# Project 3 (Parsing)

● Graded

**Student**

Giancarlos Marte

**Total Points**

**38.1 / 100 pts**

**Autograder Score**

**25.1 / 80.0**

**Failed Tests**

Problem 3: Conditional Expression (0/10)
Problem 5: For Statement (0/10)
Problem 6: Break Statement (0/10)
Problem 7: Continue Statement (0/10)
Problem 8: Switch Statement (0/10)
Problem 9: Exception Handlers (0/10)
Problem 10: Interface Type Declaration (0/10)

**Passed Tests**

Problem 0. Compiling j-- (2/2)
Problem 1: Long and Double Basic Types (10/10)
Problem 2: Operators (10/10)
Problem 4: Do Statement (10/10)

**Question 2**

## Code Clarity and Efficiency

**3** / 10 pts

### Long and Double Basic Types

- ✔ **+ 0.5 pts** Passed all tests
- ✔ **+ 0.25 pts** Changes to parser commented adequately
- ✔ **+ 0.25 pts** Followed good programming practices

### Operators

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

### Conditional Expression

- **+ 0.25 pts** Passed all tests
- **+ 0.25 pts** Precedence captured correctly
- **+ 0.25 pts** Changes to parser commented adequately
- **+ 0.25 pts** Followed good programming practices

### Do Statement

- ✔ **+ 0.5 pts** Passed all tests
- ✔ **+ 0.25 pts** Changes to parser commented adequately
- ✔ **+ 0.25 pts** Followed good programming practices

### For Statement

- **+ 0.5 pts** Passed all tests
- **+ 0.25 pts** Changes to parser commented adequately
- **+ 0.25 pts** Followed good programming practices

### Break Statement

- **+ 0.5 pts** Passed all tests
- **+ 0.25 pts** Changes to parser commented adequately

+ **0.25 pts** Followed good programming practices

---

Continue Statement

+ **0.5 pts** Passed all tests

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

+ **0.25 pts** Followed good programming practices

---

Switch Statement

+ **0.5 pts** Passed all tests

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

+ **0.25 pts** Followed good programming practices

---

Exception handlers

+ **0.5 pts** Passed all tests

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

+ **0.25 pts** Followed good programming practices

---

Interface Type Declaration

+ **0.5 pts** Passed all tests

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

+ **0.25 pts** Followed good programming practices

---

+ **0 pts** Do 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: Long and Double Basic Types (10/10)

j-- -p tests/BasicTypes.java

### Problem 2: Operators (10/10)

j-- -p tests/Operators.java

### Problem 3: Conditional Expression (0/10)

j-- -p tests/ConditionalExpression.java □
'tests/ConditionalExpression.java:7: error[93 chars]ound' != ''
- tests/ConditionalExpression.java:7: error: ? found where ; sought
- tests/ConditionalExpression.java:7: error: Literal sought where ? found

### Problem 4: Do Statement (10/10)

j-- -p tests/DoStatement.java

### Problem 5: For Statement (0/10)

j-- -p tests/ForStatement.java □
'tests/ForStatement.java:8: error: Literal[115 chars]fect' != ''
- tests/ForStatement.java:8: error: Literal sought where int found
- tests/ForStatement.java:8: error: Invalid statement expression; it does not have a side-effect

### Problem 6: Break Statement (0/10)

j-- -p tests/BreakStatement.java □
'tests/BreakStatement.java:8: error: Liter[477 chars]ound' != ''
- tests/BreakStatement.java:8: error: Literal sought where int found
- tests/BreakStatement.java:8: error: Invalid statement expression; it does not have a side-effect
- tests/BreakStatement.java:8: error: Literal sought where ; found
- tests/BreakStatement.java:8: error: Literal sought where ) found
- tests/BreakStatement.java:8: error: Invalid statement expression; it does not have a side-effect
- tests/BreakStatement.java:19: error: ? found where ; sought
- tests/BreakStatement.java:19: error: Literal sought where ? found

### Problem 7: Continue Statement (0/10)

j-- -p tests/ContinueStatement.java □
'tests/ContinueStatement.java:10: error: ;[2200 chars]109)' != ''
Diff is 2273 characters long. Set self.maxDiff to None to see it.

