CS 2112 Fall 2020
Assignment 4
Parsing and Fault Injection

Due: Tuesday, November 3, 11:59PM
Design Document due: Sunday, October 18, 11:59PM
Groups must be formed by: Thursday, October 15, 11:59PM

This assignment requires you to build a **parser** for a simple language, a **pretty-printer** that can print out parsed programs in a nice format, and a **fault injector** that mutates these parsed programs into other legal program representations.

1 Changes

- The AST is covered in 6.1, not 7.1. Fixed this typo in the design doc section.
- It is not required that mutations are able to transform any program into any other program.

2 Instructions

2.1 Grading

Solutions will be graded on design, correctness, and style. A good design makes the implementation easy to understand and maximizes code sharing. A correct program compiles without errors or warnings, and behaves according the requirements given here. A program with good style is clear, concise, and easy to read.

A few suggestions regarding good style may be helpful. You should use brief but mnemonic variable names and proper indentation. Keep your code within an 80-character width. Methods should be accompanied by Javadoc-compliant specifications, and class invariants should be documented. Other comments may be included to explain nonobvious implementation details.

2.2 Final project

This assignment is the first part of the final project for the course. Read the Project Specification to find out more about the final project and the language you will be working with in this assignment. The faults to be injected correspond to the mutations in §11 of the Project Specification.

2.3 Partners

You will work in groups of two or three for this assignment. Find your partner(s) as soon as possible, and set up your group on CMS so we know who has a partner and who does

not. One person has to invite the others and the others have to accept. Piazza also has support for soliciting partners. If you are having trouble finding a partner, ask the course staff, and we will try to match you up with someone in a fair way.

You must form a group on CMS on or before Thursday, October 15th. If you have not joined a group by this point, we will randomly assign you a group, just as in A3. You will be working with this group for the remainder of the semester.

Once you have formed a group on CMS, make a private Piazza post containing you and your partner(s) NetIDs and we will provide you with a repository pre-populated with the release code.

Remember that the course staff are happy to help with problems you run into. Read all Piazza posts and ask questions that have not been addressed, attend office hours, or set up meetings with any course staff member for help.

2.4 Restrictions

You are permitted to use any standard Java libraries from the Java SDK. However, use of a parser generator (e.g., CUP) is prohibited.

The method signatures and specifications of the following classes and interfaces may not be altered:

- `ast.Mutation`
- `ast.MutationFactory`
- `ast.Node`
- `ast.Program`
- `parse.Parser`
- `parse.ParserFactory`
- `main.ParseAndMutateApp`

Additions to these files or changes to the existing class hierarchy are allowed, provided they do not break the subtype relationships among the classes and interfaces listed above. In particular, in the release code you will see some other classes and interfaces in addition to those above. These are given as a suggested organization, but you may change them if you wish.

You may not alter the command line interface used in `ParseAndMutateApp`.

3 Design overview document

We expect your group to submit an overview document. The Overview Document Specification outlines our expectations. Writing a clear document with good use of language is important. Note this document is far more extensive than the ones you wrote for A2 and A3.

We require that you submit an early draft of your design overview document in advance before the assignment due date. Your design and testing strategy might not be

complete at that point, but we would like to see your progress. Feedback on this draft will be given promptly after the overview is due.

These are key topics to cover in your design overview document:

- What are the key data structures you will use for this assignment? In particular, how will you represent an AST (*abstract syntax tree* – see §6.1) and what data structures will you use? Having a reasonable AST design early on will be important for success. The need to support both pretty-printing and mutation will create some design challenges.
- What are the key algorithms you will need? Which ones will be challenging to implement, and why?
- What will be your implementation strategy, and how will you go about dividing responsibilities between the group members?
- What will be your testing strategy to cover the wide range of possible inputs and the different kinds of functionality you are implementing?

4 Version control

As with A3, you will be using Git. Although you might have safely gotten along without it up to now, it will be absolutely essential for A4-A6 given the scope of the project and number of collaborators. Using a version control system provides many benefits, including allowing you to easily compare your current code to any previous version. This is especially helpful if you introduce a bug and need to find out when it was introduced. It also allows teams to easily collaborate without emailing code back and forth, which quickly gets very confusing as it is hard to keep track of who has what version. You must submit a file `log.txt` that lists your commit history from your group. This is not extra work, as Git already provides this file for you.

