CS 444

Assignment 1 Report

Arturo Pie
Ryan Tinianov
Brandon Walderman

Friday, February 15, 2013

Overview

Our project is composed of a generator and a compiler written in Java 1.6. The generator converts specifications of the lexical and syntactic Joos grammars into classes that the compiler can use to validate input. The compiler contains a Lexer and Parser that use these DFAs the generator creates. The output of the compiler is a parse tree for a valid program, or an error in cases where the program is not valid. To build the tree, the Parser implements the LALR(1) parsing algorithm, obtaining its tokens from the Lexer class. The Lexer reads in ASCII input from a source file, and generates a stream of Token objects.

The Lexer

The Java Language Specification presents a set of rules for Java 1.3's lexical grammar. It might have been possible to implement a Lexer for these rules by hand, but our team chose to build a Lexer generator instead. This was to reduce the chance of error due to an improperly constructed DFA, and also allow for greater flexibility if the specification had to change. Feeding the grammar specification into the generator as a text file was an option, but this itself would have required some basic form of parsing. Therefore, we decided that the Joos lexical grammar would be specified in pure Java, and compiled straight into the Lexer generator. The advantage of this technique is that we can ensure the grammar specification is well-formed at compile time.

To specify a lexical grammar, we construct a series of NFA objects. The NFA class has a number of static methods such as union(), concatenate(), literal(), or digit() that can be chained together to form larger NFAs capable of recognizing Joos tokens. The JoosGrammar class is a big map of string token names to NFAs. Internally, an NFA is a collection of NFA states, each with transitions to other NFA states. To simplify the process of building NFAs, there are CharacterTransitions, RangeTransitions, and EpsilonTransitions. RangeTransitions allow us to specify that a range of characters should cause a transition from one state to another.

Once the NFAs have been constructed, the generator takes the union of all of them, creating a master NFA. This master NFA is then converted into a DFA suitable for use by the Lexer. The generator does this by applying the standard NFA-to-DFA algorithm, taking the epsilon-closure of the various NFA states to create equivalent DFA states. In cases where an accepting DFA state contains more than one accepting NFA state, the NFA that was declared last takes priority.

The DFA from the lexer generator is copied into the compiler project directory, and the Lexer class makes an instance of this DFA. As the lexer reads each character of the source file, it adds the character to the current lexeme. It attempts to return the longest acceptable token it can find, or throws an exception if it enters an error state. This is also the stage where we ensure that the source file is ASCII only. If the lexer encounters a code point beyond 127, it throws an exception. The Token object is the basic unit of output from the Lexer stage. It has fields to

represent what type of token it is, as well as the lexeme that was pulled from the source file. When a token has been scanned, the Lexer constructs and returns a new Token.

The Parser

We completed the lexer generator before proceeding with the parser for this assignment. Since the lexer generator was fully operational, we now had a tool to parse regular languages in text files, so this meant we could specify a context-free grammar (CFG) in a plain text file and read that file to generate a parser. JoosSyntax.txt is a text file that holds all of the rules to our Joos syntactic grammar. A sample structure of this file is:

```
Line 1:    NonTerminal1:
Line 2:    terminal NonTerminal2 terminal
Line 3:    NonTerminal2:
Line 4:    ( NonTerminal3 | NonTerminal4 )
```

This file follows the conventions used in Chapter 18 of the JLS 2nd edition [1]. The name of a non-terminal is given on a single line, followed by a colon. The productions for that rule are given on the lines below. There may be more than one production.

Throughout this phase of the assignment, our team found that the JLS 2nd edition grammar contained some errors and omissions. We attempted to correct for these errors manually by comparing the 2nd edition grammar with the most recent grammar online [2]. Since neither of these grammars were LALR(1) compatible, our team sourced our grammar from an LALR(1) Java grammar available online [3]. This is the grammar that is currently represented in JoosSyntax.txt.

The parser generator interprets the input from JoosSyntax.txt using our Lexer class, and feeds the grammar specification into the Jlalr1.java table generator given on the CS 444 course page. Our first design attempted to convert Jlalr1.java's output into a Java class that contained the parse tablet statically. Unfortunately, since Java methods are limited to 65536 bytes of bytecode, this class's constructor became too large. To work around this limitation, the LALR(1) shift-reduce rules are saved into the file JoosRules.txt, and the Parser initializes itself using this file at runtime.

The ISymbol interface is the base interface for all symbols in the parse tree. ISymbols can be Terminals or NonTerminals. The ISymbol interface has a method to return the Token type represented by a Terminal, or the name of a rule represented by a NonTerminal as strings. With this information, the Compiler can inspect parse tree nodes and make further decisions to build an abstract syntax tree (AST). The AST is capable of weeding invalid input, and presents

friendlier methods to access information such as a method's implementation level, or protection level.

Abstract Syntax Tree

The compiler contains a number of abstract syntax tree (AST) classes to simplify our interpretation of the source code and also to perform weeding. Once a parse tree has been constructed, it is passed to a JoosASTBuilder instance. The JoosASTBuilder applies a set of filters to the entire tree that optionally replace certain parse tree nodes with equivalent AST classes. For example, the IntegerLiteralFactory is a filter that when applied to the parse tree replaces all Terminals representing integer literals with an IntegerLiteralSymbol. The IntegerLiteralSymbol inspects the source text upon construction, and throws an exception if the source is invalid (i.e. if the integer is out of range). If all filters in the JoosASTBuilder are applied without an exception being thrown, the source code is considered valid and the compiler returns a success code (0). Any exceptions that are thrown, from the lexer, parser, or AST builder are caught and an error (42) code is returned.

Testing

Our project uses JUnit for functional testing of the lexer and parser, plus integration and acceptance testing for the whole compiler. Functional testing ensures that the lexer outputs the expected token stream for a given character stream, and similarly, that the parser outputs an expected parse tree for a given token stream. Integration testing exercises the lexer and parser together on simple input strings. Acceptance testing executes our compiler on a set of input files that were created to verify our weeding and parsing rules. Additionally, our acceptance tests include the Marmoset test cases for verification before submitting to marmoset.

To facilitate testing, both the Lexer and Parser classes were designed to be loosely coupled. For example, rather than a Lexer opening a file reader given a file path, the Lexer simply takes a generic Reader as a constructor argument. This means that we can exercise the Lexer in a test environment by simply passing a StringReader instead of a FileReader. The Parser similarly takes a reference to a generic ILexer in its constructor. This way, it can be tested using a MockLexer that returns a predetermined list of Tokens rather than a real Lexer which operates on input streams (Readers).

References

[1] Gosling et al. "The Java Language Specification." Second Edition. *Addison-Wesley,* 2000.

[2] Oracle. "Java Language Specification." http://docs.oracle.com/javase/specs/jls/se7/html/jls-18.html [Accessed 7 Feb 2013]

[3] Radu Iosif. “Java Syntactic Grammar.” *Politecnico Di Torino.* http://www.dai-arc.polito.it/dai-arc/manual/tools/jcat/main/node224.html#javasyntacticgrammar [Accessed 13 Feb 2013]