complications with the Parsing. The main challenge for this assignment was choosing a design and being consistent with the chosen design. For our implementation, our fault injection will mutate, but on occasion it will mutate "incorrectly" while still producing a legal rule. Another major challenge was adjusting to the pace of a team project.

3 Specification

Finished Specification: Parse.jar will take in a critter program, parse it, and print the program neatly to a standard output. Parse.jar can also apply a set number of guaranteed mutations to the input program, parse it, and return it neatly.

4 Design and Implementation

4.1 Classes and Architecture

Interfaces

- Condition
  Implemented by: BinaryCondition, Relation
  Responsibilities: Represents the different Conditions possible in a Critter Program.
- Node
  Implemented by: Expression, AbstractNode
  Extended by: Condition
  Responsibilities: Represents the Nodes of the AST of a Critter Program.
- Parser
  Implemented by: ParserImpl
  Responsibilities: Parses the Critter Program

Abstract Classes

- Expression
  Extended by: Num, BinaryExpression, SensorMem
  Responsibilities: Represents a value in the Critter Program that has an integer value.

Classes for Sharing Code

- AbstractNode
  Extended by: BinaryCondition, Expression, Program, Relation, Rule, Update
  Responsibilities: Facilities the sharing and organization of code between the different Nodes of the AST.

- Mutation
  Extended by: RuleSetMutation, AttributeMutation
  Responsibilities: Handles the probability of Mutations and calls an AttributeMutation or RuleSetMutation with equal probability.

Enum Types

- Action
  Responsibilities: Represents the different possible values of Action.

- BinaryConditionOperator
  Responsibilities: Represents the possible operations between instances of BinaryCondition.

- BinaryOp (inside BinaryExpression)
  Responsibilities: Represents the possible operations between classes that extend Expression.

- RelOperator
  Responsibilities: Represents the possible operations between instances of Relation.

4.2 Code design

We used a large amount of recursion for methods that calculated size, methods that called mutate, and methods that called printing. These recursive calls would start at the top of the tree or at the top of a subtree, and call its child nodes until it hit a terminal node, and then return to the top.

For data structures, we mainly employed the use of ArrayLists, with some help from LinkedLists. Our tree is mostly binary, with Arraylists to store multiple children (as with Updates and Rules. We used LinkedLists to keep track of parents while traversing down the tree during a recursive mutation call. A single ArrayList is used to store Critter memory in the class Critter.

4.3 Programming

We implemented our AST top-down, starting with the root Node, Program. Challenges we encountered while coding was how to abstract the tree and use interfaces effectively in our design. We did not realize the ability of the given consume method until very late in the project and decided to not incorporate it. This may have led to more complicated or lengthy code.

We also were unsure of how to separate the different types of mutations while only calling one mutate method present in each class that extended AbstractNode. It was also a challenge to coordinate schedules as Charlie's laptop broke a week into the project, forcing the team to meet in 24-hour labs. Working as a team to maximize efficiency was initially a problem - the original plan was to both work on Parser, but eventually we decided to divide and conquer.

Both of us designed and discussed the AST and class structure. Charlie implemented the main program, command-line handling, pretty-printing, and the Parser class. Kelly focused on accurately using git (due to the large quantity of merge conflicts early on in the project), performing fault-injection, doing the written problem, and writing overviews. We both took part in writing tests and debugging.

5 Testing

5.1 Testing Strategy

We incrementally tested our Parser during its creation. As for Fault Injection, it was not until after both the size methods and the mutation methods appeared correct that we started to create test cases. The tests for Parse seem to be complete, while the tests for Mutate are not as substantial. We found that many of our mutations would return incorrect mutated code that were still legal Programs.

5.2 Test Results

Tests for PrettyPrint and the Parser work some of the time, whereas the tests for Fault Injection only pass randomly.

6 Known Problems

- We added an extra class ActionSwitch, in the early stages of the project, just to realize later that the functionality it was supposed to present was not in the scope of the project.

- For Design, we feel in hindsight that perhaps Mutation should be an interface, with classes that implement it.

- For Fault Injections, we guess that our probability models may not be accurate, as the Fault Injections only work part of the time.

- When randomly choosing a Condition, as required by some of the mutations, we only pick from Conditions that are children of Rules, rather than searching the entire tree for Conditions that may not be children of Rules (e.g. may be children of Relations).

- We have a LinkedList to keep track of the traversal down the tree and call functions to get the first node (assuming it is a Program) and the last node (assuming it is the most recent parent). These assumptions have not been tested, and could cause errors.

- The way we call mutation methods could have been simplified by being more direct instead of using a RuleMutation or AttributeMutation class.

7 Comments

7.1 Time

For the both of us combined: We spent about 6-8 hours attending office hours; 3 hours designing the AST, 14-18 hours coding, and 6-8 hours testing/debugging.

7.2 Thoughts

The assignment surprisingly took a lot of thought in our part before we even started coding. Looking back, we could have benefited from even more time thinking before coding.

The hardest part of the project for us was the Fault Injection. We had to reread the documentation for the mutations multiple times before we had a grasp over what was expected from us.

Overall, this was our favorite project so far. It was very satisfying to see prettyPrint work, and seeing our program mutate. We appreciated the provided Tokenizer, but feel that the quantity of provided interfaces/classes limited our creative thinking in other ways to structure the project.

If we could do the project again, we would change it by providing less interfaces/classes.