### Problem 8: Switch Statement (0/10)

j-- -p tests/SwitchStatement.java 
'tests/SwitchStatement.java:11: error: Lit[2471 chars]109)' != ''
Diff is 2546 characters long. Set self.maxDiff to None to see it.

### Problem 9: Exception Handlers (0/10)

j-- -p tests/ExceptionHandlers.java 
'tests/ExceptionHandlers.java:3: error: .E[1278 chars]109)' != ''
Diff is 1337 characters long. Set self.maxDiff to None to see it.

### Problem 10: Interface Type Declaration (0/10)

j-- -p tests/Interface.java 
'tests/Interface.java:4: error: interface [3011 chars]109)' != ''
Diff is 3101 characters long. Set self.maxDiff to None to see it.

**Submitted Files**

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

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

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

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

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

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

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

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

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

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

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

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         JStatement statementExpression = statementExpression();
331         ArrayList<JStatement> update = new ArrayList<>();
332         update.add(statementExpression);
333         while(have(COMMA)) {
334             update.add(statementExpression());
335         }
336         return update;
337     }
338 
339     /**
340      * Parses forInit and returns a list of JStatements.
```

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

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

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

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

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

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

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

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

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

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

```
828         JExpression lhs = conditionalAndExpression();
829         if (have(ASSIGN)) {
830             return new JAssignOp(line, lhs, assignmentExpression());
831         } else if (have(PLUS_ASSIGN)) {
832             return new JPlusAssignOp(line, lhs, assignmentExpression());
833         } else if (have(MINUS_ASSIGN)) {
834             return new JMinusAssignOp(line, lhs, assignmentExpression());
835         } else if (have(STAR_ASSIGN)) {
836             return new JStarAssignOp(line, lhs, assignmentExpression());
837         } else if (have(DIV_ASSIGN)) {
838             return new JDivAssignOp(line, lhs, assignmentExpression());
839         } else if (have(REM_ASSIGN)) {
840             return new JRemAssignOp(line, lhs, assignmentExpression());
841         } else if (have(LRSHIFT_ASSIGN)) {
842             return new JLRightShiftAssignOp(line, lhs, assignmentExpression());
843         } else if (have(ALSHIFT_ASSIGN)) {
844             return new JALeftShiftAssignOp(line, lhs, assignmentExpression());
845         } else if (have(ARIGHTSHIFT_ASSIGN)) {
846             return new JARightShiftAssignOp(line, lhs, assignmentExpression());
847         } else if (have (AND_ASSIGN)) {
848             return new JAndAssignOp(line, lhs, assignmentExpression());
849         } else if (have(XOR_ASSIGN)) {
850             return new JXorAssignOp(line, lhs, assignmentExpression());
851         } else if (have(OR_ASSIGN)) {
852             return new JOrAssignOp(line, lhs, assignmentExpression());
853         }
854         else {
855             return lhs;
856         }
857     }
858 
859     /**
860      * Parses a conditional-and expression and returns an AST for it.
861      *
862      * <pre>
863      *   conditionalAndExpression ::= equalityExpression { LAND equalityExpression }
864      * </pre>
865      *
866      * @return an AST for a conditional-and expression.
867      */
868     private JExpression conditionalAndExpression() {
869         int line = scanner.token().line();
870         boolean more = true;
871         JExpression lhs = equalityExpression();
872         while (more) {
873             if (have(LAND)) {
874                 lhs = new JLogicalAndOp(line, lhs, equalityExpression());
875             } else if (have(LOR)) {
876                 lhs = new JLogicalOrOp(line, lhs, equalityExpression());
```

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

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

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

