Project 5 (Type Checking and Code Generation)

● Graded

3 Days, 13 Hours Late

Student

Giancarlos Marte

Total Points

46.03 / 100 pts

Autograder Score

40.78 / 80.0

Failed Tests

Problem 4. Switch Statement (0/10)
Problem 7. Break Statement (0/10)
Problem 8. Continue 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 3. Conditional Expression (10/10)
Problem 5. Do-while Statement (10/10)
Problem 6. For Statement (10/10)

Question 2

Code Clarity and Efficiency

5.25 / 10 pts

Long and Double Basic Types

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

Operators

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

Conditional Expression

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

Switch Statement

+ 0.5 pts Passed all tests

+ 0.25 pts Changes commented adequately

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

Do Statement

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

For Statement

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

Break Statement

+ 0.5 pts Passed all tests

+ 0.25 pts Changes commented adequately

+ 0.25 pts Followed good programming practices

Continue Statement

+ 0.5 pts Passed all tests

+ 0.25 pts Changes commented adequately

+ 0.25 pts Followed good programming practices

Exception handlers

+ 0.5 pts Passed all tests

+ 0.25 pts Changes commented adequately

+ 0.25 pts Followed good programming practices

Interface Type Declaration

+ 0.5 pts Passed all tests

+ 0.25 pts Changes commented adequately

+ 0.25 pts Followed good programming practices

+ 0 pts Do not meet expectations

Question 3

Notes File **0** / 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-- tests/BasicTypes.java
java BasicTypes 1 -5 6 6
j-- tests/Stats.java
java Stats
```

Problem 2. Operators (10/10)

```
j-- tests/Operators.java
java Operators 23 3
```

Problem 3. Conditional Expression (10/10)

```
j-- tests/ConditionalExpression.java
java ConditionalExpression
```

Problem 4. Switch Statement (0/10)

j-- tests/SwitchStatement.java
'Exception in thread "main" java.lang.Null[562 chars]131)' != ''
- Exception in thread "main" java.lang.NullPointerException: Cannot invoke "jminusminus.Type.mustMatchEx
-	at jminusminus.JSwitchStatement.analyze(JSwitchStatement.java:42)
-	at jminusminus.JSwitchStatement.analyze(JSwitchStatement.java:12)
-	at jminusminus.JBlock.analyze(JBlock.java:45)
-	at jminusminus.JMethodDeclaration.analyze(JMethodDeclaration.java:154)
-	at jminusminus.JClassDeclaration.analyze(JClassDeclaration.java:166)
-	at jminusminus.JCompilationUnit.analyze(JCompilationUnit.java:157)
-	at jminusminus.Main.main(Main.java:131)
java SwitchStatement

Error: cannot find file 'SwitchStatement.class'
java SwitchStatement

Error: cannot find file 'SwitchStatement.class'
java SwitchStatement

Error: cannot find file 'SwitchStatement.class'
java SwitchStatement

Error: cannot find file 'SwitchStatement.class'
java SwitchStatement

Error: cannot find file 'SwitchStatement.class'
java SwitchStatement

Error: cannot find file 'SwitchStatement.class'
java SwitchStatement

Error: cannot find file 'SwitchStatement.class'
java SwitchStatement

Error: cannot find file 'SwitchStatement.class'
java SwitchStatement

Error: cannot find file 'SwitchStatement.class'

Problem 5. Do-while Statement (10/10)

j-- tests/DoStatement.java
java DoStatement 100

Problem 6. For Statement (10/10)

j-- tests/ForStatement.java
java ForStatement 100

Problem 7. Break Statement (0/10)

j-- tests/BreakStatement.java □
'Exception in thread "main" java.lang.Null[549 chars]131)' != ''
- Exception in thread "main" java.lang.NullPointerException: Cannot invoke "jminusminus.JExpression.analyz
- at jminusminus.JForStatement.analyze(JForStatement.java:53)
- at jminusminus.JForStatement.analyze(JForStatement.java:12)
- at jminusminus.JBlock.analyze(JBlock.java:45)
- at jminusminus.JMethodDeclaration.analyze(JMethodDeclaration.java:154)
- at jminusminus.JClassDeclaration.analyze(JClassDeclaration.java:166)
- at jminusminus.JCompilationUnit.analyze(JCompilationUnit.java:157)
- at jminusminus.Main.main(Main.java:131)
java BreakStatement 1000 □

Error: cannot find file 'BreakStatement.class'

Problem 8. Continue Statement (0/10)

j-- tests/ContinueStatement.java □
'Exception in thread "main" java.lang.Null[518 chars]131)' != ''
- Exception in thread "main" java.lang.NullPointerException: Cannot invoke "java.util.ArrayList.size()" because
- at jminusminus.JForStatement.analyze(JForStatement.java:49)
- at jminusminus.JForStatement.analyze(JForStatement.java:12)
- at jminusminus.JBlock.analyze(JBlock.java:45)
- at jminusminus.JMethodDeclaration.analyze(JMethodDeclaration.java:154)
- at jminusminus.JClassDeclaration.analyze(JClassDeclaration.java:166)
- at jminusminus.JCompilationUnit.analyze(JCompilationUnit.java:157)
- at jminusminus.Main.main(Main.java:131)
java ContinueStatement 100 □

Error: cannot find file 'ContinueStatement.class'

Problem 9. Exception Handlers (0/10)

```
j-- tests/ExceptionHandlers.java 
"tests/ExceptionHandlers.java:27: error: Type double doesn't match type int" != "
- tests/ExceptionHandlers.java:27: error: Type double doesn't match type int
+

java ExceptionHandlers 

Error: cannot find file 'ExceptionHandlers.class'
java ExceptionHandlers two 

Error: cannot find file 'ExceptionHandlers.class'
java ExceptionHandlers -2 

Error: cannot find file 'ExceptionHandlers.class'
java ExceptionHandlers 2 

Error: cannot find file 'ExceptionHandlers.class'
```

Problem 10. Interface Type Declaration (0/10)

```
j-- tests/Interface.java 
'Exception in thread "main" java.lang.Null[328 chars]146)' != "
- Exception in thread "main" java.lang.NullPointerException: Cannot invoke "java.util.ArrayList.size()" because
-       at jminusminus.CLEmitter.endOpenMethodIfAny(CLEmitter.java:1124)
-       at jminusminus.CLEmitter.write(CLEmitter.java:1004)
-       at jminusminus.JCompilationUnit.codegen(JCompilationUnit.java:168)
-       at jminusminus.Main.main(Main.java:146)
java Interface 10 

Error: cannot find file 'Interface.class'
```

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

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

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

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

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

    // The token kind's string representation.
    private String image;

    /**
```

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

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

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

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(BREAK.image(), BREAK);
54        reserved.put(CASE.image(), CASE);
55        reserved.put(CATCH.image(), CATCH);
56        reserved.put(CHAR.image(), CHAR);
57        reserved.put(CLASS.image(), CLASS);
58        reserved.put(CONTINUE.image(), CONTINUE);
59        reserved.put(DEFLT.image(), DEFLT);
60        reserved.put(DO.image(), DO);
61        reserved.put(DOUBLE.image(), DOUBLE);
62        reserved.put(ELSE.image(), ELSE);
63        reserved.put(EXTENDS.image(), EXTENDS);
64        reserved.put(FALSE.image(), FALSE);
65        reserved.put(FINALLY.image(), FINALLY);
66        reserved.put(FOR.image(), FOR);
67        reserved.put(IF.image(), IF);
68        reserved.put(IMPLEMENTS.image(), IMPLEMENTS);
69        reserved.put(IMPORT.image(), IMPORT);
70        reserved.put(INSTANCEOF.image(), INSTANCEOF);
71        reserved.put(INT.image(), INT);
72        reserved.put(INTERFACE.image(), INTERFACE);
73        reserved.put(LONG.image(), LONG);
74        reserved.put(NEW.image(), NEW);
75        reserved.put(NULL.image(), NULL);
76        reserved.put(PACKAGE.image(), PACKAGE);
77        reserved.put(PRIVATE.image(), PRIVATE);
78        reserved.put(PROTECTED.image(), PROTECTED);
79        reserved.put(PUBLIC.image(), PUBLIC);
80        reserved.put(RETURN.image(), RETURN);
81        reserved.put(STATIC.image(), STATIC);
82        reserved.put(SUPER.image(), SUPER);
83        reserved.put(SWITCH.image(), SWITCH);
84        reserved.put(THIS.image(), THIS);
85        reserved.put(THROW.image(), THROW);
86        reserved.put(THROWS.image(), THROWS);
87        reserved.put(TRUE.image(), TRUE);
88        reserved.put(TRY.image(), TRY);
89        reserved.put(VOID.image(), VOID);
90        reserved.put(WHILE.image(), WHILE);
91
92        // Prime the pump.
93        nextCh();
94    }
95
```

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

