Project 4 (Scanning and Parsing With JavaCC)

● Graded

2 Days, 23 Hours Late

Student

Giancarlos Marte

Total Points

99.75 / 100 pts

Autograder Score

80.0 / 80.0

Passed Tests

Problem 0. Compiling j-- (2/2)
Problem 1. Multiline Comment (15/15)
Problem 2. Operators (15/15)
Problem 3. Reserved Words (15/15)
Problem 4. Literals (15/15)
Problem 5: Long and Double Basic Types (10/10)
Problem 6: Operators (10/10)
Problem 7: Conditional Expression (10/10)
Problem 8: Do Statement (10/10)
Problem 9: For Statement (10/10)
Problem 10: Break Statement (10/10)
Problem 11: Continue Statement (10/10)
Problem 12: Switch Statement (10/10)
Problem 13: Exception Handlers (10/10)
Problem 14: Interface Type Declaration (10/10)

Question 2

Code Clarity and Efficiency

9.75 / 10 pts

Multiline Comment

- ✔ **+ 0.25 pts** Passed all tests
- ✔ **+ 0.125 pts** Changes to scanner part of j--.jj commented adequately
- ✔ **+ 0.125 pts** Followed good programming practices

Operators

- ✔ **+ 0.25 pts** Passed all tests
- ✔ **+ 0.125 pts** Changes to scanner part of j--.jj commented adequately
- ✔ **+ 0.125 pts** Followed good programming practices

Reserved Words

- ✔ **+ 0.25 pts** Passed all tests
- ✔ **+ 0.125 pts** Changes to scanner part of j--.jj commented adequately
- ✔ **+ 0.125 pts** Followed good programming practices

Literals

+ 0.25 pts Passed all tests

- ✔ **+ 0.125 pts** Changes to scanner part of j--.jj commented adequately
- ✔ **+ 0.125 pts** Followed good programming practices

Long and Double Basic Types

- ✔ **+ 0.3 pts** Passed all tests
- ✔ **+ 0.25 pts** Changes to parser commented adequately
- ✔ **+ 0.25 pts** Followed good programming practices

Operators

- ✔ **+ 0.3 pts** Passed all tests
- ✔ **+ 0.25 pts** Precedence captured correctly
- ✔ **+ 0.125 pts** Changes to parser commented adequately

✔ **+ 0.125 pts** Followed good programming practices

Conditional Expression

✔ **+ 0.3 pts** Passed all tests

✔ **+ 0.25 pts** Precedence captured correctly

✔ **+ 0.125 pts** Changes to parser commented adequately

✔ **+ 0.125 pts** Followed good programming practices

Do Statement

✔ **+ 0.3 pts** Passed all tests

✔ **+ 0.25 pts** Changes to parser commented adequately

✔ **+ 0.25 pts** Followed good programming practices

For Statement

✔ **+ 0.3 pts** Passed all tests

✔ **+ 0.25 pts** Changes to parser commented adequately

✔ **+ 0.25 pts** Followed good programming practices

Break Statement

✔ **+ 0.3 pts** Passed all tests

✔ **+ 0.25 pts** Changes to parser commented adequately

✔ **+ 0.25 pts** Followed good programming practices

Switch Statement

✔ **+ 0.3 pts** Passed all tests

✔ **+ 0.25 pts** Changes to parser commented adequately

✔ **+ 0.25 pts** Followed good programming practices

Continue Statement

✔ **+ 0.3 pts** Passed all tests

✔ **+ 0.25 pts** Changes to parser commented adequately

✔ **+ 0.25 pts** Followed good programming practices

Exception handlers

✔ **+ 0.3 pts** Passed all tests

✔ **+ 0.25 pts** Changes to parser commented adequately

✔ **+ 0.25 pts** Followed good programming practices

Interface Type Declaration

✔ **+ 0.3 pts** Passed all tests

✔ **+ 0.25 pts** Changes to parser commented adequately

✔ **+ 0.25 pts** Followed good programming practices

+ 0 pts Does not meet expectations

Question 3

Notes File **10** / 10 pts

✔ **+ 10 pts** Provides a clear high-level description of the project in no more than 200 words

+ 0 pts Does not meet our expectations (see point adjustment and associated comment)

+ 0 pts Missing

Autograder Results

Problem 0. Compiling j-- (2/2)

ant

Problem 1. Multiline Comment (15/15)

javaccj-- -t tests/MultiLineComment.java

Problem 2. Operators (15/15)

javaccj-- -t tests/Operators1.java

Problem 3. Reserved Words (15/15)

javaccj-- -t tests/Keywords.java

Problem 4. Literals (15/15)

javaccj-- -t tests/IntLiterals.java
javaccj-- -t tests/LongLiterals.java
javaccj-- -t tests/DoubleLiterals.java
javaccj-- -t tests/MalformedLiterals.java □
'1\t [256 chars] = .1e\n5\t : <IDENTIFIER> = eE\n5\t : "+" = +[1915 chars]F> =' != '1\t [256 chars] = .1\n5\t : <IDE
Diff is 3065 characters long. Set self.maxDiff to None to see it.

Problem 5: Long and Double Basic Types (10/10)

javaccj-- -p tests/BasicTypes.java

Problem 6: Operators (10/10)

javaccj-- -p tests/Operators2.java

Problem 7: Conditional Expression (10/10)

javaccj-- -p tests/ConditionalExpression.java

Problem 8: Do Statement (10/10)

javaccj-- -p tests/DoStatement.java

Problem 9: For Statement (10/10)

javaccj-- -p tests/ForStatement.java

Problem 10: Break Statement (10/10)

javaccj-- -p tests/BreakStatement.java

Problem 11: Continue Statement (10/10)

javaccj-- -p tests/ContinueStatement.java

Problem 12: Switch Statement (10/10)

javaccj-- -p tests/SwitchStatement.java

Problem 13: Exception Handlers (10/10)

javaccj-- -p tests/ExceptionHandlers.java

Problem 14: Interface Type Declaration (10/10)

javaccj-- -p tests/Interface.java

Submitted Files

notes.txt

Download

1. Provide a high-level description (ie, using minimal amount of technical
jargon) of the project in no more than 200 words.
The goal of the project was to make a generated scanner and parser for j-- using JavaCC. The implementation was done
by using the Java grammar documentation as a guide and the base j--.jj code. The first part of the assignment was to
build the scanner by creating the operators and literals added in the previous projects. The next part was to work on
the parser to support these new features. Lastly, the implementation for things like for, do while, try, error
handlers and interfaces were added to the parser section. To add all of these parts, several methods based off the
Java grammar documentation were made as well. Overall, this project was a way to learn about an alternative and
simpler way to make a parser and scanner.

2. Did you receive help from anyone? List their names, status (classmate,
CS451/651 grad, TA, other), and the nature of help received.

Name	Status	Help Received
Swami Iyer	professor	Piazza help on problem 9
Ramsey Harrison	TA	Piazza help on problem 9 and office hours for the same problem

3. List any other comments here. Feel free to provide any feedback on how
much you learned from doing the assignment, and whether you enjoyed
doing it.

It was interesting to combine the JavaCC code and java code. The project was a bit difficult at the beginning due
to having to learn how to use JavaCC and the lack of help that IntelliJ provided. However, after spending time
doing it, things became a lot easier and more enjoyable.

TokenInfo.java Download

```java
// Copyright 2012- Bill Campbell, Swami Iyer and Bahar Akbal-Delibas

package jminusminus;

/**
 * An enum of token kinds. Each entry in this enum represents the kind of a token along with its
 * image (string representation).
 */
enum TokenKind {
    // End of file.
    EOF(""),

    // Reserved words.
    ABSTRACT("abstract"), BOOLEAN("boolean"), CHAR("char"), CLASS("class"), ELSE("else"),
    EXTENDS("extends"), IF("if"), IMPORT("import"), INSTANCEOF("instanceof"), INT("int"),
    NEW("new"), PACKAGE("package"), PRIVATE("private"), PROTECTED("protected"),
    PUBLIC("public"), RETURN("return"), STATIC("static"), SUPER("super"), THIS("this"),
    VOID("void"), WHILE("while"),
    BREAK("break"), CASE("case"), CATCH("catch"), CONTINUE("continue"), DEFAULT("default"),
    DO("do"), DOUBLE("double"), FINALLY("finally"), FOR("for"), IMPLEMENTS("implements"),
    INTERFACE("interface"), LONG("long"), SWITCH("switch"), THROW("throw"),
    THROWS("throws"), TRY("try"),

    // Operators.
    ASSIGN("="), DEC("--"), EQUAL("=="), GT(">"), INC("++"), LAND("&&"),
    LE("<="), LNOT("!"), MINUS("-"), PLUS("+"), PLUS_ASSIGN("+="), STAR("*"),
    DIV("/"), REM("%"), ALSHIFT("<<"), ARSHIFT(">>"), LRSHIFT(">>>"), NOT("~"),
    AND("&"), XOR("^"), OR("|"), QUESTION("?"), NOT_EQUAL("!="), DIV_ASSIGN("/="),
    MINUS_ASSIGN("-="), STAR_ASSIGN("*="), REM_ASSIGN("%="),
    ARIGHTSHIFT_ASSIGN(">>="), LRSHIFT_ASSIGN(">>>="), GE(">="), ALSHIFT_ASSIGN("<<="), LT("<"),
    XOR_ASSIGN("^="), OR_ASSIGN("|="), LOR("||"), AND_ASSIGN("&="),

    // Separators.
    COMMA(","), DOT("."), LBRACK("["), LCURLY("{"), LPAREN("("), RBRACK("]"), RCURLY("}"),
    RPAREN(")"), SEMI(";"), COLON(":"),

    // Identifiers.
    IDENTIFIER("<IDENTIFIER>"),

    // Literals.
    CHAR_LITERAL("<CHAR_LITERAL>"), FALSE("false"), INT_LITERAL("<INT_LITERAL>"), NULL("null"),
    STRING_LITERAL("<STRING_LITERAL>"), TRUE("true"),
    // New Literals
    LONG_LITERAL("<LONG_LITERAL>"), DOUBLE_LITERAL("<DOUBLE_LITERAL>");

    // The token kind's string representation.
```

```
47      private String image;
48
49      /**
50       * Constructs an instance of TokenKind given its string representation.
51       *
52       * @param image string representation of the token kind.
53       */
54      private TokenKind(String image) {
55          this.image = image;
56      }
57
58      /**
59       * Returns the token kind's string representation.
60       *
61       * @return the token kind's string representation.
62       */
63      public String tokenRep() {
64          if (this == EOF) {
65              return "<EOF>";
66          }
67          if (image.startsWith("<") && image.endsWith(">")) {
68              return image;
69          }
70          return "\"" + image + "\"";
71      }
72
73      /**
74       * Returns the token kind's image.
75       *
76       * @return the token kind's image.
77       */
78      public String image() {
79          return image;
80      }
81  }
82
83  /**
84   * A representation of tokens returned by the Scanner method getNextToken(). A token has a kind
85   * identifying what kind of token it is, an image for providing any semantic text, and the line in
86   * which it occurred in the source file.
87   */
88  public class TokenInfo {
89      // Token kind.
90      private TokenKind kind;
91
92      // Semantic text (if any). For example, the identifier name when the token kind is IDENTIFIER
93      // . For tokens without a semantic text, it is simply its string representation. For example,
94      // "+=" when the token kind is PLUS_ASSIGN.
95      private String image;
```

```
96
97      // Line in which the token occurs in the source file.
98      private int line;
99
100     /**
101      * Constructs a TokenInfo object given its kind, the semantic text forming the token, and its
102      * line number.
103      *
104      * @param kind  the token's kind.
105      * @param image the semantic text forming the token.
106      * @param line  the line in which the token occurs in the source file.
107      */
108     public TokenInfo(TokenKind kind, String image, int line) {
109        this.kind = kind;
110        this.image = image;
111        this.line = line;
112     }
113
114     /**
115      * Constructs a TokenInfo object given its kind and its line number. Its image is simply the
116      * token kind's string representation.
117      *
118      * @param kind the token's identifying number.
119      * @param line the line in which the token occurs in the source file.
120      */
121     public TokenInfo(TokenKind kind, int line) {
122        this(kind, kind.image(), line);
123     }
124
125     /**
126      * Returns the token's kind.
127      *
128      * @return the token's kind.
129      */
130     public TokenKind kind() {
131        return kind;
132     }
133
134     /**
135      * Returns the line number associated with the token.
136      *
137      * @return the line number associated with the token.
138      */
139     public int line() {
140        return line;
141     }
142
143     /**
144      * Returns the token's string representation.
```

```
145      *
146      * @return the token's string representation.
147      */
148     public String tokenRep() {
149         return kind.tokenRep();
150     }
151 
152     /**
153      * Returns the token's image.
154      *
155      * @return the token's image.
156      */
157     public String image() {
158         return image;
159     }
160 }
161 
```

Scanner.java

Download

```java
// Copyright 2012- Bill Campbell, Swami Iyer and Bahar Akbal-Delibas

package jminusminus;

import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.io.LineNumberReader;
import java.util.Hashtable;

import static jminusminus.TokenKind.*;

/**
 * A lexical analyzer for j--, that has no backtracking mechanism.
 */
class Scanner {
    // End of file character.
    public final static char EOFCH = CharReader.EOFCH;

    // Keywords in j--.
    private Hashtable<String, TokenKind> reserved;

    // Source characters.
    private CharReader input;

    // Next unscanned character.
    private char ch;

    // Whether a scanner error has been found.
    private boolean isInError;

    // Source file name.
    private String fileName;

    // Line number of current token.
    private int line;

    /**
     * Constructs a Scanner from a file name.
     *
     * @param fileName name of the source file.
     * @throws FileNotFoundException when the named file cannot be found.
     */
    public Scanner(String fileName) throws FileNotFoundException {
        this.input = new CharReader(fileName);
        this.fileName = fileName;
```

```
47        isInError = false;
48
49        // Keywords in j--
50        reserved = new Hashtable<String, TokenKind>();
51        reserved.put(ABSTRACT.image(), ABSTRACT);
52        reserved.put(BOOLEAN.image(), BOOLEAN);
53        reserved.put(CHAR.image(), CHAR);
54        reserved.put(CLASS.image(), CLASS);
55        reserved.put(ELSE.image(), ELSE);
56        reserved.put(EXTENDS.image(), EXTENDS);
57        reserved.put(FALSE.image(), FALSE);
58        reserved.put(IF.image(), IF);
59        reserved.put(IMPORT.image(), IMPORT);
60        reserved.put(INSTANCEOF.image(), INSTANCEOF);
61        reserved.put(INT.image(), INT);
62        reserved.put(NEW.image(), NEW);
63        reserved.put(NULL.image(), NULL);
64        reserved.put(PACKAGE.image(), PACKAGE);
65        reserved.put(PRIVATE.image(), PRIVATE);
66        reserved.put(PROTECTED.image(), PROTECTED);
67        reserved.put(PUBLIC.image(), PUBLIC);
68        reserved.put(RETURN.image(), RETURN);
69        reserved.put(STATIC.image(), STATIC);
70        reserved.put(SUPER.image(), SUPER);
71        reserved.put(THIS.image(), THIS);
72        reserved.put(TRUE.image(), TRUE);
73        reserved.put(VOID.image(), VOID);
74        reserved.put(WHILE.image(), WHILE);
75
76        // New reserved words
77        reserved.put(BREAK.image(), BREAK);
78        reserved.put(CASE.image(), CASE);
79        reserved.put(CATCH.image(), CATCH);
80        reserved.put(CONTINUE.image(), CONTINUE);
81        reserved.put(DEFAULT.image(), DEFAULT);
82        reserved.put(DO.image(), DO);
83        reserved.put(DOUBLE.image(), DOUBLE);
84        reserved.put(FINALLY.image(), FINALLY);
85        reserved.put(FOR.image(), FOR);
86        reserved.put(IMPLEMENTS.image(), IMPLEMENTS);
87        reserved.put(INTERFACE.image(), INTERFACE);
88        reserved.put(LONG.image(), LONG);
89        reserved.put(SWITCH.image(), SWITCH);
90        reserved.put(THROW.image(), THROW);
91        reserved.put(THROWS.image(), THROWS);
92        reserved.put(TRY.image(), TRY);
93
94        // Prime the pump.
95        nextCh();
```

```
96      }
97
98      /**
99       * Scans and returns the next token from input.
100      *
101      * @return the next scanned token.
102      */
103     public TokenInfo getNextToken() {
104
105         StringBuffer buffer;
106         boolean moreWhiteSpace = true;
107         while (moreWhiteSpace) {
108             while (isWhitespace(ch)) {
109                 nextCh();
110             }
111             if (ch == '/') {
112                 nextCh();
113                 if (ch == '/') {
114                     // CharReader maps all new lines to '\n'.
115                     while (ch != '\n' && ch != EOFCH) {
116                         nextCh();
117                         int x = 3;
118                     }
119                 }
120                 // Division assignment
121                 else if (ch == '=') {
122                     nextCh();
123                     return new TokenInfo(DIV_ASSIGN, line);
124                 }
125                 // Multiline comments
126                 else if (ch == '*') {
127                     boolean end = true;
128                     while (end) {
129                         nextCh();
130                         if (ch == '*') {
131                             nextCh();
132                             if (ch == '/') {
133                                 nextCh();
134                                 end = false;
135                             }
136                         }
137                     }
138                 }
139                 else {
140                     // Division
141                     return new TokenInfo(DIV, line);
142                 }
143             } else {
144                 moreWhiteSpace = false;
```

```
145             }
146         }
147         line = input.line();
148         switch (ch) {
149             case ':':
150                 nextCh();
151                 return new TokenInfo(COLON, line);
152             case '?':
153                 nextCh();
154                 return new TokenInfo(QUESTION, line);
155             case ',':
156                 nextCh();
157                 return new TokenInfo(COMMA, line);
158             case '.':
159                 buffer = new StringBuffer();
160                 buffer.append(ch);
161                 nextCh();
162                 // Check if double
163                 while (isDigit(ch) || ch == 'd' || ch == 'D' || ch == 'e' || ch == 'E' || ch == '-' || ch == '+') {
164                     if (ch == 'D' || ch == 'd') {
165                         if (buffer.length() >= 2) {
166                             if (buffer.indexOf("d") == -1 && buffer.indexOf("D") == -1) {
167                                 buffer.append(ch);
168                             }
169                             nextCh();
170                             break;
171                         }
172                         break;
173                     }
174                     else if (ch == 'E' || ch == 'e' && buffer.length() >= 2) {
175                         if (buffer.indexOf("e") == -1 && buffer.indexOf("E") == -1) {
176                             buffer.append(ch);
177                         }
178                         nextCh();
179                     }
180                     else if (ch == '+' || ch == '-' && buffer.length() >= 2 && (buffer.indexOf("e") == buffer.length()
    - 1
181                     || buffer.indexOf("E") == buffer.length() - 1) && (buffer.indexOf("+") == -1 &&
    buffer.indexOf("-") == -1)) {
182                         buffer.append(ch);
183                         nextCh();
184                     }
185                     else {
186                         buffer.append(ch);
187                         nextCh();
188                     }
189                 }
190                 if (buffer.length() > 1) {
191                     return new TokenInfo(DOUBLE_LITERAL, buffer.toString(), line);
```

```
192             }
193             else {
194                 return new TokenInfo(DOT, line);
195             }
196         case '[':
197             nextCh();
198             return new TokenInfo(LBRACK, line);
199         case '{':
200             nextCh();
201             return new TokenInfo(LCURLY, line);
202         case '(':
203             nextCh();
204             return new TokenInfo(LPAREN, line);
205         case ']':
206             nextCh();
207             return new TokenInfo(RBRACK, line);
208         case '}':
209             nextCh();
210             return new TokenInfo(RCURLY, line);
211         case ')':
212             nextCh();
213             return new TokenInfo(RPAREN, line);
214         case ';':
215             nextCh();
216             return new TokenInfo(SEMI, line);
217         case '*':
218             nextCh();
219             if (ch == '=') {
220                 nextCh();
221                 return new TokenInfo(STAR_ASSIGN, line);
222             }
223             else {
224                 return new TokenInfo(STAR, line);
225             }
226         case '%':
227             nextCh();
228             if (ch == '=') {
229                 nextCh();
230                 return new TokenInfo(REM_ASSIGN, line);
231             }
232             return new TokenInfo(REM, line);
233         case '+':
234             nextCh();
235             if (ch == '=') {
236                 nextCh();
237                 return new TokenInfo(PLUS_ASSIGN, line);
238             } else if (ch == '+') {
239                 nextCh();
240                 return new TokenInfo(INC, line);
```

```
241             } else {
242                 return new TokenInfo(PLUS, line);
243             }
244         case '-':
245             nextCh();
246             if (ch == '=') {
247                 nextCh();
248                 return new TokenInfo(MINUS_ASSIGN, line);
249             }
250             if (ch == '-') {
251                 nextCh();
252                 return new TokenInfo(DEC, line);
253             } else {
254                 return new TokenInfo(MINUS, line);
255             }
256         case '=':
257             nextCh();
258             if (ch == '=') {
259                 nextCh();
260                 return new TokenInfo(EQUAL, line);
261             }
262             else if (ch == '+') {
263                 nextCh();
264                 return new TokenInfo(PLUS, line);
265             }
266             else {
267                 return new TokenInfo(ASSIGN, line);
268             }
269         case '~':
270             nextCh();
271             return new TokenInfo(NOT, line);
272         case '>':
273             nextCh();
274             if (ch == '>') {
275                 nextCh();
276                 if (ch == '>') {
277                     nextCh();
278                     if (ch == '=') {
279                         nextCh();
280                         return new TokenInfo(LRSHIFT_ASSIGN, line);
281                     }
282                     else {
283                         return new TokenInfo(LRSHIFT, line);
284                     }
285                 }
286                 else if (ch == '=') {
287                     nextCh();
288                     return new TokenInfo(ARIGHTSHIFT_ASSIGN, line);
289                 }
```

```
290                 else {
291                     return new TokenInfo(ARSHIFT, line);
292                 }
293             }
294             else if (ch == '=') {
295                 nextCh();
296                 return new TokenInfo(GE, line);
297             }
298             else {
299                 return new TokenInfo(GT, line);
300             }
301         case '<':
302             nextCh();
303             if (ch == '=') {
304                 nextCh();
305                 return new TokenInfo(LE, line);
306             }
307             else if (ch == '<') {
308                 nextCh();
309                 if (ch == '=') {
310                     nextCh();
311                     return new TokenInfo(ALSHIFT_ASSIGN, line);
312                 }
313                 else {
314                     return new TokenInfo(ALSHIFT, line);
315                 }
316             }
317             else {
318                 return new TokenInfo(LT, line);
319             }
320         case '!':
321             nextCh();
322             if (ch == '=') {
323                 nextCh();
324                 return new TokenInfo(NOT_EQUAL, line);
325             }
326             else {
327                 nextCh();
328                 return new TokenInfo(LNOT, line);
329             }
330         case '&':
331             nextCh();
332             if (ch == '&') {
333                 nextCh();
334                 return new TokenInfo(LAND, line);
335             }
336             else if (ch == '=') {
337                 nextCh();
338                 return new TokenInfo(AND_ASSIGN, line);
```

```
339             }
340             else {
341                 return new TokenInfo(AND, line);
342             }
343         case '^':
344             nextCh();
345             if (ch == '=') {
346                 nextCh();
347                 return new TokenInfo(XOR_ASSIGN, line);
348             }
349             else {
350                 return new TokenInfo(XOR, line);
351             }
352         case '|':
353             nextCh();
354             if (ch == '=') {
355                 nextCh();
356                 return new TokenInfo(OR_ASSIGN, line);
357             }
358             else if (ch == '|') {
359                 nextCh();
360                 return new TokenInfo(LOR, line);
361             }
362             else {
363                 return new TokenInfo(OR, line);
364             }
365         case '\'':
366             buffer = new StringBuffer();
367             buffer.append('\'');
368             nextCh();
369             if (ch == '\\') {
370                 nextCh();
371                 buffer.append(escape());
372             } else {
373                 buffer.append(ch);
374                 nextCh();
375             }
376             if (ch == '\'') {
377                 buffer.append('\'');
378                 nextCh();
379                 return new TokenInfo(CHAR_LITERAL, buffer.toString(), line);
380             } else {
381                 // Expected a ' ; report error and try to recover.
382                 reportScannerError(ch + " found by scanner where closing ' was expected");
383                 while (ch != '\'' && ch != ';' && ch != '\n') {
384                     nextCh();
385                 }
386                 return new TokenInfo(CHAR_LITERAL, buffer.toString(), line);
387             }
```

```
388             case '"':
389                 buffer = new StringBuffer();
390                 buffer.append("\"");
391                 nextCh();
392                 while (ch != '"' && ch != '\n' && ch != EOFCH) {
393                     if (ch == '\\') {
394                         nextCh();
395                         buffer.append(escape());
396                     } else {
397                         buffer.append(ch);
398                         nextCh();
399                     }
400                 }
401                 if (ch == '\n') {
402                     reportScannerError("Unexpected end of line found in string");
403                 } else if (ch == EOFCH) {
404                     reportScannerError("Unexpected end of file found in string");
405                 } else {
406                     // Scan the closing "
407                     nextCh();
408                     buffer.append("\"");
409                 }
410                 return new TokenInfo(STRING_LITERAL, buffer.toString(), line);
411             case EOFCH:
412                 return new TokenInfo(EOF, line);
413             case '0':
414             case '1':
415             case '2':
416             case '3':
417             case '4':
418             case '5':
419             case '6':
420             case '7':
421             case '8':
422             case '9':
423                 buffer = new StringBuffer();
424                 // Accept integer, double and long
425                 while (isDigit(ch) || ch == '.' || ch == 'e' || ch == 'E' || ch == 'd' || ch == 'D' ||
426                 ch == '-' || ch == '+' | ch == 'l' || ch == 'L') {
427                     buffer.append(ch);
428                     nextCh();
429                 }
430                 // Check if double
431                 if (buffer.indexOf(".") != -1 || buffer.indexOf("e") != -1 || buffer.indexOf("E") != -1 ||
    buffer.indexOf("d") != -1 ||
432                         buffer.indexOf("D") != -1 || buffer.indexOf("+") != -1 || buffer.indexOf("-") != -1) {
433                     return new TokenInfo(DOUBLE_LITERAL, buffer.toString(), line);
434                 }
435                 // Check if long
```

```
436             else if (buffer.indexOf("l") != -1 || buffer.indexOf("L") != -1) {
437                 return new TokenInfo(LONG_LITERAL, buffer.toString(), line);
438             }
439             // Buffer is int
440             else {
441                 return new TokenInfo(INT_LITERAL, buffer.toString(), line);
442             }
443         default:
444             if (isIdentifierStart(ch)) {
445                 buffer = new StringBuffer();
446                 while (isIdentifierPart(ch)) {
447                     buffer.append(ch);
448                     nextCh();
449                 }
450                 String identifier = buffer.toString();
451                 if (reserved.containsKey(identifier)) {
452                     return new TokenInfo(reserved.get(identifier), line);
453                 } else {
454                     return new TokenInfo(IDENTIFIER, identifier, line);
455                 }
456             } else {
457                 reportScannerError("Unidentified input token: '%c'", ch);
458                 nextCh();
459                 return getNextToken();
460             }
461     }
462 }
463 
464 /**
465  * Returns true if an error has occurred, and false otherwise.
466  *
467  * @return true if an error has occurred, and false otherwise.
468  */
469 public boolean errorHasOccurred() {
470     return isInError;
471 }
472 
473 /**
474  * Returns the name of the source file.
475  *
476  * @return the name of the source file.
477  */
478 public String fileName() {
479     return fileName;
480 }
481 
482 // Scans and returns an escaped character.
483 private String escape() {
484     switch (ch) {
```

```
485             case 'b':
486                 nextCh();
487                 return "\\b";
488             case 't':
489                 nextCh();
490                 return "\\t";
491             case 'n':
492                 nextCh();
493                 return "\\n";
494             case 'f':
495                 nextCh();
496                 return "\\f";
497             case 'r':
498                 nextCh();
499                 return "\\r";
500             case '"':
501                 nextCh();
502                 return "\\\"";
503             case '\'':
504                 nextCh();
505                 return "\\'";
506             case '\\':
507                 nextCh();
508                 return "\\\\";
509             default:
510                 reportScannerError("Badly formed escape: \\%c", ch);
511                 nextCh();
512                 return "";
513         }
514     }
515 
516     // Advances ch to the next character from input, and updates the line number.
517     private void nextCh() {
518         line = input.line();
519         try {
520             ch = input.nextChar();
521         } catch (Exception e) {
522             reportScannerError("Unable to read characters from input");
523         }
524     }
525 
526     // Reports a lexical error and records the fact that an error has occurred. This fact can be
527     // ascertained from the Scanner by sending it an errorHasOccurred message.
528     private void reportScannerError(String message, Object... args) {
529         isInError = true;
530         System.err.printf("%s:%d: error: ", fileName, line);
531         System.err.printf(message, args);
532         System.err.println();
533     }
```

```
534
535     // Returns true if the specified character is a digit (0-9), and false otherwise.
536     private boolean isDigit(char c) {
537         return (c >= '0' && c <= '9');
538     }
539
540     // Returns true if the specified character is a whitespace, and false otherwise.
541     private boolean isWhitespace(char c) {
542         return (c == ' ' || c == '\t' || c == '\n' || c == '\f');
543     }
544
545     // Returns true if the specified character can start an identifier name, and false otherwise.
546     private boolean isIdentifierStart(char c) {
547         return (c >= 'a' && c <= 'z' || c >= 'A' && c <= 'Z' || c == '_' || c == '$');
548     }
549
550     // Returns true if the specified character can be part of an identifier name, and false
551     // otherwise.
552     private boolean isIdentifierPart(char c) {
553         return (isIdentifierStart(c) || isDigit(c));
554     }
555 }
556
557 /**
558  * A buffered character reader, which abstracts out differences between platforms, mapping all new
559  * lines to '\n', and also keeps track of line numbers.
560  */
561 class CharReader {
562     // Representation of the end of file as a character.
563     public final static char EOFCH = (char) -1;
564
565     // The underlying reader records line numbers.
566     private LineNumberReader lineNumberReader;
567
568     // Name of the file that is being read.
569     private String fileName;
570
571     /**
572      * Constructs a CharReader from a file name.
573      *
574      * @param fileName the name of the input file.
575      * @throws FileNotFoundException if the file is not found.
576      */
577     public CharReader(String fileName) throws FileNotFoundException {
578         lineNumberReader = new LineNumberReader(new FileReader(fileName));
579         this.fileName = fileName;
580     }
581
582     /**
```

```
583      * Scans and returns the next character.
584      *
585      * @return the character scanned.
586      * @throws IOException if an I/O error occurs.
587      */
588     public char nextChar() throws IOException {
589         return (char) lineNumberReader.read();
590     }
591 
592     /**
593      * Returns the current line number in the source file.
594      *
595      * @return the current line number in the source file.
596      */
597     public int line() {
598         return lineNumberReader.getLineNumber() + 1; // LineNumberReader counts lines from 0
599     }
600 
601     /**
602      * Returns the file name.
603      *
604      * @return the file name.
605      */
606     public String fileName() {
607         return fileName;
608     }
609 
610     /**
611      * Closes the file.
612      *
613      * @throws IOException if an I/O error occurs.
614      */
615     public void close() throws IOException {
616         lineNumberReader.close();
617     }
618 }
619 
```

