WCTC - Who Compiles The Compiler

Aditya Tanwar 200057
Janhvi Rochwani 200467
Soham Samaddar 200990

February 2023

1 Running the Files

The implementation of the Parser and Lexer is in Flex+Bison, written in C++. Further, DOT is used to create an AST. If these are not already installed, the following commands should be executed:

```
sudo apt-get update
sudo apt-get install flex
sudo apt-get install bison
sudo apt-get install graphviz
```

A makefile has been provided along with the source code files, the executable `WCTC.o` can be made by simply running `make`.

The command line options provided by our program have been documented below:

- `-i` or `-input`: Filename from which the compiler reads the Java program
- `-o` or `-output`: Filename to which the compiler outputs the `DOT` file
- `-v` or `-verbose`: Prints the derivation (parse tree) in the command line
- `-h` or `-help`: Manual page

All the above commands are optional. By default, input is read from `test.java` and output is redirected to `tree.gv`. After the `DOT` file has been created, `dot` has to be used generate the `png/svg/ps` file. An example of a successful execution is given below:

```
$ make
$ ./WCTC.o -i BubbleSort.java -o tree.dot --verbose
$ dot tree.dot -Tpng -o AST.png
```

2 Augmentations

The CFG available for Java 17 on the Oracle docs in true form lead to a lot of conflicts. We analysed the conflicts using `--verbose` command, and tried to remove the conflicts one by one.

We applied a thumb rule of merging some commonly occurring "substrings". For instance, we noticed a lot of production rules being "essentially" `ID(.ID)*`, therefore, we introduced a new non-terminal `Name` and replaced all occurrences of `ID(.ID)*` with it, and further simplify the production rules using the newly introduced non-terminal.
Another heuristic used was to replace all occurrences of a non-terminal, which had a single production rule, with a single non-terminal/terminal on the right hand side. This rule was applied recursively, until no such "trivial" production rules were left. Most of these arose due to ignoring `TypeArguments` and `Annotations`.
Later, to resolve shift-reduce conflicts, we made use of the associativity and precedence offered officially, and implemented them in Bison using commands like `%prec` (for precedence among some conflicting production rules), `%right`, `%left` (both for exploiting associativity and precedence), etc.

3 Optional Features

All of the expected language features have been supported. Further, we have implemented the following **optional features**:

- Support for Strings: Complete support has been provided for text blocks, strings, and characters (including escape characters) at this stage. However, to print them in the AST, the quotes at the beginning and end of the lexeme have been dropped.
- Import statements
- Support for the `protected` Modifier
- Partial support for Type Casting has been provided.

4 AST from Parse Tree

The `parser` is responsible for constructing the parse tree. This parse tree is taken and then some nodes are removed according to the following rules (in the same order):

1. Removing occurrences of `;` (the semicolon delimiter). This was done to make the parse tree more concise, since there are a lot of occurrences of the character.
2. Replacing nodes which have a single child. This was done to hide longer derivations in the AST (for example, it takes >10 derivations to go from `Expression` to a `Literal`. In such cases, the lower-most node replaces the higher-up nodes.
3. Lastly, the operators are shifted one level above (than their original level in the parse tree).

A colour coding scheme has been followed in the visualization of the AST for better readability.

The same has been been provided below:

Delimiter Keyword Identifier Operator

For the following example code:

```
import java.util.*;
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
}
```

The Parse-Tree generated by our parser and the Abstract-Syntax-Tree is provided in the submission in files `PT.svg` and `AST.svg` respectively.
As mentioned, some productions require >10 derivations to reach a terminal/non-terminal. Using this logic for pruning, the Abstract Syntax Tree thus made, can be seen to be much more compact in comparison.

5 References

We have written some of the test cases on our own and some have been taken from the internet. The references are as follows:

1. test_1.java: Matrix Multiplication
2. test_2.java: Dijkstra's Algorithm
3. test_3.java: Deletion in Red-Black Trees
4. test_8.java: Binary Exponentiation
5. test_9.java: Knapsack Problem