CS 444, Winter Term 2018

A1 Documentation

By Alexandru Gatea, Carl Zhou

Due on February 16

Preprocessing

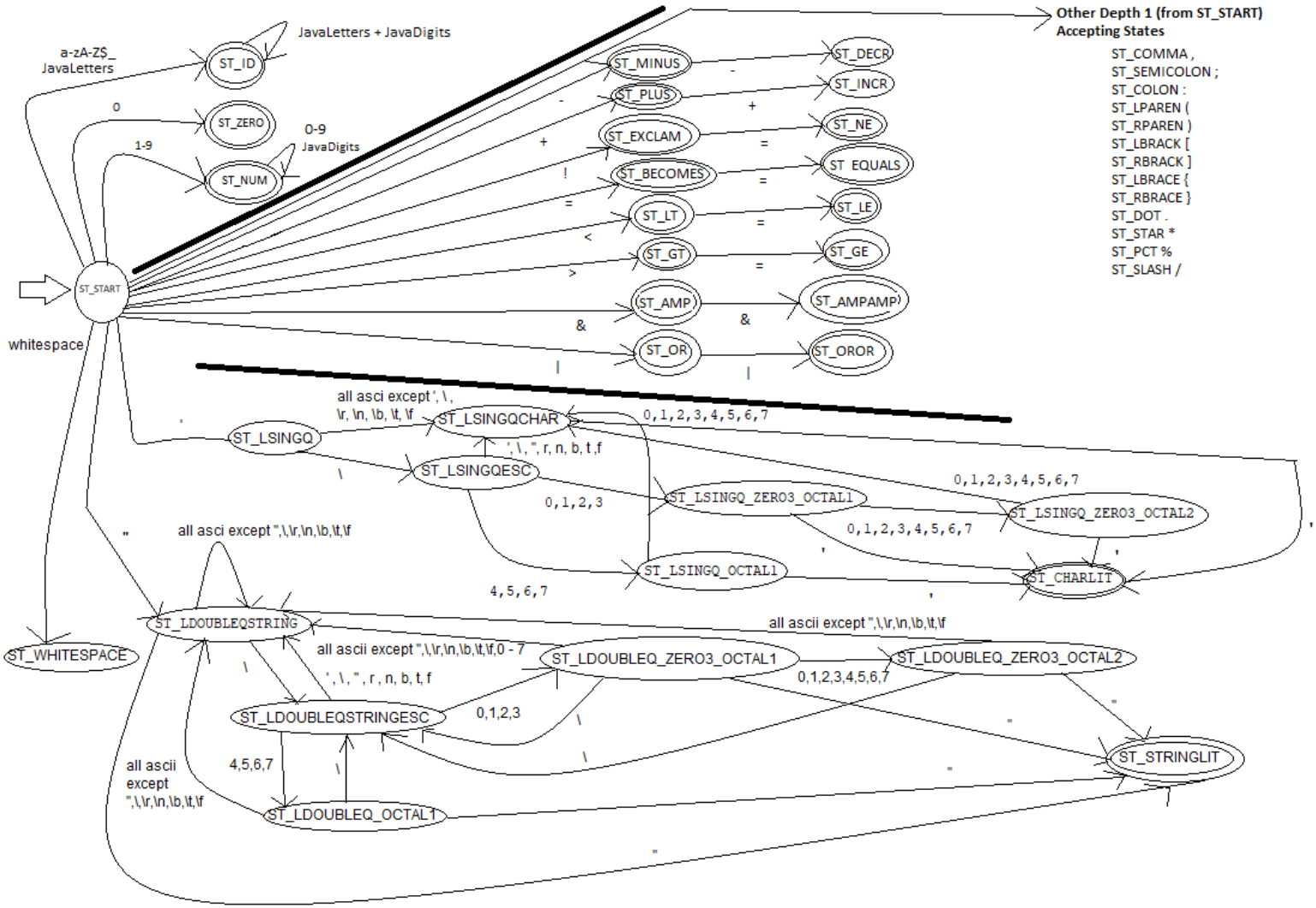
The main program creates a JoosCompiler and passes the first argument to it. The JoosCompiler has a method compile that opens and reads the input file, constructs and applies a CommentsFilter, an InputScanner, an InputParser, and a Weeder in this sequence. If any one of them throws an exception, it returns 42 to the main program, otherwise it returns 0.

The CommentsFilter class uses a Regex to find in-line comments (starting with //) and multi-line comments (delimited by /*, */) and to remove the contents of the comments.

