CS444/644 Assignment 1 Technical Report

Ho-Yi Fung, Jianchu Li, Zhiyuan Lin

February 5, 2016

Abstract

The purpose of this assignment is to design and implement the lexical and syntactic analysis modules of a Joos 1W compiler. The original objective has been successfully achieved. In this report we discuss in detail the design and implementation of our program, and the issues that the team faced in the process.

Contents

Chapter 1

Introduction

The Joos 1W language [3] is a subset of the Java 1.3 language. The purpose of this project to create a compiler for the Joos 1W language in Java 8. In this assignment we've designed and implemented the modules in compiler that are responsible for lexical and syntactic analysis. More specifically, we implemented the *scanner*, *parser*, *weeder* and *abstract syntax tree* building module for the compiler. The *scanner* takes as input the source code and convert them into a list of tokens. The *parser* then reads the tokens and ensure that the program is grammatically correct. It also builds a parse tree in the process. The *weeder* module runs after the parser to check for errors that are too complicated to encode in the grammar. Eventually we create an *abstract syntax tree* based on the verified parse tree.

Chapter 2 discusses separately how the modules were designed and what issues were dealt with in each module. Chapter 3 discusses the detail of our implementation. Chapter 4 introduces the testing techniques and tools used to ensure the correctness of our program.

Chapter 2

Design

2.1 Scanner

The first step of our design process was to sketch out the transitions for the Joos scanner DFA. We made the design decision to keep the scanner as simple as possible and create a clear separation of functions, leaving the more complicated checking and syntactic analysis for the parser or weeder.

One of the first challenges we faced was how to differentiate between identifiers, keywords, null literals, and boolean literals, as the Java language specification defines a set of reserved keywords that cannot be used as identifiers. One possible way to implement this would be to have states for each keyword, null, and boolean, but since there are over fifty such elements, this would result in a very large DFA and affects readability of our code. So, instead, we would simply have one accepting state for anything that could be an identifier (i.e., everything that starts with a letter and contains only alphanumeric characters, underscores, or dollar signs), then perform a second scan of the lexeme (by checking for equality) to determine which type the token should be. In other words, we read keywords in the same manners as identifiers and perform a hash table lookup afterwards to determine whether it is a reserved keyword. This approach has no effects asymptotically on the performance yet results in succinct and readable code.

Another important matter at this stage was to decide what qualifies as a lexical error in our scanner. A lexical error arises when a character sequence could be recognized by the scanner as a valid token. In the scanner, we deal with the following lexical errors:

1. Invalid characters: the scanner would ensure that every character of input was a seven-bit ASCII value, as the scanner reads one character of input at a time.

2. Invalid octal escapes: an octal escape in a string or character must be checked so that it does not exceed the ASCII limit.
3. Runaway strings, characters and comments: String literals in Java must not span multiple lines. An error should be raised when an end-of-line (or end-of-file) character is detected within the string body. This type of error is called runaway strings. The same kind of problem can arise for characters and block comments (with EOF) too.

2.2 Parser

We have decide to implement a table-driven LR(1) parser in this project. The table-driven approach separates the code from the parse table. It allows for easy update of the grammar. Although SLR(1) and LALR(1) would result in a smaller parse table, we have decided that it is more important to keep the grammar correct as LR(1), and this is unlikely to become the bottleneck of our program. Besides should it be necessary, we could easily switch to SLR(1) or LALR(1) with the table-driven approach.

The context free grammar was created based on the LALR(1) grammar in [2] and the Joos 1W requirements [3]. Production rules that were unnecessary under Joos 1W specification were removed. Small modifications were also made to the grammar to ensure correctness and compactness of the parse tree. The context free grammar was then passed into the *Jlalr1* generator to produce the parse table.

It is important for later stages that parser not only checks the grammar but also constructs a valid parse tree. The generic table-driven parser implements a driver algorithm from [1] and is listed in Algorithm 1.

It was also decided that the parser would not perform any checking beyond the grammar. Whatever errors that could not be checked by the scanner and the grammar are left for the weeder to deal with.