```
        switch (ch) {
            case ',':
                nextCh();
                return new TokenInfo(COMMA, line);
            case '[':
                nextCh();
                return new TokenInfo(LBRACK, line);
            case '{':
                nextCh();
                return new TokenInfo(LCURLY, line);
            case '(':
                nextCh();
                return new TokenInfo(LPAREN, line);
            case ']':
                nextCh();
                return new TokenInfo(RBRACK, line);
            case '}':
                nextCh();
                return new TokenInfo(RCURLY, line);
            case ')':
                nextCh();
                return new TokenInfo(RPAREN, line);
            case ';':
                nextCh();
                return new TokenInfo(SEMI, line);
            case '*':
                nextCh();
                if (ch == '=') {
                    nextCh();
                    return new TokenInfo(STAR_ASSIGN, line);
                } else {
                    return new TokenInfo(STAR, line);
                }
            case '%':
                nextCh();
                if (ch == '=') {
                    nextCh();
                    return new TokenInfo(REM_ASSIGN, line);
                } else {
                    return new TokenInfo(REM, line);
                }
            case '+':
                nextCh();
                if (ch == '=') {
                    nextCh();
                    return new TokenInfo(PLUS_ASSIGN, line);
                } else if (ch == '+') {
                    nextCh();
                    return new TokenInfo(INC, line);
```

```
194             } else {
195                 return new TokenInfo(PLUS, line);
196             }
197         case '-':
198             nextCh();
199             if (ch == '-') {
200                 nextCh();
201                 return new TokenInfo(DEC, line);
202             } else if (ch == '=') {
203                 nextCh();
204                 return new TokenInfo(MINUS_ASSIGN, line);
205             } else {
206                 return new TokenInfo(MINUS, line);
207             }
208         case '=':
209             nextCh();
210             if (ch == '=') {
211                 nextCh();
212                 return new TokenInfo(EQUAL, line);
213             } else {
214                 return new TokenInfo(ASSIGN, line);
215             }
216         case '>':
217             nextCh();
218             if (ch == '>') {
219                 nextCh();
220                 if (ch == '>') {
221                     nextCh();
222                     if (ch == '=') {
223                         nextCh();
224                         return new TokenInfo(LRSHIFT_ASSIGN, line);
225                     } else {
226                         return new TokenInfo(LRSHIFT, line);
227                     }
228                 } else if (ch == '=') {
229                     nextCh();
230                     return new TokenInfo(ARSHIFT_ASSIGN, line);
231                 } else {
232                     return new TokenInfo(ARSHIFT, line);
233                 }
234             } else if (ch == '=') {
235                 nextCh();
236                 return new TokenInfo(GE, line);
237             } else {
238                 return new TokenInfo(GT, line);
239             }
240         case '<':
241             nextCh();
242             if (ch == '<') {
```

```
243                 nextCh();
244                 if (ch == '=') {
245                     nextCh();
246                     return new TokenInfo(ALSHIFT_ASSIGN, line);
247                 } else {
248                     return new TokenInfo(ALSHIFT, line);
249                 }
250             } else if (ch == '=') {
251                 nextCh();
252                 return new TokenInfo(LE, line);
253             } else {
254                 return new TokenInfo(LT, line);
255             }
256         case '!':
257             nextCh();
258             if (ch == '=') {
259                 nextCh();
260                 return new TokenInfo(NOT_EQUAL, line);
261             } else {
262                 return new TokenInfo(LNOT, line);
263             }
264         case '~':
265             nextCh();
266             return new TokenInfo(NOT, line);
267         case '|':
268             nextCh();
269             if (ch == '|') {
270                 nextCh();
271                 return new TokenInfo(LOR, line);
272             } else if (ch == '=') {
273                 nextCh();
274                 return new TokenInfo(OR_ASSIGN, line);
275             } else {
276                 return new TokenInfo(OR, line);
277             }
278         case '^':
279             nextCh();
280             if (ch == '=') {
281                 nextCh();
282                 return new TokenInfo(XOR_ASSIGN, line);
283             } else {
284                 return new TokenInfo(XOR, line);
285             }
286         case '&':
287             nextCh();
288             if (ch == '&') {
289                 nextCh();
290                 return new TokenInfo(LAND, line);
291             } else if (ch == '=') {
```

```
292                 nextCh();
293                 return new TokenInfo(AND_ASSIGN, line);
294             } else {
295                 return new TokenInfo(AND, line);
296             }
297         case '?':
298             nextCh();
299             return new TokenInfo(QUESTION, line);
300         case ':':
301             nextCh();
302             return new TokenInfo(COLON, line);
303         case '\'':
304             buffer = new StringBuffer();
305             buffer.append('\'');
306             nextCh();
307             if (ch == '\\') {
308                 nextCh();
309                 buffer.append(escape());
310             } else {
311                 buffer.append(ch);
312                 nextCh();
313             }
314             if (ch == '\'') {
315                 buffer.append('\'');
316                 nextCh();
317                 return new TokenInfo(CHAR_LITERAL, buffer.toString(), line);
318             } else {
319                 // Expected a ' ; report error and try to recover.
320                 reportScannerError(ch + " found by scanner where closing ' was expected");
321                 while (ch != '\'' && ch != ';' && ch != '\n') {
322                     nextCh();
323                 }
324                 return new TokenInfo(CHAR_LITERAL, buffer.toString(), line);
325             }
326         case '"':
327             buffer = new StringBuffer();
328             buffer.append("\"");
329             nextCh();
330             while (ch != '"' && ch != '\n' && ch != EOFCH) {
331                 if (ch == '\\') {
332                     nextCh();
333                     buffer.append(escape());
334                 } else {
335                     buffer.append(ch);
336                     nextCh();
337                 }
338             }
339             if (ch == '\n') {
340                 reportScannerError("Unexpected end of line found in string");
```

```
341             } else if (ch == EOFCH) {
342                 reportScannerError("Unexpected end of file found in string");
343             } else {
344                 // Scan the closing "
345                 nextCh();
346                 buffer.append("\"");
347             }
348             return new TokenInfo(STRING_LITERAL, buffer.toString(), line);
349         case EOFCH:
350             return new TokenInfo(EOF, line);
351         case '.':
352         case '0':
353         case '1':
354         case '2':
355         case '3':
356         case '4':
357         case '5':
358         case '6':
359         case '7':
360         case '8':
361         case '9':
362             buffer = new StringBuffer();
363             if (isDigit(ch)) {
364                 buffer.append(digits());
365                 if (ch != '.' && ch != 'd' && ch != 'D' && ch != 'e' && ch != 'E' &&
366                         ch != 'l' && ch != 'L') {
367                     return new TokenInfo(INT_LITERAL, buffer.toString(), line);
368                 }
369                 TokenInfo token = suffix(buffer); // double, long, or null
370                 if (token != null) {
371                     return token;
372                 }
373                 if (ch == '.') {
374                     buffer.append(ch);
375                     nextCh();
376                 }
377                 buffer.append(digits());
378                 if (ch == 'd' || ch == 'D') {
379                     buffer.append(ch);
380                     nextCh();
381                     return new TokenInfo(DOUBLE_LITERAL, buffer.toString(), line);
382                 }
383                 if (ch == 'e' || ch == 'E') {
384                     buffer.append(exp());
385                     if (ch == 'd' || ch == 'D') {
386                         buffer.append(ch);
387                         nextCh();
388                         return new TokenInfo(DOUBLE_LITERAL, buffer.toString(), line);
389                     }
```