Scanner

We implemented the Tokenizer by implementing a DFA class that has a list of Accepting States, a Map that maps each State to a Character $\mapsto$ State Map, and methods for constructing the DFA and for applying the transition function (i.e. the Map) to a given (state, character) pair. The State in this DFA class is an Enum type containing states named ST_statename (e.g. ST_ID) for every state in the DFA we developped for the tokens (see page 2). Subsequently, in the InputScanner constructor, we construct a DFA object with precisely the transitions in this diagram.

The InputScanner class contains this DFA object, and has two methods: one for expanding escape sequences within an input string and one for scanning. The scan method takes as input the comments-free input and attempts to tokenize it; if successful, it returns a list of Token objects and if not successful, it throws a ScanningError exception. For every line in the input, the scan method starts at ST_START and keeps getting the next character and applying DFA transitions while keeping track of the lastAccepting state found and the index where it occured, and the startIndex of the current token. After each application of the DFA, it checks if the new state is accepting (and updates the last-accepting-State info appropriately). When the end of the line is reached or when the state returned by the DFA is ST_ERR, if there was a last accepting state seen then the input from the start of the current token is tokenized (using the Token method getKind), the resulting Token is added to the list of tokens and the process begins again from ST_START. It is possible that the getKind method in Token returns the kind "ERROR", in which case scan throws a ScanningError; also, if getKind returns "StringLiteral" or "CharacterLiteral", the method expandEscape is applied to the lexeme. Otherwise, if either the end of the line is reached without the current state being accepting, or if the state returned by the DFA is ST_ERR without there being a previous accepting state seen, or if the state returned by the DFA is ST_INCR or ST_DECR (indicating ++ or –) the scan method throws a ScanningError.


Other Depth 1 (from ST_START)
Accepting States
ST_COMMA ,
ST_SEMICOLON ;
ST_COLON :
ST_LPAREN (
ST_RPAREN)
ST_LBRACK [
ST_RBRACK]
ST_LBRACE {
ST_RBRACE }
ST_DOT .
ST_STAR *
ST_PCT %
ST_SLASH /
ST_START
whitespace
a-zA-Z$_
JavaLetters
ST_ID
JavaLetters + JavaDigits
0
ST_ZERO
1-9
ST_NUM
0-9
JavaDigits
ST_MINUS
ST_DECR
ST_PLUS
ST_INCR
ST_EXCLAM
ST_NE
ST_BECOMES
ST_EQUALS
ST_LT
ST_LE
ST_GT
ST_GE
ST_AMP
ST_AMPAMP
ST_OR
ST_OROR
ST_LSINGQ
all asci except ', \ ,
\r, \n, \b, \t, \f
ST_LSINGQCHAR
0,1,2,3,4,5,6,7
', \, ", r, n, b, t ,f
ST_LSINGQESC
0,1,2,3
ST_LSINGQ_ZERO3_OCTAL1
ST_LSINGQ_ZERO3_OCTAL2
ST_LSINGQ_OCTAL1
ST_CHARLIT
all asci except ",\,\r,\n,\b,\t,\f
4,5,6,7
ST_LDOUBLEQSTRING
ST_WHITESPACE
all ascii except ",\,\r,\n,\b,\t,\f,0 - 7
', \, ", r, n, b, t, f
ST_LDOUBLEQ_ZERO3_OCTAL1
ST_LDOUBLEQ_ZERO3_OCTAL2
ST_LDOUBLEQSTRINGESC
ST_STRINGLIT
all ascii
except
",\,\r,\n,\b,\t,\f
ST_LDOUBLEQ_OCTAL1

The Token class contains fields for the kind (e.g. "ID", "IntegerLiteral") which match the literals in the Joos grammar (for the parser), for the lexeme (the actual token), for the line + col number of the token, and for a static flag _isLastTokenMinus indicating if the last seen token was a minus, and a method getKind that takes a State and a lexeme and determines (and returns) the kind. This is done by a large case analysis based on the State. Most of the cases are straightforward, but there are 2 special cases:
-for ST_NUM it checks that the lexeme is at most 2^31 if _isLastTokenMinus is true, and at most $2^31 - 1$ if it is false
-for ST_ID, it checks that the lexeme is any of the allowed keywords (and the kind is that keyword); if not, it checks that the lexeme is not contained in the list of the non-allowed reserved keywords
If any error is found, it returns "ERROR". At the end, it updates the ST_MINUS it updates the _isLastTokenMinus flag.