Parser.java

Download

```java
// Copyright 2012- Bill Campbell, Swami Iyer and Bahar Akbal-Delibas

package jminusminus;

import java.util.ArrayList;

import static jminusminus.TokenKind.*;

/**
 * A recursive descent parser that, given a lexical analyzer (a LookaheadScanner), parses a j--
 * compilation unit (program file), taking tokens from the LookaheadScanner, and produces an
 * abstract syntax tree (AST) for it.
 */
public class Parser {
    // The lexical analyzer with which tokens are scanned.
    private LookaheadScanner scanner;

    // Whether a parser error has been found.
    private boolean isInError;

    // Whether we have recovered from a parser error.
    private boolean isRecovered;

    /**
     * Constructs a parser from the given lexical analyzer.
     *
     * @param scanner the lexical analyzer with which tokens are scanned.
     */
    public Parser(LookaheadScanner scanner) {
        this.scanner = scanner;
        isInError = false;
        isRecovered = true;

        // Prime the pump.
        scanner.next();
    }

    /**
     * Returns true if a parser error has occurred up to now, and false otherwise.
     *
     * @return true if a parser error has occurred up to now, and false otherwise.
     */
    public boolean errorHasOccurred() {
        return isInError;
    }

```

```
47      /**
48       * Parses a compilation unit (a program file) and returns an AST for it.
49       *
50       * <pre>
51       *    compilationUnit ::= [ PACKAGE qualifiedIdentifier SEMI ]
52       *                        { IMPORT  qualifiedIdentifier SEMI }
53       *                        { typeDeclaration }
54       *                        EOF
55       * </pre>
56       *
57       * @return an AST for a compilation unit.
58       */
59      public JCompilationUnit compilationUnit() {
60          int line = scanner.token().line();
61          String fileName = scanner.fileName();
62          TypeName packageName = null;
63          if (have(PACKAGE)) {
64              packageName = qualifiedIdentifier();
65              mustBe(SEMI);
66          }
67          ArrayList<TypeName> imports = new ArrayList<TypeName>();
68          while (have(IMPORT)) {
69              imports.add(qualifiedIdentifier());
70              mustBe(SEMI);
71          }
72          ArrayList<JAST> typeDeclarations = new ArrayList<JAST>();
73          while (!see(EOF)) {
74              JAST typeDeclaration = typeDeclaration();
75              if (typeDeclaration != null) {
76                  typeDeclarations.add(typeDeclaration);
77              }
78          }
79          mustBe(EOF);
80          return new JCompilationUnit(fileName, line, packageName, imports, typeDeclarations);
81      }
82
83      /**
84       * Parses and returns a qualified identifier.
85       *
86       * <pre>
87       *   qualifiedIdentifier ::= IDENTIFIER { DOT IDENTIFIER }
88       * </pre>
89       *
90       * @return a qualified identifier.
91       */
92      private TypeName qualifiedIdentifier() {
93          int line = scanner.token().line();
94          mustBe(IDENTIFIER);
95          String qualifiedIdentifier = scanner.previousToken().image();
```

```
96          while (have(DOT)) {
97              mustBe(IDENTIFIER);
98              qualifiedIdentifier += "." + scanner.previousToken().image();
99          }
100         return new TypeName(line, qualifiedIdentifier);
101     }
102
103     /**
104      * Parses a type declaration and returns an AST for it.
105      *
106      * <pre>
107      *   typeDeclaration ::= modifiers classDeclaration
108      * </pre>
109      *
110      * @return an AST for a type declaration.
111      */
112     private JAST typeDeclaration() {
113         ArrayList<String> mods = modifiers();
114         return classDeclaration(mods);
115     }
116
117     /**
118      * Parses and returns a list of modifiers.
119      *
120      * <pre>
121      *   modifiers ::= { ABSTRACT | PRIVATE | PROTECTED | PUBLIC | STATIC }
122      * </pre>
123      *
124      * @return a list of modifiers.
125      */
126     private ArrayList<String> modifiers() {
127         ArrayList<String> mods = new ArrayList<String>();
128         boolean scannedPUBLIC = false;
129         boolean scannedPROTECTED = false;
130         boolean scannedPRIVATE = false;
131         boolean scannedSTATIC = false;
132         boolean scannedABSTRACT = false;
133         boolean more = true;
134         while (more) {
135             if (have(ABSTRACT)) {
136                 mods.add("abstract");
137                 if (scannedABSTRACT) {
138                     reportParserError("Repeated modifier: abstract");
139                 }
140                 scannedABSTRACT = true;
141             } else if (have(PRIVATE)) {
142                 mods.add("private");
143                 if (scannedPRIVATE) {
144                     reportParserError("Repeated modifier: private");
```

```
145             }
146             if (scannedPUBLIC || scannedPROTECTED) {
147                 reportParserError("Access conflict in modifiers");
148             }
149             scannedPRIVATE = true;
150         } else if (have(PROTECTED)) {
151             mods.add("protected");
152             if (scannedPROTECTED) {
153                 reportParserError("Repeated modifier: protected");
154             }
155             if (scannedPUBLIC || scannedPRIVATE) {
156                 reportParserError("Access conflict in modifiers");
157             }
158             scannedPROTECTED = true;
159         } else if (have(PUBLIC)) {
160             mods.add("public");
161             if (scannedPUBLIC) {
162                 reportParserError("Repeated modifier: public");
163             }
164             if (scannedPROTECTED || scannedPRIVATE) {
165                 reportParserError("Access conflict in modifiers");
166             }
167             scannedPUBLIC = true;
168         } else if (have(STATIC)) {
169             mods.add("static");
170             if (scannedSTATIC) {
171                 reportParserError("Repeated modifier: static");
172             }
173             scannedSTATIC = true;
174         } else if (have(ABSTRACT)) {
175             mods.add("abstract");
176             if (scannedABSTRACT) {
177                 reportParserError("Repeated modifier: abstract");
178             }
179             scannedABSTRACT = true;
180         } else {
181             more = false;
182         }
183     }
184     return mods;
185 }
186 
187 /**
188  * Parses a class declaration and returns an AST for it.
189  *
190  * <pre>
191  *   classDeclaration ::= CLASS IDENTIFIER [ EXTENDS qualifiedIdentifier ] classBody
192  * </pre>
193  *
```

```
194     * @param mods the class modifiers.
195     * @return an AST for a class declaration.
196     */
197    private JClassDeclaration classDeclaration(ArrayList<String> mods) {
198      int line = scanner.token().line();
199      mustBe(CLASS);
200      mustBe(IDENTIFIER);
201      String name = scanner.previousToken().image();
202      Type superClass;
203      if (have(EXTENDS)) {
204        superClass = qualifiedIdentifier();
205      } else {
206        superClass = Type.OBJECT;
207      }
208      return new JClassDeclaration(line, mods, name, superClass, null, classBody());
209    }
210
211    /**
212     * Parses a class body and returns a list of members in the body.
213     *
214     * <pre>
215     *   classBody ::= LCURLY { modifiers memberDecl } RCURLY
216     * </pre>
217     *
218     * @return a list of members in the class body.
219     */
220    private ArrayList<JMember> classBody() {
221      ArrayList<JMember> members = new ArrayList<JMember>();
222      mustBe(LCURLY);
223      while (!see(RCURLY) && !see(EOF)) {
224        ArrayList<String> mods = modifiers();
225        members.add(memberDecl(mods));
226      }
227      mustBe(RCURLY);
228      return members;
229    }
230
231    /**
232     * Parses a member declaration and returns an AST for it.
233     *
234     * <pre>
235     *   memberDecl ::= IDENTIFIER formalParameters block
236     *                | ( VOID | type ) IDENTIFIER formalParameters ( block | SEMI )
237     *                | type variableDeclarators SEMI
238     * </pre>
239     *
240     * @param mods the class member modifiers.
241     * @return an AST for a member declaration.
242     */
```

```
243     private JMember memberDecl(ArrayList<String> mods) {
244         int line = scanner.token().line();
245         JMember memberDecl = null;
246         if (seeIdentLParen()) {
247             // A constructor.
248             mustBe(IDENTIFIER);
249             String name = scanner.previousToken().image();
250             ArrayList<JFormalParameter> params = formalParameters();
251             JBlock body = block();
252             memberDecl = new JConstructorDeclaration(line, mods, name, params, null, body);
253         } else {
254             Type type = null;
255             if (have(VOID)) {
256                 // A void method.
257                 type = Type.VOID;
258                 mustBe(IDENTIFIER);
259                 String name = scanner.previousToken().image();
260                 ArrayList<JFormalParameter> params = formalParameters();
261                 JBlock body = have(SEMI) ? null : block();
262                 memberDecl = new JMethodDeclaration(line, mods, name, type, params, null, body);
263             } else {
264                 type = type();
265                 if (seeIdentLParen()) {
266                     // A non void method.
267                     mustBe(IDENTIFIER);
268                     String name = scanner.previousToken().image();
269                     ArrayList<JFormalParameter> params = formalParameters();
270                     JBlock body = have(SEMI) ? null : block();
271                     memberDecl = new JMethodDeclaration(line, mods, name, type, params, null, body);
272                 } else {
273                     // A field.
274                     memberDecl = new JFieldDeclaration(line, mods, variableDeclarators(type));
275                     mustBe(SEMI);
276                 }
277             }
278         }
279         return memberDecl;
280     }
281 
282     /**
283      * Parses a block and returns an AST for it.
284      *
285      * <pre>
286      *   block ::= LCURLY { blockStatement } RCURLY
287      * </pre>
288      *
289      * @return an AST for a block.
290      */
291     private JBlock block() {
```

```
292         int line = scanner.token().line();
293         ArrayList<JStatement> statements = new ArrayList<JStatement>();
294         mustBe(LCURLY);
295         while (!see(RCURLY) && !see(EOF)) {
296             statements.add(blockStatement());
297         }
298         mustBe(RCURLY);
299         return new JBlock(line, statements);
300     }
301 
302     /**
303      * Parses a block statement and returns an AST for it.
304      *
305      * <pre>
306      *   blockStatement ::= localVariableDeclarationStatement
307      *                    | statement
308      * </pre>
309      *
310      * @return an AST for a block statement.
311      */
312     private JStatement blockStatement() {
313         if (seeLocalVariableDeclaration()) {
314             return localVariableDeclarationStatement();
315         } else {
316             return statement();
317         }
318     }
319 
320     /**
321      * Parses forUpdate and returns a list of JStatements.
322      *
323      * <pre>
324      *   forUpdate ::= statementExpression {COMMA statementExpression}
325      * </pre>
326      *
327      * @return a list of JStatements.
328      */
329     private ArrayList<JStatement> forUpdate() {
330 
331 
332 
333         JStatement statementExpression = statementExpression();
334         ArrayList<JStatement> update = new ArrayList<>();
335         update.add(statementExpression);
336         while(have(COMMA)) {
337             update.add(statementExpression());
338         }
339         return update;
340     }
```

```
341
342     /**
343      * Parses forInit and returns a list of JStatements.
344      *
345      * <pre>
346      *   forInit ::= statementExpression {COMMA statementExpression}
347      *                         | type variableDeclarators
348      * </pre>
349      *
350      * @return a list of JStatements.
351      */
352     private ArrayList<JStatement> forInit() {
353
354
355         int line = scanner.token().line();
356         JStatement statementExpression = statementExpression();
357         ArrayList<JStatement> init = new ArrayList<>();
358         if (!seeLocalVariableDeclaration()) {
359             init.add(statementExpression);
360             while (have(COMMA)) {
361                 init.add(statementExpression());
362             }
363         } else {
364             Type type = type();
365             JVariableDeclaration jVariableDeclaration= new JVariableDeclaration(line,
    variableDeclarators(type));
366             init.add(jVariableDeclaration);
367         }
368         return init;
369     }
370
371     /**
372      * Parses conditionalOrExpression and returns a JExpression.
373      *
374      * <pre>
375      *   conditionalOrExpression ::= conditionalAndExpression {LOR conditionalAndExpression}
376      * </pre>
377      *
378      * @return a JExpression.
379      */
380     private JExpression conditionalOrExpression() {
381         int line = scanner.token().line();
382         boolean more = true;
383         JExpression lhs = conditionalAndExpression();
384         while (more) {
385             if (have(LOR)) {
386                 lhs = new JLogicalOrOp(line, lhs, conditionalAndExpression());
387             } else {
388                 more = false;
```

```
389             }
390         }
391         return lhs;
392     }
393
394     private JExpression conditionalOrExpression2() {
395         int line = scanner.token().line();
396         JExpression condAndExpress = conditionalAndExpression();
397         while(have(LOR)) {
398             condAndExpress = new JLogicalOrOp(line, condAndExpress, conditionalAndExpression());
399         }
400         return condAndExpress;
401     }
402
403     /**
404      * Parses a statement and returns an AST for it.
405      *
406      * <pre>
407      *   statement ::= block
408      *               | IF parExpression statement [ ELSE statement ]
409      *               | RETURN [ expression ] SEMI
410      *               | SEMI
411      *               | WHILE parExpression statement
412      *               | expression [QUESTION expression COLON expression]
413      *               | Do statementExpression WHILE parExpression SEMI
414      *               | FOR LPAREN [forInit] SEMI [expresison] SEMI [forUpdate] RPAREN statement
415      *               | BREAK SEMI
416      *               | CONTINUE SEMI
417      *               | SWITCH parExpression LCURLY {switchBlockStatementGroup} RCURLY
418      *               | TRY block {CATCH LPAREN formalParameter RPAREN block} [FINALLY block]
419      *               | THROW expression SEMI
420      *               | INTERFACE IDENTIFIER
421      *               | statementExpression SEMI
422      * </pre>
423      *
424      * @return an AST for a statement.
425      */
426
427     private JStatement statement() {
428         int line = scanner.token().line();
429         if (see(LCURLY)) {
430             return block();
431         } else if (have(IF)) {
432             JExpression test = parExpression();
433             JStatement consequent = statement();
434             JStatement alternate = have(ELSE) ? statement() : null;
435             return new JIfStatement(line, test, consequent, alternate);
436         } else if (have(RETURN)) {
437             if (have(SEMI)) {
```

```
438                 return new JReturnStatement(line, null);
439             } else {
440                 JExpression expr = expression();
441                 mustBe(SEMI);
442                 return new JReturnStatement(line, expr);
443             }
444         } else if (have(SEMI)) {
445             return new JEmptyStatement(line);
446         } else if (have(WHILE)) {
447             JExpression test = parExpression();
448             JStatement statement = statement();
449             return new JWhileStatement(line, test, statement);
450         }
451 //        // Conditional expression
452 //        else if (see(QUESTION)) {
453 //            JExpression condOrExpr = conditionalOrExpression();
454 //            if (have(QUESTION)) {
455 //                JExpression expression = expression();
456 //                mustBe(COLON);
457 //                JExpression condExpr = expression();
458 //                return new JConditionalExpression(line, condOrExpr, expression, condExpr);
459 //            }
460 //            return new JConditionalExpression(line, condOrExpr, null, null);
461 //        }
462         // Do statement
463         else if (have(DO)) {
464             JStatement statement = statement();
465             mustBe(WHILE);
466             JExpression pe = parExpression();
467             mustBe(SEMI);
468             return new JDoStatement(line, statement, pe);
469         }
470         // For statement
471         else if (have(FOR)) {
472             mustBe(LPAREN);
473             if (see(SEMI)) {
474                 if (see(SEMI)) {
475                     if (see(RPAREN)) {
476                         JStatement body = statement();
477                         return new JForStatement(line, null, null, null, body);
478                     } else {
479                         mustBe(RPAREN);
480                         JStatement body = statement();
481                         return new JForStatement(line, null, null, null, body);
482                     }
483                 } else {
484                     JExpression condition = expression();
485                     mustBe(SEMI);
486                     ArrayList<JStatement> update = forUpdate();
```

```
487                 mustBe(RPAREN);
488                 JStatement body = statement();
489                 return new JForStatement(line, null, condition, update, body);
490             }
491         } else {
492             ArrayList<JStatement> init = forInit();
493             mustBe(SEMI);
494             JExpression condition = expression();
495             mustBe(SEMI);
496             ArrayList<JStatement> update = forUpdate();
497             mustBe(RPAREN);
498             JStatement body = statement();
499             return new JForStatement(line, init, condition, update, body);
500         }
501     }
502     // Break
503     else if (have(BREAK)) {
504         mustBe(SEMI);
505         return new JBreakStatement(line);
506     }
507     // Continue
508     else if (have(CONTINUE)) {
509         mustBe(SEMI);
510         return new JContinueStatement(line);
511     }
512     // Exception Handlers
513     else if (have(THROW)) {
514         JExpression jExpression = expression();
515         mustBe(SEMI);
516         return new JThrowStatement(line, jExpression);
517     } else if (have(TRY)) {
518         JBlock tryBlock = block();
519         JBlock finallyBlock = null;
520         ArrayList<JFormalParameter> parameters = new ArrayList<>();
521         ArrayList<JBlock> catchBlocks = new ArrayList<>();
522         while(have(CATCH)) {
523             mustBe(LPAREN);
524             parameters.add(formalParameter());
525             mustBe(RPAREN);
526             catchBlocks.add(block());
527         }
528         if (have(FINALLY)) {
529             finallyBlock = block();
530         }
531         return new JTryStatement(line, tryBlock, parameters, catchBlocks, finallyBlock);
532     }
533     else {
534         // Must be a statementExpression.
535         JStatement statement = statementExpression();
```

```
536             mustBe(SEMI);
537             return statement;
538         }
539     }
540 
541     /**
542      * Parses and returns a list of formal parameters.
543      *
544      * <pre>
545      *   formalParameters ::= LPAREN [ formalParameter { COMMA formalParameter } ] RPAREN
546      * </pre>
547      *
548      * @return a list of formal parameters.
549      */
550     private ArrayList<JFormalParameter> formalParameters() {
551         ArrayList<JFormalParameter> parameters = new ArrayList<JFormalParameter>();
552         mustBe(LPAREN);
553         if (have(RPAREN)) {
554             return parameters;
555         }
556         do {
557             parameters.add(formalParameter());
558         } while (have(COMMA));
559         mustBe(RPAREN);
560         return parameters;
561     }
562 
563     /**
564      * Parses a formal parameter and returns an AST for it.
565      *
566      * <pre>
567      *   formalParameter ::= type IDENTIFIER
568      * </pre>
569      *
570      * @return an AST for a formal parameter.
571      */
572     private JFormalParameter formalParameter() {
573         int line = scanner.token().line();
574         Type type = type();
575         mustBe(IDENTIFIER);
576         String name = scanner.previousToken().image();
577         return new JFormalParameter(line, name, type);
578     }
579 
580     /**
581      * Parses a parenthesized expression and returns an AST for it.
582      *
583      * <pre>
584      *   parExpression ::= LPAREN expression RPAREN
```

```
585     * </pre>
586     *
587     * @return an AST for a parenthesized expression.
588     */
589    private JExpression parExpression() {
590        mustBe(LPAREN);
591        JExpression expr = expression();
592        mustBe(RPAREN);
593        return expr;
594    }
595
596    /**
597     * Parses a local variable declaration statement and returns an AST for it.
598     *
599     * <pre>
600     *   localVariableDeclarationStatement ::= type variableDeclarators SEMI
601     * </pre>
602     *
603     * @return an AST for a local variable declaration statement.
604     */
605    private JVariableDeclaration localVariableDeclarationStatement() {
606        int line = scanner.token().line();
607        Type type = type();
608        ArrayList<JVariableDeclarator> vdecls = variableDeclarators(type);
609        mustBe(SEMI);
610        return new JVariableDeclaration(line, vdecls);
611    }
612
613    /**
614     * Parses and returns a list of variable declarators.
615     *
616     * <pre>
617     *   variableDeclarators ::= variableDeclarator { COMMA variableDeclarator }
618     * </pre>
619     *
620     * @param type type of the variables.
621     * @return a list of variable declarators.
622     */
623    private ArrayList<JVariableDeclarator> variableDeclarators(Type type) {
624        ArrayList<JVariableDeclarator> variableDeclarators = new ArrayList<JVariableDeclarator>();
625        do {
626            variableDeclarators.add(variableDeclarator(type));
627        } while (have(COMMA));
628        return variableDeclarators;
629    }
630
631    /**
632     * Parses a variable declarator and returns an AST for it.
633     *
```

```
634     * <pre>
635     *   variableDeclarator ::= IDENTIFIER [ ASSIGN variableInitializer ]
636     * </pre>
637     *
638     * @param type type of the variable.
639     * @return an AST for a variable declarator.
640     */
641    private JVariableDeclarator variableDeclarator(Type type) {
642        int line = scanner.token().line();
643        mustBe(IDENTIFIER);
644        String name = scanner.previousToken().image();
645        JExpression initial = have(ASSIGN) ? variableInitializer(type) : null;
646        return new JVariableDeclarator(line, name, type, initial);
647    }
648
649    /**
650     * Parses a variable initializer and returns an AST for it.
651     *
652     * <pre>
653     *   variableInitializer ::= arrayInitializer | expression
654     * </pre>
655     *
656     * @param type type of the variable.
657     * @return an AST for a variable initializer.
658     */
659    private JExpression variableInitializer(Type type) {
660        if (see(LCURLY)) {
661            return arrayInitializer(type);
662        }
663        return expression();
664    }
665
666    /**
667     * Parses an array initializer and returns an AST for it.
668     *
669     * <pre>
670     *   arrayInitializer ::= LCURLY [ variableInitializer { COMMA variableInitializer }
671     *                                 [ COMMA ] ] RCURLY
672     * </pre>
673     *
674     * @param type type of the array.
675     * @return an AST for an array initializer.
676     */
677    private JArrayInitializer arrayInitializer(Type type) {
678        int line = scanner.token().line();
679        ArrayList<JExpression> initials = new ArrayList<JExpression>();
680        mustBe(LCURLY);
681        if (have(RCURLY)) {
682            return new JArrayInitializer(line, type, initials);
```

```
683         }
684         initials.add(variableInitializer(type.componentType()));
685         while (have(COMMA)) {
686             initials.add(see(RCURLY) ? null : variableInitializer(type.componentType()));
687         }
688         mustBe(RCURLY);
689         return new JArrayInitializer(line, type, initials);
690     }
691 
692     /**
693      * Parses and returns a list of arguments.
694      *
695      * <pre>
696      *   arguments ::= LPAREN [ expression { COMMA expression } ] RPAREN
697      * </pre>
698      *
699      * @return a list of arguments.
700      */
701     private ArrayList<JExpression> arguments() {
702         ArrayList<JExpression> args = new ArrayList<JExpression>();
703         mustBe(LPAREN);
704         if (have(RPAREN)) {
705             return args;
706         }
707         do {
708             args.add(expression());
709         } while (have(COMMA));
710         mustBe(RPAREN);
711         return args;
712     }
713 
714     /**
715      * Parses and returns a type.
716      *
717      * <pre>
718      *   type ::= referenceType | basicType
719      * </pre>
720      *
721      * @return a type.
722      */
723     private Type type() {
724         if (seeReferenceType()) {
725             return referenceType();
726         }
727         return basicType();
728     }
729 
730     /**
731      * Parses and returns a basic type.
```

```
732     *
733     * <pre>
734     *   basicType ::= BOOLEAN | CHAR | INT | DOUBLE | LONG
735     * </pre>
736     *
737     * @return a basic type.
738     */
739    private Type basicType() {
740        if (have(BOOLEAN)) {
741            return Type.BOOLEAN;
742        } else if (have(CHAR)) {
743            return Type.CHAR;
744        } else if (have(INT)) {
745            return Type.INT;
746        } else if (have(DOUBLE)) {
747            return Type.DOUBLE;
748        } else if (have(LONG)) {
749            return Type.LONG;
750        } else {
751            reportParserError("Type sought where %s found", scanner.token().image());
752            return Type.ANY;
753        }
754    }
755
756    /**
757     * Parses and returns a reference type.
758     *
759     * <pre>
760     *   referenceType ::= basicType LBRACK RBRACK { LBRACK RBRACK }
761     *                   | qualifiedIdentifier { LBRACK RBRACK }
762     * </pre>
763     *
764     * @return a reference type.
765     */
766    private Type referenceType() {
767        Type type = null;
768        if (!see(IDENTIFIER)) {
769            type = basicType();
770            mustBe(LBRACK);
771            mustBe(RBRACK);
772            type = new ArrayTypeName(type);
773        } else {
774            type = qualifiedIdentifier();
775        }
776        while (seeDims()) {
777            mustBe(LBRACK);
778            mustBe(RBRACK);
779            type = new ArrayTypeName(type);
780        }
```

```
781         return type;
782     }
783 
784     /**
785      * Parses a statement expression and returns an AST for it.
786      *
787      * <pre>