```
390                 }
391                 return new TokenInfo(DOUBLE_LITERAL, buffer.toString(), line);
392             }
393             if (ch == '.') {
394                 buffer.append(ch);
395                 nextCh();
396                 if (!isDigit(ch)) {
397                     return new TokenInfo(DOT, line);
398                 }
399                 buffer.append(digits());
400                 if (ch == 'e' || ch == 'E') {
401                     buffer.append(exp());
402                 }
403                 if (ch == 'd' || ch == 'D') {
404                     buffer.append(ch);
405                     nextCh();
406                 }
407                 return new TokenInfo(DOUBLE_LITERAL, buffer.toString(), line);
408             }
409 
410             // Shouldn't get here.
411             reportScannerError("Freak out!", ch);
412             return getNextToken();
413         default:
414             if (isIdentifierStart(ch)) {
415                 buffer = new StringBuffer();
416                 while (isIdentifierPart(ch)) {
417                     buffer.append(ch);
418                     nextCh();
419                 }
420                 String identifier = buffer.toString();
421                 if (reserved.containsKey(identifier)) {
422                     return new TokenInfo(reserved.get(identifier), line);
423                 } else {
424                     return new TokenInfo(IDENTIFIER, identifier, line);
425                 }
426             } else {
427                 reportScannerError("Unidentified input token: '%c'", ch);
428                 nextCh();
429                 return getNextToken();
430             }
431     }
432 }
433 
434 /**
435  * Returns true if an error has occurred, and false otherwise.
436  *
437  * @return true if an error has occurred, and false otherwise.
438  */
```

```
439     public boolean errorHasOccurred() {
440         return isInError;
441     }
442 
443     /**
444      * Returns the name of the source file.
445      *
446      * @return the name of the source file.
447      */
448     public String fileName() {
449         return fileName;
450     }
451 
452     // Scans and returns an escaped character.
453     private String escape() {
454         switch (ch) {
455             case 'b':
456                 nextCh();
457                 return "\\b";
458             case 't':
459                 nextCh();
460                 return "\\t";
461             case 'n':
462                 nextCh();
463                 return "\\n";
464             case 'f':
465                 nextCh();
466                 return "\\f";
467             case 'r':
468                 nextCh();
469                 return "\\r";
470             case '"':
471                 nextCh();
472                 return "\\\"";
473             case '\'':
474                 nextCh();
475                 return "\\'";
476             case '\\':
477                 nextCh();
478                 return "\\\\";
479             default:
480                 reportScannerError("Badly formed escape: \\%c", ch);
481                 nextCh();
482                 return "";
483         }
484     }
485 
486     // Advances ch to the next character from input, and updates the line number.
487     private void nextCh() {
```

```
488         line = input.line();
489         try {
490             ch = input.nextChar();
491         } catch (Exception e) {
492             reportScannerError("Unable to read characters from input");
493         }
494     }
495
496     // Reports a lexical error and records the fact that an error has occurred. This fact can be
497     // ascertained from the Scanner by sending it an errorHasOccurred message.
498     private void reportScannerError(String message, Object... args) {
499         isInError = true;
500         System.err.printf("%s:%d: error: ", fileName, line);
501         System.err.printf(message, args);
502         System.err.println();
503     }
504
505     // Returns true if the specified character is a digit (0-9), and false otherwise.
506     private boolean isDigit(char c) {
507         return (c >= '0' && c <= '9');
508     }
509
510     // Returns true if the specified character is a whitespace, and false otherwise.
511     private boolean isWhitespace(char c) {
512         return (c == ' ' || c == '\t' || c == '\n' || c == '\f');
513     }
514
515     // Returns true if the specified character can start an identifier name, and false otherwise.
516     private boolean isIdentifierStart(char c) {
517         return (c >= 'a' && c <= 'z' || c >= 'A' && c <= 'Z' || c == '_' || c == '$');
518     }
519
520     // Returns true if the specified character can be part of an identifier name, and false
521     // otherwise.
522     private boolean isIdentifierPart(char c) {
523         return (isIdentifierStart(c) || isDigit(c));
524     }
525
526     // Scans and returns a string of digits (0-9).
527     private String digits() {
528         StringBuffer buffer = new StringBuffer();
529         while (isDigit(ch)) {
530             buffer.append(ch);
531             nextCh();
532         }
533         return buffer.toString();
534     }
535
536     // Scans and returns the exponent grammar rule.
```

```
537     //    EXPONENT       ::= ( "e" | "E" ) [ ( "+" | "-" ) ] DIGITS
538     private String exp() {
539         StringBuffer buffer = new StringBuffer();
540         buffer.append(ch);
541         nextCh();
542         if (ch == '+' || ch == '-') {
543             buffer.append(ch);
544             nextCh();
545         }
546         String digits = digits();
547         buffer.append(digits);
548         if (digits.length() == 0) {
549             reportScannerError("malformed exponent " + buffer.toString());
550         }
551         return buffer.toString();
552     }
553 
554     // Returns the TokenInfo object for the literal represented by the given buffer based on the
555     // suffix ("d" | "D") for doubles and ("l" | "L") for longs, or null.
556     private TokenInfo suffix(StringBuffer buffer) {
557         switch (ch) {
558             case 'd':
559             case 'D':
560                 buffer.append(ch);
561                 nextCh();
562                 return new TokenInfo(DOUBLE_LITERAL, buffer.toString(), line);
563             case 'l':
564             case 'L':
565                 buffer.append(ch);
566                 nextCh();
567                 return new TokenInfo(LONG_LITERAL, buffer.toString(), line);
568         }
569         return null;
570     }
571 }
572 
573 /**
574  * A buffered character reader, which abstracts out differences between platforms, mapping all new
575  * lines to '\n', and also keeps track of line numbers.
576  */
577 class CharReader {
578     // Representation of the end of file as a character.
579     public final static char EOFCH = (char) -1;
580 
581     // The underlying reader records line numbers.
582     private LineNumberReader lineNumberReader;
583 
584     // Name of the file that is being read.
585     private String fileName;
```