The biggest challenge in the Scanner was determining the DFA to be used (the transitions for the String Literals and Character Literals were the most complicated, because of the many escape sequence rules). Nevertheless, we decided to hardcode these DFA transitions in the InputScanner, using the diagram from pg. 2, rather than using an NFA and writing a method for the NFAtoDFA conversion, because it allowed us to know exactly what the States in the DFA were, which made it easier to determine the kind for each Token.

Parser

For the parser, we used the grammar from the JLS 2 document (with some modifications to account for shift-reduce and reduce-reduce conflicts due to the dangling else problem for example) and used the provided LALR1-machine generating code to get a file with the LALR1 DFA. The InputParser class has a constructor that reads in this file and stores the rules (as a Map from an integer (the rule #) to a list of strings (the actual rule)) as well as the actions (a list of Action objects). InputParser also has a method findAction that searches the list of actions for the appropriate action based on a state and a lookahead-symbol, and returns an ERROR action if no action is found.

The actual parsing is done in the parse method, that takes in as input a list of tokens and returns the root Node of the resulting parse tree (or throws a ScanningError exception if an error is found). While keeping a State stack (with LALR1 DFA states) and a Node Stack, it reads the list of tokens; for each token, it gets its kind (which will be the lookahead-symbol) and keeps applying findAction and performing the resulting reduce action (by popping the correct number of objects off the stacks, and pushing the state and node corresponding to the rule given by the reduction; the popped nodes are put as the children of the pushed node) until the action is a shift action. At this point, it performs the shift (by pushing a leaf node and the new state) and goes into the next iteration (for the next token). If at any point findAction returns ERROR, or the stack is empty prematurely, or if the stack at the end does not have 3 objects (one for BOF, one for the parse tree/start state, one for EOF) then it throws a ScanningError.

The challenge in this phase was the grammar (because of some inherent conflicts, and be-

cause we had to figure out which rules given in the JLS document for Java needed to be changed to accomodate the restrictions of Joos). We overcame this by trying different variations of the rules, as well as following advice from class and from the JLS document and the Joos language features from the CS 444 website.

Weeder

First, we added a method in the Node class for flattening lists (such as the list of modifiers). The Weeder class has several methods to perform the weeding, and a method performWeeding that calls flattenTree followed by the weeding methods. All of the weeding methods throw a WeedingError if they catch an error. To simplify these methods (most of them have to do with modifiers), the first weeding method determines the modifiers for each Modifiers node (and stores them as a list of strings in the node), checking that no duplicates are found and that exactly one of PUBLIC, PROTECTED occurs in each modifier list, and stores that modifiers for the top class/interface as well as CLASS/INTERFACE in the root node. The rest of the methods are straightforward tree traversals: rootModifiersCheck (checks for the class/interface modifiers), methodModifiersCheck, interfaceMethodCheck, fieldCheck, constructorCheck (these last three are very similar so we wrote one more general method and used three wrapper methods for these three), largeIntCheck (checking that 2^{31} occurs only after a unary minus), baseNameCheck (checking that the name of the class/interface matches the name of the file), explicitConstructorCheck, castCheck (checking that the cast parentheses only enclose a basic type or an object type (not a more general expression)).

We decided to perform the weeding before the AST building because, even some of the checks would have benefited from the AST structure, others such as checking that there are no duplicate modifiers would have been more difficult. Moreover, this allowed us to work on the AST without the time pressure of having to the Weeding right after.

AST Construction

This phase was done by following the suggestion from class, namely to write different node classes for the node types that implement an abstract node and by setting things up so that tree traversals are performed using the visitor pattern.

Testing

Other than the Marmoset tests, we wrote unit tests for every part of the code (each method in each class) as well tests for conditions not checked in the Marmoset tests but that are present either in the JLS or in the Joos page on the CS 444 website. To elaborate on this latter set of tests, we had tests for:
-an empty input file or an input file with only whitespace
-an input file with only imports and/or a package (no class/interface)
-a class or a method that has both "public" and "protected" as modifiers, or has duplicate modifiers
-an abstract constructor
-an abstract method in a non-abstract class
-an array initialized with array data
-an invalid input program (e.g. inter/* */face A{} or input with an invalid character such

as @)
-a file where the imports, package and class/interface declaration are in the wrong order
-a cast to an expression (e.g. int j = (4 + 54) k;)
-multiple variables in a declaration
-multiple classes in the input file or nested classes
-no explicit constructor
and many others. All of these tests passed. Together with the Marmoset tests, these give a thorough check against all possible errors in the input program that can be caught at the scanning/parsing stage.