788      *   statementExpression ::= expression
789      * </pre>
790      *
791      * @return an AST for a statement expression.
792      */
793     private JStatement statementExpression() {
794         int line = scanner.token().line();
795         JExpression expr = expression();
796         if (expr instanceof JAssignment
797                 || expr instanceof JPreIncrementOp
798                 || expr instanceof JPreDecrementOp
799                 || expr instanceof JPostIncrementOp
800                 || expr instanceof JPostDecrementOp
801                 || expr instanceof JMessageExpression
802                 || expr instanceof JSuperConstruction
803                 || expr instanceof JThisConstruction
804                 || expr instanceof JNewOp
805                 || expr instanceof JNewArrayOp) {
806             // So as not to save on stack.
807             expr.isStatementExpression = true;
808         }
809         else {
810             reportParserError("Invalid statement expression; it does not have a side-effect");
811         }
812         return new JStatementExpression(line, expr);
813     }
814 
815     /**
816      * Parses an expression and returns an AST for it.
817      *
818      * <pre>
819      *   expression ::= assignmentExpression
820      * </pre>
821      *
822      * @return an AST for an expression.
823      */
824     private JExpression expression() {
825         return assignmentExpression();
826     }
827 
828     /**
829      * Parses an assignment expression and returns an AST for it.
```

```
830     *
831     * <pre>
832     *   assignmentExpression ::= conditionalAndExpression
833     *                            [ ( ASSIGN | PLUS_ASSIGN | MINUS_ASSIGN | STAR_ASSIGN | DIV_ASSIGN |
    REM_ASSIGN
834     *                            | LRSHIFT_ASSIGN | ALSHIFT_ASSIGN | ARSHIFT_ASSIGN | AND_ASSIGN |
    XOR_ASSIGN
835     *                            | OR_ASSIGN) assignmentExpression ]
836     * </pre>
837     *
838     * @return an AST for an assignment expression.
839     */
840    private JExpression conditionalExpression() {
841        int line = scanner.token().line();
842        JExpression jorexp = conditionalOrExpression();
843        if (have(QUESTION)) {
844            JExpression expression = expression();
845            mustBe(COLON);
846            JExpression condExp = conditionalOrExpression();
847            return new JConditionalExpression(line, jorexp, expression, condExp);
848        }
849        return new JConditionalExpression(line, jorexp, null, null);
850    }
851
852    private JExpression assignmentExpression() {
853        int line = scanner.token().line();
854        JExpression lhs = conditionalExpression();
855        if (have(ASSIGN)) {
856            return new JAssignOp(line, lhs, assignmentExpression());
857        } else if (have(PLUS_ASSIGN)) {
858            return new JPlusAssignOp(line, lhs, assignmentExpression());
859        } else if (have(MINUS_ASSIGN)) {
860            return new JMinusAssignOp(line, lhs, assignmentExpression());
861        } else if (have(STAR_ASSIGN)) {
862            return new JStarAssignOp(line, lhs, assignmentExpression());
863        } else if (have(DIV_ASSIGN)) {
864            return new JDivAssignOp(line, lhs, assignmentExpression());
865        } else if (have(REM_ASSIGN)) {
866            return new JRemAssignOp(line, lhs, assignmentExpression());
867        } else if (have(LRSHIFT_ASSIGN)) {
868            return new JLRightShiftAssignOp(line, lhs, assignmentExpression());
869        } else if (have(ALSHIFT_ASSIGN)) {
870            return new JALeftShiftAssignOp(line, lhs, assignmentExpression());
871        } else if (have(ARIGHTSHIFT_ASSIGN)) {
872            return new JARightShiftAssignOp(line, lhs, assignmentExpression());
873        } else if (have (AND_ASSIGN)) {
874            return new JAndAssignOp(line, lhs, assignmentExpression());
875        } else if (have(XOR_ASSIGN)) {
876            return new JXorAssignOp(line, lhs, assignmentExpression());
```

```
877         } else if (have(OR_ASSIGN)) {
878             return new JOrAssignOp(line, lhs, assignmentExpression());
879         }
880         else {
881             return lhs;
882         }
883     }
884 
885     /**
886      * Parses a conditional-and expression and returns an AST for it.
887      *
888      * <pre>
889      *   conditionalAndExpression ::= equalityExpression { LAND equalityExpression }
890      * </pre>
891      *
892      * @return an AST for a conditional-and expression.
893      */
894     private JExpression conditionalAndExpression() {
895         int line = scanner.token().line();
896         boolean more = true;
897         JExpression lhs = equalityExpression();
898         while (more) {
899             if (have(LAND)) {
900                 lhs = new JLogicalAndOp(line, lhs, equalityExpression());
901             } else if (have(LOR)) {
902                 lhs = new JLogicalOrOp(line, lhs, equalityExpression());
903             }
904             else {
905                 more = false;
906             }
907         }
908         return lhs;
909     }
910 
911     /**
912      * Parses an equality expression and returns an AST for it.
913      *
914      * <pre>
915      *   equalityExpression ::= relationalExpression { (EQUAL | NOT_EQUAL) relationalExpression }
916      * </pre>
917      *
918      * @return an AST for an equality expression.
919      */
920     private JExpression equalityExpression() {
921         int line = scanner.token().line();
922         boolean more = true;
923         JExpression lhs = relationalExpression();
924         while (more) {
925             if (have(EQUAL)) {
```

```
926             lhs = new JEqualOp(line, lhs, relationalExpression());
927         } else if (have(NOT_EQUAL)) {
928             lhs = new JNotEqualOp(line, lhs, relationalExpression());
929         }
930         else {
931             more = false;
932         }
933       }
934       return lhs;
935     }
936 
937     /**
938      * Parses a relational expression and returns an AST for it.
939      *
940      * <pre>
941      *   relationalExpression ::= additiveExpression [ ( GT | LE | LT | GE ) additiveExpression
942      *                                       | INSTANCEOF referenceType ]
943      * </pre>
944      *
945      * @return an AST for a relational expression.
946      */
947     private JExpression relationalExpression() {
948       int line = scanner.token().line();
949       JExpression lhs = additiveExpression();
950       if (have(GT)) {
951         return new JGreaterThanOp(line, lhs, additiveExpression());
952       } else if (have(LT)) {
953         return new JLessThanOp(line, lhs, additiveExpression());
954       } else if (have(GE)) {
955         return new JGreaterEqualOp(line, lhs, additiveExpression());
956       } else if (have(LE)) {
957         return new JLessEqualOp(line, lhs, additiveExpression());
958       } else if (have(INSTANCEOF)) {
959         return new JInstanceOfOp(line, lhs, referenceType());
960       } else {
961         return lhs;
962       }
963     }
964 
965     /**
966      * Parses an additive expression and returns an AST for it.
967      *
968      * <pre>
969      *   additiveExpression ::= multiplicativeExpression { (MINUS | PLUS) multiplicativeExpression }
970      * </pre>
971      *
972      * @return an AST for an additive expression.
973      */
974     private JExpression additiveExpression() {
```

```
975         int line = scanner.token().line();
976         boolean more = true;
977         JExpression lhs = multiplicativeExpression();
978         while (more) {
979           if (have(MINUS)) {
980             lhs = new JSubtractOp(line, lhs, multiplicativeExpression());
981           } else if (have(PLUS)) {
982             lhs = new JPlusOp(line, lhs, multiplicativeExpression());
983           } else {
984             more = false;
985           }
986         }
987         return lhs;
988       }
989
990       /**
991        * Parses a multiplicative expression and returns an AST for it.
992        *
993        * <pre>
994        *   multiplicativeExpression ::= unaryExpression { (STAR | DIV | REM
995        *   | ALSHIFT | ARSHIFT | LRSHIFT | AND | XOR | OR) unaryExpression }
996        * </pre>
997        *
998        * @return an AST for a multiplicative expression.
999        */
1000      private JExpression multiplicativeExpression() {
1001        int line = scanner.token().line();
1002        boolean more = true;
1003        JExpression lhs = unaryExpression();
1004
1005
1006
1007        while (more) {
1008          if (have(STAR)) {
1009            lhs = new JMultiplyOp(line, lhs, unaryExpression());
1010          }
1011          else if (have(DIV)) {
1012            lhs = new JDivideOp(line, lhs, unaryExpression());
1013          }
1014          else if (have(REM)) {
1015            lhs = new JRemainderOp(line, lhs, unaryExpression());
1016          }
1017          else if (have(ALSHIFT)) {
1018            lhs = new JALeftShiftOp(line, lhs, unaryExpression());
1019          }
1020          else if (have(ARSHIFT)) {
1021            lhs = new JARightShiftOp(line, lhs, unaryExpression());
1022          }
1023          else if (have(LRSHIFT)) {
```

```
1024                lhs = new JLRightShiftOp(line, lhs, unaryExpression());
1025            }
1026            else if (have(AND)) {
1027                lhs = new JAndOp(line, lhs, unaryExpression());
1028            }
1029            else if (have(XOR)) {
1030                lhs = new JXorOp(line, lhs, unaryExpression());
1031            }
1032            else if (have(OR)) {
1033                lhs = new JOrOp(line, lhs, unaryExpression());
1034            }
1035            else {
1036                more = false;
1037            }
1038        }
1039        return lhs;
1040    }
1041
1042    /**
1043     * Parses an unary expression and returns an AST for it.
1044     *
1045     * <pre>
1046     *   unaryExpression ::= INC unaryExpression
1047     *                     | MINUS unaryExpression
1048     *                     | simpleUnaryExpression
1049     *                     | COMP unaryExpression
1050     * </pre>
1051     *
1052     * @return an AST for an unary expression.
1053     */
1054    private JExpression unaryExpression() {
1055        int line = scanner.token().line();
1056        if (have(INC)) {
1057            return new JPreIncrementOp(line, unaryExpression());
1058        } else if (have (DEC)) {
1059            return new JPreDecrementOp(line, unaryExpression());
1060        }
1061        else if (have(MINUS)) {
1062            return new JNegateOp(line, unaryExpression());
1063        }
1064        else if (have(PLUS)) {
1065            return new JUnaryPlusOp(line, unaryExpression());
1066        }
1067        else if (have(NOT)) {
1068            return new JComplementOp(line, unaryExpression());
1069        }
1070        else {
1071            return simpleUnaryExpression();
1072        }
```

```
1073    }
1074
1075    /**
1076     * Parses a simple unary expression and returns an AST for it.
1077     *
1078     * <pre>
1079     *   simpleUnaryExpression ::= LNOT unaryExpression
1080     *                           | LPAREN basicType RPAREN unaryExpression
1081     *                           | LPAREN referenceType RPAREN simpleUnaryExpression
1082     *                           | postfixExpression
1083     * </pre>
1084     *
1085     * @return an AST for a simple unary expression.
1086     */
1087    private JExpression simpleUnaryExpression() {
1088        int line = scanner.token().line();
1089        if (have(LNOT)) {
1090            return new JLogicalNotOp(line, unaryExpression());
1091        } else if (seeCast()) {
1092            mustBe(LPAREN);
1093            boolean isBasicType = seeBasicType();
1094            Type type = type();
1095            mustBe(RPAREN);
1096            JExpression expr = isBasicType ? unaryExpression() : simpleUnaryExpression();
1097            return new JCastOp(line, type, expr);
1098        } else {
1099            return postfixExpression();
1100        }
1101    }
1102
1103    /**
1104     * Parses a postfix expression and returns an AST for it.
1105     *
1106     * <pre>
1107     *   postfixExpression ::= primary { selector } { DEC | INC }
1108     * </pre>
1109     *
1110     * @return an AST for a postfix expression.
1111     */
1112    private JExpression postfixExpression() {
1113        int line = scanner.token().line();
1114        JExpression primaryExpr = primary();
1115        while (see(DOT) || see(LBRACK)) {
1116            primaryExpr = selector(primaryExpr);
1117        }
1118        while (have(DEC)) {
1119            primaryExpr = new JPostDecrementOp(line, primaryExpr);
1120        }
1121        while (have(INC)) {
```

```
1122            primaryExpr = new JPostIncrementOp(line, primaryExpr);
1123        }
1124        return primaryExpr;
1125    }
1126
1127    /**
1128     * Parses a selector and returns an AST for it.
1129     *
1130     * <pre>
1131     *   selector ::= DOT qualifiedIdentifier [ arguments ]
1132     *              | LBRACK expression RBRACK
1133     * </pre>
1134     *
1135     * @param target the target expression for this selector.
1136     * @return an AST for a selector.
1137     */
1138    private JExpression selector(JExpression target) {
1139        int line = scanner.token().line();
1140        if (have(DOT)) {
1141            // target.selector.
1142            mustBe(IDENTIFIER);
1143            String name = scanner.previousToken().image();
1144            if (see(LPAREN)) {
1145                ArrayList<JExpression> args = arguments();
1146                return new JMessageExpression(line, target, name, args);
1147            } else {
1148                return new JFieldSelection(line, target, name);
1149            }
1150        } else {
1151            mustBe(LBRACK);
1152            JExpression index = expression();
1153            mustBe(RBRACK);
1154            return new JArrayExpression(line, target, index);
1155        }
1156    }
1157
1158    /**
1159     * Parses a primary expression and returns an AST for it.
1160     *
1161     * <pre>
1162     *   primary ::= parExpression
1163     *             | NEW creator
1164     *             | THIS [ arguments ]
1165     *             | SUPER ( arguments | DOT IDENTIFIER [ arguments ] )
1166     *             | qualifiedIdentifier [ arguments ]
1167     *             | literal
1168     * </pre>
1169     *
1170     * @return an AST for a primary expression.
```

```
1171      */
1172     private JExpression primary() {
1173         int line = scanner.token().line();
1174         if (see(LPAREN)) {
1175             return parExpression();
1176         } else if (have(NEW)) {
1177             return creator();
1178         } else if (have(THIS)) {
1179             if (see(LPAREN)) {
1180                 ArrayList<JExpression> args = arguments();
1181                 return new JThisConstruction(line, args);
1182             } else {
1183                 return new JThis(line);
1184             }
1185         } else if (have(SUPER)) {
1186             if (!have(DOT)) {
1187                 ArrayList<JExpression> args = arguments();
1188                 return new JSuperConstruction(line, args);
1189             } else {
1190                 mustBe(IDENTIFIER);
1191                 String name = scanner.previousToken().image();
1192                 JExpression newTarget = new JSuper(line);
1193                 if (see(LPAREN)) {
1194                     ArrayList<JExpression> args = arguments();
1195                     return new JMessageExpression(line, newTarget, null, name, args);
1196                 } else {
1197                     return new JFieldSelection(line, newTarget, name);
1198                 }
1199             }
1200         } else if (see(IDENTIFIER)) {
1201             TypeName id = qualifiedIdentifier();
1202             if (see(LPAREN)) {
1203                 // ambiguousPart.messageName(...).
1204                 ArrayList<JExpression> args = arguments();
1205                 return new JMessageExpression(line, null, ambiguousPart(id), id.simpleName(), args);
1206             } else if (ambiguousPart(id) == null) {
1207                 // A simple name.
1208                 return new JVariable(line, id.simpleName());
1209             } else {
1210                 // ambiguousPart.fieldName.
1211                 return new JFieldSelection(line, ambiguousPart(id), null, id.simpleName());
1212             }
1213         } else {
1214             return literal();
1215         }
1216     }
1217
1218     /**
1219      * Parses a creator and returns an AST for it.
```

```
1220      *
1221      * <pre>
1222      *   creator ::= ( basicType | qualifiedIdentifier )
1223      *                 ( arguments
1224      *                 | LBRACK RBRACK { LBRACK RBRACK } [ arrayInitializer ]
1225      *                 | newArrayDeclarator
1226      *                 )
1227      * </pre>
1228      *
1229      * @return an AST for a creator.
1230      */
1231     private JExpression creator() {
1232       int line = scanner.token().line();
1233       Type type = seeBasicType() ? basicType() : qualifiedIdentifier();
1234       if (see(LPAREN)) {
1235         ArrayList<JExpression> args = arguments();
1236         return new JNewOp(line, type, args);
1237       } else if (see(LBRACK)) {
1238         if (seeDims()) {
1239           Type expected = type;
1240           while (have(LBRACK)) {
1241             mustBe(RBRACK);
1242             expected = new ArrayTypeName(expected);
1243           }
1244           return arrayInitializer(expected);
1245         } else {
1246           return newArrayDeclarator(line, type);
1247         }
1248       } else {
1249         reportParserError("( or [ sought where %s found", scanner.token().image());
1250         return new JWildExpression(line);
1251       }
1252     }
1253
1254     /**
1255      * Parses a new array declarator and returns an AST for it.
1256      *
1257      * <pre>
1258      *   newArrayDeclarator ::= LBRACK expression RBRACK
1259      *                          { LBRACK expression RBRACK } { LBRACK RBRACK }
1260      * </pre>
1261      *
1262      * @param line line in which the declarator occurred.
1263      * @param type type of the array.
1264      * @return an AST for a new array declarator.
1265      */
1266     private JNewArrayOp newArrayDeclarator(int line, Type type) {
1267       ArrayList<JExpression> dimensions = new ArrayList<JExpression>();
1268       mustBe(LBRACK);
```

```
1269        dimensions.add(expression());
1270        mustBe(RBRACK);
1271        type = new ArrayTypeName(type);
1272        while (have(LBRACK)) {
1273          if (have(RBRACK)) {
1274            // We're done with dimension expressions.
1275            type = new ArrayTypeName(type);
1276            while (have(LBRACK)) {
1277              mustBe(RBRACK);
1278              type = new ArrayTypeName(type);
1279            }
1280            return new JNewArrayOp(line, type, dimensions);
1281          } else {
1282            dimensions.add(expression());
1283            type = new ArrayTypeName(type);
1284            mustBe(RBRACK);
1285          }
1286        }
1287        return new JNewArrayOp(line, type, dimensions);
1288      }
1289
1290      /**
1291       * Parses a literal and returns an AST for it.
1292       *
1293       * <pre>
1294       *   literal ::= CHAR_LITERAL | FALSE | INT_LITERAL | DOUBLE_LITERAL | LONG_LITERAL | NULL |
      STRING_LITERAL | TRUE
1295       * </pre>
1296       *
1297       * @return an AST for a literal.
1298       */
1299      private JExpression literal() {
1300        int line = scanner.token().line();
1301        if (have(CHAR_LITERAL)) {
1302          return new JLiteralChar(line, scanner.previousToken().image());
1303        } else if (have(FALSE)) {
1304          return new JLiteralBoolean(line, scanner.previousToken().image());
1305        } else if (have(INT_LITERAL)) {
1306          return new JLiteralInt(line, scanner.previousToken().image());
1307        } else if (have(DOUBLE_LITERAL)) {
1308          return new JLiteralDouble(line, scanner.previousToken().image());
1309        } else if (have(LONG_LITERAL)) {
1310          return new JLiteralLong(line, scanner.previousToken().image());
1311        } else if (have(NULL)) {
1312          return new JLiteralNull(line);
1313        } else if (have(STRING_LITERAL)) {
1314          return new JLiteralString(line, scanner.previousToken().image());
1315        } else if (have(TRUE)) {
1316          return new JLiteralBoolean(line, scanner.previousToken().image());
```

```
1317        } else {
1318            reportParserError("Literal sought where %s found", scanner.token().image());
1319            return new JWildExpression(line);
1320        }
1321    }
1322
1323    ////////////////////////////////////////////////
1324    // Parsing Support
1325    // ////////////////////////////////////////////
1326
1327    // Returns true if the current token equals sought, and false otherwise.
1328    private boolean see(TokenKind sought) {
1329        return (sought == scanner.token().kind());
1330    }
1331
1332    // If the current token equals sought, scans it and returns true. Otherwise, returns false
1333    // without scanning the token.
1334    private boolean have(TokenKind sought) {
1335        if (see(sought)) {
1336            scanner.next();
1337            return true;
1338        } else {
1339            return false;
1340        }
1341    }
1342
1343    // Attempts to match a token we're looking for with the current input token. On success,
1344    // scans the token and goes into a "Recovered" state. On failure, what happens next depends
1345    // on whether or not the parser is currently in a "Recovered" state: if so, it reports the
1346    // error and goes into an "Unrecovered" state; if not, it repeatedly scans tokens until it
1347    // finds the one it is looking for (or EOF) and then returns to a "Recovered" state. This
1348    // gives us a kind of poor man's syntactic error recovery, a strategy due to David Turner and
1349    // Ron Morrison.
1350    private void mustBe(TokenKind sought) {
1351        if (scanner.token().kind() == sought) {
1352            scanner.next();
1353            isRecovered = true;
1354        } else if (isRecovered) {
1355            isRecovered = false;
1356            reportParserError("%s found where %s sought", scanner.token().image(), sought.image());
1357        } else {
1358            // Do not report the (possibly spurious) error, but rather attempt to recover by
1359            // forcing a match.
1360            while (!see(sought) && !see(EOF)) {
1361                scanner.next();
1362            }
1363            if (see(sought)) {
1364                scanner.next();
1365                isRecovered = true;
```

```
1366            }
1367        }
1368    }
1369
1370    // Pulls out and returns the ambiguous part of a name.
1371    private AmbiguousName ambiguousPart(TypeName name) {
1372        String qualifiedName = name.toString();
1373        int i = qualifiedName.lastIndexOf('.');
1374        return i == -1 ? null : new AmbiguousName(name.line(), qualifiedName.substring(0, i));
1375    }
1376
1377    // Reports a syntax error.
1378    private void reportParserError(String message, Object... args) {
1379        isInError = true;
1380        isRecovered = false;
1381        System.err.printf("%s:%d: error: ", scanner.fileName(), scanner.token().line());
1382        System.err.printf(message, args);
1383        System.err.println();
1384    }
1385
1386    ///////////////////////////////////////////////
1387    // Lookahead Methods
1388    ///////////////////////////////////////////////
1389
1390    // Returns true if we are looking at an IDENTIFIER followed by a LPAREN, and false otherwise.
1391    private boolean seeIdentLParen() {
1392        scanner.recordPosition();
1393        boolean result = have(IDENTIFIER) && see(LPAREN);
1394        scanner.returnToPosition();
1395        return result;
1396    }
1397
1398    // Returns true if we are looking at a cast (basic or reference), and false otherwise.
1399    private boolean seeCast() {
1400        scanner.recordPosition();
1401        if (!have(LPAREN)) {
1402            scanner.returnToPosition();
1403            return false;
1404        }
1405        if (seeBasicType()) {
1406            scanner.returnToPosition();
1407            return true;
1408        }
1409        if (!see(IDENTIFIER)) {
1410            scanner.returnToPosition();
1411            return false;
1412        } else {
1413            scanner.next();
1414            // A qualified identifier is ok.
```

```
1415            while (have(DOT)) {
1416                if (!have(IDENTIFIER)) {
1417                    scanner.returnToPosition();
1418                    return false;
1419                }
1420            }
1421        }
1422        while (have(LBRACK)) {
1423            if (!have(RBRACK)) {
1424                scanner.returnToPosition();
1425                return false;
1426            }
1427        }
1428        if (!have(RPAREN)) {
1429            scanner.returnToPosition();
1430            return false;
1431        }
1432        scanner.returnToPosition();
1433        return true;
1434    }
1435
1436    // Returns true if we are looking at a local variable declaration, and false otherwise.
1437    private boolean seeLocalVariableDeclaration() {
1438        scanner.recordPosition();
1439        if (have(IDENTIFIER)) {
1440            // A qualified identifier is ok.
1441            while (have(DOT)) {
1442                if (!have(IDENTIFIER)) {
1443                    scanner.returnToPosition();
1444                    return false;
1445                }
1446            }
1447        } else if (seeBasicType()) {
1448            scanner.next();
1449        } else {
1450            scanner.returnToPosition();
1451            return false;
1452        }
1453        while (have(LBRACK)) {
1454            if (!have(RBRACK)) {
1455                scanner.returnToPosition();
1456                return false;
1457            }
1458        }
1459        if (!have(IDENTIFIER)) {
1460            scanner.returnToPosition();
1461            return false;
1462        }
1463        while (have(LBRACK)) {
```

```
1464            if (!have(RBRACK)) {
1465                scanner.returnToPosition();
1466                return false;
1467            }
1468        }
1469        scanner.returnToPosition();
1470        return true;
1471    }
1472
1473    // Returns true if we are looking at a basic type, and false otherwise.
1474    private boolean seeBasicType() {
1475        return (see(BOOLEAN) || see(CHAR) || see(INT)
1476                || see(DOUBLE) || see(LONG));
1477    }
1478
1479    // Returns true if we are looking at a reference type, and false otherwise.
1480    private boolean seeReferenceType() {
1481        if (see(IDENTIFIER)) {
1482            return true;
1483        } else {
1484            scanner.recordPosition();
1485            if (have(BOOLEAN) || have(CHAR) || have(INT) || have(DOUBLE) || have(LONG)) {
1486                if (have(LBRACK) && see(RBRACK)) {
1487                    scanner.returnToPosition();
1488                    return true;
1489                }
1490            }
1491            scanner.returnToPosition();
1492        }
1493        return false;
1494    }
1495
1496    // Returns true if we are looking at a [] pair, and false otherwise.
1497    private boolean seeDims() {
1498        scanner.recordPosition();
1499        boolean result = have(LBRACK) && see(RBRACK);
1500        scanner.returnToPosition();
1501        return result;
1502    }
1503 }
1504
```