```
586
587     /**
588      * Constructs a CharReader from a file name.
589      *
590      * @param fileName the name of the input file.
591      * @throws FileNotFoundException if the file is not found.
592      */
593     public CharReader(String fileName) throws FileNotFoundException {
594         lineNumberReader = new LineNumberReader(new FileReader(fileName));
595         this.fileName = fileName;
596     }
597
598     /**
599      * Scans and returns the next character.
600      *
601      * @return the character scanned.
602      * @throws IOException if an I/O error occurs.
603      */
604     public char nextChar() throws IOException {
605         return (char) lineNumberReader.read();
606     }
607
608     /**
609      * Returns the current line number in the source file.
610      *
611      * @return the current line number in the source file.
612      */
613     public int line() {
614         return lineNumberReader.getLineNumber() + 1; // LineNumberReader counts lines from 0
615     }
616
617     /**
618      * Returns the file name.
619      *
620      * @return the file name.
621      */
622     public String fileName() {
623         return fileName;
624     }
625
626     /**
627      * Closes the file.
628      *
629      * @throws IOException if an I/O error occurs.
630      */
631     public void close() throws IOException {
632         lineNumberReader.close();
633     }
634 }
```

635

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 | interfaceDeclaration )
108      * </pre>
109      *
110      * @return an AST for a type declaration.
111      */
112     private JAST typeDeclaration() {
113         ArrayList<String> mods = modifiers();
114         if (see(INTERFACE)) {
115             return interfaceDeclaration(mods);
116         }
117         return classDeclaration(mods);
118     }
119
120     /**
121      * Parses and returns a list of modifiers.
122      *
123      * <pre>
124      *   modifiers ::= { ABSTRACT | PRIVATE | PROTECTED | PUBLIC | STATIC }
125      * </pre>
126      *
127      * @return a list of modifiers.
128      */
129     private ArrayList<String> modifiers() {
130         ArrayList<String> mods = new ArrayList<String>();
131         boolean scannedPUBLIC = false;
132         boolean scannedPROTECTED = false;
133         boolean scannedPRIVATE = false;
134         boolean scannedSTATIC = false;
135         boolean scannedABSTRACT = false;
136         boolean more = true;
137         while (more) {
138             if (have(ABSTRACT)) {
139                 mods.add("abstract");
140                 if (scannedABSTRACT) {
141                     reportParserError("Repeated modifier: abstract");
142                 }
143                 scannedABSTRACT = true;
144             } else if (have(PRIVATE)) {
```

```
            mods.add("private");
            if (scannedPRIVATE) {
                reportParserError("Repeated modifier: private");
            }
            if (scannedPUBLIC || scannedPROTECTED) {
                reportParserError("Access conflict in modifiers");
            }
            scannedPRIVATE = true;
        } else if (have(PROTECTED)) {
            mods.add("protected");
            if (scannedPROTECTED) {
                reportParserError("Repeated modifier: protected");
            }
            if (scannedPUBLIC || scannedPRIVATE) {
                reportParserError("Access conflict in modifiers");
            }
            scannedPROTECTED = true;
        } else if (have(PUBLIC)) {
            mods.add("public");
            if (scannedPUBLIC) {
                reportParserError("Repeated modifier: public");
            }
            if (scannedPROTECTED || scannedPRIVATE) {
                reportParserError("Access conflict in modifiers");
            }
            scannedPUBLIC = true;
        } else if (have(STATIC)) {
            mods.add("static");
            if (scannedSTATIC) {
                reportParserError("Repeated modifier: static");
            }
            scannedSTATIC = true;
        } else if (have(ABSTRACT)) {
            mods.add("abstract");
            if (scannedABSTRACT) {
                reportParserError("Repeated modifier: abstract");
            }
            scannedABSTRACT = true;
        } else {
            more = false;
        }
    }
    return mods;
}

/**
 * Parses a class declaration and returns an AST for it.
 *
 * <pre>
```

```
194     *   classDeclaration ::= CLASS IDENTIFIER [ EXTENDS qualifiedIdentifier ]
195     *                        [ IMPLEMENTS qualifiedIdentifier { COMMA qualifiedIdentifier } ]
196     *                           classBody
197     * </pre>
198     *
199     * @param mods the class modifiers.
200     * @return an AST for a class declaration.
201     */
202    private JClassDeclaration classDeclaration(ArrayList<String> mods) {
203      int line = scanner.token().line();
204      mustBe(CLASS);
205      mustBe(IDENTIFIER);
206      String name = scanner.previousToken().image();
207      Type superClass;
208      if (have(EXTENDS)) {
209        superClass = qualifiedIdentifier();
210      } else {
211        superClass = Type.OBJECT;
212      }
213      ArrayList<TypeName> superInterfaces = null;
214      if (have(IMPLEMENTS)) {
215        superInterfaces = new ArrayList<TypeName>();
216        do {
217          superInterfaces.add(qualifiedIdentifier());
218        } while (have(COMMA));
219      }
220      return new JClassDeclaration(line, mods, name, superClass, superInterfaces, classBody());
221    }
222
223    /**
224     * Parses an interface declaration and returns an AST for it.
225     *
226     * <pre>
227     *   interfaceDeclaration ::= INTERFACE IDENTIFIER
228     *                            [ EXTENDS qualifiedIdentifier { COMMA qualifiedIdentifier } ]
229     *                               interfaceBody
230     * </pre>
231     *
232     * @param mods the interface modifiers
233     * @return an AST for an interface declaration.
234     */
235    private JInterfaceDeclaration interfaceDeclaration(ArrayList<String> mods) {
236      int line = scanner.token().line();
237      mustBe(INTERFACE);
238      mustBe(IDENTIFIER);
239      String name = scanner.previousToken().image();
240      ArrayList<TypeName> superInterfaces = null;
241      if (have(EXTENDS)) {
242        superInterfaces = new ArrayList<TypeName>();
```

```
243             do {
244                 superInterfaces.add(qualifiedIdentifier());
245             } while (have(COMMA));
246         }
247         return new JInterfaceDeclaration(line, mods, name, superInterfaces, interfaceBody());
248     }
249 
250     /**
251      * Parses a class body and returns a list of members in the body.
252      *
253      * <pre>
254      *   classBody ::= LCURLY { modifiers memberDecl } RCURLY
255      * </pre>
256      *
257      * @return a list of members in the class body.
258      */
259     private ArrayList<JMember> classBody() {
260         ArrayList<JMember> members = new ArrayList<JMember>();
261         mustBe(LCURLY);
262         while (!see(RCURLY) && !see(EOF)) {
263             ArrayList<String> mods = modifiers();
264             members.add(memberDecl(mods));
265         }
266         mustBe(RCURLY);
267         return members;
268     }
269 
270     /**
271      * Parses an interface body and returns a list of members in the body.
272      *
273      * <pre>
274      *   interfaceBody ::= LCURLY { modifiers interfaceMemberDecl } RCURLY
275      * </pre>
276      *
277      * @return list of members in the interface body.
278      */
279     private ArrayList<JMember> interfaceBody() {
280         ArrayList<JMember> members = new ArrayList<JMember>();
281         mustBe(LCURLY);
282         while (!see(RCURLY) && !see(EOF)) {
283             members.add(interfaceMemberDecl(modifiers()));
284         }
285         mustBe(RCURLY);
286         return members;
287     }
288 
289     /**
290      * Parses a member declaration and returns an AST for it.
291      *
```

```
292     * <pre>
293     *   memberDecl ::= IDENTIFIER formalParameters
294     *                  [ THROWS qualifiedIdentifier { COMMA qualifiedIdentifier } ] block
295     *              | ( VOID | type ) IDENTIFIER formalParameters
296     *                  [ THROWS qualifiedIdentifier { COMMA qualifiedIdentifier } ]
297     *                     ( block | SEMI )
298     *              | type variableDeclarators SEMI
299     * </pre>
300     *
301     * @param mods the class member modifiers.
302     * @return an AST for a member declaration.
303     */
304    private JMember memberDecl(ArrayList<String> mods) {
305        int line = scanner.token().line();
306        JMember memberDecl = null;
307        if (seeIdentLParen()) {
