CS444 Assignment 1 - Scanning, Parsing, and Weeding

Amrit Sehgal - a38sehga
Aryaman Dhingra - a23dhing
Jagvir Dehal - jdehal
Paranjay Agarwal - p38agarw

Building the Compiler

Our compiler can be built with `make`, using either `make build` or just `make`. The system it's built on needs the following dependencies, all of which are available in the student linux environment: g++ 11.4.0, GNU Make 4.3, cmake 3.22.1, flex 2.6.4, and bison (GNU Bison) 3.8.2. After building, the executable `joosc` will be placed in the project root directory.

Design and Implementation

Our compiler is implemented in C++. The important components of our program are the build system, the scanner, the parser, the weeder, and the main function.

Build System

We use a combination of `make` and `cmake` for our build system. The `Makefile` lives in the root directory of the project, and contains useful utility commands for development such as `make submission` and `make integration-test`. It also calls into `cmake` with `make build`. In our program, `cmake` is used to fetch dependencies and recursively compile all the `.cc` files and link them together for `joosc` and other targets such as `unit-tests`. We chose to use `cmake` due to its prevalence in the C++ ecosystem, which means it's easy to use it for dependencies such as GoogleTest.

Lexing

We use the Flex lexer generator to tokenize the input. Flex is useful because it cooperates well with Bison. The Flex input file, `scanner.ll`, lives in `src/parsing/bison` due to being closely coupled to the bison code. The tokens used are defined in the `src/parsing/bison/parser.yy` file, and then we define the regular expressions to match these tokens in the `src/parsing/bison/scanner.ll` file.

Our scanner file generates a `yylex` function that Bison uses, which matches each token and returns an instance of a token defined in the parser file, by invocating the Bison-generated function `yy::parser::make_{TOKEN_NAME}`. We also catch invalid Joos tokens that are valid Java tokens in the scanner, such as "`volatile`", "`super`", and others, by throwing a `yy::parser::syntax_error(..)` whenever these are matched and printing a debug message of which Java token was matched. The scanner also has some very basic definitions, to make the regular expressions more readable.

Parsing

As hinted to in the previous section, Bison is our LALR(1) parser generator of choice. We decided to use Bison because it is compatible with C++ and supports counterexample generation, which was a huge aid in our debugging efforts. Our grammar is defined in the `src/parsing/bison/parser.yy` file.

We implemented the grammar rules using the Java 1.3 specification from the course website. A major challenge we encountered early on while implementing the grammar was handling inconsistencies in the specification. Our first strategy was to use the glossary of rules in §18 and convert them to Bison rules, which we thought would make this step very quick as there was a note at the top of the chapter mentioning that section was meant to make it easier to write a parser than the rules presented throughout earlier chapters, which were meant for exposition.

However, there were two main problems with this approach. One was that the rules presented in §18 were not just a collection of the rules from early chapters; in fact, the rules from earlier chapters were more suitable with a LALR(1) grammar as many of the rules from §18 could not be non-ambiguously parsed with one token lookahead. The second problem was that the rules in §18 did not have descriptive names, using names such as `Expression3` instead of `MultiplicativeExpression` which made it harder to understand what was going on when debugging conflicts. Thus, we ended up resolving these problems by using the chapters with heavy exposition for our grammar rules instead, and rewrote our grammar to follow suit.

Another major challenge we had in the implementation of the parser was dealing with conflicts. Bison's counterexample generation was useful, but we struggled to understand the output at first. Thankfully, the first part of the paper which discovered parser generator counterexamples was a great help in explaining how they work and helped us understand how to change our grammar rules for one token lookahead.

Weeder
Our weeder is a class that traverses through the codes AST and finds issues such as different file naming, missing constructors, and mixing modifiers. It does this through the creation of a generic AST which is done through bison actions. The AST is built up of a tree of AstNode pointers. Each Node has a type, a parent, and a list of children. We built it this way because it makes traversing the tree extremely simple. The weeder essentially traverses this data structure to weed out any problems in the source code. We decided to build a weeder class so that we could decouple our parser from our weeder. Our parser only creates the actual AstNodes and creates the root of the AST, and our weeder uses this root and other data to weed out problematic code. Our weeder is defined in `src/weeder` and the AstNode is defined in `src/ast`.

