CS 2112 Fall 2025

Assignment 3

Parsing and Fault Injection

Due: Thursday, October 23, 11:59PM
Design document due: Thursday, October 16, 11:59PM
Groups must be formed by: Friday, October 10, 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 Updates

Nothing yet.

2 Instructions

2.1 Grading

Solutions will be graded on design, correctness, testing, 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 to the requirements given here. A good test plan ensures good coverage of features and edge cases. 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. Spacing and bracket placement should be consistent - make use of IntelliJ's automatic formatting tools. 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 Teams

You will work in groups of two or three for this assignment. Find your teammates as soon as possible, and set up your group on CMSX so we know who has a team and who does not. One person has to invite the others and the others have to accept. Ed also has support for soliciting teammates. If you are having trouble finding a group, ask the course staff, and we will try to match you up with someone in a fair way.

You must form a group on CMSX on or before Friday, October 10th. If you have not joined a group by this point, we will randomly assign you a group. You will be working with this group for the remainder of the semester.

Once you have formed a group on CMSX, make a private Ed post containing you and your team'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 Ed posts and ask questions that have not been addressed, attend office hours, or set up meetings with any course staff member for help.

After each assignment, you will be asked to fill out a brief survey on CMSX providing peer evaluations for your teammates, and they will do the same for you. Peer evaluations will be reviewed by course staff and become a component of your grade.

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 A1 and A2.

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

Despite its learning curve, version control is an extremely 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.

You will be using Git to coordinate with your project group. Although you might have gotten along without it up to now, it will be absolutely essential for the remaining assignments given the scope of the project and number of collaborators. Using a version control system provides many benefits, including the ability to handle merge conflicts and track changes over time, such as finding where a bug came from.

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.

In order to create `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 using Git through IntelliJ, open the built-in terminal by selecting the terminal icon in the bottom left corner. Type the command `git log > log.txt` to print the Git log to a file called `log.txt` in your current directory.

If you do not have Git installed on your command line, you can download it from the official Git website.

For more details about Git, refer to Monday Lab 5 / Wednesday Lab 6 or the official Git documentation.

5 Getting started

First, create a group on CMSX under both the Assignment 3 and A3 Design Doc entries. Once your groups are formed, make a private post on Ed, and we will create your repository with the release code.

Next, clone your repository and open it with IntelliJ. To run your code, select Tasks, Application, Run under the Gradle pane. This should be the same process you followed for Assignment 2. For more instructions on Gradle projects in IntelliJ, refer to the IntelliJ website.

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.

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.

6.2 Critter grammar

The grammar for the language to be parsed is given in the Project Specification (– see §6). Please see the course staff in office hours or use Ed 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 (– see §16).