308            // A constructor.
309            mustBe(IDENTIFIER);
310            String name = scanner.previousToken().image();
311            ArrayList<JFormalParameter> params = formalParameters();
312            ArrayList<TypeName> exceptions = null;
313            if (have(THROWS)) {
314                exceptions = new ArrayList<TypeName>();
315                do {
316                    exceptions.add(qualifiedIdentifier());
317                } while (have(COMMA));
318            }
319            JBlock body = block();
320            memberDecl = new JConstructorDeclaration(line, mods, name, params, exceptions, body);
321        } else {
322            Type type = null;
323            if (have(VOID)) {
324                // A void method.
325                type = Type.VOID;
326                mustBe(IDENTIFIER);
327                String name = scanner.previousToken().image();
328                ArrayList<JFormalParameter> params = formalParameters();
329                ArrayList<TypeName> exceptions = null;
330                if (have(THROWS)) {
331                    exceptions = new ArrayList<TypeName>();
332                    do {
333                        exceptions.add(qualifiedIdentifier());
334                    } while (have(COMMA));
335                }
336                JBlock body = have(SEMI) ? null : block();
337                memberDecl = new JMethodDeclaration(line, mods, name, type, params, exceptions,
338                        body);
339            } else {
340                type = type();
```

```
341             if (seeIdentLParen()) {
342                 // A non void method.
343                 mustBe(IDENTIFIER);
344                 String name = scanner.previousToken().image();
345                 ArrayList<JFormalParameter> params = formalParameters();
346                 ArrayList<TypeName> exceptions = null;
347                 if (have(THROWS)) {
348                     exceptions = new ArrayList<TypeName>();
349                     do {
350                         exceptions.add(qualifiedIdentifier());
351                     } while (have(COMMA));
352                 }
353                 JBlock body = have(SEMI) ? null : block();
354                 memberDecl = new JMethodDeclaration(line, mods, name, type, params, exceptions,
355                         body);
356             } else {
357                 // A field.
358                 memberDecl = new JFieldDeclaration(line, mods, variableDeclarators(type));
359                 mustBe(SEMI);
360             }
361         }
362     }
363     return memberDecl;
364 }
365
366 /**
367  * Parses an interface member declaration and return an AST for it.
368  *
369  * <pre>
370  *   interfaceMemberDecl ::= ( VOID | type ) IDENTIFIER formalParameters
371  *                           [ THROWS qualifiedIdentifier { COMMA qualifiedIdentifier } ] SEMI
372  *                         | type variableDeclarators SEMI
373  * </pre>
374  *
375  * @param mods the interface member modifiers.
376  * @return an AST for an interface member declaration.
377  */
378 private JMember interfaceMemberDecl(ArrayList<String> mods) {
379     int line = scanner.token().line();
380     JMember member = null;
381     if (have(CLASS)) {
382         reportParserError("No inner classes right now.");
383     } else if (have(INTERFACE)) {
384         reportParserError("No inner interfaces right now.");
385     } else {
386         Type type = null;
387         if (have(VOID)) {
388             // void method
389             if (!mods.contains("abstract")) {
```

```
390                 mods.add("abstract");
391             }
392             if (!mods.contains("public")) {
393                 mods.add("public");
394             }
395             type = Type.VOID;
396             mustBe(IDENTIFIER);
397             String name = scanner.previousToken().image();
398             ArrayList<JFormalParameter> params = formalParameters();
399             ArrayList<TypeName> exceptions = null;
400             if (have(THROWS)) {
401                 exceptions = new ArrayList<TypeName>();
402                 do {
403                     exceptions.add(qualifiedIdentifier());
404                 } while (have(COMMA));
405             }
406             mustBe(SEMI);
407             member = new JMethodDeclaration(line, mods, name, type, params, exceptions, null);
408         } else {
409             type = type();
410             if (seeIdentLParen()) {
411                 // Non void method
412                 if (!mods.contains("abstract")) {
413                     mods.add("abstract");
414                 }
415                 if (!mods.contains("public")) {
416                     mods.add("public");
417                 }
418                 mustBe(IDENTIFIER);
419                 String name = scanner.previousToken().image();
420                 ArrayList<JFormalParameter> params = formalParameters();
421                 ArrayList<TypeName> exceptions = null;
422                 if (have(THROWS)) {
423                     exceptions = new ArrayList<TypeName>();
424                     do {
425                         exceptions.add(qualifiedIdentifier());
426                     } while (have(COMMA));
427                 }
428                 mustBe(SEMI);
429                 member = new JMethodDeclaration(line, mods, name, type, params, exceptions,
430                         null);
431             } else {
432                 // Field
433                 member = new JFieldDeclaration(line, mods, variableDeclarators(type));
434                 mustBe(SEMI);
435             }
436         }
437     }
438     return member;
```

```
439     }
440
441     /**
442      * Parses a block and returns an AST for it.
443      *
444      * <pre>
445      *   block ::= LCURLY { blockStatement } RCURLY
446      * </pre>
447      *
448      * @return an AST for a block.
449      */
450     private JBlock block() {
451         int line = scanner.token().line();
452         ArrayList<JStatement> statements = new ArrayList<JStatement>();
453         mustBe(LCURLY);
454         while (!see(RCURLY) && !see(EOF)) {
455             statements.add(blockStatement());
456         }
457         mustBe(RCURLY);
458         return new JBlock(line, statements);
459     }
460
461     /**
462      * Parses a block statement and returns an AST for it.
463      *
464      * <pre>
465      *   blockStatement ::= localVariableDeclarationStatement
466      *                    | statement
467      * </pre>
468      *
469      * @return an AST for a block statement.
470      */
471     private JStatement blockStatement() {
472         if (seeLocalVariableDeclaration()) {
473             return localVariableDeclarationStatement();
474         } else {
475             return statement();
476         }
477     }
478
479     /**
480      * Parses a statement and returns an AST for it.
481      *
482      * <pre>
483      *   statement ::= block
484      *               | BREAK SEMI
485      *               | CONTINUE SEMI
486      *               | DO statement WHILE parExpression SEMI
487      *               | FOR LPAREN [ forInit ] SEMI [ expression ] SEMI [ forUpdate ] RPAREN statement
```

```
488     *             | IF parExpression statement [ ELSE statement ]
489     *             | RETURN [ expression ] SEMI
490     *             | SEMI
491     *             | SWITCH parExpression LCURLY { switchBlockStatementGroup } RCURLY
492     *             | THROW expression SEMI
493     *             | TRY block { CATCH LPAREN formalParameter RPAREN block } [ FINALLY block ]
494     *             | WHILE parExpression statement
495     *             | statementExpression SEMI
496     * </pre>
497     *
498     * @return an AST for a statement.
499     */
500    private JStatement statement() {
501        int line = scanner.token().line();
502        if (see(LCURLY)) {
503            return block();
504        } else if (have(BREAK)) {
505            mustBe(SEMI);
506            return new JBreakStatement(line);
507        } else if (have(CONTINUE)) {
508            mustBe(SEMI);
509            return new JContinueStatement(line);
510        } else if (have(DO)) {
511            JStatement statement = statement();
512            mustBe(WHILE);
513            JExpression test = parExpression();
514            mustBe(SEMI);
515            return new JDoStatement(line, statement, test);
516        } else if (have(FOR)) {
517            mustBe(LPAREN);
518            ArrayList<JStatement> inits = null;
519            if (!see(SEMI)) {
520                inits = forInit();
521                mustBe(SEMI);
522            } else {
523                mustBe(SEMI);
524            }
525            JExpression expr = null;