While it requires some learning to understand all the features, version control is a most valuable skill to have. In the short term, you will reap the benefits as you delve further into the final project. In the long run, any large piece of modern software is always managed with version control.

4.1 Git Review

Here is a brief review of some useful Git commands:

1. `git status`: shows you the status of your repository, and indicate which files have changed since the last commit.
2. `git add <File1> <File2> <etc>`: adds the listed files to the *staging area*. The staging area holds files which you are about to commit, but which are not tracked yet. There are many useful options, such as -A, which adds all changed files.
3. `git commit`: saves all the currently staged changes as a new commit. Once you commit files, you will be able to browse through their states at previous commits. It's a good idea to commit code once you get it working. Then, if you break it later, you can look at the old version and see what you changed. Git stores a message with each commit.

It is good practice to have a short description, followed by a blank line, followed by a detailed description of what you changed. You can use the `-m` flag to provide a description on the command line. Note that this command only commits the changes to your local repository. Your partner(s) will not be able to see them until you push to the remote repository.

4. `git push`: pushes all your changes to your remote repository.
5. `git pull`: fetches code from the remote repository and merges it into your code. Use this command to pull changes your partner(s) have made and pushed to the remote repository.

When you pull and push, you might run into *merge conflicts*. These occur when team members have simultaneously changed the same parts of the same files. In this case, you will have to modify the file manually, then commit and push or pull again.

Another important command is `git checkout`, which is how you navigate to previous commits. You can use `git log` to view a list of all commits, and their associated hashes. Once you have the hash of a commit you want to checkout, run `git checkout <commit SHA-1 here>` to revert back to that commit. Run `git checkout master` to return to the normal state once you are done.

In order to produce your `log.txt`, you must first open Git in a terminal. If you are already using Git from the command line, you have already done this. If you are on GitHub Desktop, select `Repository -> Open In` and select your operating system's terminal (Terminal, Command Prompt, Power Shell, etc). Type the command `git log > log.txt` to print the Git log to a file called `log.txt` in your current directory.

For more details, see the Git documentation.

5 Getting started

5.1 Importing Your Project

To get started, you will want to clone your git repository, which we have already set up for you. To do this, look for the green `Clone or download` button on the right. Once you click on this, a URL should pop up. Copy this URL, and then pull up your terminal and navigate to whatever folder you would like to download your repository to. Then, run the following command:

```
git clone <your url here>
```

Alternatively, if you use GitHub Desktop, go to File/Clone Repository, and select your repo from GitHub Enterprise Server. Enter the local path of the location you want to clone to. Your project should be downloaded automatically.

Once you've cloned your project into your Eclipse workspace folder, you will need to change the name of the project in your settings.gradle file. Right now, this file lists the project name as CHANGE ME. It is key that you update this to match the name of the folder containing your project. This will also be the name of the project in Eclipse.

One of the TAs has been kind enough to make two tutorial videos on importing your project. Follow these links for the Mac and Windows instructions.

We have already provided a `.gitignore` file that ignores files generated by Gradle. This prevents generated files, and files that don't really matter, from being tracked by Git.

Now that you have downloaded and set up your git repository, import your project as a Gradle project in Eclipse. Right click in the project explorer, then click `Import`. `Select Gradle -> Existing Gradle Project`.

Note: Gradle, Git, and general project setup have also been covered in Lab.

6 Parsing

Parsing involves converting an input text, such as a program, into an internal tree structure according to a grammar. For example, the Java compiler includes a parser that converts Java programs you write, which are just strings, into an internal form called an *abstract syntax tree* (AST). You will apply this same idea to parsing a program written in a critter language into an internal AST representation that your program can understand, execute, and modify.

A *grammar* for a language gives a concise formal specification of the syntax of the language, including all the tokens that are part of the input. The parser must accept all and only input strings that are valid according to the grammar. The job of the parser is to convert a valid input string to an AST.

6.1 Abstract syntax trees

An abstract syntax tree (AST) is called *abstract* because it ignores differences in inputs that do not affect meaning. As a result, different inputs with different *concrete syntax trees* (parse trees) but the same meaning will have the same AST. For example, the expressions `(2+3*4)`, `2+(3*4)`, and `(2) + (3)*(4)` have different concrete syntax trees, but the same AST, because parentheses are only there to guide the construction of the AST. Figure 1 shows this AST along with the concrete syntax tree (parse tree) for `2+(3*4)`. The AST is shown on the left in two different forms: the top represents how we might think of the AST, while the bottom corresponds more closely to the code and might help with your AST implementation.