JUnaryExpression.java

Download

```java
// Copyright 2012- Bill Campbell, Swami Iyer and Bahar Akbal-Delibas

package jminusminus;

import static jminusminus.CLConstants.*;

/**
 * This abstract base class is the AST node for an unary expression --- an expression with a
 * single operand.
 */
abstract class JUnaryExpression extends JExpression {
    /**
     * The unary operator.
     */
    protected String operator;

    /**
     * The operand.
     */
    protected JExpression operand;

    /**
     * Constructs an AST node for an unary expression.
     *
     * @param line     line in which the unary expression occurs in the source file.
     * @param operator the unary operator.
     * @param operand  the operand.
     */
    protected JUnaryExpression(int line, String operator, JExpression operand) {
        super(line);
        this.operator = operator;
        this.operand = operand;
    }

    /**
     * {@inheritDoc}
     */
    public void toJSON(JSONElement json) {
        JSONElement e = new JSONElement();
        json.addChild("JUnaryExpression:" + line, e);
        e.addAttribute("operator", operator);
        e.addAttribute("type", type == null ? "" : type.toString());
        JSONElement e1 = new JSONElement();
        e.addChild("Operand", e1);
        operand.toJSON(e1);
    }
```

```
47  }
48
49  /**
50   * The AST node for a logical NOT (!) expression.
51   */
52  class JLogicalNotOp extends JUnaryExpression {
53      /**
54       * Constructs an AST for a logical NOT expression.
55       *
56       * @param line line in which the logical NOT expression occurs in the source file.
57       * @param arg  the operand.
58       */
59      public JLogicalNotOp(int line, JExpression arg) {
60          super(line, "!", arg);
61      }
62
63      /**
64       * {@inheritDoc}
65       */
66      public JExpression analyze(Context context) {
67          operand = (JExpression) operand.analyze(context);
68          operand.type().mustMatchExpected(line(), Type.BOOLEAN);
69          type = Type.BOOLEAN;
70          return this;
71      }
72
73      /**
74       * {@inheritDoc}
75       */
76      public void codegen(CLEmitter output) {
77          String falseLabel = output.createLabel();
78          String trueLabel = output.createLabel();
79          this.codegen(output, falseLabel, false);
80          output.addNoArgInstruction(ICONST_1); // true
81          output.addBranchInstruction(GOTO, trueLabel);
82          output.addLabel(falseLabel);
83          output.addNoArgInstruction(ICONST_0); // false
84          output.addLabel(trueLabel);
85      }
86
87      /**
88       * {@inheritDoc}
89       */
90      public void codegen(CLEmitter output, String targetLabel, boolean onTrue) {
91          operand.codegen(output, targetLabel, !onTrue);
92      }
93  }
94
95  /**
```

```
 * The AST node for a unary negation (-) expression.
 */
class JNegateOp extends JUnaryExpression {
    /**
     * Constructs an AST node for a negation expression.
     *
     * @param line    line in which the negation expression occurs in the source file.
     * @param operand the operand.
     */
    public JNegateOp(int line, JExpression operand) {
        super(line, "-", operand);
    }

    /**
     * {@inheritDoc}
     */
    public JExpression analyze(Context context) {
        operand = operand.analyze(context);
        operand.type().mustMatchExpected(line(), Type.INT);
        type = Type.INT;
        return this;
    }

    /**
     * {@inheritDoc}
     */
    public void codegen(CLEmitter output) {
        operand.codegen(output);
        output.addNoArgInstruction(INEG);
    }
}

/**
 * The AST node for a post-decrement (--) expression.
 */
class JPostDecrementOp extends JUnaryExpression {
    /**
     * Constructs an AST node for a post-decrement expression.
     *
     * @param line    line in which the expression occurs in the source file.
     * @param operand the operand.
     */
    public JPostDecrementOp(int line, JExpression operand) {
        super(line, "-- (post)", operand);
    }

    /**
     * {@inheritDoc}
     */
```

```
145     public JExpression analyze(Context context) {
146       if (!(operand instanceof JLhs)) {
147         JAST.compilationUnit.reportSemanticError(line, "Operand to -- must have an LValue.");
148         type = Type.ANY;
149       } else {
150         operand = (JExpression) operand.analyze(context);
151         operand.type().mustMatchExpected(line(), Type.INT);
152         type = Type.INT;
153       }
154       return this;
155     }
156 
157     /**
158      * {@inheritDoc}
159      */
160     public void codegen(CLEmitter output) {
161       if (operand instanceof JVariable) {
162         // A local variable; otherwise analyze() would have replaced it with an explicit
163         // field selection.
164         int offset = ((LocalVariableDefn) ((JVariable) operand).iDefn()).offset();
165         if (!isStatementExpression) {
166           // Loading its original rvalue.
167           operand.codegen(output);
168         }
169         output.addIINCInstruction(offset, -1);
170       } else {
171         ((JLhs) operand).codegenLoadLhsLvalue(output);
172         ((JLhs) operand).codegenLoadLhsRvalue(output);
173         if (!isStatementExpression) {
174           // Loading its original rvalue.
175           ((JLhs) operand).codegenDuplicateRvalue(output);
176         }
177         output.addNoArgInstruction(ICONST_1);
178         output.addNoArgInstruction(ISUB);
179         ((JLhs) operand).codegenStore(output);
180       }
181     }
182 }
183 
184 /**
185  * The AST node for pre-increment (++) expression.
186  */
187 class JPreIncrementOp extends JUnaryExpression {
188     /**
189      * Constructs an AST node for a pre-increment expression.
190      *
191      * @param line    line in which the expression occurs in the source file.
192      * @param operand the operand.
193      */
```

```
194     public JPreIncrementOp(int line, JExpression operand) {
195       super(line, "++ (pre)", operand);
196     }
197 
198     /**
199      * {@inheritDoc}
200      */
201     public JExpression analyze(Context context) {
202       if (!(operand instanceof JLhs)) {
203         JAST.compilationUnit.reportSemanticError(line, "Operand to ++ must have an LValue.");
204         type = Type.ANY;
205       } else {
206         operand = (JExpression) operand.analyze(context);
207         operand.type().mustMatchExpected(line(), Type.INT);
208         type = Type.INT;
209       }
210       return this;
211     }
212 
213     /**
214      * {@inheritDoc}
215      */
216     public void codegen(CLEmitter output) {
217       if (operand instanceof JVariable) {
218         // A local variable; otherwise analyze() would have replaced it with an explicit
219         // field selection.
220         int offset = ((LocalVariableDefn) ((JVariable) operand).iDefn()).offset();
221         output.addIINCInstruction(offset, 1);
222         if (!isStatementExpression) {
223           // Loading its original rvalue.
224           operand.codegen(output);
225         }
226       } else {
227         ((JLhs) operand).codegenLoadLhsLvalue(output);
228         ((JLhs) operand).codegenLoadLhsRvalue(output);
229         output.addNoArgInstruction(ICONST_1);
230         output.addNoArgInstruction(IADD);
231         if (!isStatementExpression) {
232           // Loading its original rvalue.
233           ((JLhs) operand).codegenDuplicateRvalue(output);
234         }
235         ((JLhs) operand).codegenStore(output);
236       }
237     }
238 }
239 
240 /**
241  * The AST node for a unary plus (+) expression.
242  */
```

```
243 class JUnaryPlusOp extends JUnaryExpression {
244     /**
245      * Constructs an AST node for a unary plus expression.
246      *
247      * @param line    line in which the unary plus expression occurs in the source file.
248      * @param operand the operand.
249      */
250     public JUnaryPlusOp(int line, JExpression operand) {
251         super(line, "+", operand);
252     }
253 
254     /**
255      * {@inheritDoc}
256      */
257     public JExpression analyze(Context context) {
258         operand = operand.analyze(context);
259         operand.type().mustMatchExpected(line(), Type.INT);
260         type = Type.INT;
261         return this;
262     }
263 
264     /**
265      * {@inheritDoc}
266      */
267     public void codegen(CLEmitter output) {
268         operand.codegen(output);
269         output.addNoArgInstruction(ICONST_0);
270         output.addNoArgInstruction(IADD);
271     }
272 }
273 
274 /**
275  * The AST node for a unary complement (~) expression.
276  */
277 class JComplementOp extends JUnaryExpression {
278     /**
279      * Constructs an AST node for a unary complement expression.
280      *
281      * @param line    line in which the unary complement expression occurs in the source file.
282      * @param operand the operand.
283      */
284     public JComplementOp(int line, JExpression operand) {
285         super(line, "~", operand);
286     }
287 
288     /**
289      * {@inheritDoc}
290      */
291     public JExpression analyze(Context context) {
```

```
292         operand = operand.analyze(context);
293         operand.type().mustMatchExpected(line(), Type.INT);
294         type = Type.INT;
295         return this;
296     }
297 
298     /**
299      * {@inheritDoc}
300      */
301     public void codegen(CLEmitter output) {
302         operand.codegen(output);
303         output.addLDCInstruction(-1);
304         output.addNoArgInstruction(IXOR);
305     }
306 }
307 
308 /**
309  * The AST node for post-increment (++) expression.
310  */
311 class JPostIncrementOp extends JUnaryExpression {
312     /**
313      * Constructs an AST node for a post-increment expression.
314      *
315      * @param line    line in which the expression occurs in the source file.
316      * @param operand the operand.
317      */
318     public JPostIncrementOp(int line, JExpression operand) {
319         super(line, "++ (post)", operand);
320     }
321 
322     /**
323      * {@inheritDoc}
324      */
325     public JExpression analyze(Context context) {
326         // TODO
327         return this;
328     }
329 
330     /**
331      * {@inheritDoc}
332      */
333     public void codegen(CLEmitter output) {
334         // TODO
335     }
336 }
337 
338 /**
339  * The AST node for a pre-decrement (--) expression.
340  */
```

```
class JPreDecrementOp extends JUnaryExpression {
    /**
     * Constructs an AST node for a pre-decrement expression.
     *
     * @param line    line in which the expression occurs in the source file.
     * @param operand the operand.
     */
    public JPreDecrementOp(int line, JExpression operand) {
        super(line, "-- (pre)", operand);
    }

    /**
     * {@inheritDoc}
     */
    public JExpression analyze(Context context) {
        // TODO
        return this;
    }

    /**
     * {@inheritDoc}
     */
    public void codegen(CLEmitter output) {
        // TODO
    }
}
```