526            if (!see(SEMI)) {
527                expr = expression();
528                mustBe(SEMI);
529            } else {
530                mustBe(SEMI);
531            }
532            ArrayList<JStatement> updates = null;
533            if (!see(RPAREN)) {
534                updates = forUpdate();
535                mustBe(RPAREN);
536            } else {
```

```
537             mustBe(RPAREN);
538           }
539           return new JForStatement(line, inits, expr, updates, statement());
540         } else if (have(IF)) {
541           JExpression test = parExpression();
542           JStatement consequent = statement();
543           JStatement alternate = have(ELSE) ? statement() : null;
544           return new JIfStatement(line, test, consequent, alternate);
545         } else if (have(RETURN)) {
546           if (have(SEMI)) {
547             return new JReturnStatement(line, null);
548           } else {
549             JExpression expr = expression();
550             mustBe(SEMI);
551             return new JReturnStatement(line, expr);
552           }
553         } else if (have(SEMI)) {
554           return new JEmptyStatement(line);
555         } else if (have(SWITCH)) {
556           JExpression test = parExpression();
557           mustBe(LCURLY);
558           ArrayList<SwitchStatementGroup> switchBlockStmtGroup =
559               new ArrayList<SwitchStatementGroup>();
560           while (!see(RCURLY) && !see(EOF)) {
561             switchBlockStmtGroup.add(switchBlockStatementGroup());
562           }
563           mustBe(RCURLY);
564           return new JSwitchStatement(line, test, switchBlockStmtGroup);
565         } else if (have(THROW)) {
566           JExpression expr = expression();
567           mustBe(SEMI);
568           return new JThrowStatement(line, expr);
569         } else if (have(TRY)) {
570           JBlock tryBlock = block();
571           ArrayList<JFormalParameter> params = null;
572           ArrayList<JBlock> catchBlocks = null;
573           if (see(CATCH)) {
574             params = new ArrayList<JFormalParameter>();
575             catchBlocks = new ArrayList<JBlock>();
576             while (have(CATCH)) {
577               mustBe(LPAREN);
578               params.add(formalParameter());
579               mustBe(RPAREN);
580               catchBlocks.add(block());
581             }
582           }
583           if (have(FINALLY)) {
584             return new JTryStatement(line, tryBlock, params, catchBlocks, block());
585           }
```

```
586             return new JTryStatement(line, tryBlock, params, catchBlocks, null);
587         } else if (have(WHILE)) {
588             JExpression test = parExpression();
589             JStatement statement = statement();
590             return new JWhileStatement(line, test, statement);
591         } else {
592             // Must be a statementExpression.
593             JStatement statement = statementExpression();
594             mustBe(SEMI);
595             return statement;
596         }
597     }
598 
599     /**
600      * Parses and returns a list of formal parameters.
601      *
602      * <pre>
603      *   formalParameters ::= LPAREN [ formalParameter { COMMA formalParameter } ] RPAREN
604      * </pre>
605      *
606      * @return a list of formal parameters.
607      */
608     private ArrayList<JFormalParameter> formalParameters() {
609         ArrayList<JFormalParameter> parameters = new ArrayList<JFormalParameter>();
610         mustBe(LPAREN);
611         if (have(RPAREN)) {
612             return parameters;
613         }
614         do {
615             parameters.add(formalParameter());
616         } while (have(COMMA));
617         mustBe(RPAREN);
618         return parameters;
619     }
620 
621     /**
622      * Parses a formal parameter and returns an AST for it.
623      *
624      * <pre>
625      *   formalParameter ::= type IDENTIFIER
626      * </pre>
627      *
628      * @return an AST for a formal parameter.
629      */
630     private JFormalParameter formalParameter() {
631         int line = scanner.token().line();
632         Type type = type();
633         mustBe(IDENTIFIER);
634         String name = scanner.previousToken().image();
```

```
635         return new JFormalParameter(line, name, type);
636     }
637 
638     /**
639      * Parses a parenthesized expression and returns an AST for it.
640      *
641      * <pre>
642      *   parExpression ::= LPAREN expression RPAREN
643      * </pre>
644      *
645      * @return an AST for a parenthesized expression.
646      */
647     private JExpression parExpression() {
648         mustBe(LPAREN);
649         JExpression expr = expression();
650         mustBe(RPAREN);
651         return expr;
652     }
653 
654     /**
655      * Parses the initializations for a for loop.
656      *
657      * <pre>
658      *   forInit ::= statementExpression { COMMA statementExpression }
659      *             | type variableDeclarators
660      * </pre>
661      *
662      * @return a list of initialization statements.
663      */
664     private ArrayList<JStatement> forInit() {
665         ArrayList<JStatement> init = new ArrayList<JStatement>();
666         if (!seeLocalVariableDeclaration()) {
667             do {
668                 init.add(statementExpression());
669             } while (have(COMMA));
670             return init;
671         }
672         ArrayList<JVariableDeclarator> vdecls = variableDeclarators(type());
673         init.add(new JVariableDeclaration(scanner.token().line(), vdecls));
674         return init;
675     }
676 
677     /**
678      * Parses the update expressions for a for loop.
679      *
680      * <pre>
681      *   forUpdate ::= statementExpression { COMMA statementExpression }
682      * </pre>
683      *
```

```
684      * @return a list of update statements.
685      */
686     private ArrayList<JStatement> forUpdate() {
687         ArrayList<JStatement> updates = new ArrayList<JStatement>();
688         do {
689             updates.add(statementExpression());
690         } while (have(COMMA));
691         return updates;
692     }
693 
694     /**
695      * Parses a switch block statement group.
696      *
697      * <pre>
698      *   switchBlockStatementGroup ::= switchLabel { switchLabel } { blockStatement }
699      * </pre>
700      *
701      * @return an object that contains the labels and block statements.
702      */
703     private SwitchStatementGroup switchBlockStatementGroup() {
704         ArrayList<JExpression> switchLabels = new ArrayList<JExpression>();
705         do {
706             switchLabels.add(switchLabel());
707         } while (see(CASE) || see(DEFLT));
708         ArrayList<JStatement> blockStatements = new ArrayList<JStatement>();
709         while (!see(CASE) && !see(DEFLT) && !see(RCURLY)) {
710             blockStatements.add(blockStatement());
711         }
712         return new SwitchStatementGroup(switchLabels, blockStatements);
713     }
714 
715     /**
716      * Parses a switch label.
717      *
718      * <pre>
719      *   switchLabel ::= CASE expression COLON
720      *                 | DEFLT COLON
721      * </pre>
722      *
723      * @return the expression for the case label or null for the default label.
724      */
725     private JExpression switchLabel() {
726         JExpression expr = null;
727         if (have(CASE)) {
728             expr = expression();
729             mustBe(COLON);
730         } else if (have(DEFLT)) {
731             mustBe(COLON);
732         } else {
```

```
733             reportParserError("case, default, or '}' expected");
734         }
735         return expr;
736     }
737 
738     /**
739      * Parses a local variable declaration statement and returns an AST for it.
740      *
741      * <pre>
742      *   localVariableDeclarationStatement ::= type variableDeclarators SEMI
743      * </pre>
744      *
745      * @return an AST for a local variable declaration statement.
746      */
747     private JVariableDeclaration localVariableDeclarationStatement() {
748         int line = scanner.token().line();
749         Type type = type();
750         ArrayList<JVariableDeclarator> vdecls = variableDeclarators(type);
751         mustBe(SEMI);
752         return new JVariableDeclaration(line, vdecls);
753     }
754 
755     /**
756      * Parses and returns a list of variable declarators.
757      *
758      * <pre>
759      *   variableDeclarators ::= variableDeclarator { COMMA variableDeclarator }
760      * </pre>
761      *
762      * @param type type of the variables.
763      * @return a list of variable declarators.
764      */
765     private ArrayList<JVariableDeclarator> variableDeclarators(Type type) {
