CS 444 - Compiler Validation Documentation
Assignment 1

Yimeng Wang, Wenzhu Man, Matt Baker

February 15, 2013

Introduction

The initial phase of the compiler deals with the scanning and parsing of a proposed joos source file. The Joos W1 specification is a set of rules which must be followed by any valid joos source file. At the lowest level, a joos source file can be separated into a collection of symbols consisting of one or more characters called Tokens. A token is considered valid if it is defined in the Joos specification. The specification also places restrictions and derives meaning on the way tokens are ordered. These restrictions form the logical syntax of the Joos language. To validate a file, the compiler takes a proposed joos source file, splits it into a sequence of Joos tokens, and then checks that these tokens conform to the syntactical structure described in the specification. A file is considered valid if it can be completely split into a sequence of tokens and if the sequence conforms to the syntactical rules laid out by the Joos specification.

The process of validating a proposed joos file is split into 4 steps:

1. Initialization
2. Lexing
3. Parsing
4. Weeding

At a glance, initialization sets the compiler up to prepare it to accept a source file for input. Lexing attempts to break the input up into a list of valid Joos tokens. Parsing then uses the joos syntax to determine if the list of tokens are in a meaningful order. While parsing, the compiler also builds a tree representation of the syntax production rules which is scanned once more in the weeding stage. Weeding then checks the compilation tree against a further set of syntax restrictions, finalizing the parsing process. Weeding allows the parser to ignore some aspects of the joos syntax rules and simplifies the parsing process. If a proposed source file does not follow joos specification, the compiler will fail at one of these steps and return a value of `42`. If the file is valid, it will be successfully processed through each of these steps and output a value of `0`.

The compiler itself is written in Java divided into a set of packages. Each package contains a collection of files used to accomplish a specific step of the validation process. Initialization is handled in the Main.java file, lexing is handled by the scanner package, parsing is handled by the parser package, and weeding handled with the weeding package. In addition, the compiler also makes use of a collection of resource files. These files are seperate from the compiler and together describe the lexical and syntactical rules which make up the Joos specification.

Initialization

Primary files: Main.java
The first stage of validation process configures the compiler. This step initializes each of the components used in validation. This step also facilitates communication between each of the components, by passing the output of one step into the next. As part of the initialization process for each component, this step also links the parser and lexer with the proper resource file.

Important Methods and Structures

Scanner.`scanner`: Used to tokenize a proposed joos file. LR1Parser.parser: Used to parse the joos file

Preprocessor.preprocessor(): Modifies the token list before it is parsed.

Main(): Initializes each of the above variables. At this point, any resource file needed by a structure is passed as a parameter of its constructor. At the end of initialization, the compiler has all of the validation rules stored internally.

execute(): Called after the **Main()** constructor finishes. This method actually invokes each step of the validation process. It returns an *AST* generated by the weeder.

List **`tokens`:** A list to hold the tokens in a proposed joos source file.

ParseTree **`parsetree`:** A structure holding a tree generated by the parser.

Lexing

Primary Files: DFA.java, Scanner.java, Token.java, joos.dfa
The lexing phase is the first phase to follow initialization. This phase is concerned with splitting up a proposed joos source file into a sequence of valid joos tokens. From **Main()** it is given both the location of the **joos.dfa** file and the source file to be validated. Upon completion, this stage returns to **Main()** the list of tokens that make up the source file.

Token.java

This file contains the structure used to define a single joos source token.

Important Methods and Structures

Token.`kind`: A string describing what kind of token is represented.
Token.`lexeme`: A string containing the characters used to represent this token in the source.
getKind(), getLexeme(): Returns the respective string in the Token.

DFA.java

This file defines the collection of tokens which are found in a valid joos source file. The class is given the location of a joos.dfa resource file. This file defines a DFA describing the language of all acceptable joos tokens and constructs a method which allows Scanner.java to traverse it. The resource file follows the structure used to represent DFAs in CS 241.

One early problem encountered in the design of the compiler was how to compactly describe all of the possible character transitions for the scanner to take. For example, a token representing an ID could accept any alphanumeric input and a block comment accepts anything, including whitespace, as long as it is surrounded by delimiters. Cases like this made the joos.dfa rather long as each character that can be read from a state would be given its own line. To circumvent this, a couple of keywords were created to be used in the Character slot of the tuple to represent multiple possible characters. For instance, the Character 'digit' represents any numeric character. These special character representations are defined specifically within a *HashMap* **symbolregextable** in **DFA.java** and is constructed in the **DFA.regexForKeyword()** function.

Important Methods and Structures

Map<String, String> `symbolregextable`**:** A map holding all of the keywords used in a transition rule.

Map<String, Map<String, String> > `Transactions`**:** Holds the transition function of the DFA. The first key represents a state in the DFA. The second key is the input given at that state and is mapped to the state to transition to.

Class RegexTransaction: A structure containing two strings. Defines a keyword used in a .dfa resource file.

void parseDfaFile(): Reads the .dfa resource file. Constructs a transition function and places it in Transactions. It also finds the start state and accepting states of the DFA.