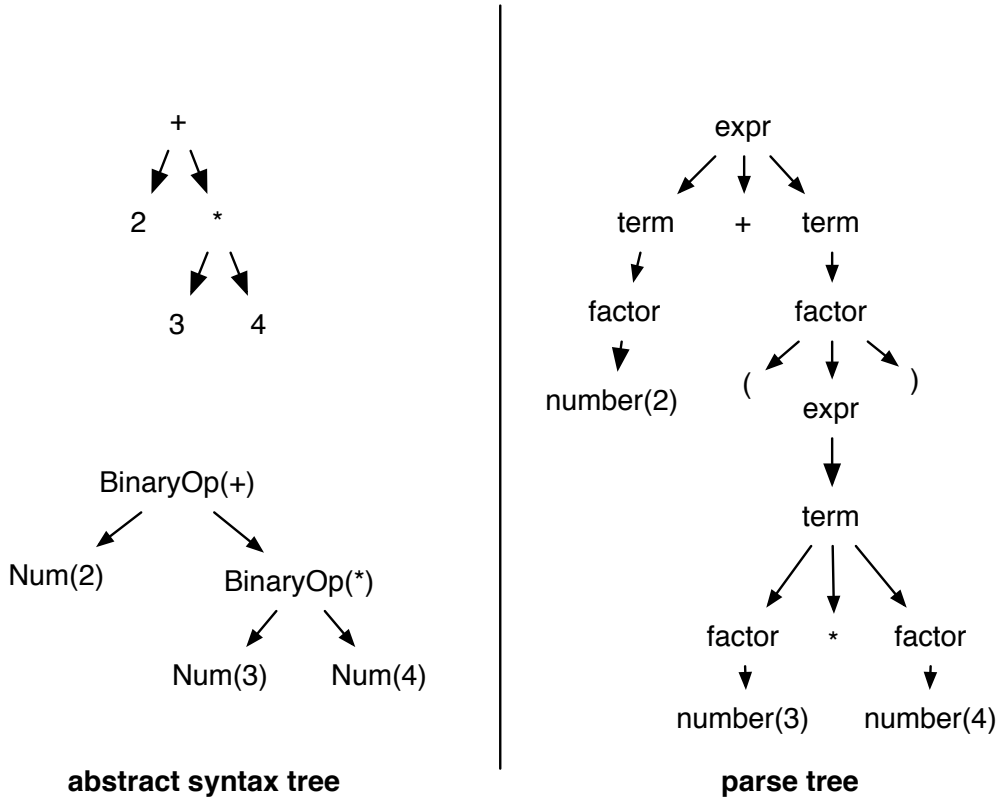

Figure 1: Abstract and concrete syntax trees for `2+(3*4)`

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 / mutations

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. The idea of mutating existing test inputs is also a core part of modern fuzzers.

In this assignment, you will build a 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 (– see §11) defines the possible genome mutations.

Some of the mutation specifications are open-ended, as not all AST designs will be entirely the same. As usual, if you find an ambiguity, make a reasonable design decision (and be sure to document it)!

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

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.

Note that many mutations are more complicated than they initially appear. For example, the replace mutation must be able to replace a node with any other valid subtree. This means that a number node not only can be replaced by any other number, for example, but it can also be replaced by a `smell` node!

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.

Try to remove redundant parentheses when pretty printing. For example, the AST in Fig. 1 should be printed as "`2+3*4`" instead of the inputted "`2+(3*4)`". If the two operators were swapped, "`2*(3+4)`" should be printed, not "`2*3+4`". Refer to Discussion 7 for more details.

9 Testing

A portion of your grade will be based on the quality and exhaustiveness of your test cases. Due to the open ended nature of this assignment, our test cases used when grading tend to test end-to-end functionality. This means that a small mistake anywhere in your project can cause a very large number of our tests to fail against your submission. For this reason, an exhaustive test strategy will be paramount to ensure your project is functional.

Writing good tests for this assignment is difficult and time-consuming, yet done well becomes invaluable for helping you complete the rest of the project. In our experience, successful teams spend a significant portion of their time planning and implementing their tests, so be sure to start testing early and allocate plenty of time towards writing good tests.

To help you, we've provided the following starter template for a test plan. **A submission that does not feature at least the tests described here will not receive full credit.** The testing resource directory `src/test/resources` contains several critter programs that you will use to test your parser and mutator. You will need to come up with more of your own critter programs and add them to this directory as you test. Examples of critter programs you must write include:

- Small critter programs with only one rule to test only one or a few nodes. You'll need to write at least enough programs to test every node type.
- Small critter programs with edge cases. The exact number and nature will depend on your AST design, but create programs that exploit as many edge cases as you can identify in your AST nodes.
- Small critter programs to compute many different types of expressions.
- Slightly larger critter programs with a rule that has multiple updates, or an update and an action, and every combination thereof.
- Critter programs with multiple rules.
- Invalid critter programs that should fail to parse.

With a bank of test critter programs written, you will then write JUnit test cases that parse those critter programs and check for correct behavior. Since ways of testing correctness for this assignment is less obvious than previous ones, we offer the following list of tests you need to write.

