CS2112—Fall 2015
Assignment 4
Parsing and Fault Injection

Due: Thursday, October 15, 11:59PM
Design Overview due: Wednesday, October 7, 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.

0 Changes

- None yet; watch this space.

1 Instructions

1.1 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.

1.2 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 Section 10 of the Project Specification.

1.3 Partners

You will work in a group of two students for this assignment. Find your partner as soon as possible, and set up your group on CMS so we know who has a partner and who does not. Piazza also has support for soliciting partners. If you are having trouble finding a partner, ask the course staff, and we will try to pair you up with someone in a fair way.

Remember that the course staff is 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.

1.4 Restrictions

Use of any standard Java libraries from the Java SDK is permitted. Use of a parser generator (e.g., CUP) is prohibited, however.

The existing class hierarchy, 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`

Additions to these files are allowed as long as they do not break the existing structures.

2 Design overview document

Starting with this assignment, 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.

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 within 72 hours after the overview is due.

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

- Have you thought about the key data structures in this assignment? In particular, 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.
- Have you thought through the key algorithms and identified what will be challenging to implement?
- Have you thought about your implementation strategy and how you will go about dividing responsibilities between the group members?
- Do you have a testing strategy that covers the possible inputs and the different kinds of functionality you are implementing?

3 Version control

Working with a partner effectively is a key learning goal for this project. To facilitate this goal, we would like you to use version control in managing your partnership. You may choose to use any system you like; common industry standards include Git, Subversion, and Mercurial. At least one of these systems will be covered in lab. You must submit file `log.txt` that lists your commit

history from your group. This is not extra work, as version control systems already provide this functionality. While it may require some learning, using version control is a 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 Parsing

Parsing involves converting an input sequence of text, such as a program, into a tree structure according to a grammar. The Java compiler, for example, is a parser that converts programs you write into an executable form. You will apply this same idea to parsing a program written in a critter language into an internal abstract syntax tree representation that your program can understand, execute, and modify.

The grammar for a language describes the *concrete syntax* of programs, including all the tokens that are part of the input. Rather than to construct the concrete syntax tree, the job of the parser is to build an *abstract syntax tree*.

4.1 Abstract syntax trees

An abstract syntax tree (AST) represents the given input without parts that do not affect its meaning, inherent in the structure of the AST. As a result, different inputs that have the same meaning will have an identical AST. For example, the expressions `(2+3*4)`, `2+(3*4)`, and `(2) + (3)*(4)` have the same abstract syntax tree, because parentheses are only there to guide the construction of the tree. Figure 1 shows this abstract syntax tree 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 syntax that is unnecessary, 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` appropriate methods, various useful functionality, including fault injection in this assignment and evaluation in a later assignment, can be implemented recursively.

4.2 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.

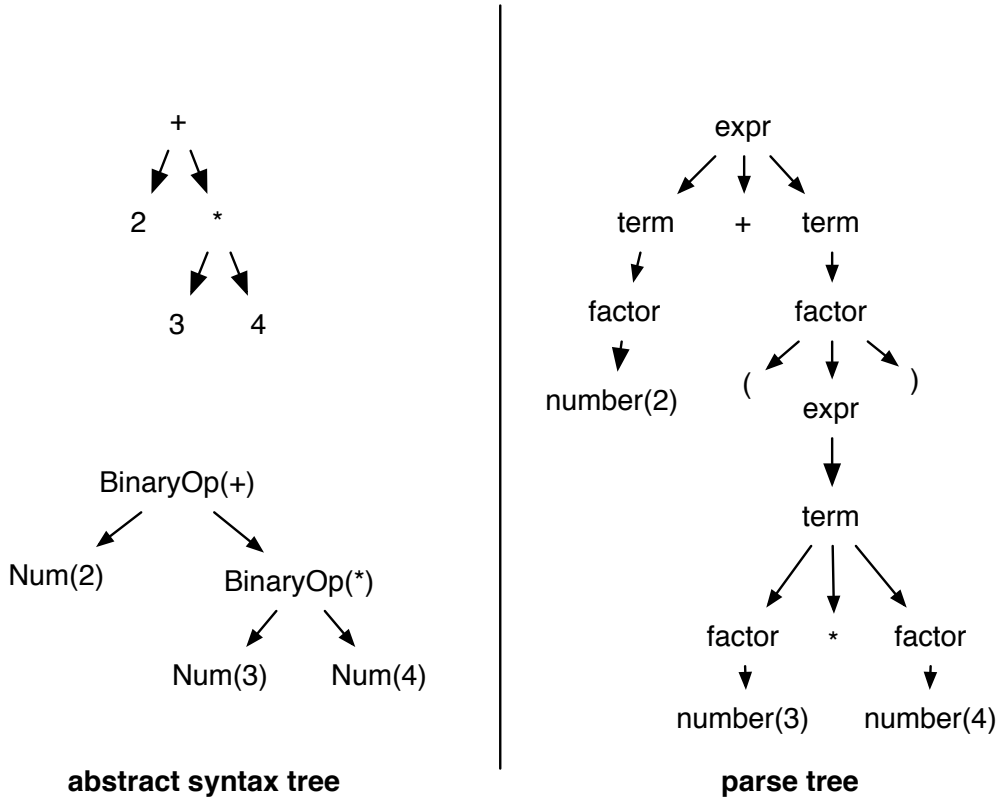


Figure 1: Abstract and concrete syntax trees for `2+(3*4)`

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

4.3 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 and help critters understand themselves more easily.

