Assignment 1: Scanning, Parsing, Weeding

Zhenyan Zhu - z277zhu Zijian Feng - z63feng Richella Li - j2333li

Instructions

To build the compiler, run `make` at the root directory.

`make lexer` will allow you to generate parser and lexer code only.

To compile a java file, we use the following usage. `./joos [java file]`

Ex. to use the lexer `./joos ./test/test.in`

Design and Implementation

Our project is developed with java and a structure with sbt as our build tool.
The structure of the files are listed below:

```
- build (store our intermediate files while building.)
- lib (store the third-party library and also the package we built)
- src (store the source files)
   |-- ast [TODO]
   |-- flex (jflex/cup config files)
   L– lexer (parser/lexer generated by flex)
- test (store the test cases)
- test.sh (bash script to run each test case for A1)
- boot.sh (boot the docker env)
- Dockerfile (to build the docker env)
- Makefile (build this compiler)
```

Design Idea: We use the third-party libraries, jflex-1.6.1 and java_cup to generate the scanner rules and the parser rulers.

Lexer: Comments, strings literals are handled by the regular expression grammar in jflex. For every word, a `Token` Class is here constructed and stored for further use in parsing.

Parser: All production rules are originally copied from jls. They are ambiguous in the beginning. To solve this ambiguity, we merge all non-terminal symbols that may generate conflicts together. For instance, JLS's grammar contains several kinds of modifiers, such as field_modifier, method_modifier, and so on. Meanwhile, a "PUBLIC" terminal can either become field_modifier or method_modifier, so we merge these kinds of modifiers to 'modifier' type that can produce any kinds of modifiers, and we made restriction on modifiers in weeding part.

AST: We design the AST to be a simplified parse tree. The structure of ASTNode is implemented in src/ast/ASTNode.java. It has a List<ASTNode> field to represent its children. To simplify the parse tree, we use inheritance. For instance, let's say we want to parse '1+1' from 'expr', if you follow the production gramma ,a parse tree would be like 'expr' -> 'assignment_expr' -> 'condition_or_expr' -> … -> additive_expr -> additive_expr PLUS multiplicative_expr -> …, which is quite complex and hard for traversal, so we utilize inheritance and polymorphism. We built up an inheritance hierarchy, we let expr to be the base class; let assignment_expr inherit expr; let condition_or_expr inherit assignment_expr; … ; let additive_expr inherit multiplicative_expr. Thus, we can directly assign expr to be an additive_expr, and our final AST for this example would be 'additive_expr' -> 'NumericLiteral(1)' 'PLUS' 'NumericLiteral(1)'

Weeder: In weeder we mostly rely on removing, rewriting production rules and mostly introducing helper functions in the production rule. Using the helper methods allows us to check every ASTNode and their children to see if they are syntactically correct or throwing the exception otherwise.

Testing and Known issues

As public tests cover most cases, our personal tests focus on small parts and detailed restrictions. Here are some topics we have tested:

- Redundant modifiers
- Restrictions on modifiers: must have one and only one access modifiers, restriction of native, and abstract.
- Unicode input
- Interface
- One file can have only one public class which has the same name as the filename
- Package, import declarations
- Simple and complicated for statements.

Some of the tests went well, but some of them did fail, like the unicode, some modifier test cases and so on. But we were able to fix them on time and make them work. In general, our tests were going really well.

Work

- Zijian Feng(z63feng)
  - Dockerfile and environment for multi-platforms.
  - Designed and implemented scanner used in joos.
  - Design the regular expression used for string literal, comments handler.
  - Tested the scanner compatibility with joos's features.
- Zhenyan Zhu(z277zhu)
  - Designed and implemented joos context-free grammar; solved several conflicts (together with Richella).
  - Implemented a bash test script and tested parser correctness.
  - Designed and built AST in parser (together with Richella).
  - Tested AST correctness (together with Richella).

    - Solved integer overflow, class name and filename matching in Weeding.
    - General correctness test (together with Richella and Alfred).
- Richella Li(j2333li)
    - Designed and built AST in parser (together with Zhenyan).
    - Fixed problems for abstract method, restrictions on modifiers for class/field/constructor/method…, Escape Characters and Strings, Casting Expression(together with Zhenyan), explicit constructor declaration, invalid super, this method name,
    - General correctness test (together with Zhenyan and Alfred).

Thoughts

- How much time did you spend on the assignment? What did you like or dislike about the assignment? What would you change about it?
    - For parser, we spent about 3 hours going through all production rules originally listed in JLS; we spent 5 hours solving all ambiguity/conflicts; we spent 5 hours constructing AST; we spent 5 hours on weeding; we spent 5 hours on compiling and testing.
    - The part we like about the assignment is we could use some generators to do less redundant work, at the same time, as we need to write tokens, production rules, helper methods that weed out syntactically incorrect tokens, we still can have a good understanding of the scanning, parsing, weeding stage. We have choices do decide how to design our compiler by ourselves instead of hardly following the guideline. The part we think it could be improved is Joos1W has many differences compared to Java. But the resources we have are Java language specifications and some simple requirements of Joos1W, it is relatively difficult for us to distinguish all the differences by ourselves.
    - We may add more instructions about the restrictions and requirements of Joos1W for A1 particularly. We may add more guidelines of building A1.
- What aspects of your chosen programming language did you find helpful in writing the compiler? What aspects caused annoyance?
    - Originally, we plan to use Scala for its functional and OO feature. However, we found that there is few resources of third party that can generate a lexer or parser written by Scala; so we change our plan to use Java for lexer and parser but built up AST by Scala since we may easily traverse an AST in Scala by its functional feature. In ASTNode class we use List<ASTNode> to represent the children of that Node, and the type of the List is a Scala List; however, in Joos.cup we can only create Java List to initialize ASTNode, and here comes the conflict: if we use Java List ub ASTNode class written in Scala, we may lose its functional feature such as map, foreach, and so on. Otherwise, we may hardly initialize a Scala List in Java (maybe by some third-party jar or sth). Finally we decide to use Java for all later assignments.
    - The aspect that we found helpful in writing the compiler is that Java has an instanceof operator to identify the class type, this is quite useful in weeding because we can identify the actual type of the ASTNode and made restrictions on them.
    - The aspect that we found annoy in Java is that one java file can only contain one public class, and for ASTNode, we may have tons of different subclasses inherited from ASTNode, and for each class, we have to create a new file, which makes our repo looks quite huge.