- **Examine Expected Nodes.** The exact layout of your nodes will depend on your design for your AST, but you should be able to compute, by hand, what you expect your AST to look like for a given program. Write test cases that parse programs and then, using the `nodeAt(int)` method inside the `Node` interface, extract and compare each individual node in the resulting AST to verify it matches what you expect.
- **Compare ASTs.** A parse of a pretty-printed AST should result in the exact same AST. For each critter program you wrote, you should be able to turn it into an AST, pretty-print that AST, and then parse the output into a second AST. Write a test that ensures the two ASTs are identical.
- **Compare Pretty-Printing.** While your pretty-printer may modify the exact contents of the string you originally parsed, the same AST should always pretty-print to the same string. Therefore, much like the last test, if you pretty-print once, parse the result, and then pretty-print it a second time, the two outputs should be identical.
- **Syntax Errors.** For the invalid programs, ensure that your parser correctly fails to parse them in ways you expect.

Mutations are even harder to test. For each mutation, start with the following tests:

- Create a small critter program for every node that would be valid for the mutation to change. Then, list all possible outputs of running the specific mutation on each program, and check to see if the output of your mutation is in that set.

- Create critter programs that contain only nodes that would not be valid for the mutation to change. Ensure your mutation does not modify the program.
- Create larger critter programs that contain multiple nodes which can be mutated. Enumerating every valid output becomes infeasible at this stage, but you can test that the result of your mutation at least still parses. See the pretty-print, parse, and pretty-print again bullet point from above. Do this in a loop, repeatedly applying mutations to the output from the last iteration and checking that it still parses.

This is **not** an exhaustive list, both for the parser and mutations, and a good testing strategy will include additional tests that are tailored to your specific class and AST design. Be sure to spend time discussing your test strategy as a group and in your overview document draft.

We have provided a sample unit test in the release code 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.

Remember that tests are code just like anything else you write. If you find yourself copy and pasting the same code over and over again between multiple tests, consider refactoring it out into a shared method or class that all the tests can use.

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 problem

11.1 Trees

1. Draw an AST representing the program in the file `draw_critter.txt` provided with the assignment (located in `src/test/resources/files`). Feel free to draw the individual AST for each rule if that is easier.

2. Write a critter program that causes the critter to compute the sum of the numbers 1 to 100 in `MEM[10]` and to then execute `wait` actions forever. The only numeric literals permitted in the program are `1` and `100`. You may assume the critter has enough memory to store the result and enough energy to carry on the computation. Recall that the critter language spec is available in the project handout.

12 Overview of tasks

Determine with your team 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.
- Design and write thorough test cases for your parser.
- Implement classes to perform fault injection (mutations).
- Design and write thorough test cases for your mutations.
- Solve the written problems.

13 Tips

The key to success on this assignment is for all team members to contribute equally and effectively. However, working with a team can add challenges. Some tips:

- **Read the entire assignment.** That's this entire document, and also the Project Specification on the course website. These are long and dense documents, but they contain answers to many common questions and a lot of useful information.
- **Meet with your team 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 teammates for what to do, and to explain the work you have done. Good communication is essential.
- A good way to partition an assignment into parts that can be worked on separately is to 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.
- **Test early and often.** Test cases are not something that can be put off until the end on this project, at least not without catastrophic consequences. Start writing test cases early and continue adding to your test suite as you go. You should start writing tests as soon as your AST design is done, even if you have not started code yet.
- 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.
- 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.
- 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.
- 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 teammates. 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 teammates. Remember that when you go to implement something, it tends to take about twice as long as you thought it would. So what your teammates are doing is also twice as hard as it looks. If you think you are working twice as hard as

your teammates, you are probably about even!

14 Submission

You should submit these items on CMSX:

- `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 hidden files like `.DS_Store` or `__MACOSX`, or compiler outputs such as files ending in `.class` or `.jar` or the `META-INF` folder. The other files mentioned in this section should be uploaded separately to CMSX.
- `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.