766         ArrayList<JVariableDeclarator> variableDeclarators = new ArrayList<JVariableDeclarator>();
767         do {
768             variableDeclarators.add(variableDeclarator(type));
769         } while (have(COMMA));
770         return variableDeclarators;
771     }
772 
773     /**
774      * Parses a variable declarator and returns an AST for it.
775      *
776      * <pre>
777      *   variableDeclarator ::= IDENTIFIER [ ASSIGN variableInitializer ]
778      * </pre>
779      *
780      * @param type type of the variable.
781      * @return an AST for a variable declarator.
```

```
782      */
783     private JVariableDeclarator variableDeclarator(Type type) {
784         int line = scanner.token().line();
785         mustBe(IDENTIFIER);
786         String name = scanner.previousToken().image();
787         JExpression initial = have(ASSIGN) ? variableInitializer(type) : null;
788         return new JVariableDeclarator(line, name, type, initial);
789     }
790 
791     /**
792      * Parses a variable initializer and returns an AST for it.
793      *
794      * <pre>
795      *   variableInitializer ::= arrayInitializer | expression
796      * </pre>
797      *
798      * @param type type of the variable.
799      * @return an AST for a variable initializer.
800      */
801     private JExpression variableInitializer(Type type) {
802         if (see(LCURLY)) {
803             return arrayInitializer(type);
804         }
805         return expression();
806     }
807 
808     /**
809      * Parses an array initializer and returns an AST for it.
810      *
811      * <pre>
812      *   arrayInitializer ::= LCURLY [ variableInitializer { COMMA variableInitializer }
813      *                                 [ COMMA ] ] RCURLY
814      * </pre>
815      *
816      * @param type type of the array.
817      * @return an AST for an array initializer.
818      */
819     private JArrayInitializer arrayInitializer(Type type) {
820         int line = scanner.token().line();
821         ArrayList<JExpression> initials = new ArrayList<JExpression>();
822         mustBe(LCURLY);
823         if (have(RCURLY)) {
824             return new JArrayInitializer(line, type, initials);
825         }
826         initials.add(variableInitializer(type.componentType()));
827         while (have(COMMA)) {
828             initials.add(see(RCURLY) ? null : variableInitializer(type.componentType()));
829         }
830         mustBe(RCURLY);
```

```
831         return new JArrayInitializer(line, type, initials);
832     }
833 
834     /**
835      * Parses and returns a list of arguments.
836      *
837      * <pre>
838      *   arguments ::= LPAREN [ expression { COMMA expression } ] RPAREN
839      * </pre>
840      *
841      * @return a list of arguments.
842      */
843     private ArrayList<JExpression> arguments() {
844         ArrayList<JExpression> args = new ArrayList<JExpression>();
845         mustBe(LPAREN);
846         if (have(RPAREN)) {
847             return args;
848         }
849         do {
850             args.add(expression());
851         } while (have(COMMA));
852         mustBe(RPAREN);
853         return args;
854     }
855 
856     /**
857      * Parses and returns a type.
858      *
859      * <pre>
860      *   type ::= referenceType | basicType
861      * </pre>
862      *
863      * @return a type.
864      */
865     private Type type() {
866         if (seeReferenceType()) {
867             return referenceType();
868         }
869         return basicType();
870     }
871 
872     /**
873      * Parses and returns a basic type.
874      *
875      * <pre>
876      *   basicType ::= BOOLEAN | CHAR | DOUBLE | INT | LONG
877      * </pre>
878      *
879      * @return a basic type.
```

```
880      */
881     private Type basicType() {
882         if (have(BOOLEAN)) {
883             return Type.BOOLEAN;
884         } else if (have(CHAR)) {
885             return Type.CHAR;
886         } else if (have(DOUBLE)) {
887             return Type.DOUBLE;
888         } else if (have(INT)) {
889             return Type.INT;
890         } else if (have(LONG)) {
891             return Type.LONG;
892         } else {
893             reportParserError("Type sought where %s found", scanner.token().image());
894             return Type.ANY;
895         }
896     }
897 
898     /**
899      * Parses and returns a reference type.
900      *
901      * <pre>
902      *   referenceType ::= basicType LBRACK RBRACK { LBRACK RBRACK }
903      *                   | qualifiedIdentifier { LBRACK RBRACK }
904      * </pre>
905      *
906      * @return a reference type.
907      */
908     private Type referenceType() {
909         Type type = null;
910         if (!see(IDENTIFIER)) {
911             type = basicType();
912             mustBe(LBRACK);
913             mustBe(RBRACK);
914             type = new ArrayTypeName(type);
915         } else {
916             type = qualifiedIdentifier();
917         }
918         while (seeDims()) {
919             mustBe(LBRACK);
920             mustBe(RBRACK);
921             type = new ArrayTypeName(type);
922         }
923         return type;
924     }
925 
926     /**
927      * Parses a statement expression and returns an AST for it.
928      *
```

```
929     * <pre>
930     *   statementExpression ::= expression
931     * </pre>
932     *
933     * @return an AST for a statement expression.
934     */
935    private JStatement statementExpression() {
936      int line = scanner.token().line();
937      JExpression expr = expression();
938      if (expr instanceof JAssignment
939              || expr instanceof JPreIncrementOp
940              || expr instanceof JPreDecrementOp
941              || expr instanceof JPostIncrementOp
942              || expr instanceof JPostDecrementOp
943              || expr instanceof JMessageExpression
944              || expr instanceof JSuperConstruction
945              || expr instanceof JThisConstruction
946              || expr instanceof JNewOp
947              || expr instanceof JNewArrayOp) {
948          // So as not to save on stack.
949          expr.isStatementExpression = true;
950      } else {
951          reportParserError("Invalid statement expression; it does not have a side-effect");
952      }
953      return new JStatementExpression(line, expr);
954    }
955
956    /**
957     * Parses an expression and returns an AST for it.
958     *
959     * <pre>
960     *   expression ::= assignmentExpression
961     * </pre>
962     *
963     * @return an AST for an expression.
964     */
965    private JExpression expression() {
966      return assignmentExpression();
967    }
968
969    /**
970     * Parses an assignment expression and returns an AST for it.
971     *
972     * <pre>
973     *   assignmentExpression ::= conditionalExpression
974     *                          [ ( ALSHIFT_ASSIGN | AND_ASSIGN | ARSHIFT_ASSIGN | ASSIGN
975     *                            | DIV_ASSIGN | LRSHIFT_ASSIGN | MINUS_ASSIGN | OR_ASSIGN
976     *                            | PLUS_ASSIGN | REM_ASSIGN | STAR_ASSIGN | XOR_ASSIGN )
977     *                             assignmentExpression ]
```

```
978       * </pre>
979       *
980       * @return an AST for an assignment expression.
981       */
982      private JExpression assignmentExpression() {
983        int line = scanner.token().line();
984        JExpression lhs = conditionalExpression();
985        if (have(ALSHIFT_ASSIGN)) {
986          return new JALeftShiftAssignOp(line, lhs, assignmentExpression());
987        } else if (have(AND_ASSIGN)) {
988          return new JAndAssignOp(line, lhs, assignmentExpression());
989        } else if (have(ARSHIFT_ASSIGN)) {
990          return new JARightShiftAssignOp(line, lhs, assignmentExpression());
991        } else if (have(ASSIGN)) {
992          return new JAssignOp(line, lhs, assignmentExpression());
993        } else if (have(DIV_ASSIGN)) {
994          return new JDivAssignOp(line, lhs, assignmentExpression());
995        } else if (have(LRSHIFT_ASSIGN)) {
996          return new JLRightShiftAssignOp(line, lhs, assignmentExpression());
997        } else if (have(MINUS_ASSIGN)) {
998          return new JMinusAssignOp(line, lhs, assignmentExpression());
999        } else if (have(OR_ASSIGN)) {
1000         return new JOrAssignOp(line, lhs, assignmentExpression());
1001       } else if (have(PLUS_ASSIGN)) {
1002         return new JPlusAssignOp(line, lhs, assignmentExpression());
1003       } else if (have(REM_ASSIGN)) {
1004         return new JRemAssignOp(line, lhs, assignmentExpression());
1005       } else if (have(STAR_ASSIGN)) {
1006         return new JStarAssignOp(line, lhs, assignmentExpression());
1007       } else if (have(XOR_ASSIGN)) {