5 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.

5.1 Examples

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

5.2 Implementation

Some flexibility exists in how to interpret the mutation rules. Identify any ambiguities you see and explain how you have resolved them. One rule of thumb is that it should be possible to change a valid critter program into any other valid program through some sequence of mutations.

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.

6 Pretty-printing

You should be able to print out programs in the same syntax they were written in. That is, the printed programs 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.

7 User interface

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

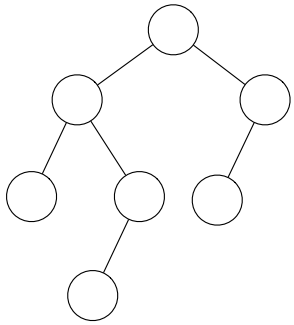
- `java -jar <your_jar> <input_file>`
  Parse the file `input_file` as a critter program and pretty-print the program to standard output if the program is valid.
- `java -jar <your_jar> --mutate <n> <input_file>`
  Parse the file `input_file` as a critter program and apply *n* mutations if the program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

8 Written problems

8.1 Trees

Wandering through Mirkwood, Bilbo and the dwarves fall into an Elvish oubliette deep in the ground. In the dim light filtering in from the forest above, Bilbo searches for a way to escape the prison.

1. A small, apparently dead tree sprouts from the dirt. However, Bilbo discovers that by chanting the even natural numbers starting from 0 (0, 2, 4, . . .) as he touches the nodes of this magical tree, he is able to make it come back to life and grow into a tree they can use to climb their way out. However, it only works if the numbers assigned to each node respect the binary search tree invariant. Show where on the tree Bilbo places each number. (The root is drawn at the top, as usual.)



2. The tree can only be climbed successfully by doing a preorder traversal of its nodes. Report the order in which the nodes are visited.

3. Bilbo can almost climb out but the tree is too short and unbalanced. He needs to add nodes for the numbers 5 and 15 to the binary search tree. Draw the resulting tree after the new nodes are inserted.

4. Draw the AST for each of the rules in the file `draw_critter.txt`, provided with the assignment.

8.2 Asymptotic complexity

Recall that a function $f(n)$ is $O(g(n))$ if there exist positive constants k and n_0 such that for all $n \geq n_0$, $f(n) \leq kg(n)$. The constants k and n_0 together are a *witness* to the fact that $f(n)$ is $O(g(n))$. For each of the following functions, either show that it is $O(n^2)$ by giving a witness, or show that it is not $O(n^2)$ by arguing that no such witness can exist.

5. $1000n$

6. $5n^2 + 10n - 10$

7. n^3

8. 2^n

9. $f_1(n) \times f_2(n)$, where each of $f_i(n)$ is $O(n)$.

9 Overview of tasks

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

- Altering `Tokenizer` to support end-of-line comments
- Implementing the `main` method and command-line interface.
- Designing and implementing a class hierarchy for representing abstract syntax trees.
- Implementing interface `Parser` to generate abstract syntax trees.

- Implementing pretty-printing functionality as methods on AST nodes.
- Implementing a class or classes to perform fault injection.
- Solving the written problems.

10 KARMA

KARMA questions do not affect your raw score for any assignment. They are given as interesting problems that present a challenge.

10. In the code below, let `a` be an array of integers. Show that the code satisfies the given loop invariants. What does the Postcondition part of the loop invariant argument tell you about what the code does?

```
1 int sum = 0, max = 0;
2 for (int i = 0; i < a.length; i++) {
3     sum += a[i];
4     if (sum < 0)
5         sum = 0;
6     if (sum > max)
7         max = sum;
8 }
```

Invariants:

$$0 \le \texttt{i} \le \texttt{a.length} \tag{1}$$

$$\texttt{sum} = \max_{0 \le k \le \texttt{i}} \left(\sum_{j=k}^{\texttt{i}-1} \texttt{a}[j] \right) \tag{2}$$

$$\texttt{max} = \max_{0 \le k \le \texttt{i}} \left(\max_{0 \le j \le k} \left(\sum_{\ell=j}^{k-1} \texttt{a}[\ell] \right) \right) \tag{3}$$

Assume that the empty summation is zero.

KARMA:

11 Tips

The key to success on this assignment is for both partners to contribute effectively. Working with a partner, however, may add challenges. Some tips:

- Meet with your partner 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 for what to do, and to explain the work you have done. Good communication is essential.
- One way to partition an assignment into parts that can be worked on separately is to agree on, first, what the different modules will be, and further, exactly what their interfaces are, including detailed specifications.
- 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 person 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. Walk through your code and explain to your partner what you have done, and convince your partner 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. 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 is doing is also twice as hard as it looks. If you think you are working twice as hard as your partner, you two are probably about even!

12 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.
- A `zip` file containing these items:
  - *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, in a package named `tests`, separate from the rest of your source code. Subpackages are permitted.

  Do not include any files ending in `.class`.
- `log.txt`: A dump of your commit log from the version control system of your choice.
- `written_problems.txt/pdf`: This file should include your response to the written problems.