CS 6410: Compilers

Fall 2023

Part 2 – Parser and AST

Acknowledgment: Project assignment modified from an assignment developed by Hal Perkins, faculty member of the University of Washington, Allen School of Computer Science and Engineering

Posted on Wednesday, October 4, 2023

Instructor: Tamara Bonaci
Khoury College of Computer Science
Northeastern University – Seattle

Please submit your project by 11:59pm on Saturday, November 4, 2023. You should submit your project by pushing it to your Khoury GitHub repository, and providing a suitable tag. See the end of this writeup for details.

1 Assignment Overview

For this part of the project, construct a parser (recognizer), and build abstract syntax trees (ASTs) for MiniJava. You should use the **CUP** parser generator tool to interface with your **JFlex scanner**. Be sure that your parser and scanner can together successfully parse legal MiniJava programs.

The semantic actions in the parser should create an **Abstract Syntax Tree** (AST) representation of the parsed program. For now, just build the AST. Semantic analysis, and type checking will be included in the next part of the project.

In addition to building the AST, you should provide an implementation of the Visitor pattern to print a nicely indented representation of the tree on standard output. You should also use the PrettyPrintVisitor supplied with the starter code to print a "decompiled" version of the AST as a Java source program when requested. These output formats are different, and more details are given below.
Modify your MiniJava main program so that when it is executed using command:

```
java MiniJava -A filename.java
```

it will parse the MiniJava program in the named input file, and print an abstract representation of the AST on standard output. Similarly, if your compiler is executed using command:

```
java MiniJava -P filename.java
```

it should parse the MiniJava program in the named input file, and print on standard output a decompiled version of the AST ("prettyprint") in a format that is a legal Java program that could be processed by any Java compiler. The -P option should use the PrettyPrintVisitor class provided in our project starter code, with whatever small modifications might be needed due to changes, or additions made to the AST classes in your compiler.

As with the scanner, when your compiler terminates it should return an exit or status code of 0 if no errors were detected while compiling (scanning and parsing) the input program. It should return an exit code of 1 if any errors were detected.

As before, if you are using a different implementation language or additional libraries, please be sure that your compiler continues to work as similarly as possible to the specification above, and you must add to your README file any new or additional information we need to build, run, and test your compiler.

2 Example

To make this a bit more concrete, suppose we use this input file: Foo.java, whose code is given below:

```
class Foo {
  public static void main(String[] args) {
    System.out.println(5 + 4 * 3 - 1);
  }
}

class Bar extends Nothing {
  public int bar(Baz f, Bar b) {
    System.out.println(new Thing().method());
    return 5;
    }
}
```

The MiniJava -A abstract AST printout should resemble the output provided below:

```
Program
  MainClass Foo (line 1)
    Print (line 3)
      ((5 + (4 * 3)) - 1)
  Class Bar extends Nothing (line 7)
    MethodDecl bar (line 7)
      returns int
      parameters:
        Baz f
        Bar b
      Print (line 8)
        new Thing().method()
      Return 5 (line 9)
```

Similarly, MiniJava -P should prettyprint the file, and produce something resembling the following output:

```
class Foo {
  public static void main(String[] args) {
    System.out.println(((5 + (4 * 3)) - 1));
  }
}
class Bar extends Nothing {
  public int bar(Baz f, Bar b) {
    System.out.println(new Thing().method());
    return 5;
  }
}
```

Your actual output may differ slightly, but it should be similar.
Note that `Foo.java` *is clearly an illegal Java program, but it is syntactically legal, so you should be able to parse it.*

If the input program contains syntax errors it is up to you how you handle the situation. You can simply report the error, and quit without producing the AST, or, if you wish, you can try to do better than that.

The java commands shown above will also need a -cp argument, or CLASSPATH variable as before to locate the compiled .class files and libraries. See the scanner assignment if you need a refresher on the details.

Your MiniJava compiler should still be able to print out scanner tokens if the -S option is used instead of -P or -A. There is no requirement for how your compiler should behave if more than one of -A, -P, and -S are specified at the same time. That is up to you. You could treat that as an error, or, maybe more useful, the compiler could print tokens followed by the AST, or the abstract AST followed by the prettyprinted one, etc.

Feel free to experiment with language extensions (additional Java constructs not included in MiniJava) or syntactic error recovery if you wish, but be sure to get the basic parser/AST/visitors working first. You should document any extensions, or error handling in the PARSER-NOTES file, described in the "What to Submit" section.

