CS2112—Fall 2012

Homework 4

Parsing and Fault Injection

Due: Tuesday, October 16, 11:59PM

Compilers and bug-finding systems operate on source code to produce, respectively, compiled code and lists of possible bugs detected. This software needs a lot of test cases, and test cases that are programs are expensive to produce. Fault injection is a technique for inexpensively generating such test cases. The idea is to take a single correct, valid program and make small random changes to it to produce many useful test cases.

For testing a compiler, we want both test cases that are valid programs in the programming language, and also test cases that are invalid programs. For bug finders, the test cases are valid programs that contain bugs. In this assignment, you will be using fault injection to implement part of the final project for the course. You will use fault injection to implement mutations in a program controlling a simulated critter.

In this assignment, you will also build a parser for a simple language and a *pretty-printer* that can print out parsed programs in a nicely formatted form.

0 Changes

- 10/5: The due date has been moved back to compensate for Fall Break and the prelim.

1 Instructions

1.1 Group Project

This assignment is the first part of the group project for the course. The programming language you will be parsing, pretty-printing, and injecting faults into will be the language controlling simulated creatures. You will need to 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 that are being injected are those corresponding to the mutations in Section 9 of the project specification.

1.2 Partners

You will work in a group of two students for this assignment. Obviously, it is important that you find your partner very soon. Piazza has support for finding a partner.

1.3 Overview Document

Starting with this assignment, we expect your group to submit an overview document. You will want to read the Overview Document Specification to learn what we are expecting. Writing a clear

document with good use of language is important.

We are requiring you to submit an early draft of your design overview document on Friday before the assignment is due. You may not be able to predict what your design and testing strategy will look like in full at that point, but we want to see how far you have gotten. We will aim to get you quick feedback on this draft.

1.4 Restrictions

You may use any standard Java libraries from the Java SDK. However, you may not use a parser generator. As usual, we expect you to stick to Java 6 features and avoid features found only in Java 7.

2 Parsing

2.1 Overview

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 parse 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 the language you will be parsing is given in the project specification. The grammar describes the *concrete syntax* of critter programs, including all the tokens that are part of the input.

However, the job of the parser is not construct the concrete syntax tree; instead, it should build an *abstract syntax tree*.

2.2 Abstract Syntax Trees

An abstract syntax tree (AST) represents the syntax of some input while avoiding representing parts of the syntax that do not affect the meaning of the input. For example, the expressions `(2+3*4)`, `2+(3*4)`, and `(2) + (3)*(4)` all would have the same abstract syntax tree, because the 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 uses some of the classes we have supplied to you for use in AST construction.

Because the tree structure implicitly represents many syntax details, it omits any syntax that is unnecessary. This is what makes it different from a concrete syntax tree or a parse tree, which include all syntax included in a program.

You will need to design and implement a class hierarchy to represent this tree where the leaves are subclasses of `Node`. By giving `Node` the right methods, it will be possible to recursively