The abstract syntax tree omits any unnecessary syntax, which makes it different from a concrete syntax tree (parse tree). This distinction becomes critically important when you implement fault injection. Fault injection will be much more difficult if your abstract syntax tree has concrete parse tree nodes such as parentheses, or nonterminals that exist only to represent different levels of precedence.

You will need to design and implement a class hierarchy to represent this tree, in which element types are subclasses of `Node`. By giving `Node` the appropriate methods, various useful kinds of functionality, including fault injection in this assignment and evaluation in a later assignment, can be implemented recursively.

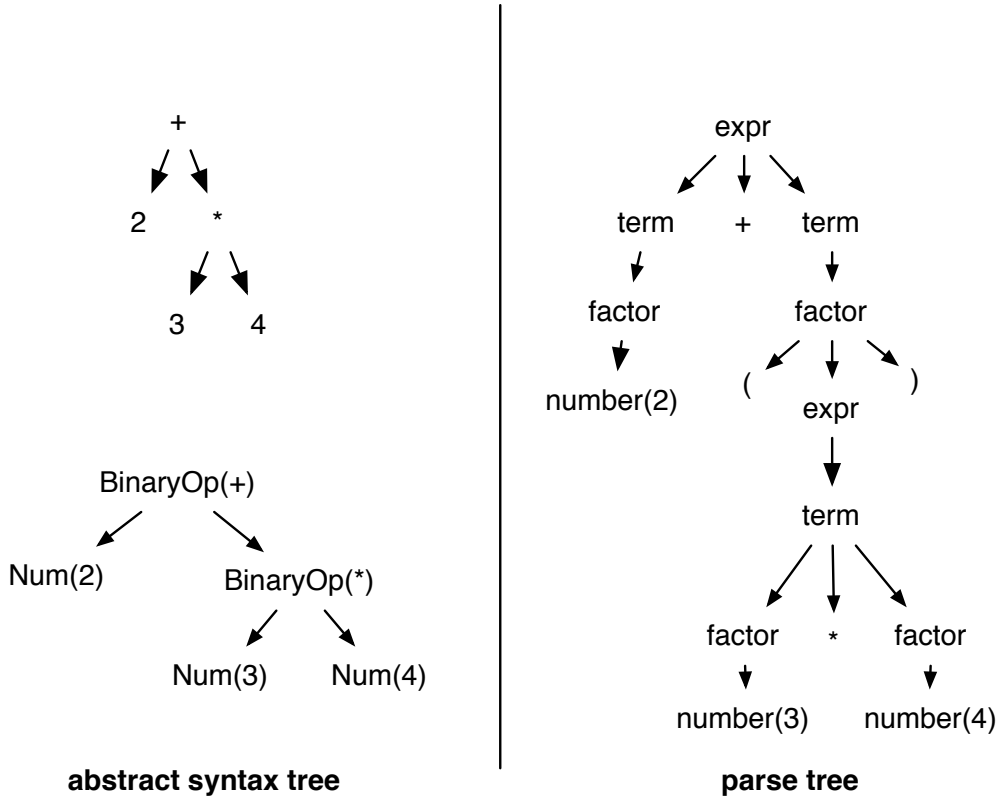

Figure 1: Abstract and concrete syntax trees for `2+(3*4)`

Valid	Invalid

Rules:
`1 > 0 --> mem[0] := 10`
`mem[1] := 4`
`mem[5] := 3;`
 	
`1 > 0 --> mem[0] := 10`
`mem[1] := 4`
`mem[5] := 3`
`wait;`
 	
Conditions:
`1 > 2 and { 3 <= 4 or 5 = 6 }`
 	
Expressions:
`(1 + 2) * 3`
	Rules:
	`1 > 0 --> mem[0] := 1`
	`forward`
	`attack;`
	
	(Two actions instead of one.)
	Conditions:
	`1 > 2 and (3 <= 4 or 5 = 6)`
	
	(Curly braces should be used.)
	`mem[1] > 3 { or mem[2] < 4 }`
	
	(`or` should be before `{`.)
	Expressions:
	`{ 1 + 2 } * 3`
	
	(Parentheses should be used.)