JBinaryExpression.java

Download

```java
// Copyright 2012- Bill Campbell, Swami Iyer and Bahar Akbal-Delibas

package jminusminus;

import static jminusminus.CLConstants.*;

/**
 * This abstract base class is the AST node for a binary expression --- an expression with a binary
 * operator and two operands: lhs and rhs.
 */
abstract class JBinaryExpression extends JExpression {
    /**
     * The binary operator.
     */
    protected String operator;

    /**
     * The lhs operand.
     */
    protected JExpression lhs;

    /**
     * The rhs operand.
     */
    protected JExpression rhs;

    /**
     * Constructs an AST node for a binary expression.
     *
     * @param line     line in which the binary expression occurs in the source file.
     * @param operator the binary operator.
     * @param lhs      the lhs operand.
     * @param rhs      the rhs operand.
     */
    protected JBinaryExpression(int line, String operator, JExpression lhs, JExpression rhs) {
        super(line);
        this.operator = operator;
        this.lhs = lhs;
        this.rhs = rhs;
    }

    /**
     * {@inheritDoc}
     */
    public void toJSON(JSONElement json) {
        JSONElement e = new JSONElement();
```

```
47        json.addChild("JBinaryExpression:" + line, e);
48        e.addAttribute("operator", operator);
49        e.addAttribute("type", type == null ? "" : type.toString());
50        JSONElement e1 = new JSONElement();
51        e.addChild("Operand1", e1);
52        lhs.toJSON(e1);
53        JSONElement e2 = new JSONElement();
54        e.addChild("Operand2", e2);
55        rhs.toJSON(e2);
56    }
57 }
58
59 /**
60  * The AST node for a multiplication (*) expression.
61  */
62 class JMultiplyOp extends JBinaryExpression {
63    /**
64     * Constructs an AST for a multiplication expression.
65     *
66     * @param line line in which the multiplication expression occurs in the source file.
67     * @param lhs  the lhs operand.
68     * @param rhs  the rhs operand.
69     */
70    public JMultiplyOp(int line, JExpression lhs, JExpression rhs) {
71        super(line, "*", lhs, rhs);
72    }
73
74    /**
75     * {@inheritDoc}
76     */
77    public JExpression analyze(Context context) {
78        lhs = (JExpression) lhs.analyze(context);
79        rhs = (JExpression) rhs.analyze(context);
80        lhs.type().mustMatchExpected(line(), Type.INT);
81        rhs.type().mustMatchExpected(line(), Type.INT);
82        type = Type.INT;
83        return this;
84    }
85
86    /**
87     * {@inheritDoc}
88     */
89    public void codegen(CLEmitter output) {
90        lhs.codegen(output);
91        rhs.codegen(output);
92        output.addNoArgInstruction(IMUL);
93    }
94 }
95
```

```
96  /**
97   * The AST node for a plus (+) expression. In j--, as in Java, + is overloaded to denote addition
98   * for numbers and concatenation for Strings.
99   */
100 class JPlusOp extends JBinaryExpression {
101     /**
102      * Constructs an AST node for an addition expression.
103      *
104      * @param line line in which the addition expression occurs in the source file.
105      * @param lhs  the lhs operand.
106      * @param rhs  the rhs operand.
107      */
108     public JPlusOp(int line, JExpression lhs, JExpression rhs) {
109         super(line, "+", lhs, rhs);
110     }
111 
112     /**
113      * {@inheritDoc}
114      */
115     public JExpression analyze(Context context) {
116         lhs = (JExpression) lhs.analyze(context);
117         rhs = (JExpression) rhs.analyze(context);
118         if (lhs.type() == Type.STRING || rhs.type() == Type.STRING) {
119             return (new JStringConcatenationOp(line, lhs, rhs)).analyze(context);
120         } else if (lhs.type() == Type.INT && rhs.type() == Type.INT) {
121             type = Type.INT;
122         } else {
123             type = Type.ANY;
124             JAST.compilationUnit.reportSemanticError(line(), "Invalid operand types for +");
125         }
126         return this;
127     }
128 
129     /**
130      * {@inheritDoc}
131      */
132     public void codegen(CLEmitter output) {
133         lhs.codegen(output);
134         rhs.codegen(output);
135         output.addNoArgInstruction(IADD);
136     }
137 }
138 
139 /**
140  * The AST node for a subtraction (-) expression.
141  */
142 class JSubtractOp extends JBinaryExpression {
143     /**
144      * Constructs an AST node for a subtraction expression.
```

```
145     *
146     * @param line line in which the subtraction expression occurs in the source file.
147     * @param lhs  the lhs operand.
148     * @param rhs  the rhs operand.
149     */
150    public JSubtractOp(int line, JExpression lhs, JExpression rhs) {
151        super(line, "-", lhs, rhs);
152    }
153
154    /**
155     * {@inheritDoc}
156     */
157    public JExpression analyze(Context context) {
158        lhs = (JExpression) lhs.analyze(context);
159        rhs = (JExpression) rhs.analyze(context);
160        lhs.type().mustMatchExpected(line(), Type.INT);
161        rhs.type().mustMatchExpected(line(), Type.INT);
162        type = Type.INT;
163        return this;
164    }
165
166    /**
167     * {@inheritDoc}
168     */
169    public void codegen(CLEmitter output) {
170        lhs.codegen(output);
171        rhs.codegen(output);
172        output.addNoArgInstruction(ISUB);
173    }
174}
175
176/**
177 * The AST node for a division (/) expression.
178 */
179class JDivideOp extends JBinaryExpression {
180    /**
181     * Constructs an AST node for a division expression.
182     *
183     * @param line line in which the division expression occurs in the source file.
184     * @param lhs  the lhs operand.
185     * @param rhs  the rhs operand.
186     */
187    public JDivideOp(int line, JExpression lhs, JExpression rhs) {
188        super(line, "/", lhs, rhs);
189    }
190
191    /**
192     * {@inheritDoc}
193     */
```

```
194     public JExpression analyze(Context context) {
195         lhs = (JExpression) lhs.analyze(context);
196         rhs = (JExpression) rhs.analyze(context);
197         lhs.type().mustMatchExpected(line(), Type.INT);
198         rhs.type().mustMatchExpected(line(), Type.INT);
199         type = Type.INT;
200         return this;
201     }
202 
203     /**
204      * {@inheritDoc}
205      */
206     public void codegen(CLEmitter output) {
207         lhs.codegen(output);
208         rhs.codegen(output);
209         output.addNoArgInstruction(IDIV);
210     }
211 }
212 
213 /**
214  * The AST node for a remainder (%) expression.
215  */
216 class JRemainderOp extends JBinaryExpression {
217     /**
218      * Constructs an AST node for a remainder expression.
219      *
220      * @param line line in which the division expression occurs in the source file.
221      * @param lhs  the lhs operand.
222      * @param rhs  the rhs operand.
223      */
224     public JRemainderOp(int line, JExpression lhs, JExpression rhs) {
225         super(line, "%", lhs, rhs);
226     }
227 
228     /**
229      * {@inheritDoc}
230      */
231     public JExpression analyze(Context context) {
232         lhs = (JExpression) lhs.analyze(context);
233         rhs = (JExpression) rhs.analyze(context);
234         lhs.type().mustMatchExpected(line(), Type.INT);
235         rhs.type().mustMatchExpected(line(), Type.INT);
236         type = Type.INT;
237         return this;
238     }
239 
240     /**
241      * {@inheritDoc}
242      */
```

```
243     public void codegen(CLEmitter output) {
244         lhs.codegen(output);
245         rhs.codegen(output);
246         output.addNoArgInstruction(IREM);
247     }
248 }
249 
250 /**
251  * The AST node for an inclusive or (|) expression.
252  */
253 class JOrOp extends JBinaryExpression {
254     /**
255      * Constructs an AST node for an inclusive or expression.
256      *
257      * @param line line in which the inclusive or expression occurs in the source file.
258      * @param lhs  the lhs operand.
259      * @param rhs  the rhs operand.
260      */
261     public JOrOp(int line, JExpression lhs, JExpression rhs) {
262         super(line, "|", lhs, rhs);
263     }
264 
265     /**
266      * {@inheritDoc}
267      */
268     public JExpression analyze(Context context) {
269         lhs = (JExpression) lhs.analyze(context);
270         rhs = (JExpression) rhs.analyze(context);
271         lhs.type().mustMatchExpected(line(), Type.INT);
272         rhs.type().mustMatchExpected(line(), Type.INT);
273         type = Type.INT;
274         return this;
275     }
276 
277     /**
278      * {@inheritDoc}
279      */
280     public void codegen(CLEmitter output) {
281         lhs.codegen(output);
282         rhs.codegen(output);
283         output.addNoArgInstruction(IOR);
284     }
285 }
286 
287 /**
288  * The AST node for an exclusive or (^) expression.
289  */
290 class JXorOp extends JBinaryExpression {
291     /**
```

```
292     * Constructs an AST node for an exclusive or expression.
293     *
294     * @param line line in which the exclusive or expression occurs in the source file.
295     * @param lhs  the lhs operand.
296     * @param rhs  the rhs operand.
297     */
298    public JXorOp(int line, JExpression lhs, JExpression rhs) {
299        super(line, "^", lhs, rhs);
300    }
301
302    /**
303     * {@inheritDoc}
304     */
305    public JExpression analyze(Context context) {
306        lhs = (JExpression) lhs.analyze(context);
307        rhs = (JExpression) rhs.analyze(context);
308        lhs.type().mustMatchExpected(line(), Type.INT);
309        rhs.type().mustMatchExpected(line(), Type.INT);
310        type = Type.INT;
311        return this;
312    }
313
314    /**
315     * {@inheritDoc}
316     */
317    public void codegen(CLEmitter output) {
318        lhs.codegen(output);
319        rhs.codegen(output);
320        output.addNoArgInstruction(IXOR);
321    }
322 }
323
324 /**
325  * The AST node for an and (&amp;amp;) expression.
326  */
327 class JAndOp extends JBinaryExpression {
328    /**
329     * Constructs an AST node for an and expression.
330     *
331     * @param line line in which the and expression occurs in the source file.
332     * @param lhs  the lhs operand.
333     * @param rhs  the rhs operand.
334     */
335    public JAndOp(int line, JExpression lhs, JExpression rhs) {
336        super(line, "&", lhs, rhs);
337    }
338
339    /**
340     * {@inheritDoc}
```

```
341      */
342     public JExpression analyze(Context context) {
343         lhs = (JExpression) lhs.analyze(context);
344         rhs = (JExpression) rhs.analyze(context);
345         lhs.type().mustMatchExpected(line(), Type.INT);
346         rhs.type().mustMatchExpected(line(), Type.INT);
347         type = Type.INT;
348         return this;
349     }
350 
351     /**
352      * {@inheritDoc}
353      */
354     public void codegen(CLEmitter output) {
355         lhs.codegen(output);
356         rhs.codegen(output);
357         output.addNoArgInstruction(IAND);
358     }
359 }
360 
361 /**
362  * The AST node for an arithmetic left shift (&lt;&lt;) expression.
363  */
364 class JALeftShiftOp extends JBinaryExpression {
365     /**
366      * Constructs an AST node for an arithmetic left shift expression.
367      *
368      * @param line line in which the arithmetic left shift expression occurs in the source file.
369      * @param lhs  the lhs operand.
370      * @param rhs  the rhs operand.
371      */
372     public JALeftShiftOp(int line, JExpression lhs, JExpression rhs) {
373         super(line, "<<", lhs, rhs);
374     }
375 
376     /**
377      * {@inheritDoc}
378      */
379     public JExpression analyze(Context context) {
380         lhs = (JExpression) lhs.analyze(context);
381         rhs = (JExpression) rhs.analyze(context);
382         lhs.type().mustMatchExpected(line(), Type.INT);
383         rhs.type().mustMatchExpected(line(), Type.INT);
384         type = Type.INT;
385         return this;
386     }
387 
388     /**
389      * {@inheritDoc}
```

```
390      */
391     public void codegen(CLEmitter output) {
392         lhs.codegen(output);
393         rhs.codegen(output);
394         output.addNoArgInstruction(ISHL);
395     }
396 }
397 
398 /**
399  * The AST node for an arithmetic right shift (&rt;&rt;) expression.
400  */
401 class JARightShiftOp extends JBinaryExpression {
402     /**
403      * Constructs an AST node for an arithmetic right shift expression.
404      *
405      * @param line line in which the arithmetic right shift expression occurs in the source file.
406      * @param lhs  the lhs operand.
407      * @param rhs  the rhs operand.
408      */
409     public JARightShiftOp(int line, JExpression lhs, JExpression rhs) {
410         super(line, ">>", lhs, rhs);
411     }
412 
413     /**
414      * {@inheritDoc}
415      */
416     public JExpression analyze(Context context) {
417         lhs = (JExpression) lhs.analyze(context);
418         rhs = (JExpression) rhs.analyze(context);
419         lhs.type().mustMatchExpected(line(), Type.INT);
420         rhs.type().mustMatchExpected(line(), Type.INT);
421         type = Type.INT;
422         return this;
423     }
424 
425     /**
426      * {@inheritDoc}
427      */
428     public void codegen(CLEmitter output) {
429         lhs.codegen(output);
430         rhs.codegen(output);
431         output.addNoArgInstruction(ISHR);
432     }
433 }
434 
435 /**
436  * The AST node for a logical right shift (&rt;&rt;&rt;) expression.
437  */
438 class JLRightShiftOp extends JBinaryExpression {
```

```
439     /**
440      * Constructs an AST node for a logical right shift expression.
441      *
442      * @param line line in which the logical right shift expression occurs in the source file.
443      * @param lhs  the lhs operand.
444      * @param rhs  the rhs operand.
445      */
446     public JLRightShiftOp(int line, JExpression lhs, JExpression rhs) {
447         super(line, ">>>", lhs, rhs);
448     }
449 
450     /**
451      * {@inheritDoc}
452      */
453     public JExpression analyze(Context context) {
454         lhs = (JExpression) lhs.analyze(context);
455         rhs = (JExpression) rhs.analyze(context);
456         lhs.type().mustMatchExpected(line(), Type.INT);
457         rhs.type().mustMatchExpected(line(), Type.INT);
458         type = Type.INT;
459         return this;
460     }
461 
462     /**
463      * {@inheritDoc}
464      */
465     public void codegen(CLEmitter output) {
466         lhs.codegen(output);
467         rhs.codegen(output);
468         output.addNoArgInstruction(IUSHR);
469     }
470 }
471 
```