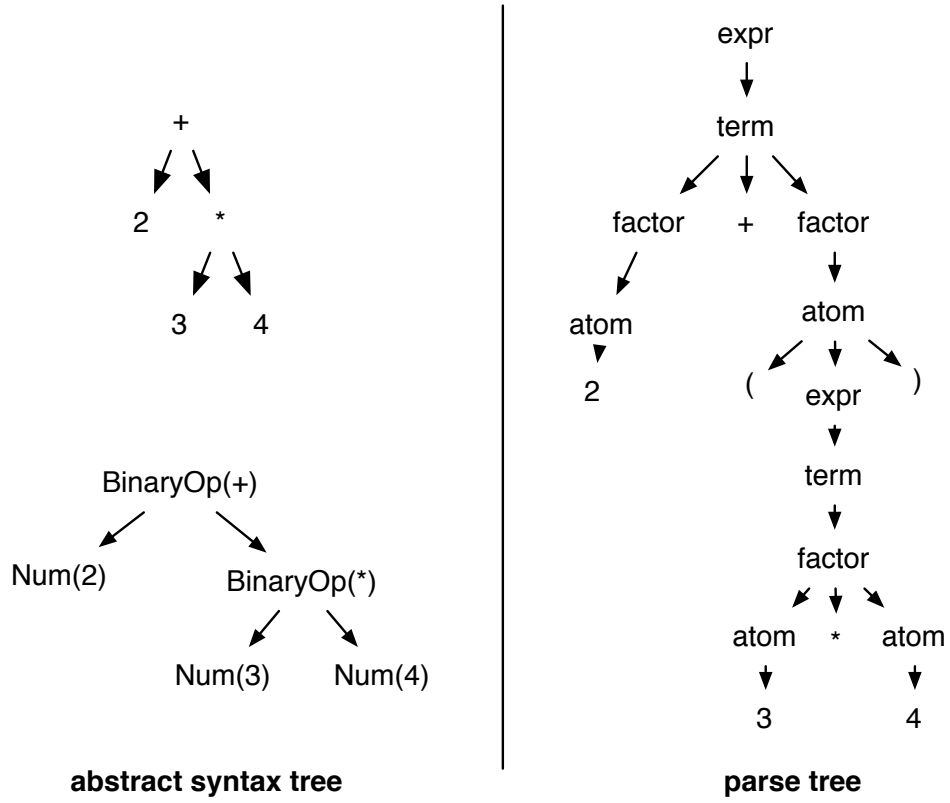

Figure 1: Abstract and concrete syntax trees for `2+(3*4)`

implement various useful functionality, including fault injection and, later, evaluation.

2.3 Provided Classes

We have provided an implementation of a `Tokenizer` as well as interfaces and some classes to get you started with defining your AST. You may not need to use all of these but you will probably need to add more.

3 Fault Injection

Since we are using fault injection to simulate a genome mutation for a critter, see the project specification for how this mutation is to be done. The key to correct fault injection is that for this project, it will mutate a program in such a way that the resulting program is still a legal critter program, though it does not, perhaps, do what it was originally intended to do.

There is some flexibility in how to interpret the mutation rules given in the project specification. You should identify any ambiguities you see and explain how you have resolved them. One rule of thumb is that it should be possible, though some sequence of mutations, to change any program into any other program.

4 Pretty-Printing

You should be able to print out programs in the same syntax they were written in, meaning that the programs you print would generate the same abstract syntax tree if you were to parse them again. Pretty-printing should make use of indentation (though not the ASCII tab character!) and line breaks in order to make output readable and, well, pretty.

5 User Interface

Your program must support the following command-line interface:

- `java -jar <your_jar> <input_file>`
  parse the file `input_file` as a critter program and pretty-print the the program to standard output.
- `java -jar <your_jar> --mutate <n> <input_file>`
  parse the file `input_file` as a critter program and apply n mutations. After each mutation, print a description of the kind of mutation that has been applied and pretty-print the program.

6 Overview of Programming Tasks

Because you will want to figure out with your partner how to break up the work involved in this assignment, it is good to start thinking about some of the major tasks involved:

- Implementing the main program and command-line handling.
- Designing and implementing a class hierarchy of classes for representing abstract syntax trees. These will be subclasses of `Node`. We have given you a start on some of these classes, but you will likely need to add more.
- Implementing the `Parser` class to generate abstract syntax trees.
- Implementing pretty-printing functionality, as methods on AST nodes.
- Implementing a class or classes to perform fault injection. It is up to you to design the interfaces for these classes.

7 Written problem

Recall that a function $f(n)$ is $O(g(n))$ if there exist 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, show by giving a witness that it is $O(n^2)$, or else show that it isn't $O(n^2)$ by arguing that no such witness can exist.

- $3n^2 + 10n + 1$
- 2^n
- $n \lg n$
- $n^3 / \lg n$
- $f(n) + h(n)$, where each of $f(n)$ and $h(n)$ are $O(n^2)$.

8 Submission

You should compress exactly these files into a zip file that you will then submit on CMS:

- *Source code*: You should include all source code required to compile and run the project.
- *Tests*: You should include code for all your test cases.
- `overview.txt/html/pdf`: This file should contain your overview document.

Do not include any files ending in `.class`.

You should also separately submit your work on the written problems in `written-problems.txt` (`.doc` and `.pdf` also permitted).

9 Tips

Working with a partner may be challenging. Some tips:

- Meet with your partner as early as possible to work out the design and to divide up 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.
- The 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 further, exactly what their interfaces are, including detailed specs.

- This project is a great opportunity to try out *pair programming*, in which you program in a pilot/copilot mode. It's fun. It also tends to result in fewer bugs. A key ingredient is to give the person typing the job of convincing the other person that the code meets the spec. Of course, you need to agree on a spec first.
- It might seem as if you are working harder than your partner. But remember that when you go to implement something, it typically turns out to be twice as hard as you thought. 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're probably about even!
- We encourage you to use a version control system such as Subversion or Git to manage your code so that you and your partner can work separately when necessary. We'll be covering this in lab.

As usual, we encourage you to read all Piazza posts because often your questions have already been asked by someone else—even before it occurs to you to ask. You will save a lot of time this way.