String nextStateFor(): Called by the scanner. This method takes a state and a character and returns the transition described by the DFA if it exists. Otherwise it returns null.

Scanner.java

This class contains the methods used to read and tokenize a proposed joos source file. It relies on the methods and structures defined in **DFA.java** and **Token.java**. The class is invoked from **Main.execute()** with a call to **Scanner.filetoTokens()** where it is passed the location to the proposed file to be scanned. As Scanner reads the proposed joos source file, it traverses the DFA defined in **DFA.java** maintaining a state and calling **DFA.NextStateFor**().

Important Methods and Structures

String **extractToken.state:** The state the scanner is at in the token DFA.

String **extractToken.lexeme:** The lexeme of the read token.

fileToTokens(): This method takes a File as a parameter, opens an input stream to the file and reads the entire file into the byte array InBytes. The byte array is then converted to the string *inStr* which is passed to **Scanner.stringToTokens()**.

stringToTokens(): This method takes the string passed by **fileToTokens()** and splits it into constituent Tokens. **extractToken()** is called on each character within the given string.

extractToken(): This method takes an array of characters which are separated by whitespace in the proposed source file. Each time the method is called, the scanner is just beginning to scan look for a new token so it is placed in the starting state defined in the **.dfa** resource file. The method then reads each char within the given array and uses **DFA.nextStateFor()** to determine a new state to enter given the current state and last character read. After the character array is read, the method checks to see if the scanner is in an accept state and if so adds the token represented by that state to a list of read tokens.

Parsing

Primary Files: LR1.java, LR1Parser.java, ParseTree.java
After the **lexer** finishes successfully, the proposed joos file is represented as a *List* of type *Token.* The parsing phase then takes this list and applies an LR(1) analysis on it.

LR1.java
This file reads an **.lr1** resource file and constructs a transition table for the parser to follow.

Important Methods and Structures

parseFile(): This method accepts the lr1 resource file as a File parameter. The given file is then used to construct the transition table. The rules described in the lr1 file are stored in a HashMap denoted transitionRules. This HashMap contains Integer, HashMap pairs. The inner HashMap stores String, Action pairs. Like the Scanner, as the parser scans the list of tokens, it enters different states. The Integer key represents a state the parser is in and the second HashMap takes the token's kind variable as the key for the next Action to take. The Action describes whether the parser is to shift to another state or to reduce a string of tokens to the lefthand side of a CFG production rule.

ParseTransition(): This method takes a line from the lr1 resource file and enters it into the transition table.

Class Action: One of the things to consider when building the parser is the two different actions it can take on reading a token. The Action class is an abstract class which is extended by two other classes, ActionReduce which defines a reduce action and ActionShift which defines a shift action. Both subclasses contain an integer which either represents a state to shift to or a production rule to reduce by. This integer is given through the Action.getInt() method.

LR1Parser.java
This file holds methods responsible for actually parsing a list of tokens. Like the Scanner it relies on a structure created in another java class, namely LR1.java. As the parser scans through the list of tokens it records its current state which is used as the Integer key for the transitionrules HashMap constructed in LR1.java. The scanned token's kind variable is used as the string key for the internal HashMap.

Important Methods and Structures

parseStack: A Stack of TransitionState structures. When the parser encounters a shift action it creates a new TransitionState instance, stores the token and current state, and pushes it onto the stack.

TransitionState: An internal class which holds a token read by the parser the state the parser was in when the token was read.

parseTokens(): This method takes as input the list of tokens generated by Scanner.java. The method then iterates over the tokens. On each iteration, the parser's current Symbol and the next token's kind are used as input into the function LR1.actionFor(). This returns an object of type Action. The method checks the subclass of the returned value and either performs a shift or reduce action. On a shift action, the current token is pushed onto the parseStack stack and the Action's getInt() function returns the new state of the parser. On a reduce, the getInt()function returns the number of symbols to be popped off of the parseStack stack. These are then placed into nodes and pushed into a parse tree. The method succeeds if on the last token parsed it is in the start state described in the .lr1file.

ParseTree.java: This file is a collection of classes used to construct a parse tree as the proposed source is parsed. Each node in the tree uses a bridge pattern to provide a standard interface for the weeder to traverse it.

Weeding

Primary Files: joos.ast.*

The final stage scans through the tree generated in the parsing stage and a few more lexical validations. This stage is kept separate from the Parsing stage to reduce the complexity of the syntax CFG. There is one primary structure, the *AST* which represents an abstract symbol tree. This tree is constructed using a nested tree structure containing a data structure at each node. Each node in the tree makes use of the visitor pattern. The weeder creates multiple visitors on each node for each check which is run on a node. A check verifies a specific rule of the Joos specification ignored by the Parser.

Test Methodology

A separate java file was used to facilitate testing. This file takes a list of testcases and runs the validation process against each one. Each test case has a comment describing the expected result of the validation process. These comments are extracted and compared to the validation result. Any mismatched result raises an exception and is recorded.