```
1024      */
1025     private JExpression unaryExpression() {
1026         int line = scanner.token().line();
1027         if (have(INC)) {
1028             return new JPreIncrementOp(line, unaryExpression());
1029         } else if (have (DEC)) {
1030             return new JPreDecrementOp(line, unaryExpression());
1031         }
1032         else if (have(MINUS)) {
1033             return new JNegateOp(line, unaryExpression());
1034         }
1035         else if (have(PLUS)) {
1036             return new JUnaryPlusOp(line, unaryExpression());
1037         }
1038         else if (have(NOT)) {
1039             return new JComplementOp(line, unaryExpression());
1040         }
1041         else {
1042             return simpleUnaryExpression();
1043         }
1044     }
1045 
1046     /**
1047      * Parses a simple unary expression and returns an AST for it.
1048      *
1049      * <pre>
1050      *   simpleUnaryExpression ::= LNOT unaryExpression
1051      *                           | LPAREN basicType RPAREN unaryExpression
1052      *                           | LPAREN referenceType RPAREN simpleUnaryExpression
1053      *                           | postfixExpression
1054      * </pre>
1055      *
1056      * @return an AST for a simple unary expression.
1057      */
1058     private JExpression simpleUnaryExpression() {
1059         int line = scanner.token().line();
1060         if (have(LNOT)) {
1061             return new JLogicalNotOp(line, unaryExpression());
1062         } else if (seeCast()) {
1063             mustBe(LPAREN);
1064             boolean isBasicType = seeBasicType();
1065             Type type = type();
1066             mustBe(RPAREN);
1067             JExpression expr = isBasicType ? unaryExpression() : simpleUnaryExpression();
1068             return new JCastOp(line, type, expr);
1069         } else {
1070             return postfixExpression();
1071         }
1072     }
```

```
1073
1074    /**
1075     * Parses a postfix expression and returns an AST for it.
1076     *
1077     * <pre>
1078     *   postfixExpression ::= primary { selector } { DEC | INC }
1079     * </pre>
1080     *
1081     * @return an AST for a postfix expression.
1082     */
1083    private JExpression postfixExpression() {
1084        int line = scanner.token().line();
1085        JExpression primaryExpr = primary();
1086        while (see(DOT) || see(LBRACK)) {
1087            primaryExpr = selector(primaryExpr);
1088        }
1089        while (have(DEC)) {
1090            primaryExpr = new JPostDecrementOp(line, primaryExpr);
1091        }
1092        while (have(INC)) {
1093            primaryExpr = new JPostIncrementOp(line, primaryExpr);
1094        }
1095        return primaryExpr;
1096    }
1097
1098    /**
1099     * Parses a selector and returns an AST for it.
1100     *
1101     * <pre>
1102     *   selector ::= DOT qualifiedIdentifier [ arguments ]
1103     *              | LBRACK expression RBRACK
1104     * </pre>
1105     *
1106     * @param target the target expression for this selector.
1107     * @return an AST for a selector.
1108     */
1109    private JExpression selector(JExpression target) {
1110        int line = scanner.token().line();
1111        if (have(DOT)) {
1112            // target.selector.
1113            mustBe(IDENTIFIER);
1114            String name = scanner.previousToken().image();
1115            if (see(LPAREN)) {
1116                ArrayList<JExpression> args = arguments();
1117                return new JMessageExpression(line, target, name, args);
1118            } else {
1119                return new JFieldSelection(line, target, name);
1120            }
1121        } else {
```

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