Main
Since we're using C++, the entrypoint to our program is a `main` function, which in our case lives in `src/main.cc`. This function is the driver for calling into the parser to produce a tree, and then calling the weeder on the tree. We also implemented `-p`, `-s`, and `-r` command line flags to enable parse tracing, scan tracing, and print the return code from executing on a file respectively. These additional flags were very useful for debugging failures in our test suite.

Testing and Known Issues
We created a python integration test script, `valid_invalid_prog_test.py` for verifying that `joosc` correctly validates and invalidates valid and invalid programs respectively. Valid programs were placed in `tests/programs/valid` and invalid programs in `tests/programs/invalid`. We attempted to cover as many cases as possible, by looking at the `Joos1W` supported features on the course website and Java 1.3 specification to test almost every supported syntactic feature, as well as verify that unsupported Java 1.3 features are not considered syntactically valid by our compiler.

We also created test cases specifically for lexical analysis. These tests simply take text and create an output that resembles the text in token form. A sample input and expected output is below.

```
public class A { int b = 0; }

PUBLIC CLASS IDENTIFIER OPENING_BRACE
INT IDENTIFIER ASSIGNMENT INTEGER SEMI_COLON
CLOSING_BRACE EOF
```

Work
The contributions of each group member is listed below. Throughout the development process, we regularly communicated synchronously with multiple meetings per week and asynchronously through Discord, so each member was supported in their tasks.

Amrit

- Main focus on the parser.
- Implemented grammar rules for packages, expressions, and types.
- Resolved reduce/reduce and shift/reduce conflicts for all components of the grammar and when combining the rules each person did separately.
- Bootstrapped project structure and build system, and wrote integration test python script for testing that `joosc` returns the correct error codes on valid/invalid programs.
- Worked on the weeder.
- Worked on this report.

Aryaman

- Mainly worked on building the lexer.
- Defined the token set, and wrote matching expressions for half the token set.
- Set up basic CI/CD on Gitlab.
- Wrote a lot of lexer tests and valid/invalid tests to test A1 requirements
- Wrote the grammar for the interfaces subset of Joosc
- Worked on building the AST definition, as well as the weeder.
- Worked on this report

Jagvir

- Had a hand in every part of the project
- Worked on part of the lexer and the entirety of the lex testing cpp & script
- Wrote the initial grammar for classes, methods, and fields (before bug fixing due to integration)
- Pair programmed with Aryaman to create the parsetree code
- Added the parsetree code to each of the grammar rules
- Added support for graphical visualization of the parse tree to aid debugging
- Supported weeding for string, char and integer literals
- Created the makefile for bison/flex

Paranjay

- Mainly focusing on the grammar rules for the parser.
- Implemented the entirety of the grammar rules once which resulted in a lot of conflicts so had to be rewritten in parts.
- Wrote the rules for statements and resolved conflicts whenever they arose.
- Wrote a lot of invalid tests for the parser covering all possible invalid scenarios.
- Debugging the weeder after submission to cover the test cases our compiler failed to take care of.

Thoughts

We spent a **lot** of time on this assignment. In the last week, we worked on it almost every day. The primary reason it took up a long time was likely all the different specifications to meet, as each specification in the lexer and parser needed to be tested and implemented precisely. Another reason we spent a long time was having to rewrite our grammar, and dealing with all of the conflicts which took hours. This also delayed our implementation of the weeder, which we couldn't design until we knew what the parsing looked like.

Overall, we generally enjoyed the assignment. The aspects we liked were learning about how parser generators worked, and carefully designing our program. Something we disliked about the assignment was some of the ambiguity, since we're given a Java 1.3 specification but not a Joos 1W specification aside from the list of unsupported features in the form of code snippets on the website.

We found C++'s ecosystem containing good lexing/parsing tools in Flex and Bison to be a large advantage of using the language. The build system caused some annoyance, since C++ is not known for having great build systems.