j--.jj

Download

```
// Copyright 2012- Bill Campbell, Swami Iyer and Bahar Akbal-Delibas

/**
 * This is the input file to JavaCC for generating a scanner and a parser for j--. From the
 * specification in this file, JavaCC generates, among other files, a JavaCCParser.java program
 * (the parser) and a JavaCCParserTokenManager.java program (the scanner).
 */
PARSER_BEGIN(JavaCCParser)

package jminusminus;

import java.util.ArrayList;

/**
 * Parser generated by JavaCC. It parses a j-- compilation unit (program file), taking tokens from
 * the scanner (also generated by JavaCC), and produces an abstract syntax tree (AST) for it.
 */
class JavaCCParser {
    // Whether a parser error has been found.
    private boolean errorHasOccurred;

    // Name of the file that is parsed.
    private String fileName;

    /**
     * Sets the name of the file being parsed.
     *
     * @param fileName name of the file being parsed.
     */
    public void fileName(String fileName) {
        this.fileName = fileName;
    }

    /**
     * Returns {@code true} if a parser error has occurred up to now, and {@code false} otherwise.
     *
     * @return {@code true} if a parser error has occurred up to now, and {@code false} otherwise.
     */
    public boolean errorHasOccurred() {
        return errorHasOccurred;
    }

    // Pulls out and returns the ambiguous part of a name.
    private AmbiguousName ambiguousPart(TypeName name) {
        String qualifiedName = name.toString();
        int i = qualifiedName.lastIndexOf('.');
```

```
47        return i == -1 ? null : new AmbiguousName(name.line(), qualifiedName.substring(0, i));
48     }
49
50     // Reports a syntax error.
51     private void reportParserError(String message, Object... args) {
52        errorHasOccurred = true;
53        System.err.printf("%s:%d: error: ", fileName, token.beginLine);
54        System.err.printf(message, args);
55        System.err.println();
56     }
57
58     // Recover from the parser error that occurred by skipping to any of the specified tokens.
59     // Current error recovery mechanism is rather simple-minded and is based on skipping all the
60     // tokens until a SEMI or an EOF is encountered. This scheme can be enhanced by passing in the
61     // FOLLOW-SET of the non-terminal at hand.
62     private void recoverFromError(int[] skipTo, ParseException e) {
63        // Get the possible expected tokens.
64        StringBuffer expected = new StringBuffer();
65        for (int i = 0; i < e.expectedTokenSequences.length; i++) {
66           for (int j = 0; j < e.expectedTokenSequences[i].length; j++) {
67              expected.append("\n");
68              expected.append("    ");
69              expected.append(tokenImage[e.expectedTokenSequences[i][j]]);
70              expected.append("...");
71           }
72        }
73
74        // Print error message.
75        if (e.expectedTokenSequences.length == 1) {
76           reportParserError("\"%s\" found where %s sought", getToken(1), expected);
77        } else {
78           reportParserError("\"%s\" found where one of %s sought", getToken(1), expected);
79        }
80
81        // Recover.
82        boolean loop = true;
83        do {
84           token = getNextToken();
85           for (int i = 0; i < skipTo.length; i++) {
86              if (token.kind == skipTo[i]) {
87                 loop = false;
88                 break;
89              }
90           }
91        } while(loop);
92     }
93  }
94
95  PARSER_END(JavaCCParser)
```

```
96
97  ////////////////////////////////////////////////////////
98  //         The j-- lexical grammar starts here        //
99  ////////////////////////////////////////////////////////
100
101 // Whitespace -- ignored
102 SKIP: { " " | "\t" | "\n" | "\r" | "\f" }
103
104 // Single line comment -- ignored
105 SKIP: { <BEGIN_COMMENT: "//">: IN_SINGLE_LINE_COMMENT }
106 <IN_SINGLE_LINE_COMMENT>
107 SKIP: { <END_COMMENT: "\n" | "\r" | "\r\n">: DEFAULT }
108 <IN_SINGLE_LINE_COMMENT>
109 SKIP: { <COMMENT: ~[]> }
110
111 // Multi line comment - ignored
112 SKIP: { <BEGIN_MULTI_COMMENT: "/*">: IN_MULTI_LINE_COMMENT }
113 <IN_MULTI_LINE_COMMENT>
114 SKIP: { <END_MULTI_COMMENT: "*/">: DEFAULT }
115 <IN_MULTI_LINE_COMMENT>
116 SKIP: { <MULTI_COMMENT: ~[]> }
117
118 // Reserved words
119 TOKEN: {
120   <ABSTRACT: "abstract">
121 | <BOOLEAN: "boolean">
122 | <CHAR: "char">
123 | <CLASS: "class">
124 | <ELSE: "else">
125 | <EXTENDS: "extends">
126 | <FALSE: "false">
127 | <IF: "if">
128 | <IMPORT: "import">
129 | <INSTANCEOF: "instanceof">
130 | <INT: "int">
131 | <NEW: "new">
132 | <NULL: "null">
133 | <PACKAGE: "package">
134 | <PRIVATE: "private">
135 | <PROTECTED: "protected">
136 | <PUBLIC: "public">
137 | <RETURN: "return">
138 | <STATIC: "static">
139 | <SUPER: "super">
140 | <THIS: "this">
141 | <TRUE: "true">
142 | <VOID: "void">
143 | <WHILE: "while">
144 | <BREAK: "break">
```

```
| <CASE: "case">
| <CATCH: "catch">
| <CONTINUE: "continue">
| <DEFAULT_RW: "default">
| <DO: "do">
| <DOUBLE: "double">
| <FINALLY: "finally">
| <FOR: "for">
| <IMPLEMENTS: "implements">
| <INTERFACE: "interface">
| <LONG: "long">
| <SWITCH: "switch">
| <THROW: "throw">
| <THROWS: "throws">
| <TRY: "try">
}

// Separators
TOKEN: {
  <COMMA: ",">
| <DOT: ".">
| <LBRACK: "[">
| <LCURLY: "{">
| <LPAREN: "(">
| <RPAREN: ")">
| <RBRACK: "]">
| <RCURLY: "}">
| <SEMI: ";">
}

// Operators
TOKEN: {
  <ASSIGN: "=">
| <DEC: "--">
| <EQUAL: "==">
| <GT: ">">
| <INC: "++">
| <LAND: "&&">
| <LE: "<=">
| <LNOT: "!">
| <MINUS: "-">
| <PLUS: "+">
| <PLUS_ASSIGN: "+=">
| <STAR: "*">
| <QUESTION: "?">
| <COLON: ":">
| <NOT : "~">
| <NOT_EQUAL: "!=">
| <DIV: "/">
```

```
| <DIV_ASSIGN: "/=">
| <MINUS_ASSIGN: "-=">
| <STAR_ASSIGN: "*=">
| <REM: "%">
| <REM_ASSIGN: "%=">
| <ARSHIFT: ">>">
| <ARSHIFT_ASSIGN: ">>=">
| <LRSHIFT: ">>>">
| <LRSHIFT_ASSIGN: ">>>=">
| <GE: ">=">
| <ALSHIFT: "<<">
| <ALSHIFT_ASSIGN: "<<=">
| <LT: "<">
| <XOR: "^">
| <XOR_ASSIGN: "^=">
| <OR: "|">
| <OR_ASSIGN: "|=">
| <LOR: "||">
| <AND: "&">
| <AND_ASSIGN: "&=">
}

// Identifiers
TOKEN: {
  <IDENTIFIER: ( <LETTER> | "_" | "$" ) ( <LETTER> | <DIGIT> | "_" | "$" )*>
| <#LETTER: [ "a"-"z", "A"-"Z" ]>
| <#DIGIT: [ "0"-"9" ]>
| <#EXPONENT: ["e", "E"]>
| <#PLUS_OR_MINUS: ["+", "-"]>
| <#SUFFIX_D: ["d", "D"]>
}

TOKEN: {
  <INT_LITERAL: <DIGIT> ( <DIGIT> )*>
| <LONG_LITERAL: <DIGIT> ( <DIGIT> )* [ "l", "L" ]>
| <DOUBLE_LITERAL: ["."] <DIGIT> ( <DIGIT> )* <EXPONENT> <PLUS_OR_MINUS> ( <DIGIT> )*
<SUFFIX_D>
                | <DIGIT> ( <DIGIT> )* <EXPONENT> <PLUS_OR_MINUS> ( <DIGIT> )* <SUFFIX_D>
                | <DIGIT> ["."] ( <DIGIT> )* <EXPONENT> <PLUS_OR_MINUS> ( <DIGIT> )* <SUFFIX_D>
                | <DIGIT> ( <DIGIT> )* <SUFFIX_D>
                | <DIGIT> ( <DIGIT> )* <EXPONENT> ( <DIGIT> )*
                | <DIGIT> ( <DIGIT> )* <EXPONENT> ( <DIGIT> )* <SUFFIX_D>
                | <DIGIT> ( <DIGIT> )* <EXPONENT> <PLUS_OR_MINUS> ( <DIGIT> )*
                | <DIGIT> ["."] ( <DIGIT> )*
                | <DIGIT> ["."] ( <DIGIT> )* <SUFFIX_D>
                | <DIGIT> ( <DIGIT> )* ["."]
                | <DIGIT> ( <DIGIT> )* ["."] <SUFFIX_D>
                | <DIGIT> ( <DIGIT> )* ["."] <EXPONENT> ( <DIGIT> )*
                | <DIGIT> ( <DIGIT> )* ["."] <EXPONENT> ( <DIGIT> )* <SUFFIX_D>
```

```
                | <DIGIT> ( <DIGIT> )* ["."] <EXPONENT> <PLUS_OR_MINUS> ( <DIGIT> )* <SUFFIX_D>
                | ["."] <DIGIT> ( <DIGIT> )*
                | ["."] <DIGIT> ( <DIGIT> )* <SUFFIX_D>
                | ["."] <DIGIT> ( <DIGIT> )* <EXPONENT> <SUFFIX_D>
                | ["."] <DIGIT> ( <DIGIT> )* <EXPONENT> ( <DIGIT> )*
                | ["."] <DIGIT> ( <DIGIT> )* <EXPONENT> <PLUS_OR_MINUS> ( <DIGIT> )*
                | ["."] <DIGIT> ( <DIGIT> )* <EXPONENT> ( <DIGIT> )* <SUFFIX_D>
                | <DIGIT> ( <DIGIT> )* ["."] ( <DIGIT> )*
                | <DIGIT> ( <DIGIT> )* ["."] ( <DIGIT> )* <SUFFIX_D>
                | <DIGIT> ( <DIGIT> )* ["."] ( <DIGIT> )* <EXPONENT> ( <DIGIT> )*
                | <DIGIT> ( <DIGIT> )* ["."] ( <DIGIT> )* <EXPONENT> <PLUS_OR_MINUS> ( <DIGIT> )*
                | <DIGIT> ( <DIGIT> )* ["."] ( <DIGIT> )* <EXPONENT> ( <DIGIT> )* <SUFFIX_D>>

| <CHAR_LITERAL: "'" ( <ESC> | ~[ "'", "\\" ] ) "'">
| <STRING_LITERAL: "\"" ( <ESC> | ~[ "\"", "\\" ] )* "\"">
| <#ESC: "\\" [ "n", "t", "b", "r", "f", "\\", "'", "\"" ]>
}

// For anything else, we return an ERROR token. Without this definition the TokenManager will throw
// an Error when a lexical error occurs, making it impossible to recover from it. So we define this
// ERROR token.
TOKEN: { <ERROR: ~[]> }

////////////////////////////////////////////////////////
//      The j-- syntactic grammar starts here         //
////////////////////////////////////////////////////////

/**
 * Parses a compilation unit (a program file) and returns an AST for it.
 *
 * <pre>
 *   compilationUnit ::= [ PACKAGE qualifiedIdentifier SEMI ]
 *                       { IMPORT  qualifiedIdentifier SEMI }
 *                       { typeDeclaration }
 *                       EOF
 * </pre>
 *
 * @return an AST for a compilation unit.
 */
public JCompilationUnit compilationUnit():
{
    int line = 0;
    TypeName packageName = null;
    TypeName anImport = null;
    ArrayList<TypeName> imports = new ArrayList<TypeName>();
    JAST aTypeDeclaration = null;
    ArrayList<JAST> typeDeclarations = new ArrayList<JAST>();
}
{
```

```
    try {
        [
            <PACKAGE>
            { line = token.beginLine; }
            packageName = qualifiedIdentifier()
            <SEMI>
        ]
        (
            <IMPORT>
            { line = line == 0 ? token.beginLine : line; }
            anImport = qualifiedIdentifier()
            { imports.add(anImport); }
            <SEMI>
        )*
        (
            aTypeDeclaration = typeDeclaration()
            {
                line = line == 0 ? aTypeDeclaration.line() : line;
                typeDeclarations.add(aTypeDeclaration);
            }
        )*
        <EOF>
        { line = line == 0 ? token.beginLine : line; }
    } catch (ParseException e) {
        recoverFromError(new int[] { SEMI, EOF }, e);
    }
    { return new JCompilationUnit(fileName, line, packageName, imports, typeDeclarations); }
}

/**
 * Parses and returns a qualified identifier.
 *
 * <pre>
 *   qualifiedIdentifier ::= IDENTIFIER { DOT IDENTIFIER }
 * </pre>
 *
 * @return a qualified identifier.
 */
private TypeName qualifiedIdentifier():
{
    int line = 0;
    String qualifiedIdentifier = "";
}
{
    try {
        <IDENTIFIER>
        {
            line = token.beginLine;
            qualifiedIdentifier = token.image;
```

```
340         }
341         (
342             // Lookahead added to suppress JavaCC warnings.
343             LOOKAHEAD(<DOT> <IDENTIFIER>)
344             <DOT> <IDENTIFIER>
345             { qualifiedIdentifier += "." + token.image; }
346         )*
347     } catch (ParseException e) {
348         recoverFromError(new int[] { SEMI, EOF }, e);
349     }
350     { return new TypeName(line, qualifiedIdentifier); }
351 }
352 
353 /**
354  * Parses a type declaration and returns an AST for it.
355  *
356  * <pre>
357  *   typeDeclaration ::= modifiers (classDeclaration | interfaceDeclaration)
358  * </pre>
359  *
360  * @return an AST for a type declaration.
361  */
362 private JAST typeDeclaration():
363 {
364     ArrayList<String> mods = null;
365     JAST declaration = null;
366 }
367 {
368     try {
369         mods = modifiers()
370         declaration = classDeclaration(mods) |
371         declaration = interfaceDeclaration(mods)
372 
373     } catch (ParseException e) {
374         recoverFromError(new int[] { SEMI, EOF }, e);
375     }
376     { return declaration; }
377 }
378 
379 /**
380  * Parses and returns a list of modifiers.
381  *
382  * <pre>
383  *   modifiers ::= { ABSTRACT | PRIVATE | PROTECTED | PUBLIC | STATIC }
384  * </pre>
385  *
386  * @return a list of modifiers.
387  */
388 private ArrayList<String> modifiers():
```

```
{
    ArrayList<String> mods = new ArrayList<String>();
    boolean scannedPUBLIC    = false;
    boolean scannedPROTECTED = false;
    boolean scannedPRIVATE   = false;
    boolean scannedSTATIC    = false;
    boolean scannedABSTRACT  = false;
}
{
    try {
        (
            <ABSTRACT>
            {
                mods.add("abstract");
                if (scannedABSTRACT) {
                    reportParserError("Repeated modifier: abstract");
                }
                scannedABSTRACT = true;
            } |
            <PRIVATE>
            {
                mods.add("private");
                if (scannedPRIVATE) {
                    reportParserError("Repeated modifier: private");
                }
                if (scannedPUBLIC || scannedPROTECTED) {
                    reportParserError("Access conflict in modifiers");
                }
                scannedPRIVATE = true;
            } |
            <PROTECTED>
            {
                mods.add("protected");
                if (scannedPROTECTED) {
                    reportParserError("Repeated modifier: protected");
                }
                if (scannedPUBLIC || scannedPRIVATE) {
                    reportParserError("Access conflict in modifiers");
                }
                scannedPROTECTED = true;
            } |
            <PUBLIC>
            {
                mods.add("public");
                if (scannedPUBLIC) {
                    reportParserError("Repeated modifier: public");
                }
                if (scannedPROTECTED || scannedPRIVATE) {
                    reportParserError("Access conflict in modifiers");
```

```
438             }
439             scannedPUBLIC = true;
440           } |
441           <STATIC>
442           {
443             mods.add("static");
444             if (scannedSTATIC) {
445               reportParserError("Repeated modifier: static");
446             }
447             scannedSTATIC = true;
448           }
449         )*
450     } catch (ParseException e) {
451       recoverFromError(new int[] { SEMI, EOF }, e);
452     }
453     { return mods; }
454 }
455 
456 /**
457  * Parses a class declaration and returns an AST for it.
458  *
459  * <pre>
460  *   classDeclaration ::= CLASS IDENTIFIER [ EXTENDS qualifiedIdentifier ] classBody
461  * </pre>
462  *
463  * @param mods the class modifiers.
464  * @return an AST for a class declaration.
465  */
466 private JClassDeclaration classDeclaration(ArrayList<String> mods):
467 {
468     int line = 0;
469     String name = "";
470     Type superClass = Type.OBJECT;
471     ArrayList<JMember> classBody = null;
472     ArrayList<TypeName> superInterfaces = null;
473     TypeName qualifiedId = null;
474 }
475 {
476     try {
477       <CLASS>
478       { line = token.beginLine; }
479       <IDENTIFIER>
480       { name = token.image; }
481       [
482         <EXTENDS>
483         superClass = qualifiedIdentifier()
484       ]
485       [
486         <IMPLEMENTS>
```

```
487             { superInterfaces = new ArrayList<TypeName>(); }
488             qualifiedId = qualifiedIdentifier()
489             { superInterfaces.add(qualifiedId); }
490             (
491                 <COMMA>
492                 qualifiedId = qualifiedIdentifier()
493                 { superInterfaces.add(qualifiedId); }
494             )*
495         ]
496         classBody = classBody()
497     } catch (ParseException e) {
498         recoverFromError(new int[] { SEMI, EOF }, e);
499     }
500     { return new JClassDeclaration(line, mods, name, superClass, superInterfaces, classBody); }
501 }
502 
503 /**
504  * Parses a class body and returns a list of members in the body.
505  *
506  * <pre>
507  *   classBody ::= LCURLY { modifiers memberDecl } RCURLY
508  * </pre>
509  *
510  * @return a list of members in the class body.
511  */
512 private ArrayList<JMember> classBody():
513 {
514     ArrayList<String> mods = null;
515     JMember aMember        = null;
516     ArrayList<JMember> members = new ArrayList<JMember>();
517 }
518 {
519     try {
520         <LCURLY>
521         (
522             mods = modifiers()
523             aMember = memberDecl(mods)
524             { members.add(aMember); }
525         )*
526         <RCURLY>
527     } catch (ParseException e) {
528         recoverFromError(new int[] { SEMI, EOF }, e);
529     }
530     { return members; }
531 }
532 
533 /**
534  * Parses a member declaration and returns an AST for it.
535  *
```

```
 * <pre>
 *   memberDecl ::= IDENTIFIER formalParameters
 *                  [ THROWS qualifiedIdentifier { COMMA qualifiedIdentifier } ] block
 *                | ( VOID | type ) IDENTIFIER formalParameters
 *                  [ THROWS qualifiedIdentifier { COMMA qualifiedIdentifier } ] ( block | SEMI )
 *                | type variableDeclarators SEMI
 * </pre>
 *
 * @param mods the class member modifiers.
 * @return an AST for a member declaration.
 */
private JMember memberDecl(ArrayList<String> mods):
{
   int line = 0;
   Type type = null;
   String name = "";
   ArrayList<JFormalParameter> params = null;
   JBlock body = null;
   ArrayList<JVariableDeclarator> variableDeclarators = null;
   JMember memberDecl = null;
   ArrayList<TypeName> exceptions = null;
   TypeName qualifiedId = null;
}
{
   try {
      LOOKAHEAD(<IDENTIFIER> <LPAREN>)
      (
         <IDENTIFIER>
         {
            line = token.beginLine;
            name = token.image;
         }
         params = formalParameters()
         [
            <THROWS>
            qualifiedId = qualifiedIdentifier()
            {
               exceptions = new ArrayList<TypeName>();
               exceptions.add(qualifiedId);
            }
            (
               qualifiedId = qualifiedIdentifier()
               { exceptions.add(qualifiedId); }
            )*
         ]
         body = block()
         { memberDecl = new JConstructorDeclaration(line, mods, name, params, exceptions, body); }
      ) |
      LOOKAHEAD((<VOID> | type()) <IDENTIFIER> <LPAREN>)
```

```
585         (
586           (
587             <VOID>
588             { type = Type.VOID; } |
589             type = type()
590           )
591           { line = token.beginLine; }
592           <IDENTIFIER>
593           { name = token.image; }
594           params = formalParameters()
595           [
596             <THROWS>
597             qualifiedId = qualifiedIdentifier()
598             {
599               exceptions = new ArrayList<TypeName>();
600               exceptions.add(qualifiedId);
601             }
602             (
603               qualifiedId = qualifiedIdentifier()
604               { exceptions.add(qualifiedId); }
605             )*
606           ]
607           (
608             body = block() |
609             <SEMI>
610           )
611           { memberDecl = new JMethodDeclaration(line, mods, name, type, params, exceptions, body); }
612         ) |
613         (
614           type = type()
615           { line = token.beginLine; }
616           variableDeclarators = variableDeclarators(type)
617           { memberDecl = new JFieldDeclaration(line, mods, variableDeclarators); }
618           <SEMI>
619         )
620       } catch (ParseException e) {
621         recoverFromError(new int[] { SEMI, EOF }, e);
622       }
623       { return memberDecl; }
624   }
625
626   /**
627    * Parses an interface declaration and returns an AST for it.
628    *
629    * <pre>
630    *   interfaceDeclaration ::= INTERFACE IDENTIFIER
631    *                            [ EXTENDS qualifiedIdentifier { COMMA qualifiedIdentifier } ] interfaceBody
632    * </pre>
633    *
```

```
 * @param mods the interface modifiers.
 * @return an AST for an interface declaration.
 */
private JInterfaceDeclaration interfaceDeclaration(ArrayList<String> mods):
{
    int line = 0;
    String name = "";
    ArrayList<JMember> body = null;
    ArrayList<TypeName> superInterfaces = null;
    TypeName qualifiedId = null;
    if (mods == null) {
        mods = new ArrayList<String>();
    }
}
{
    try {
        <INTERFACE>
        { line = token.beginLine; }
        <IDENTIFIER>
        { name = token.image; }
        [
            <EXTENDS>
            qualifiedId = qualifiedIdentifier()
            {
                superInterfaces = new ArrayList<TypeName>();
                superInterfaces.add(qualifiedId);
            }
            (
                qualifiedId = qualifiedIdentifier()
                { superInterfaces.add(qualifiedId); }
            )*

        ]
        body = interfaceBody()
    } catch (ParseException e) {
        recoverFromError(new int[] { SEMI, EOF }, e);
    }
    { return new JInterfaceDeclaration(line, mods, name, superInterfaces, body); }
}

/**
 * Parses an interface body and returns a list of members in the body.
 *
 * <pre>
 *   interfaceBody ::= LCURLY { modifiers interfaceMemberDecl } RCURLY
 * </pre>
 *
 * @return a list of members in the interface body.
 */
```

```
683 private ArrayList<JMember> interfaceBody():
684 {
685     ArrayList<String> mods = null;
686     JMember aMember       = null;
687     ArrayList<JMember> members = new ArrayList<JMember>();
688 }
689 {
690     try {
691         <LCURLY>
692         (
693             mods = modifiers()
694             aMember = interfaceMemberDecl(mods)
695             { members.add(aMember); }
696         )*
697         <RCURLY>
698     } catch (ParseException e) {
699         recoverFromError(new int[] { SEMI, EOF }, e);
700     }
701     { return members; }
702 }
703 
704 /**
705  * Parses an interface member declaration and returns an AST for it.
706  *
707  * <pre>
708  *   interfaceMemberDecl ::= ( VOID | type ) IDENTIFIER formalParameters
709  *                           [ THROWS qualifiedIdentifier { COMMA qualifiedIdentifier } ] SEMI
710  *                         | type variableDeclarators SEMI
711  * </pre>
712  *
713  * @param mods the interface member modifiers.
714  * @return an AST for an interface member declaration.
715  */
716 private JMember interfaceMemberDecl(ArrayList<String> mods):
717 {
718     int line = 0;
719     Type type = null;
720     String name = "";
721     ArrayList<JFormalParameter> params = null;
722     ArrayList<JVariableDeclarator> variableDeclarators = null;
723     JMember iMemberDecl = null;
724     ArrayList<TypeName> exceptions = null;
725     TypeName qualifiedId = null;
726     mods.add("abstract");
727 }
728 {
729     try {
730         LOOKAHEAD((<VOID> | type()) <IDENTIFIER> <LPAREN>)
731         (
```

```
732             (
733                 <VOID>
734                 { type = Type.VOID; } |
735                 type = type()
736             )
737             { line = token.beginLine; }
738             <IDENTIFIER>
739             { name = token.image; }
740             params = formalParameters()
741             [
742                 <THROWS>
743                 qualifiedId = qualifiedIdentifier()
744                 {
745                     exceptions = new ArrayList<TypeName>();
746                     exceptions.add(qualifiedId);
747                 }
748                 (
749                     qualifiedId = qualifiedIdentifier()
750                     { exceptions.add(qualifiedId); }
751                 )*
752             ]
753             <SEMI>
754             { iMemberDecl = new JMethodDeclaration(line, mods, name, type, params, exceptions, null); }
755         ) |