Table 1: Examples of valid and invalid critter terms

6.2 Critter grammar

The grammar for the language to be parsed is given in the Project Specification. Please see the course staff in office hours or use Piazza if you have any question about the grammar.

Table 1 shows several valid and invalid terms in the critter language. An example of a valid critter program can be found in the Project Specification.

6.3 Implementation

AST interfaces and some AST classes are provided to help you with defining your AST. You will need to add more data structures to represent the entire critter language. A skeleton of `ParserImpl`, an implementation of `Parser`, is also provided as a guideline and can be used in the `ParserFactory`. Finally, a nearly complete implementation of a `Tokenizer` is given; however, you will find that it does not support Java-style "`//`" comments that extend to the end of a line in critter programs, such as

```
POSTURE != 17 --> POSTURE := 17; // we are species 17!
```

Extend `Tokenizer` to handle this comment syntax to help critters understand themselves more easily.

7 Fault injection

Compilers operate on source code to produce compiled code. Bug finders operate on source code to produce lists of possible bugs detected. Testing such software requires a lot of programs as test cases, but writing a lot of programs is expensive. Fault injection is a cost-effective technique for generating many programs as test cases. The idea is to make small random changes to a valid program to produce many useful test cases.

Compiler testing requires not only test cases that are valid programs in the programming language, but also ones that are invalid programs. Finding bugs, on the other hand, requires test cases that are valid programs containing bugs. In this assignment, you will build the latter kind of fault injector, in which a valid program is transformed into another valid program. For the final project, mutations on the behavior of a simulated critter can be implemented using fault injection. The Project Specification defines possible kinds of genome mutations.

7.1 Examples

The critter programs in the directory `src/test/resources/files` demonstrate several steps of mutating a critter. The program `mutated_critter_1` is the result of mutating the program `unmutated_critter` with Rule 1. The program `mutated_critter_2` is the result of mutating `mutated_critter_1` with Rule 2, and similarly for the remaining programs in the sequence.

7.2 Implementation

There are six types of mutations. You are to implement all six.

There is some flexibility in the interpretation of the mutation rules. Identify any ambiguities you see and explain how you have resolved them. However, *do not use reflection* (except `getClass()`, allowed in comparisons). There are good use cases for Java reflection, and this is not one of them. Using reflection will result in code which is hard to read, hard to debug, and probably doesn't work.

We have provided you with a number of useful methods to make mutations easier. Before you begin implementing any mutations, be sure to understand which mutations can apply to different types of nodes.

There are many different kinds of nodes in the AST. Implementing mutation for each of them could involve a lot of tedious code and opportunities for mistakes. Think about how to abstract the various kind of mutations so that you can share mutation code across multiple node types. Can you create a common framework so that most mutation types can be implemented in a common way, rather than creating complex logic specific to each combination of node type and mutation type? You have a lot of flexibility on how to implement this.

8 Pretty-printing

You should be able to print out programs in the same syntax they were written in. That is, the printed program should yield the same abstract syntax tree when parsed again. Pretty-printing should use indentation and line breaks to make output readable and, well, pretty.

9 Testing

The testing resource directory `src/test/resources` contains several critter programs that you may use to test your parser and mutator. You are encouraged to come up with your own critter programs and add them to this directory as you test. There is also a sample unit test to serve as an example of how to load a critter file into your tests. Note that this test will fail until your parser is implemented.

You should include any tests and critter programs that you write along with your submission. When grading, the course staff will give feedback not only on the correctness of your program, but also on how you could improve your testing methodology to catch any bugs that remain in your submission.

10 User interface

Your program must be able to be run from the command line as follows:

- `java -jar <your_jar> <input_file>`
  Parse the file `input_file` as a critter program and pretty-print the program to standard output if the program is valid.
- `java -jar <your_jar> --mutate <n> <input_file>`
  Parse the file `input_file` as a critter program and apply n mutations if the program is valid. After each mutation, print a description of the kind of mutation applied and pretty-print the resulting program.

We have supplied a skeleton command line interface for you to use. You are free to make any changes you want to this class, so long as you continue to implement the preceding command line interface.

11 Written problems

11.1 Trees

1. Draw the AST for each of the rules in the file `draw_critter.txt` provided with the assignment (located in `src/test/resources/files`).