```
1171        } else if (see(IDENTIFIER)) {
1172          TypeName id = qualifiedIdentifier();
1173          if (see(LPAREN)) {
1174            // ambiguousPart.messageName(...).
1175            ArrayList<JExpression> args = arguments();
1176            return new JMessageExpression(line, null, ambiguousPart(id), id.simpleName(), args);
1177          } else if (ambiguousPart(id) == null) {
1178            // A simple name.
1179            return new JVariable(line, id.simpleName());
1180          } else {
1181            // ambiguousPart.fieldName.
1182            return new JFieldSelection(line, ambiguousPart(id), null, id.simpleName());
1183          }
1184        } else {
1185          return literal();
1186        }
1187      }
1188
1189      /**
1190       * Parses a creator and returns an AST for it.
1191       *
1192       * <pre>
1193       *   creator ::= ( basicType | qualifiedIdentifier )
1194       *                 ( arguments
1195       *                 | LBRACK RBRACK { LBRACK RBRACK } [ arrayInitializer ]
1196       *                 | newArrayDeclarator
1197       *                 )
1198       * </pre>
1199       *
1200       * @return an AST for a creator.
1201       */
1202      private JExpression creator() {
1203        int line = scanner.token().line();
1204        Type type = seeBasicType() ? basicType() : qualifiedIdentifier();
1205        if (see(LPAREN)) {
1206          ArrayList<JExpression> args = arguments();
1207          return new JNewOp(line, type, args);
1208        } else if (see(LBRACK)) {
1209          if (seeDims()) {
1210            Type expected = type;
1211            while (have(LBRACK)) {
1212              mustBe(RBRACK);
1213              expected = new ArrayTypeName(expected);
1214            }
1215            return arrayInitializer(expected);
1216          } else {
1217            return newArrayDeclarator(line, type);
1218          }
1219        } else {
```

```
1220            reportParserError("( or [ sought where %s found", scanner.token().image());
1221            return new JWildExpression(line);
1222          }
1223        }
1224
1225        /**
1226         * Parses a new array declarator and returns an AST for it.
1227         *
1228         * <pre>
1229         *   newArrayDeclarator ::= LBRACK expression RBRACK
1230         *                          { LBRACK expression RBRACK } { LBRACK RBRACK }
1231         * </pre>
1232         *
1233         * @param line line in which the declarator occurred.
1234         * @param type type of the array.
1235         * @return an AST for a new array declarator.
1236         */
1237        private JNewArrayOp newArrayDeclarator(int line, Type type) {
1238          ArrayList<JExpression> dimensions = new ArrayList<JExpression>();
1239          mustBe(LBRACK);
1240          dimensions.add(expression());
1241          mustBe(RBRACK);
1242          type = new ArrayTypeName(type);
1243          while (have(LBRACK)) {
1244            if (have(RBRACK)) {
1245              // We're done with dimension expressions.
1246              type = new ArrayTypeName(type);
1247              while (have(LBRACK)) {
1248                mustBe(RBRACK);
1249                type = new ArrayTypeName(type);
1250              }
1251              return new JNewArrayOp(line, type, dimensions);
1252            } else {
1253              dimensions.add(expression());
1254              type = new ArrayTypeName(type);
1255              mustBe(RBRACK);
1256            }
1257          }
1258          return new JNewArrayOp(line, type, dimensions);
1259        }
1260
1261        /**
1262         * Parses a literal and returns an AST for it.
1263         *
1264         * <pre>
1265         *   literal ::= CHAR_LITERAL | FALSE | INT_LITERAL | DOUBLE_LITERAL | LONG_LITERAL | NULL |
    STRING_LITERAL | TRUE
1266         * </pre>
1267         *
```