756         (
757             type = type()
758             { line = token.beginLine; }
759             variableDeclarators = variableDeclarators(type)
760             { iMemberDecl = new JFieldDeclaration(line, mods, variableDeclarators); }
761             <SEMI>
762         )
763     } catch (ParseException e) {
764         recoverFromError(new int[] { SEMI, EOF }, e);
765     }
766     { return iMemberDecl; }
767 }
768 
769 /**
770  * Parses a block and returns an AST for it.
771  *
772  * <pre>
773  *   block ::= LCURLY { blockStatement } RCURLY
774  * </pre>
775  *
776  * @return an AST for a block.
777  */
778 private JBlock block():
779 {
780     int line = 0;
```

```
781     JStatement aStatement = null;
782     ArrayList<JStatement> statements = new ArrayList<JStatement>();
783   }
784   {
785     try {
786       <LCURLY>
787       { line = token.beginLine; }
788       (
789         aStatement = blockStatement()
790         { statements.add(aStatement); }
791       )*
792       <RCURLY>
793     } catch (ParseException e) {
794       recoverFromError(new int[] { SEMI, EOF }, e);
795     }
796     { return new JBlock(line, statements); }
797   }
798
799   /**
800    * Parses a block statement and returns an AST for it.
801    *
802    * <pre>
803    *   blockStatement ::= localVariableDeclarationStatement
804    *                    | statement
805    * </pre>
806    *
807    * @return an AST for a block statement.
808    */
809   private JStatement blockStatement():
810   {
811     JStatement statement = null;
812   }
813   {
814     try {
815       LOOKAHEAD(type() <IDENTIFIER>)
816       statement = localVariableDeclarationStatement() |
817       statement = statement()
818     } catch (ParseException e) {
819       recoverFromError(new int[] { SEMI, EOF }, e);
820     }
821     { return statement; }
822   }
823
824   /**
825    * Parses a for-update and returns an array of JStatements.
826    *
827    * <pre>
828    *   forUpdate ::= statementExpression { COMMA statementExpression }
829    * </pre>
```

```
 *
 * @return an array list of JStatements.
 */
private ArrayList<JStatement> forUpdate():
{
   JStatement statementExpression = null;
   ArrayList<JStatement> update = new ArrayList<JStatement>();
}
{
   try {
      statementExpression = statementExpression()
      { update.add(statementExpression); }
      (
         <COMMA>
         statementExpression = statementExpression()
         { update.add(statementExpression); }
      )*
   } catch (ParseException e) {
      recoverFromError(new int[] { SEMI, EOF }, e);
   }
   { return update; }
}

/**
 * Parses a for-init and returns an array of JStatements.
 *
 * <pre>
 *   forInit ::= statementExpression { COMMA statementExpression }
 *                  | type variableDeclarators
 * </pre>
 *
 * @return an array list of JStatements.
 */
private ArrayList<JStatement> forInit():
{
   JStatement stateExp = null;
   ArrayList<JStatement> init = null;
   Type type = null;
   JVariableDeclaration jvd = null;
   ArrayList<JVariableDeclarator> jvdList = null;
   int line = 0;
}
{
   try {
      LOOKAHEAD(statementExpression())
      {
         line = token.beginLine;
         init = new ArrayList<JStatement>();
      }
```

```
879         stateExp = statementExpression()
880         { init.add(stateExp); }
881         (
882             <COMMA>
883             stateExp = statementExpression()
884             { init.add(stateExp); }
885         )* |
886         {
887             line = token.beginLine;
888             init = new ArrayList<JStatement>();
889         }
890         type = type()
891         jvdList = variableDeclarators(type)
892         {
893           jvd = new JVariableDeclaration(line, jvdList);
894           init.add(jvd); }
895     } catch (ParseException e) {
896         recoverFromError(new int[] { SEMI, EOF }, e);
897     }
898     { return init; }
899 }
900
901 /**
902  * Parses a switch label and returns a JExpression.
903  *
904  * <pre>
905  *   switchLabel ::= CASE expression COLON
906  *                 | DEFAULT_RW COLON
907  * </pre>
908  *
909  * @return a JExpression.
910  */
911 private JExpression switchLabel():
912 {
913     JExpression label = null;
914 }
915 {
916     try {
917         <CASE>
918         label = expression()
919         <COLON> |
920         <DEFAULT_RW>
921         <COLON>
922     } catch (ParseException e) {
923         recoverFromError(new int[] { SEMI, EOF }, e);
924     }
925     { return label; }
926 }
927
```

```
/**
 * Parses a switch block statement group and returns a SwitchStatementGroup.
 *
 * <pre>
 *   switchBlockStatementGroup ::= switchLabel { switchLabel } { blockStatement }
 * </pre>
 *
 * @return a SwitchStatementGroup.
 */
private SwitchStatementGroup switchBlockStatementGroup():
{
    JExpression label = null;
    JStatement block = null;
    ArrayList<JExpression> labels = new ArrayList<JExpression>();
    ArrayList<JStatement> blocks = new ArrayList<JStatement>();
    SwitchStatementGroup ssg = null;
}
{
    try {
        label = switchLabel()
        { labels.add(label); }
        (
            label = switchLabel()
            { labels.add(label); }
        )*
        (
            block = blockStatement()
            { blocks.add(block); }
        )*
        { ssg = new SwitchStatementGroup(labels, blocks); }
    } catch (ParseException e) {
        recoverFromError(new int[] { SEMI, EOF }, e);
    }
    { return ssg; }
}

/**
 * Parses a statement and returns an AST for it.
 *
 * <pre>
 *   statement ::= block
 *               | BREAK SEMI
 *               | CONTINUE SEMI
 *               | DO statement WHILE parExpression SEMI
 *               | FOR LPAREN [ forInit ] SEMI [ expression ] SEMI [ forUpdate ] RPAREN statement
 *               | SWITCH parExpression LCURLY { switchBlockStatementGroup } RCURLY
 *               | THROW expression SEMI
 *               | TRY block { CATCH LPAREN formalParameter RPAREN block } [ FINALLY block ]
 *               | IF parExpression statement [ ELSE statement ]
```

```
977  *             | RETURN [ expression ] SEMI
978  *             | SEMI
979  *             | WHILE parExpression statement
980  *             | FOR LPAREN [forInit]
981  *             | statementExpression SEMI
982  * </pre>
983  *
984  * @return an AST for a statement.
985  */
986 private JStatement statement():
987 {
988     int line = 0;
989     JStatement statement  = null;
990     JExpression test      = null;
991     JStatement consequent = null;
992     JStatement alternate  = null;
993     JStatement body       = null;
994     JExpression expr      = null;
995     ArrayList<JStatement> init = null;
996     ArrayList<JStatement> update = null;
997     SwitchStatementGroup ssg = null;
998     ArrayList<SwitchStatementGroup> ssgList = null;
999     JBlock tryBlock = null;
1000    JBlock block = null;
1001    JFormalParameter param = null;
1002    ArrayList<JFormalParameter> parameters = null;
1003    ArrayList<JBlock> catchBlocks = null;
1004    JBlock finallyBlock = null;
1005 }
1006 {
1007    try {
1008        statement = block() |
1009        <BREAK>
1010        { line = token.beginLine; }
1011        <SEMI>
1012        { statement = new JBreakStatement(line); } |
1013        <CONTINUE>
1014        { line = token.beginLine; }
1015        <SEMI>
1016        { statement = new JContinueStatement(line); } |
1017        <THROW>
1018        { line = token.beginLine; }
1019        expr = expression()
1020        <SEMI>
1021        { statement = new JThrowStatement(line, expr); } |
1022        <TRY>
1023        { line = token.beginLine; }
1024        tryBlock = block()
1025        {
```

```
1026            parameters = new ArrayList<JFormalParameter>();
1027            catchBlocks = new ArrayList<JBlock>();
1028        }
1029        (
1030            <CATCH>
1031            <LPAREN>
1032            param = formalParameter()
1033            { parameters.add(param); }
1034            <RPAREN>
1035            block = block()
1036            { catchBlocks.add(block); }
1037        )*
1038        [
1039            <FINALLY>
1040            finallyBlock = block()
1041        ]
1042        { statement = new JTryStatement(line, tryBlock, parameters, catchBlocks, finallyBlock); } |
1043        <SWITCH>
1044        { line = token.beginLine; }
1045        expr = parExpression()
1046        <LCURLY>
1047        { ssgList = new ArrayList<SwitchStatementGroup>(); }
1048        (
1049            ssg = switchBlockStatementGroup()
1050            { ssgList.add(ssg); }
1051        )*
1052        <RCURLY>
1053        { statement = new JSwitchStatement(line, expr, ssgList); } |
1054        <IF>
1055        { line = token.beginLine; }
1056        test = parExpression()
1057        consequent = statement()
1058        // Even without the lookahead below, which is added to suppress JavaCC warnings, dangling
1059        // if-else problem is resolved by binding the alternate to the closest consequent.
1060        [
1061            LOOKAHEAD(<ELSE>)
1062            <ELSE>
1063            alternate = statement()
1064        ]
1065        { statement = new JIfStatement(line, test, consequent, alternate); } |
1066        <RETURN>
1067        { line = token.beginLine; }
1068        [
1069            expr = expression()
1070        ]
1071        <SEMI>
1072        { statement = new JReturnStatement(line, expr); } |
1073        <SEMI>
1074        {
```

```
1075            line = token.beginLine;
1076            statement = new JEmptyStatement( line );
1077        } |
1078        <WHILE>
1079        { line = token.beginLine; }
1080        test = parExpression()
1081        body = statement()
1082        { statement = new JWhileStatement(line, test, body); } |
1083        <DO>
1084        { line = token.beginLine; }
1085        statement = statement()
1086        <WHILE>
1087        expr = parExpression()
1088        <SEMI>
1089        { statement = new JDoStatement(line, statement, expr); } |
1090        <FOR>
1091        { line = token.beginLine; }
1092        <LPAREN>
1093        [
1094            init = forInit()
1095        ]
1096        <SEMI>
1097        [
1098            expr = expression()
1099        ]
1100        <SEMI>
1101        [
1102            update = forUpdate()
1103        ]
1104        <RPAREN>
1105        body = statement()
1106        { statement = new JForStatement(line, init, expr, update, body); } |
1107        // Must be a statementExpression.
1108        statement = statementExpression()
1109        <SEMI>
1110    } catch (ParseException e) {
1111        recoverFromError(new int[] { SEMI, EOF }, e);
1112    }
1113    { return statement; }
1114 }
1115
1116 /**
1117  * Parses and returns a list of formal parameters.
1118  *
1119  * <pre>
1120  *   formalParameters ::= LPAREN [ formalParameter { COMMA  formalParameter } ] RPAREN
1121  * </pre>
1122  *
1123  * @return a list of formal parameters.
```

```
 */
private ArrayList<JFormalParameter> formalParameters():
{
    ArrayList<JFormalParameter> parameters = new ArrayList<JFormalParameter>();
    JFormalParameter aParameter = null;
}
{
    try {
        <LPAREN>
        [
            aParameter = formalParameter()
            { parameters.add(aParameter); }
            (
                <COMMA>
                aParameter = formalParameter()
                { parameters.add(aParameter); }
            )*
        ]
        <RPAREN>
    } catch (ParseException e) {
        recoverFromError(new int[] { SEMI, EOF }, e);
    }
    { return parameters; }
}

/**
 * Parses a formal parameter and returns an AST for it.
 *
 * <pre>
 *   formalParameter ::= type IDENTIFIER
 * </pre>
 *
 * @return an AST for a formal parameter.
 */
private JFormalParameter formalParameter():
{
    int line    = 0;
    Type type   = null;
    String name = "";
}
{
    try {
        type = type()
        { line = token.beginLine; }
        <IDENTIFIER>
        { name = token.image; }
    } catch (ParseException e) {
        recoverFromError(new int[] { SEMI, EOF }, e);
    }
```

```
1173     { return new JFormalParameter(line, name, type); }
1174 }
1175
1176 /**
1177  * Parses a parenthesized expression and returns an AST for it.
1178  *
1179  * <pre>
1180  *   parExpression ::= LPAREN expression RPAREN
1181  * </pre>
1182  *
1183  * @return an AST for a parenthesized expression.
1184  */
1185 private JExpression parExpression():
1186 {
1187     JExpression expr = null;
1188 }
1189 {
1190     try {
1191         <LPAREN>
1192         expr = expression()
1193         <RPAREN>
1194     } catch (ParseException e) {
1195         recoverFromError(new int[] { SEMI, EOF }, e);
1196     }
1197     { return expr; }
1198 }
1199
1200 /**
1201  * Parses a local variable declaration statement and returns an AST for it.
1202  *
1203  * <pre>
1204  *   localVariableDeclarationStatement ::= type variableDeclarators SEMI
1205  * </pre>
1206  *
1207  * @return an AST for a local variable declaration statement.
1208  */
1209 private JVariableDeclaration localVariableDeclarationStatement():
1210 {
1211     int line = 0;
1212     Type type = null;
1213     ArrayList<JVariableDeclarator> vdecls = null;
1214 }
1215 {
1216     try {
1217         type = type()
1218         { line = token.beginLine; }
1219         vdecls = variableDeclarators(type)
1220         <SEMI>
1221     } catch (ParseException e) {
```

```
1222        recoverFromError(new int[] { SEMI, EOF }, e);
1223    }
1224    { return new JVariableDeclaration(line, vdecls); }
1225 }
1226
1227 /**
1228  * Parses and returns a list of variable declarators.
1229  *
1230  * <pre>
1231  *   variableDeclarators ::= variableDeclarator { COMMA variableDeclarator }
1232  * </pre>
1233  *
1234  * @param type type of the variables.
1235  * @return a list of variable declarators.
1236  */
1237 private ArrayList<JVariableDeclarator> variableDeclarators(Type type):
1238 {
1239    JVariableDeclarator aVariableDeclarator = null;
1240    ArrayList<JVariableDeclarator> variableDeclarators = new ArrayList<JVariableDeclarator>();
1241 }
1242 {
1243    try {
1244        aVariableDeclarator = variableDeclarator(type)
1245        { variableDeclarators.add(aVariableDeclarator); }
1246        (
1247            <COMMA>
1248            aVariableDeclarator = variableDeclarator(type)
1249            { variableDeclarators.add(aVariableDeclarator); }
1250        )*
1251    } catch (ParseException e) {
1252        recoverFromError(new int[] { SEMI, EOF }, e);
1253    }
1254    { return variableDeclarators; }
1255 }
1256
1257 /**
1258  * Parses a variable declarator and returns an AST for it.
1259  *
1260  * <pre>
1261  *   variableDeclarator ::= IDENTIFIER [ ASSIGN variableInitializer ]
1262  * </pre>
1263  *
1264  * @param type type of the variable.
1265  * @return an AST for a variable declarator.
1266  */
1267 private JVariableDeclarator variableDeclarator(Type type):
1268 {
1269    int line = 0;
1270    JExpression initial = null;
```

```
1271    String name = "";
1272 }
1273 {
1274    try {
1275       <IDENTIFIER>
1276       {
1277          line = token.beginLine;
1278          name = token.image;
1279       }
1280       [
1281          <ASSIGN>
1282          initial = variableInitializer(type)
1283       ]
1284    } catch (ParseException e) {
1285       recoverFromError(new int[] { SEMI, EOF }, e);
1286    }
1287    { return new JVariableDeclarator(line, name, type, initial); }
1288 }
1289
1290 /**
1291  * Parses a variable initializer and returns an AST for it.
1292  *
1293  * <pre>
1294  *   variableInitializer ::= arrayInitializer | expression
1295  * </pre>
1296  *
1297  * @param type type of the variable.
1298  * @return an AST for a variable initializer.
1299  */
1300 private JExpression variableInitializer(Type type):
1301 {
1302    JExpression initializer = null;
1303 }
1304 {
1305    try {
1306       initializer = arrayInitializer(type) |
1307       initializer = expression()
1308    } catch (ParseException e) {
1309       recoverFromError(new int[] { SEMI, EOF }, e);
1310    }
1311    { return initializer; }
1312 }
1313
1314 /**
1315  * Parses an array initializer and returns an AST for it.
1316  *
1317  * <pre>
1318  *   arrayInitializer ::= LCURLY [variableInitializer {COMMA variableInitializer} [COMMA]] RCURLY
1319  * </pre>
```

```
 *
 * @param type type of the array.
 * @return an AST for an array initializer.
 */
private JArrayInitializer arrayInitializer(Type type):
{
    int line = 0;
    ArrayList<JExpression> initials = new ArrayList<JExpression>();
    JExpression anInitializer = null;
}
{
    try {
        <LCURLY>
        { line = token.beginLine; }
        [
            anInitializer = variableInitializer(type.componentType())
            { initials.add(anInitializer); }
            (
                <COMMA>
                anInitializer = variableInitializer(type.componentType())
                { initials.add(anInitializer); }
            )*
        ]
        <RCURLY>
    } catch (ParseException e) {
        recoverFromError(new int[] { SEMI, EOF }, e);
    }
    { return new JArrayInitializer(line, type, initials); }
}

/**
 * Parses and returns a list of arguments.
 *
 * <pre>
 *   arguments ::= LPAREN [ expression { COMMA expression } ] RPAREN
 * </pre>
 *
 * @return a list of arguments.
 */
private ArrayList<JExpression> arguments():
{
    ArrayList<JExpression> args = new ArrayList<JExpression>();
    JExpression anExpression = null;
}
{
    try {
        <LPAREN>
        [
            anExpression = expression()
```

```
1369            { args.add(anExpression); }
1370            (
1371                <COMMA>
1372                anExpression = expression()
1373                { args.add(anExpression); }
1374            )*
1375        ]
1376        <RPAREN>
1377    } catch (ParseException e) {
1378        recoverFromError(new int[] { SEMI, EOF }, e);
1379    }
1380    { return args; }
1381 }
1382
1383 /**
1384  * Parses and returns a type.
1385  *
1386  * <pre>
1387  *   type ::= referenceType | basicType
1388  * </pre>
1389  *
1390  * @return a type.
1391  */
1392 private Type type():
1393 {
1394    Type type = null;
1395 }
1396 {
1397    try {
1398        LOOKAHEAD(<IDENTIFIER> | basicType() <LBRACK> <RBRACK>)
1399        type = referenceType() |
1400        type = basicType()
1401    } catch (ParseException e) {
1402       recoverFromError(new int[] { SEMI, EOF }, e);
1403    }
1404    { return type; }
1405 }
1406
1407 /**
1408  * Parses and returns a basic type.
1409  *
1410  * <pre>
1411  *   basicType ::= BOOLEAN | CHAR | INT | DOUBLE | LONG
1412  * </pre>
1413  *
1414  * @return a basic type.
1415  */
1416 private Type basicType():
1417 {
```

```
1418     Type type = Type.ANY;
1419 }
1420 {
1421     try {
1422         <BOOLEAN>
1423         { type = Type.BOOLEAN; } |
1424         <CHAR>
1425         { type = Type.CHAR; } |
1426         <INT>
1427         { type = Type.INT; } |
1428         <DOUBLE>
1429         { type = Type.DOUBLE; } |
1430         <LONG>
1431         { type = Type.LONG; }
1432     } catch (ParseException e) {
1433         recoverFromError(new int[] { SEMI, EOF }, e);
1434     }
1435     {
1436         if (type == Type.ANY) {
1437             reportParserError("Type sought where %s found", token.image);
1438         }
1439         return type;
1440     }
1441 }
1442
1443 /**
1444  * Parses and returns a reference type.
1445  *
1446  * <pre>
1447  *   referenceType ::= basicType LBRACK RBRACK { LBRACK RBRACK }
1448  *                   | qualifiedIdentifier { LBRACK RBRACK }
1449  * </pre>
1450  *
1451  * @return a reference type.
1452  */
1453 private Type referenceType():
1454 {
1455     Type type = Type.ANY;
1456 }
1457 {
1458     try {
1459         type = basicType()
1460         <LBRACK> <RBRACK>
1461         { type = new ArrayTypeName(type); }
1462         (
1463             <LBRACK> <RBRACK>
1464             { type = new ArrayTypeName(type); }
1465         )* |
1466         type = qualifiedIdentifier()
```

```
1467        (
1468            <LBRACK> <RBRACK>
1469            { type = new ArrayTypeName(type); }
1470        )*
1471    } catch (ParseException e) {
1472        recoverFromError(new int[] { SEMI, EOF }, e);
1473    }
1474    { return type; }
1475 }
1476 
1477 /**
1478  * Parses a statement expression and returns an AST for it.
1479  *
1480  * <pre>
1481  *   statementExpression ::= expression
1482  * </pre>
1483  *
1484  * @return an AST for a statement expression.
1485  */
1486 private JStatement statementExpression():
1487 {
1488    int line = 0;
1489    JExpression expr = null;
1490 }
1491 {
1492    try {
1493        expr = expression()
1494        {
1495            line = expr.line();
1496            if (expr instanceof JAssignment
1497             || expr instanceof JExpression
1498             || expr instanceof JPreIncrementOp
1499             || expr instanceof JPostDecrementOp
1500             || expr instanceof JMessageExpression
1501             || expr instanceof JSuperConstruction
1502             || expr instanceof JThisConstruction
1503             || expr instanceof JNewOp
1504             || expr instanceof JNewArrayOp) {
1505               // So as not to save on stack.
1506               expr.isStatementExpression = true;
1507            } else {
1508               reportParserError("Invalid statement expression; it does not have a side-effect");
1509            }
1510        }
1511    } catch (ParseException e) {
1512        recoverFromError(new int[] { SEMI, EOF }, e);
1513    }
1514    { return new JStatementExpression( line, expr ); }
1515 }
```

```
1516
1517 /**
1518  * Parses an expression and returns an AST for it.
1519  *
1520  * <pre>
1521  *   expression ::= assignmentExpression
1522  * </pre>
1523  *
1524  * @return an AST for an expression.
1525  */
1526 private JExpression expression():
1527 {
1528     JExpression expr = null;
1529 }
1530 {
1531     try {
1532         expr = assignmentExpression()
1533     } catch (ParseException e) {
1534         recoverFromError(new int[] { SEMI, EOF }, e);
1535     }
1536     { return expr; }
1537 }
1538
1539 /**
1540  * Parses an assignment expression and returns an AST for it.
1541  *
1542  * <pre>
1543  *   assignmentExpression ::= conditionalExpression
1544  *                            [ ( ASSIGN | PLUS_ASSIGN | DIV_ASSIGN | STAR_ASSIGN | REM_ASSIGN
1545  *                              AND_ASSIGN | MINUS_ASSIGN | OR_ASSIGN | XOR_ASSIGN
1546  *                              ALSHIFT_ASSIGN | ARSHIFT_ASSIGN | LRSHIFT_ASSIGN) assignmentExpression ]
1547  * </pre>
1548  *
1549  * @return an AST for an assignment expression.
1550  */
1551 private JExpression assignmentExpression():
1552 {
1553     int line = 0;
1554     JExpression lhs = null, rhs = null;
1555 }
1556 {
1557     try {
1558         lhs = conditionalExpression()
1559         { line = lhs.line(); }
1560         [
1561             <ASSIGN>
1562             rhs = assignmentExpression()
1563             { lhs = new JAssignOp(line, lhs, rhs); } |
1564             <PLUS_ASSIGN>
```

```
1565            rhs = assignmentExpression()
1566            { lhs = new JPlusAssignOp(line, lhs, rhs); } |
1567            <DIV_ASSIGN>
1568            rhs = assignmentExpression()
1569            { lhs = new JDivAssignOp(line, lhs, rhs); } |
1570            <STAR_ASSIGN>
1571            rhs = assignmentExpression()
1572            { lhs = new JStarAssignOp(line, lhs, rhs); } |
1573            <REM_ASSIGN>
1574            rhs = assignmentExpression()