11.2 Loop Invariants

2. Give a loop invariant for the following piece of code, which finds both the minimum and maximum value of an even-length array `a` while using only $1\frac{1}{2}$ comparisons per element. Justify your answer by showing Establishment, Preservation, and Postcondition.

```
// Precondition: a.length is even
int max = Integer.MIN_VALUE,
    min = Integer.MAX_VALUE,
    i = 0;
while (i < a.length-1) {
    int x,y;
    if (a[i+1] > a[i]) {
        x = a[i];
        y = a[i+1];
    } else {
        x = a[i+1];
        y = a[i];
    }
    max = Math.max(max, y);
    min = Math.min(min, x);
    i += 2;
}
```

3. Recall the Polynomial class from project 1. Provide a loop invariant for the `while` loop in the `create()` function and use that loop invariant to argue that the loop is correct using that invariant.

12 Overview of tasks

Determine with your partner(s) how to break up the work involved in this assignment. Here is a list of the major tasks involved:

- Alter `Tokenizer` to support end-of-line comments
- Implement the `main` method and command-line interface.
- Design and implement a class hierarchy for representing abstract syntax trees.
- Implement the `Parser` interface to generate abstract syntax trees.
- Implement pretty-printing functionality as methods on AST nodes.
- Implement a class or classes to perform fault injection.
- Solve the written problems.

13 Tips

The key to success on this assignment is for all partner(s) to contribute equally and effectively. However, working with partners can add challenges. Some tips:

- Meet with your partner(s) as early as possible to work out the design and to discuss the responsibilities for the assignment. Keep meeting and talking as the project progresses. Be prepared for your meetings. Be ready to present proposals to your partner(s) for what to do, and to explain the work you have done. Good communication is essential.
- You should do the written problems together. These questions are good practice for the final exam.
- The code we have given you to work with is just a guideline with some ideas in it to get you started. Feel free to add new methods and classes as you see fit.
- Mutation is significantly harder than it looks at first. You will probably have to design your own abstractions to implement the mutations. So it is tempting to partition the programming tasks on this assignment in a way that turns out to be quite unequal.
- Here is the best way to partition an assignment into parts that can be worked on separately. First, agree on what the different modules will be and exactly what their interfaces are. Include detailed specifications. In this assignment, the AST is a key data structure because it connects the parsing and mutation parts of the assignment. You should design it together as a team, including whatever methods it needs to support mutation.
- It will be tempting to wait on implementing mutation until after parsing is working. But this strategy will mean delaying the implementation of perhaps the hardest part of the assignment. Instead, you should agree on the AST interface so you can start working on mutation immediately. You can and should test your mutation algorithms on hand-crafted ASTs if the parser is not ready yet.
- Drop by office hours and explain your design to a member of the course staff as early as possible. This will help you avoid big design errors that will cost you as you try to implement.
- This project is a great opportunity to try out **pair programming**, in which you program in a pilot/copilot mode. It can be more fun and tends to result in fewer bugs. A key ingredient is to have the pilot/typist convince the other(s) that the code meets the predefined spec. It might be tempting to let the pilot/typist be the person who is more confident on how to implement the code, but you will probably be more successful if you do the reverse.
- This project is also a great time for **code reviews** with your partner(s). Walk through your code and explain to them what you have done, and convince them that your design is good. Be ready to give and to accept constructive criticism.
- Sometimes people feel that they are working much harder than their partner(s). Remember that when you go to implement something, it tends to take about twice as long as you thought it would. So what your partner(s) are doing is also twice as hard as it looks. If you think you are working twice as hard as your partner(s), you are probably about even!

14 Submission

You should submit these items on CMS:

- `overview.txt/pdf`: Your final overview document for the assignment. It should also include descriptions of any extensions you implemented.
- Zip and submit your `src` directory. This directory contains:
  - **Source code**: You should include all source code required to compile and run the project. All source code should reside in the `src` directory with an appropriate package structure.
  - **Tests**: You should include code for all your test cases, organized along with the sample tests released to you in the testing directory. You may create subpackages to keep your tests organized.

  Do not include any files ending in `.class`. The other files mentioned in this section should be uploaded separately to CMS.
- `log.txt`: A dump of your Git commit log.
- `written_problems.txt/pdf`: This file should include your response to the written problems.

Finally, do not make any changes to your `build.gradle` file. In future assignments we may allow you to add additional libraries, but for this assignment you should not make any changes.