CS444/644 Assignment 1 Technical Report

Ho-Yi Fung, Jianchu Li, Zhiyuan Lin

Abstract

bleh bleh bleh

Contents

Chapter 1

Introduction

Introduction is stored in intro.tex. [1]

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]. Product 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 the following algorithm from [1] and is listed in Algorithm 1.

It was also decided that the parser does not do 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 → γ then                    ▷ reduce
        stateStack.pop(|γ|)
        oldNodes = nodeStack.pop(|γ|)
        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`.

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 AST was split into a hierarchy under a CompilationUnit: a CompilationUnit contains some Declarations, would could contain Names, Types, other Declarations, or Statements; Statements could contain other Statements or Expressions; and Expressions could contain other Expressions or Literals.

Chapter 3

Implementation

This project is implemented in Java 8.

3.1 Scanner

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 handwrite 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 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 a slash (/) followed by an asterisk (*) could not form a valid Joos program, the scanner would simply raise an lexical error when such situation arises.

Each token created by the scanner is assigned a type. We list every possible type in an Java enum type called *Symbol*. This enum also contains symbols that will later be used for building parse tree.

Another thing we decided was that since both strings and characters could contain escape characters, we would build a secondary DFA for escape characeters rather independently build it into both string and character scanning.

```
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

3.2 Parser

We decided 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.

To make parse table access as fast as possible, the parse table is implemented practically as an array of hash maps (under a Java class called *ParseActions*. An array is 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.

3.3 Weeder

3.4 Abstract Syntax Tree

The abstract syntax tree is constructed based on the parse tree (also sometimes called concrete syntax tree) from the previous stages. The parse tree is traverse once using recursive depth first search.

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 pattern is applied here. 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 is used to ensure the robustness of our program, especially during implementation of scanner of parser. The JUnit 4 framework provides excellent support for this method. Following the principle of unit testing, each small unit in our program is tested separately. For example, the scanning of each token type was tested separately using parameterized tests. Both positive and negative test cases, derived from the language specification, were used in this process.

Because of the continuous nature of our project, the program is also tested incrementally, e.g. after each module is completed it is 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.

A JUnit test suite is also created so that after each addition/modification all modules could be tested again conveniently to prevent bugs being introduced to the program.

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

In the testing part of assignment 1, we mainly used JUnit testing framework to test different components.
For testing scanner, we made a set of test cases for each token type, and then made them as the parameters of parameterized tests. So the input are the different valid and invalid lexemes and expected output is the name of corresponding token type, or exception for invalid cases. Parameterized tests run the test over and over again using different values, such that we can make sure the scanner could recognize all valid tokens, and throw errors for some illegal cases, for example, the first character of an identifier is a number.

For testing parser and weeder, we referred the specification of Joos1W language, made some small testcases according to the different features of the language. In that way, we can make sure that the parser can perform correct rejecting and accepting action. Then in order to check whether the parser and weeder could handle more complicated cases, we made some big test cases, combined different kinds of situation.
The last stage of testing is run the compiler on marmoset test cases. For those test cases, we also used parameterized testing strategy, so the input are all marmoset test cases for a1, and if the input is accepted by compiler, 0 will be returned, otherwise, compiler returns 42. In this step, the compiler could pass all test cases in marmoset, which means scanner, parser, and weeder are well functional.

Chapter 5

Conclusion

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.