2.3 Weeder

Now that the parser has output a parse tree, the weeder would ensure that the meaning fits the specifications of the language. The weeder deals with stntax errors that were complicated to specify in the grammar.

For example, the list of modifiers had to be checked to ensure that there were no duplicate modifiers (e.g., `final final`) and no contradictory ones (e.g., `abstract final` method). Modifiers might also be affected by other modifiers elsewhere; for example, a class must be abstract if any method within is. Also a method must have a body if it is not abstract or native.

Algorithm 1 Driver for LR(1) parser

```
BOF = tokens.removeHead()
stateStack.push(parseTable[startState][BOF])
nodeStack.push(BOF)
accepted = false
while the list of tokens is not empty do
    action = parseTable[stateStack.top()][tokens.peek()]
    if action == shift $s$ then
        stateStack.push($s$)
        nodeStack.push(tokens.removeHead())
    else if action == reduce $A \to \gamma$ then                ▷ reduce
        stateStack.pop($|\gamma|$)
        oldNodes = nodeStack.pop($|\gamma|$)
        newNode = Node($A$)
        add every node from oldNodes as child node of newNode
        tokens.prepend(newNode)
    else
        Error()
    end if
end while
EOFNode = nodeStack.pop()
return nodeStack.pop()                                  ▷ root of parse tree
```

The weeder was also when the name of the class or interface was checked against the name of the file for equality in addition to being compared with the constructor (which must be explicit for classes).

In addition, casting rules (e.g., no casting to an expression) were checked, and the range of integers had to fit in an `int` ($-2,147,483,648$ to $2,147,483,647$).

It was also desired that the weeder only traverse the parse tree once, in order to ensure the efficiency of our program.

2.4 Abstract Syntax Tree

The next phase of the program is transforming the parse tree into an abstract syntax tree (AST). The AST is a simplification of the parse tree (sometimes called concrete syntax tree) while retaining all the semantic information. It is the interface between the syntactic analysis and later stages of the compiler.

A logical and semantically correct hierarchy has been created for AST nodes. See figure 2.1 for an example of the node hierarchy. The red arrows in the figure shows the composition relationship. For instance, a *CompilationUnit* contains a *PackageDeclaration*, *ImportDeclaration*s and a *TypeDeclaration*. The