1575            { lhs = new JRemAssignOp(line, lhs, rhs); } |
1576            <AND_ASSIGN>
1577            rhs = assignmentExpression()
1578            { lhs = new JAndAssignOp(line, lhs, rhs); } |
1579            <MINUS_ASSIGN>
1580            rhs = assignmentExpression()
1581            { lhs = new JMinusAssignOp(line, lhs, rhs); } |
1582            <OR_ASSIGN>
1583            rhs = assignmentExpression()
1584            { lhs = new JOrAssignOp(line, lhs, rhs); } |
1585            <XOR_ASSIGN>
1586            rhs = assignmentExpression()
1587            { lhs = new JXorAssignOp(line, lhs, rhs); } |
1588            <ALSHIFT_ASSIGN>
1589            rhs = assignmentExpression()
1590            { lhs = new JALeftShiftAssignOp(line, lhs, rhs); } |
1591            <ARSHIFT_ASSIGN>
1592            rhs = assignmentExpression()
1593            { lhs = new JARightShiftAssignOp(line, lhs, rhs); } |
1594            <LRSHIFT_ASSIGN>
1595            rhs = assignmentExpression()
1596            { lhs = new JLRightShiftAssignOp(line, lhs, rhs); }
1597        ]
1598    } catch (ParseException e) {
1599        recoverFromError(new int[] { SEMI, EOF }, e);
1600    }
1601    { return lhs; }
1602 }
1603
1604 /**
1605  * Parses a conditional expression and returns an AST for it.
1606  *
1607  * <pre>
1608  *   conditionalExpression ::= conditionalOrExpression [ QUESTION expression COLON
      conditionalExpression ]
1609  * </pre>
1610  *
1611  * @return an AST for a conditional expression.
1612  */
```

```
1613 private JExpression conditionalExpression():
1614 {
1615     int line = 0;
1616     JExpression lhs = null, thenPart = null, elsePart = null;
1617 }
1618 {
1619     try {
1620         lhs = conditionalOrExpression()
1621         { line = lhs.line(); }
1622         [
1623             <QUESTION>
1624             thenPart = expression()
1625             <COLON>
1626             elsePart = conditionalExpression()
1627             { lhs = new JConditionalExpression(line, lhs, thenPart, elsePart); }
1628         ]
1629     } catch (ParseException e) {
1630         recoverFromError(new int[] { SEMI, EOF }, e);
1631     }
1632     { return lhs; }
1633 }
1634
1635 /**
1636  * Parses a conditional-or expression and returns an AST for it.
1637  *
1638  * <pre>
1639  *   conditionalOrExpression ::= conditionalAndExpression { LOR conditionalAndExpression }
1640  * </pre>
1641  *
1642  * @return an AST for a conditional-or expression.
1643  */
1644 private JExpression conditionalOrExpression():
1645 {
1646     int line = 0;
1647     JExpression lhs = null, rhs = null;
1648 }
1649 {
1650     try {
1651         lhs = conditionalAndExpression()
1652         { line = lhs.line(); }
1653         (
1654             <LOR>
1655             rhs = conditionalAndExpression()
1656             { lhs = new JLogicalOrOp(line, lhs, rhs); }
1657         )*
1658     } catch (ParseException e) {
1659         recoverFromError(new int[] { SEMI, EOF }, e);
1660     }
1661     { return lhs; }
```

```
}

/**
 * Parses a conditional-and expression and returns an AST for it.
 *
 * <pre>
 *   conditionalAndExpression ::= inclusiveOrExpression { LAND inclusiveOrExpression }
 * </pre>
 *
 * @return an AST for a conditional-and expression.
 */
private JExpression conditionalAndExpression():
{
    int line = 0;
    JExpression lhs = null, rhs = null;
}
{
    try {
        lhs = inclusiveOrExpression()
        { line = lhs.line(); }
        (
            <LAND>
            rhs = inclusiveOrExpression()
            { lhs = new JLogicalAndOp(line, lhs, rhs); }
        )*
    } catch (ParseException e) {
        recoverFromError(new int[] { SEMI, EOF }, e);
    }
    { return lhs; }
}

/**
 * Parses an inclusive-or expression and returns an AST for it.
 *
 * <pre>
 *   inclusiveOrExpression ::= exclusiveOrExpression { OR exclusiveOrExpression }
 * </pre>
 *
 * @return an AST for a exclusive-or expression.
 */
private JExpression inclusiveOrExpression():
{
    int line = 0;
    JExpression lhs = null, rhs = null;
}
{
    try {
        lhs = exclusiveOrExpression()
        { line = lhs.line(); }
```

```
1711        (
1712            <OR>
1713            rhs = exclusiveOrExpression()
1714            { lhs = new JOrOp(line, lhs, rhs); }
1715        )*
1716    } catch (ParseException e) {
1717        recoverFromError(new int[] { SEMI, EOF }, e);
1718    }
1719    { return lhs; }
1720 }
1721 
1722 /**
1723  * Parses an exclusive-or expression and returns an AST for it.
1724  *
1725  * <pre>
1726  *   exclusiveOrExpression ::= andExpression { XOR andExpression }
1727  * </pre>
1728  *
1729  * @return an AST for a exclusive-or expression.
1730  */
1731 private JExpression exclusiveOrExpression():
1732 {
1733     int line = 0;
1734     JExpression lhs = null, rhs = null;
1735 }
1736 {
1737     try {
1738         lhs = andExpression()
1739         { line = lhs.line(); }
1740         (
1741             <XOR>
1742             rhs = andExpression()
1743             { lhs = new JXorOp(line, lhs, rhs); }
1744         )*
1745     } catch (ParseException e) {
1746         recoverFromError(new int[] { SEMI, EOF }, e);
1747     }
1748     { return lhs; }
1749 }
1750 
1751 /**
1752  * Parses an and expression and returns an AST for it.
1753  *
1754  * <pre>
1755  *   andExpression ::= equalityExpression { AND equalityExpression }
1756  * </pre>
1757  *
1758  * @return an AST for a and expression.
1759  */
```

```
1760 private JExpression andExpression():
1761 {
1762     int line = 0;
1763     JExpression lhs = null, rhs = null;
1764 }
1765 {
1766     try {
1767         lhs = equalityExpression()
1768         { line = lhs.line(); }
1769         (
1770             <AND>
1771             rhs = equalityExpression()
1772             { lhs = new JAndOp(line, lhs, rhs); }
1773         )*
1774     } catch (ParseException e) {
1775         recoverFromError(new int[] { SEMI, EOF }, e);
1776     }
1777     { return lhs; }
1778 }
1779
1780
1781 /**
1782  * Parses an equality expression and returns an AST for it.
1783  *
1784  * <pre>
1785  *   equalityExpression ::= relationalExpression { ( EQUAL | NOT EQUAL ) relationalExpression }
1786  * </pre>
1787  *
1788  * @return an AST for an equality expression.
1789  */
1790 private JExpression equalityExpression():
1791 {
1792     int line = 0;
1793     JExpression lhs = null, rhs = null;
1794 }
1795 {
1796     try {
1797         lhs = relationalExpression()
1798         { line = lhs.line(); }
1799         (
1800             <EQUAL>
1801             rhs = relationalExpression()
1802             { lhs = new JEqualOp(line, lhs, rhs); } |
1803             <NOT_EQUAL>
1804             rhs = relationalExpression()
1805             { lhs = new JNotEqualOp(line, lhs, rhs); }
1806         )*
1807     } catch (ParseException e) {
1808         recoverFromError(new int[] { SEMI, EOF }, e);
```

```
1809     }
1810     { return lhs; }
1811 }
1812 
1813 /**
1814  * Parses a relational expression and returns an AST for it.
1815  *
1816  * <pre>
1817  *   relationalExpression ::= shiftExpression [ ( GT | LE | LT | GE ) shiftExpression
1818  *                                      | INSTANCEOF referenceType ]
1819  * </pre>
1820  *
1821  * @return an AST for a relational expression.
1822  */
1823 private JExpression relationalExpression():
1824 {
1825     int line = 0;
1826     JExpression lhs = null, rhs = null;
1827     Type type = null;
1828 }
1829 {
1830     try {
1831         lhs = shiftExpression() { line = lhs.line(); }
1832         [
1833             <GT>
1834             rhs = shiftExpression()
1835             { lhs = new JGreaterThanOp(line, lhs, rhs); } |
1836             <LE>
1837             rhs = shiftExpression()
1838             { lhs = new JLessEqualOp(line, lhs, rhs); } |
1839             <LT>
1840             rhs = shiftExpression()
1841             { lhs = new JLessThanOp(line, lhs, rhs); } |
1842             <GE>
1843             rhs = shiftExpression()
1844             { lhs = new JGreaterEqualOp(line, lhs, rhs); } |
1845             <INSTANCEOF>
1846             type = referenceType()
1847             { lhs = new JInstanceOfOp(line, lhs, type); }
1848         ]
1849     } catch (ParseException e) {
1850         recoverFromError(new int[] { SEMI, EOF }, e);
1851     }
1852     { return lhs; }
1853 }
1854 
1855 /**
1856  * Parses a shift expression and returns an AST for it.
1857  *
```

```
1858  * <pre>
1859  *   shiftExpression ::= additiveExpression { (ALSHIFT | ARSHIFT | LRSHIFT) additiveExpression }
1860  * </pre>
1861  *
1862  * @return an AST for a shift expression.
1863  */
1864
1865 private JExpression shiftExpression():
1866 {
1867     int line = 0;
1868     JExpression lhs = null, rhs = null;
1869     Type type = null;
1870 }
1871 {
1872     try {
1873         lhs = additiveExpression()
1874         { line = lhs.line(); }
1875         (
1876             <ALSHIFT>
1877             rhs = additiveExpression()
1878             { lhs = new JALeftShiftOp(line, lhs, rhs); } |
1879             <ARSHIFT>
1880             rhs = additiveExpression()
1881             { lhs = new JARightShiftOp(line, lhs, rhs); } |
1882             <LRSHIFT>
1883             rhs = additiveExpression()
1884             { lhs = new JLRightShiftOp(line, lhs, rhs); }
1885         )*
1886     } catch (ParseException e) {
1887         recoverFromError(new int[] { SEMI, EOF }, e);
1888     }
1889     { return lhs; }
1890 }
1891
1892 /**
1893  * Parses an additive expression and returns an AST for it.
1894  *
1895  * <pre>
1896  *   additiveExpression ::= multiplicativeExpression { ( MINUS | PLUS ) multiplicativeExpression }
1897  * </pre>
1898  *
1899  * @return an AST for an additive expression.
1900  */
1901 private JExpression additiveExpression():
1902 {
1903     int line = 0;
1904     JExpression lhs = null, rhs = null;
1905 }
1906 {
```

```
1907     try {
1908       lhs = multiplicativeExpression()
1909       { line = lhs.line(); }
1910       (
1911         <MINUS>
1912         rhs = multiplicativeExpression()
1913         { lhs = new JSubtractOp(line, lhs, rhs); } |
1914         <PLUS>
1915         rhs = multiplicativeExpression()
1916         { lhs = new JPlusOp(line, lhs, rhs); }
1917       )*
1918     } catch (ParseException e) {
1919       recoverFromError(new int[] { SEMI, EOF }, e);
1920     }
1921     { return lhs; }
1922 }
1923 
1924 /**
1925  * Parses a multiplicative expression and returns an AST for it.
1926  *
1927  * <pre>
1928  *   multiplicativeExpression ::= unaryExpression { ( STAR | DIV | REM ) unaryExpression }
1929  * </pre>
1930  *
1931  * @return an AST for a multiplicative expression.
1932  */
1933 private JExpression multiplicativeExpression():
1934 {
1935     int line = 0;
1936     JExpression lhs = null, rhs = null;
1937 }
1938 {
1939     try {
1940       lhs = unaryExpression()
1941       { line = lhs.line(); }
1942       (
1943         <STAR>
1944         rhs = unaryExpression()
1945         { lhs = new JMultiplyOp(line, lhs, rhs); } |
1946         <DIV>
1947         rhs = unaryExpression()
1948         { lhs = new JDivideOp(line, lhs, rhs); } |
1949         <REM>
1950         rhs = unaryExpression()
1951         { lhs = new JRemainderOp(line, lhs, rhs); }
1952       )*
1953     } catch (ParseException e) {
1954       recoverFromError(new int[] { SEMI, EOF }, e);
1955     }
```

```
    { return lhs; }
}

/**
 * Parses an unary expression and returns an AST for it.
 *
 * <pre>
 *   unaryExpression ::= INC unaryExpression
 *                     | DEC unaryExpression
 *                     | ( MINUS | PLUS ) unaryExpression
 *                     | simpleUnaryExpression
 * </pre>
 *
 * @return an AST for an unary expression.
 */
private JExpression unaryExpression():
{
    int line = 0;
    JExpression expr = null, unaryExpr = null;
}
{
    try {
        <INC>
        { line = token.beginLine; }
        unaryExpr = unaryExpression()
        { expr = new JPreIncrementOp(line, unaryExpr); } |
        <DEC>
        { line = token.beginLine; }
        unaryExpr = unaryExpression()
        { expr = new JPreDecrementOp(line, unaryExpr); } |
        <MINUS>
        { line = token.beginLine; }
        unaryExpr = unaryExpression()
        { expr = new JNegateOp(line, unaryExpr); } |
        <PLUS>
        { line = token.beginLine; }
        unaryExpr = unaryExpression()
        { expr = new JUnaryPlusOp(line, unaryExpr); } |
        expr = simpleUnaryExpression()
    } catch (ParseException e) {
        recoverFromError(new int[] { SEMI, EOF }, e);
    }
    { return expr; }
}

/**
 * Parses a simple unary expression and returns an AST for it.
 *
 * <pre>
```

```
2005  *   simpleUnaryExpression ::= LNOT unaryExpression
2006  *                           | NOT unaryExpression
2007  *                           | LPAREN basicType RPAREN unaryExpression
2008  *                           | LPAREN referenceType RPAREN simpleUnaryExpression
2009  *                           | postfixExpression
2010  * </pre>
2011  *
2012  * @return an AST for a simple unary expression.
2013  */
2014 private JExpression simpleUnaryExpression():
2015 {
2016    int line = 0;
2017    Type type = null;
2018    JExpression expr = null, unaryExpr = null, simpleUnaryExpr = null;
2019 }
2020 {
2021    try {
2022       <LNOT>
2023       { line = token.beginLine; }
2024       unaryExpr = unaryExpression()
2025       { expr = new JLogicalNotOp(line, unaryExpr); } |
2026       <NOT>
2027       { line = token.beginLine; }
2028       unaryExpr = unaryExpression()
2029       { expr = new JComplementOp(line, unaryExpr); } |
2030       LOOKAHEAD(<LPAREN> basicType() <RPAREN>)
2031       <LPAREN>
2032       { line = token.beginLine; }
2033       type = basicType()
2034       <RPAREN>
2035       unaryExpr = unaryExpression()
2036       { expr = new JCastOp(line, type, unaryExpr); } |
2037       LOOKAHEAD(<LPAREN> referenceType() <RPAREN>)
2038       <LPAREN>
2039       { line = token.beginLine; }
2040       type = referenceType()
2041       <RPAREN>
2042       simpleUnaryExpr = simpleUnaryExpression()
2043       { expr = new JCastOp(line, type, simpleUnaryExpr); } |
2044       expr = postfixExpression()
2045    } catch (ParseException e) {
2046       recoverFromError(new int[] { SEMI, EOF }, e);
2047    }
2048    { return expr ; }
2049 }
2050
2051 /**
2052  * Parses a postfix expression and returns an AST for it.
2053  *
```

```
2054  * <pre>
2055  *   postfixExpression ::= primary { selector } { DEC | INC }
2056  * </pre>
2057  *
2058  * @return an AST for a postfix expression.
2059  */
2060 private JExpression postfixExpression():
2061 {
2062     int line = 0;
2063     JExpression primaryExpr = null;
2064 }
2065 {
2066     try {
2067         primaryExpr = primary()
2068         { line = primaryExpr.line(); }
2069         (
2070             primaryExpr = selector(primaryExpr)
2071         )*
2072         (
2073             <DEC>
2074             { primaryExpr = new JPostDecrementOp(line, primaryExpr); } |
2075             <INC>
2076             { primaryExpr = new JPostIncrementOp(line, primaryExpr); }
2077         )*
2078     } catch (ParseException e) {
2079         recoverFromError(new int[] { SEMI, EOF }, e);
2080     }
2081     { return primaryExpr; }
2082 }
2083
2084 /**
2085  * Parses a selector and returns an AST for it.
2086  *
2087  * <pre>
2088  *   selector ::= DOT qualifiedIdentifier [ arguments ]
2089  *              | LBRACK expression RBRACK
2090  * </pre>
2091  *
2092  * @param target the target expression for this selector.
2093  * @return an AST for a selector.
2094  */
2095 private JExpression selector(JExpression target):
2096 {
2097     int line = 0;
2098     ArrayList<JExpression> args = null;
2099     TypeName id = null;
2100     JExpression expr = null;
2101 }
2102 {
```

```
2103     try {
2104       <DOT>
2105       { line = token.beginLine; }
2106       id = qualifiedIdentifier()
2107       { expr = new JFieldSelection(line, ambiguousPart(id), target, id.simpleName()); }
2108       [
2109         args = arguments()
2110         { expr = new JMessageExpression(line, target, ambiguousPart(id), id.simpleName(),
2111                                         args); }
2112       ] |
2113       <LBRACK>
2114       { line = token.beginLine; }
2115       { expr = new JArrayExpression(line, target, expression()); }
2116       <RBRACK>
2117     } catch (ParseException e) {
2118       recoverFromError(new int[] { SEMI, EOF }, e);
2119     }
2120     { return expr; }
2121 }
2122
2123 /**
2124  * Parses a primary expression and returns an AST for it.
2125  *
2126  * <pre>
2127  *   primary ::= parExpression
2128  *             | NEW creator
2129  *             | THIS [ arguments ]
2130  *             | SUPER ( arguments | DOT IDENTIFIER [ arguments ] )
2131  *             | qualifiedIdentifier [ arguments ]
2132  *             | literal
2133  * </pre>
2134  *
2135  * @return an AST for a primary expression.
2136  */
2137 private JExpression primary():
2138 {
2139     int line = 0;
2140     String name = "";
2141     JExpression expr = null;
2142     JExpression newTarget = null;
2143     ArrayList<JExpression> args = null;
2144     TypeName id = null;
2145 }
2146 {
2147     try {
2148       expr = parExpression() |
2149       <NEW>
2150       expr = creator() |
2151       <THIS>
```

```
2152        {
2153          line = token.beginLine;
2154          expr = new JThis(line);
2155        }
2156        [
2157          args = arguments()
2158          { expr = new JThisConstruction(line, args); }
2159        ] |
2160        <SUPER>
2161        { line = token.beginLine; }
2162        (
2163          args = arguments()
2164          { expr = new JSuperConstruction(line, args); } |
2165          <DOT> <IDENTIFIER>
2166          {
2167            name = token.image;
2168            newTarget = new JSuper(line);
2169            expr = new JFieldSelection(line, newTarget, name);
2170          }
2171          [
2172            args = arguments()
2173            { expr = new JMessageExpression(line, newTarget, null, name, args); }
2174          ]
2175        ) |
2176        // Language is ambiguous here. JavaCC is unable to choose between qualifiedIdentifier and
2177        // selector. Semantic analysis will sort it out.
2178        id = qualifiedIdentifier()
2179        {
2180          line = id.line();
2181          if (ambiguousPart(id) == null) {
2182            expr = new JVariable(line, id.simpleName());
2183          } else {
2184            expr = new JFieldSelection(line, ambiguousPart(id), null, id.simpleName());
2185          }
2186        }
2187        [
2188          args = arguments()
2189          { expr = new JMessageExpression(line, null, ambiguousPart(id), id.simpleName(), args); }
2190        ] |
2191        expr = literal()
2192      } catch (ParseException e) {
2193        recoverFromError(new int[] { SEMI, EOF }, e);
2194      }
2195      { return expr; }
2196  }
2197
2198  /**
2199   * Parses a creator and returns an AST for it.
2200   *
```

```
2201  * <pre>
2202  *   creator ::= ( basicType | qualifiedIdentifier )
2203  *                 ( arguments
2204  *                 | LBRACK RBRACK { LBRACK RBRACK } [ arrayInitializer ]
2205  *                 | newArrayDeclarator
2206  *                 )
2207  * </pre>
2208  *
2209  * @return an AST for a creator.
2210  */
2211 private JExpression creator():
2212 {
2213    int line = 0;
2214    Type type = null;
2215    ArrayList<JExpression> args = null;
2216    ArrayList<JExpression> dims = null;
2217    JArrayInitializer init =  null;
2218    JExpression expr = null;
2219    Type expected = null;
2220 }
2221 {
2222    try {
2223       (
2224          type = basicType() |
2225          type = qualifiedIdentifier()
2226       )
2227       {
2228          line = token.beginLine;
2229          expected = type;
2230       }
2231       (
2232          args = arguments()
2233          { expr = new JNewOp(line, type, args); } |
2234          LOOKAHEAD(<LBRACK> <RBRACK>)
2235          <LBRACK> <RBRACK>
2236          { expected = new ArrayTypeName(expected); }
2237          (
2238             LOOKAHEAD(<LBRACK> <RBRACK>)
2239             <LBRACK> <RBRACK>
2240             { expected = new ArrayTypeName(expected); }
2241          )*
2242          [
2243            expr = arrayInitializer(expected)
2244          ] |
2245          expr = newArrayDeclarator(type)
2246       )
2247    } catch (ParseException e) {
2248       expr = new JWildExpression(token.beginLine);
2249       recoverFromError(new int[] { SEMI, EOF }, e);
```

```
2250     }
2251     { return expr; }
2252 }
2253
2254 /**
2255  * Parses a new array declarator and returns an AST for it.
2256  *
2257  * <pre>
2258  *   newArrayDeclarator ::= LBRACK expression RBRACK { LBRACK expression RBRACK } { LBRACK
     RBRACK }
2259  * </pre>
2260  *
2261  * @param line line in which the declarator occurred.
2262  * @param type type of the array.
2263  * @return an AST for a new array declarator.
2264  */
2265 private JNewArrayOp newArrayDeclarator(Type type):
2266 {
2267     int line = 0;
2268     ArrayList<JExpression> dimensions = new ArrayList<JExpression>();
2269     JExpression expr = null;
2270 }
2271 {
2272     try {
2273         <LBRACK>
2274         { line = token.beginLine; }
2275         expr = expression()
2276         {
2277             dimensions.add(expr);
2278             type = new ArrayTypeName(type);
2279         }
2280         <RBRACK>
2281         (
2282             LOOKAHEAD(<LBRACK> expression() <RBRACK>)
2283             <LBRACK>
2284             expr = expression()
2285             {
2286                 dimensions.add(expr);
2287                 type = new ArrayTypeName(type);
2288             }
2289             <RBRACK>
2290         )*
2291         (
2292             LOOKAHEAD(<LBRACK> <RBRACK>)
2293             <LBRACK> <RBRACK>
2294             { type = new ArrayTypeName(type); }
2295         )*
2296     } catch (ParseException e) {
2297         recoverFromError(new int[] { SEMI, EOF }, e);
```

```
2298     }
2299     { return new JNewArrayOp(line, type, dimensions); }
2300 }
2301 
2302 /**
2303  * Parses a literal and returns an AST for it.
2304  *
2305  * <pre>
2306  *   literal ::= CHAR_LITERAL | FALSE | INT_LITERAL | DOUBLE_LITERAL | LONG_LITERAL | NULL |
      STRING_LITERAL | TRUE
2307  * </pre>
2308  *
2309  * @return an AST for a literal.
2310  */
2311 private JExpression literal():
2312 {
2313     JExpression expr = null;
2314 }
2315 {
2316     try {
2317         <CHAR_LITERAL>
2318         { expr = new JLiteralChar(token.beginLine, token.image); } |
2319         <FALSE>
2320         { expr = new JLiteralBoolean(token.beginLine, token.image); } |
2321         <INT_LITERAL>
2322         { expr = new JLiteralInt(token.beginLine, token.image); } |
2323         <DOUBLE_LITERAL>
2324         { expr = new JLiteralDouble(token.beginLine, token.image); } |
2325         <LONG_LITERAL>
2326         { expr = new JLiteralLong(token.beginLine, token.image); } |
2327         <NULL>
2328         { expr = new JLiteralNull(token.beginLine); } |
2329         <STRING_LITERAL>
2330         { expr = new JLiteralString(token.beginLine, token.image); } |
2331         <TRUE>
2332         { expr = new JLiteralBoolean(token.beginLine, token.image); }
2333     } catch (ParseException e) {
2334         expr = new JWildExpression(token.beginLine);
2335         recoverFromError(new int[] { SEMI, EOF }, e);
2336     }
2337     { return expr; }
2338 }
2339 
```