```
1268      * @return an AST for a literal.
1269      */
1270     private JExpression literal() {
1271         int line = scanner.token().line();
1272         if (have(CHAR_LITERAL)) {
1273             return new JLiteralChar(line, scanner.previousToken().image());
1274         } else if (have(FALSE)) {
1275             return new JLiteralBoolean(line, scanner.previousToken().image());
1276         } else if (have(INT_LITERAL)) {
1277             return new JLiteralInt(line, scanner.previousToken().image());
1278         } else if (have(DOUBLE_LITERAL)) {
1279             return new JLiteralDouble(line, scanner.previousToken().image());
1280         } else if (have(LONG_LITERAL)) {
1281             return new JLiteralLong(line, scanner.previousToken().image());
1282         } else if (have(NULL)) {
1283             return new JLiteralNull(line);
1284         } else if (have(STRING_LITERAL)) {
1285             return new JLiteralString(line, scanner.previousToken().image());
1286         } else if (have(TRUE)) {
1287             return new JLiteralBoolean(line, scanner.previousToken().image());
1288         } else {
1289             reportParserError("Literal sought where %s found", scanner.token().image());
1290             return new JWildExpression(line);
1291         }
1292     }
1293 
1294     //////////////////////////////////////////////////
1295     // Parsing Support
1296     // //////////////////////////////////////////////
1297 
1298     // Returns true if the current token equals sought, and false otherwise.
1299     private boolean see(TokenKind sought) {
1300         return (sought == scanner.token().kind());
1301     }
1302 
1303     // If the current token equals sought, scans it and returns true. Otherwise, returns false
1304     // without scanning the token.
1305     private boolean have(TokenKind sought) {
1306         if (see(sought)) {
1307             scanner.next();
1308             return true;
1309         } else {
1310             return false;
1311         }
1312     }
1313 
1314     // Attempts to match a token we're looking for with the current input token. On success,
1315     // scans the token and goes into a "Recovered" state. On failure, what happens next depends
1316     // on whether or not the parser is currently in a "Recovered" state: if so, it reports the
```

```
1317    // error and goes into an "Unrecovered" state; if not, it repeatedly scans tokens until it
1318    // finds the one it is looking for (or EOF) and then returns to a "Recovered" state. This
1319    // gives us a kind of poor man's syntactic error recovery, a strategy due to David Turner and
1320    // Ron Morrison.
1321    private void mustBe(TokenKind sought) {
1322        if (scanner.token().kind() == sought) {
1323            scanner.next();
1324            isRecovered = true;
1325        } else if (isRecovered) {
1326            isRecovered = false;
1327            reportParserError("%s found where %s sought", scanner.token().image(), sought.image());
1328        } else {
1329            // Do not report the (possibly spurious) error, but rather attempt to recover by
1330            // forcing a match.
1331            while (!see(sought) && !see(EOF)) {
1332                scanner.next();
1333            }
1334            if (see(sought)) {
1335                scanner.next();
1336                isRecovered = true;
1337            }
1338        }
1339    }
1340
1341    // Pulls out and returns the ambiguous part of a name.
1342    private AmbiguousName ambiguousPart(TypeName name) {
1343        String qualifiedName = name.toString();
1344        int i = qualifiedName.lastIndexOf('.');
1345        return i == -1 ? null : new AmbiguousName(name.line(), qualifiedName.substring(0, i));
1346    }
1347
1348    // Reports a syntax error.
1349    private void reportParserError(String message, Object... args) {
1350        isInError = true;
1351        isRecovered = false;
1352        System.err.printf("%s:%d: error: ", scanner.fileName(), scanner.token().line());
1353        System.err.printf(message, args);
1354        System.err.println();
1355    }
1356
1357    //////////////////////////////////////////////////
1358    // Lookahead Methods
1359    //////////////////////////////////////////////////
1360
1361    // Returns true if we are looking at an IDENTIFIER followed by a LPAREN, and false otherwise.
1362    private boolean seeIdentLParen() {
1363        scanner.recordPosition();
1364        boolean result = have(IDENTIFIER) && see(LPAREN);
1365        scanner.returnToPosition();
```

```
1366        return result;
1367    }
1368
1369    // Returns true if we are looking at a cast (basic or reference), and false otherwise.
1370    private boolean seeCast() {
1371        scanner.recordPosition();
1372        if (!have(LPAREN)) {
1373            scanner.returnToPosition();
1374            return false;
1375        }
1376        if (seeBasicType()) {
1377            scanner.returnToPosition();
1378            return true;
1379        }
1380        if (!see(IDENTIFIER)) {
1381            scanner.returnToPosition();
1382            return false;
1383        } else {
1384            scanner.next();
1385            // A qualified identifier is ok.
1386            while (have(DOT)) {
1387                if (!have(IDENTIFIER)) {
1388                    scanner.returnToPosition();
1389                    return false;
1390                }
1391            }
1392        }
1393        while (have(LBRACK)) {
1394            if (!have(RBRACK)) {
1395                scanner.returnToPosition();
1396                return false;
1397            }
1398        }
1399        if (!have(RPAREN)) {
1400            scanner.returnToPosition();
1401            return false;
1402        }
1403        scanner.returnToPosition();
1404        return true;
1405    }
1406
1407    // Returns true if we are looking at a local variable declaration, and false otherwise.
1408    private boolean seeLocalVariableDeclaration() {
1409        scanner.recordPosition();
1410        if (have(IDENTIFIER)) {
1411            // A qualified identifier is ok.
1412            while (have(DOT)) {
1413                if (!have(IDENTIFIER)) {
1414                    scanner.returnToPosition();
```