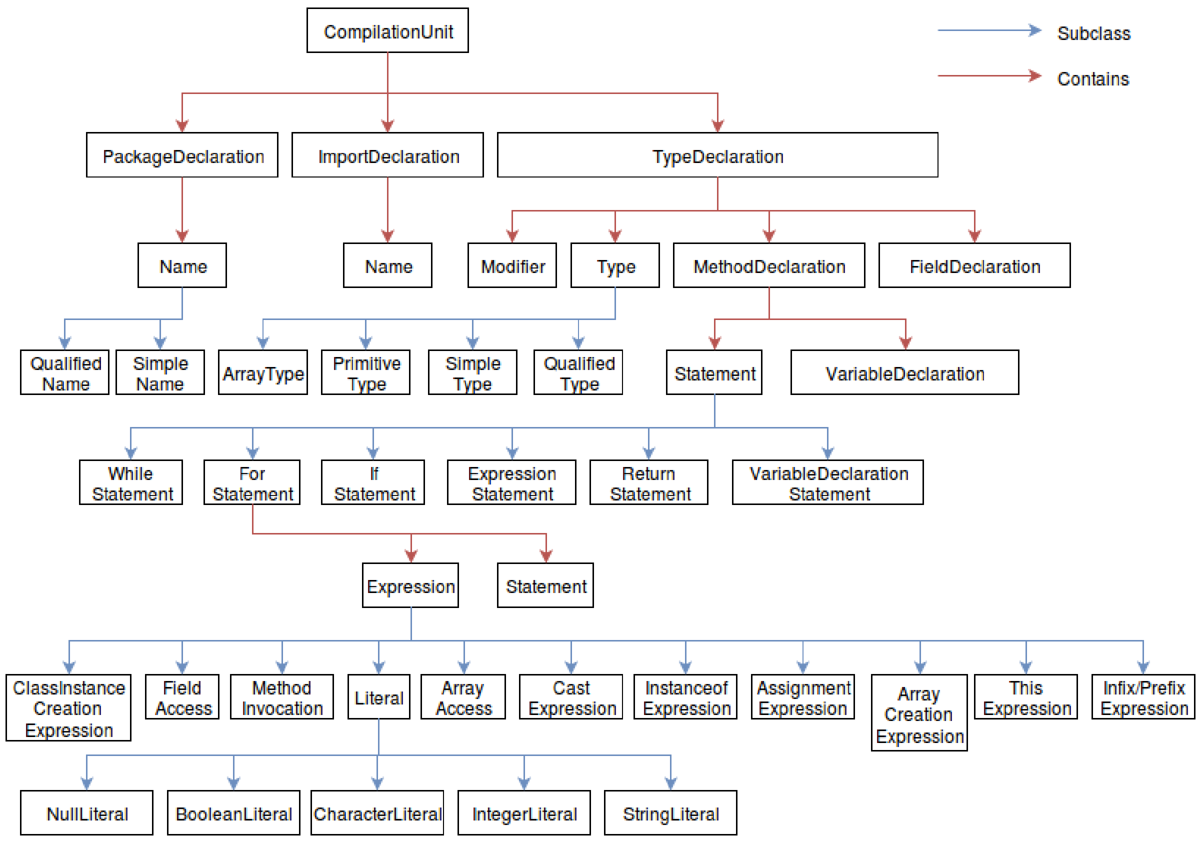

Figure 2.1: Hierarchy of AST nodes

blue arrows, on the other hand, shows the type hierarchy. Take *Expression* for example, there are many types of *Expressions*, including *Literal*, *Name*, *Method Invocation* and such.

Chapter 3

Implementation

This chapter discusses the actual implementation of our program and the issues encountered during the implementation. This project was implemented in Java 8. *Git* was used for version control and the code repository was hosted on University of Waterloo server with *Gitlab*.

3.1 Scanner

The implementation of scanner follows the DFA designed for valid tokens in our language. As explained in [1], a DFA can be implemented in two forms: *table-driven*, or *explicit control*. The table-driven approach, usually used by the scanner generators, utilizes an explicit transition table that could be interpreted by a universal driver program. The explicit control form, on the other hand, incorporates the transitions of the DFA directly into the control logic of the scanner program. In this project, we choose to hand-code the scanner in explicit control form for two reasons. First of all, incorporating the DFA transitions directly into control provides better performance. Secondly, the scanner produced in such manners is easier to debug and modify based on our requirements. The shortcoming of this approach is that, with the token definitions hard-coded into our program, the scanner could not be easily adapted for use elsewhere. This, however, is not a problem for the project. Another benefit of the explicit control implementation is that in the actual implementation there was no need to backtrack when running the Maximal Munch algorithm: the only states that were non-accepting and between accepting states were those associated with a block comment, and since an unfinished comment could not form a valid Joos program, the scanner would simply raise an lexical error when such situation arises.

Another implementation detail was that since both strings and characters could contain escape characters, we would build a secondary DFA for escape characters, which was used as subroutine by both string and character scanning functions, rather than independently build it into both string and character scanning.

Each token created by the scanner would be assigned a type. For example, *StringLiteral* is a token type, and each reserved keyword is a valid token type. We list every possible token type in an Java enum type called *Symbol*. This enum also contains symbols that will later be used for building parse tree.

As discussed in Section 2.1, we check for lexical errors such as non-ASCII characters. More specifically, the *read()* function in the implementation checks for invalid characters every time a character is read. The scanning functions for strings, characters and comments would check for runaway strings, character, and comments respectively.

3.2 Parser

It was decided during implementation that rather than embedding the grammar directly into the code, we would keep it in a separate file so that, should it be required, the grammar could be regenerated without affecting the rest of the code. This approach follows the table-driven parser design discussed in Section 2.2.