3 Implementation Details

1. You may need to massage the MiniJava grammar to make it LALR(1) so that **CUP** can use it to produce a parser. Please keep track of the changes you make, and turn in a description of them with this part of the project (see below).
2. Take advantage of precedence and associativity declarations in the parser specification to keep the overall size of the parser grammar small. In particular, `exp ::= exp op exp` productions along with precedence and associativity declarations for various operators will shorten the specification considerably compared to a grammar that encodes that information in separate productions. **CUP's** input syntax is basically the same that used by YACC and Bison, described in many compiler books. It should be easy enough to pick up the syntactic details from the CUP documentation and example code.
3. Your grammar should not contain any reduce-reduce conflicts, and should have as few shift-reduce conflicts as possible. You should describe any remaining shift-reduce conflicts in your writeup.
4. Note that the `minijava.cup` example file supplied with the starter code is intended only to show how the tools work together. The grammar given there is not a proper subset of the MiniJava grammar, and you should make appropriate changes as you produce the full grammar.
5. The starter code contains a `TestParser` program that shows how the parsing tools work together. You can run that to see what it does. You will need to adapt the ideas found there for your own compiler.
6. CUP has some useful features for debugging. First, the build.xml ant file specifies options that cause CUP to create a build.out file in the build directory. This file contains details of the parser generation, including a description of the LR states and other information about the grammar, which can be very useful for understanding conflicts and other problems with the grammar.
7. After the parser is generated, a MiniJava compiler parses its input by calling the parse() method. As described in the TestParser.java sample program, if debug_parse() is called instead the parser will print a detailed trace of all of the shift and reduce actions performed as it parses the input. That information can be very useful when trying to figure out parser problems, particularly with the information in the build.out file produced when the parser was generated, which describes the parser states in detail.
8. Once you have the parsing rules in place, and have sorted out any grammar issues, add semantic actions to your parser to create an **Abstract Syntax Tree (AST)** and add **Visitor** code to print a nicely indented representation of the AST on standard output. Also add code to use the supplied PrettyPrintVisitor class to print the decompiled version of the AST when requested.
9. The AST should represent the abstract structure of the code, not the concrete syntax. Each node in the tree should be an instance of a Java class corresponding to a production in an abstract grammar. The abstract grammar should be as simple as possible, but no simpler. Nodes for nonterminals should contain references to nodes representing non-trivial component parts of the nonterminal, i.e., the important items on the right-hand side of an abstract grammar rule. Trivial things like semicolons, braces, and parentheses should not be represented in the AST, and chain productions (like Expression $\rightarrow$ Term $\rightarrow$ Factor $\rightarrow$ Identifier) should not be included unless they convey useful semantic information. The classes defining the different abstract syntax tree nodes should take advantage of inheritance to build a strongly-typed, self-describing tree. We suggest that you start with the AST classes given on the MiniJava project web page, which are included in the starter code for our project.
10. The output of the new AST Visitor should be a readable representation of the abstract tree, not a "decompiled" version of the program, and, in particular, the output will not be a syntactically legal Java program that could then be fed back into a Java compiler. Each node in the tree should normally begin

on a separate line, and children of a node should be indented below it to show the nesting structure. The output should include source line numbers for major constructs in the tree (certainly for individual statements and for things like class, method, and instance variable declarations, but not necessarily for every minor node in, for example, expressions). The tree printout should not include syntactic noise from the original program like curly braces (), semicolons, and other punctuation that is not retained in the AST, although you should use reasonable punctuation (indentation, whitespace, parentheses, commas, etc.) in your output to make things readable. Although most tree nodes should occupy a separate line in the output, you can, if you wish, print things like expressions on fewer lines to reduce the vertical space used, as long as your output clearly shows the AST structure of the expression (perhaps by adding parentheses to show nesting if line breaks and indenting are not used).

11. We have included the MiniJava Visitor interface and PrettyPrintVisitor.java file from the MiniJava project web site in the directory src/AST/Visitor in the starter code. Note that this PrettyPrintVisitor doesn't print the AST in the abstract format required above – it is closer to the concrete syntax and doesn't have much indentation logic. But you will want to use it to pretty-print the AST (compiler -P option) and you will find it useful as a starting point for understanding the visitor pattern and creating the new AST visitor (compiler -A option). Also, cloning an existing visitor is often a good way to start developing a new one.
12. You should test your parser, semantic actions, and printing visitor by processing several MiniJava programs, both correct ones and ones with syntax errors
13. Please continue using your CS6410 GitHub repositories to store the code for this, and remaining parts of the compiler project.

4 What to Submit

The main things we'll be looking at in this phase of the project are the following:

1. The grammar specification, including semantic actions, for your parser.
2. Source files containing definitions of the AST node classes and visitors.
3. Printed AST representations produced by both print visitors (-A and -P options).
4. A brief PARSER-NOTES file describing any changes you made to the grammar, and a list of shift-reduce conflicts remaining in the grammar (if any) with an explanation of why these are not a problem.
5. If you include any error handling or extra features in your parser, or add any language extensions, include a brief description of these features in your PARSER-NOTES writeup.
6. As before, you will submit your parser project by pushing code to your GitHub repository. Once you are satisfied that everything is working properly, create a parser-final tag, and push that to the repository.