```
1415                return false;
1416            }
1417        }
1418    } else if (seeBasicType()) {
1419        scanner.next();
1420    } else {
1421        scanner.returnToPosition();
1422        return false;
1423    }
1424    while (have(LBRACK)) {
1425        if (!have(RBRACK)) {
1426            scanner.returnToPosition();
1427            return false;
1428        }
1429    }
1430    if (!have(IDENTIFIER)) {
1431        scanner.returnToPosition();
1432        return false;
1433    }
1434    while (have(LBRACK)) {
1435        if (!have(RBRACK)) {
1436            scanner.returnToPosition();
1437            return false;
1438        }
1439    }
1440    scanner.returnToPosition();
1441    return true;
1442  }
1443
1444  // Returns true if we are looking at a basic type, and false otherwise.
1445  private boolean seeBasicType() {
1446    return (see(BOOLEAN) || see(CHAR) || see(INT)
1447            || see(DOUBLE) || see(LONG));
1448  }
1449
1450  // Returns true if we are looking at a reference type, and false otherwise.
1451  private boolean seeReferenceType() {
1452    if (see(IDENTIFIER)) {
1453        return true;
1454    } else {
1455        scanner.recordPosition();
1456        if (have(BOOLEAN) || have(CHAR) || have(INT) || have(DOUBLE) || have(LONG)) {
1457            if (have(LBRACK) && see(RBRACK)) {
1458                scanner.returnToPosition();
1459                return true;
1460            }
1461        }
1462        scanner.returnToPosition();
1463    }
```

```
1464        return false;
1465    }
1466
1467    // Returns true if we are looking at a [] pair, and false otherwise.
1468    private boolean seeDims() {
1469        scanner.recordPosition();
1470        boolean result = have(LBRACK) && see(RBRACK);
1471        scanner.returnToPosition();
1472        return result;
1473    }
1474 }
1475
```

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  */
```

```
341 class JPreDecrementOp extends JUnaryExpression {
342     /**
343      * Constructs an AST node for a pre-decrement expression.
344      *
345      * @param line    line in which the expression occurs in the source file.
346      * @param operand the operand.
347      */
348     public JPreDecrementOp(int line, JExpression operand) {
349         super(line, "-- (pre)", operand);
350     }
351 
352     /**
353      * {@inheritDoc}
354      */
355     public JExpression analyze(Context context) {
356         // TODO
357         return this;
358     }
359 
360     /**
361      * {@inheritDoc}
362      */
363     public void codegen(CLEmitter output) {
364         // TODO
365     }
366 }
```

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 
```

**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 this project was to learn more about the parser by adding support for extra features. In order
to implement the parsing of these new features, the java grammar had to be used as a guide. There were also times
in which helper methods were created to support further grammars that were not in j--. The first problem was about
parsing long and double basic types. The second part was adding support for the operators introduced in project 2.
The next things to parse in the same method were conditional expressions, do, for, break, continue, and switch
statements. Lastly, we also had to add support for exception handlers and interface type declaration, which needed
other methods in parser to be changed.

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 | Helped with debugging and clarifications on Piazza for problems 1, 3 and 5. |

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.

This project was a lot more challenging than I expected. I underestimated the time it would take and was
not able to finish it or get most of the code working. The experience could have went better.