Implementation of the parse table in Java was an important step. One of the key issue in the implementation is to make parse table access as fast as possible. The parse table was implemented practically as an array of hash maps (under a Java class called *ParseActions*). An array was used because each state in the parse table is represented as an integer. Given the current state s and the next symbol t, we first access the s-th hash map in the array, and then retrieve the action value with key t. If the hash map does not contain the key t, an error is raised. This implementation allows for action retrieval in time constant to the number of actions, therefore it is time efficient.

Another important detail at this stage was the construction of parse tree. To make the code succinct, we have decide that the parse tree node would inherit from the *Token* class used by the scanner, and they both share use of the *Symbol* enum type, which contains both terminal and non-terminal symbols in our context free grammar.

3.3 Weeder

The implementation of the weeding phase was straight forward compared to the other modules. The weeder does a single traversal of the parse tree with a

```
public interface Visitor {
        public int visit(Statement s);
        public int visit(Expression e);
        public int visit(Literal l);
        ...
}
```

Figure 3.1: Example of Visitor Interface

breadth-first search. And it checks for the errors discussed in Section 2.3.

3.4 Abstract Syntax Tree

The abstract syntax tree is constructed based on the parse tree from the previous stages. The parse tree is traversed once using recursive depth first search.

The type hierarchy shown in figure 2.1 was completely implemented. All AST nodes inherits directly or indirectly from the class *ASTNode*. The functions that builds the AST are incorporated into the AST nodes. For example, the *Statement* abstract class contains functions that transform a statement node in the parse tree into a *Statement* node in AST, and this new statement node could be a *WhileStatement*, *ForStatement* node, etc.

With more than 40 nodes types and a complex type hierarchy, later stages such as type checking could have a lot of interacts with the AST nodes. To help better organize the code, the *Visitor* design pattern was applied. The pattern makes it easy to encode a logical operation (a phase of compilation) for different AST node types under different methods in one file. An example of the *Visitor* interface is provided in Figure 3.1. The *Visitor* interface will be implemented by future phases.

Chapter 4

Testing

The unit testing method was used to ensure the robustness of our program, especially during implementation of scanner and parser. The JUnit 4 framework provides excellent support for this method. Following the principle of unit testing, each small unit in our program was tested separately. For example, the scanning of each token type was tested individually using parameterized tests. Both positive and negative test cases, derived from the language specification, were used in this process. For example, each error that should be detected by the weeder has its own test cases and was verified.

Because of the continuous nature of our project, the program was also tested incrementally, e.g. after each module (scanner, parser, etc.) was completed it was tested against all test cases that had been created even for previous modules. This approach ensures that any valid input could be run on all modules, while invalid input would fail at some point.

We've also created test case based on the example code from Joos 1W language specification [3]. These tests ensure that the program satisfies the requirements for the language.

Parse trees created by the parser was checked visually to ensure that it follows the grammar design. The weeder also provides confidence that the parse tree was constructed correctly. The effectiveness and correctness of the AST module was partly based on the correctness of parse tree due to their similarities. The AST will also be tested more extensively in later stages where it would be used as an input.

A JUnit test suite was also created so that after each addition/modification all modules could be tested again conveniently to prevent bugs being introduced to the program. After all parsing module of the program was completed, the tests cases from Marmoset was introduced and tested against. Parameterized testing was again used at this step to make sure the program passes every test case.

The program so far successfully passes all the test case devised by the team. This demonstrates the robustness and correctness of our implementation.

Chapter 5

Conclusion

The implementation of lexical and syntactic analysis for our compiler has been a success. All the modules were implemented as designed and passes all the test cases. There are however improvements we could make to the program. Time complexity was always carefully considered when designing the program, but space efficiency could be improved. This was a conscious choice made by the development team as it was believed that any modern machine would have enough memory for these operations. The AST nodes have been implemented but further development and modification are expected to make it easier to use for later phases. Any future changes for this module will be documented in the next report.

Bibliography

[1] C. N. Fischer, R. K. Cytron, and R. J. LeBlanc, *Crafting a compiler.* Addison-Wesley Publishing Company, 2009.

[2] J. Gosling, *The Java language specification.* Addison-Wesley Professional, 2000.

[3] "The Joos language." `https://www.student.cs.uwaterloo.ca/~cs444/joos.html`. Accessed: 2016-02-03.