1008         return new JXorAssignOp(line, lhs, assignmentExpression());
1009       } else {
1010         return lhs;
1011       }
1012     }
1013
1014     /**
1015      * Parses a conditional expression and return an AST for it.
1016      *
1017      * <pre>
1018      *   conditionalExpression ::= conditionalOrExpression [ QUESTION expression COLON
1019      *                                                       conditionalExpression ]
1020      * </pre>
1021      *
1022      * @return an AST for a conditional expression.
1023      */
1024     private JExpression conditionalExpression() {
1025       int line = scanner.token().line();
1026       JExpression expr = conditionalOrExpression();
```

```
1027        if (have(QUESTION)) {
1028            JExpression thenPart = expression();
1029            mustBe(COLON);
1030            JExpression elsePart = conditionalExpression();
1031            return new JConditionalExpression(line, expr, thenPart, elsePart);
1032        }
1033        return expr;
1034    }
1035
1036    /**
1037     * Parses a conditional-or expression and returns an AST for it.
1038     *
1039     * <pre>
1040     *   conditionalOrExpression ::= conditionalAndExpression { LOR conditionalAndExpression }
1041     * </pre>
1042     *
1043     * @return an AST for a conditional-and expression.
1044     */
1045    private JExpression conditionalOrExpression() {
1046        int line = scanner.token().line();
1047        boolean more = true;
1048        JExpression lhs = conditionalAndExpression();
1049        while (more) {
1050            if (have(LOR)) {
1051                lhs = new JLogicalOrOp(line, lhs, conditionalAndExpression());
1052            } else {
1053                more = false;
1054            }
1055        }
1056        return lhs;
1057    }
1058
1059    /**
1060     * Parses a conditional-and expression and returns an AST for it.
1061     *
1062     * <pre>
1063     *   conditionalAndExpression ::= inclusiveOrExpression { LAND inclusiveOrExpression }
1064     * </pre>
1065     *
1066     * @return an AST for a conditional-and expression.
1067     */
1068    private JExpression conditionalAndExpression() {
1069        int line = scanner.token().line();
1070        boolean more = true;
1071        JExpression lhs = inclusiveOrExpression();
1072        while (more) {
1073            if (have(LAND)) {
1074                lhs = new JLogicalAndOp(line, lhs, inclusiveOrExpression());
1075            } else {
```

```
1076                more = false;
1077            }
1078        }
1079        return lhs;
1080    }
1081
1082    /**
1083     * Parses an inclusive-or expression and returns an AST for it.
1084     *
1085     * <pre>
1086     *   inclusiveOrExpression ::= exclusiveOrExpression { OR exclusiveOrExpression }
1087     * </pre>
1088     *
1089     * @return an AST for an inclusive-or expression.
1090     */
1091    private JExpression inclusiveOrExpression() {
1092        int line = scanner.token().line();
1093        boolean more = true;
1094        JExpression lhs = exclusiveOrExpression();
1095        while (more) {
1096            if (have(OR)) {
1097                lhs = new JOrOp(line, lhs, exclusiveOrExpression());
1098            } else {
1099                more = false;
1100            }
1101        }
1102        return lhs;
1103    }
1104
1105    /**
1106     * Parses an exclusive-or expression and returns an AST for it.
1107     *
1108     * <pre>
1109     *   exclusiveOrExpression ::= andExpression { XOR andExpression }
1110     * </pre>
1111     *
1112     * @return an AST for a exclusive-or expression.
1113     */
1114    private JExpression exclusiveOrExpression() {
1115        int line = scanner.token().line();
1116        boolean more = true;
1117        JExpression lhs = andExpression();
1118        while (more) {
1119            if (have(XOR)) {
1120                lhs = new JXorOp(line, lhs, andExpression());
1121            } else {
1122                more = false;
1123            }
1124        }
```

```
1125        return lhs;
1126    }
1127
1128    /**
1129     * Parses an and expression and returns an AST for it.
1130     *
1131     * <pre>
1132     *   andExpression ::= equalityExpression { AND equalityExpression }
1133     * </pre>
1134     *
1135     * @return an AST for an and expression.
1136     */
1137    private JExpression andExpression() {
1138        int line = scanner.token().line();
1139        boolean more = true;
1140        JExpression lhs = equalityExpression();
1141        while (more) {
1142            if (have(AND)) {
1143                lhs = new JAndOp(line, lhs, equalityExpression());
1144            } else {
1145                more = false;
1146            }
1147        }
1148        return lhs;
1149    }
1150
1151    /**
1152     * Parses an equality expression and returns an AST for it.
1153     *
1154     * <pre>
1155     *   equalityExpression ::= relationalExpression { ( EQUAL | NOT_EQUAL ) relationalExpression
1156     * </pre>
1157     *
1158     * @return an AST for an equality expression.
1159     */
1160    private JExpression equalityExpression() {
1161        int line = scanner.token().line();
1162        boolean more = true;
1163        JExpression lhs = relationalExpression();
1164        while (more) {
1165            if (have(EQUAL)) {
1166                lhs = new JEqualOp(line, lhs, relationalExpression());
1167            } else if (have(NOT_EQUAL)) {
1168                lhs = new JNotEqualOp(line, lhs, relationalExpression());
1169            } else {
1170                more = false;
1171            }
1172        }
1173        return lhs;
```

```
1174    }
1175
1176    /**
1177     * Parses a relational expression and returns an AST for it.
1178     *
1179     * <pre>
1180     *   relationalExpression ::= shiftExpression [ ( GE | GT | LE | LT ) shiftExpression
1181     *                                    | INSTANCEOF referenceType ]
1182     * </pre>
1183     *
1184     * @return an AST for a relational expression.
1185     */
1186    private JExpression relationalExpression() {
1187        int line = scanner.token().line();
1188        JExpression lhs = shiftExpression();
1189        if (have(GE)) {
1190            return new JGreaterEqualOp(line, lhs, shiftExpression());
1191        } else if (have(GT)) {
1192            return new JGreaterThanOp(line, lhs, shiftExpression());
1193        } else if (have(LE)) {
1194            return new JLessEqualOp(line, lhs, shiftExpression());
1195        } else if (have(LT)) {
1196            return new JLessThanOp(line, lhs, shiftExpression());
1197        } else if (have(INSTANCEOF)) {
1198            return new JInstanceOfOp(line, lhs, referenceType());
1199        } else {
1200            return lhs;
1201        }
1202    }
1203
1204    /**
1205     * Parses a shift expression and returns an AST for it.
1206     *
1207     * <pre>
1208     *   shiftExpression ::= additiveExpression
1209     *                       { ( ALSHIFT | ARSHIFT | LRSHIFT ) additiveExpression }
1210     * </pre>
1211     *
1212     * @return an AST for a shift expression.
1213     */
1214    private JExpression shiftExpression() {
1215        int line = scanner.token().line();
1216        boolean more = true;
1217        JExpression lhs = additiveExpression();
1218        while (more) {
1219            if (have(ALSHIFT)) {
1220                lhs = new JALeftShiftOp(line, lhs, additiveExpression());
1221            } else if (have(ARSHIFT)) {
1222                lhs = new JARightShiftOp(line, lhs, additiveExpression());
```

```
1223            } else if (have(LRSHIFT)) {
1224                lhs = new JLRightShiftOp(line, lhs, additiveExpression());
1225            } else {
1226                more = false;
1227            }
1228        }
1229        return lhs;
1230    }
1231
1232    /**
1233     * Parses an additive expression and returns an AST for it.
1234     *
1235     * <pre>
1236     *   additiveExpression ::= multiplicativeExpression
1237     *                          { ( MINUS | PLUS ) multiplicativeExpression }
1238     * </pre>
1239     *
1240     * @return an AST for an additive expression.
1241     */
1242    private JExpression additiveExpression() {
1243        int line = scanner.token().line();
1244        boolean more = true;
1245        JExpression lhs = multiplicativeExpression();
1246        while (more) {
1247            if (have(MINUS)) {
1248                lhs = new JSubtractOp(line, lhs, multiplicativeExpression());
1249            } else if (have(PLUS)) {
1250                lhs = new JPlusOp(line, lhs, multiplicativeExpression());
1251            } else {
1252                more = false;
1253            }
1254        }
1255        return lhs;
1256    }
1257
1258    /**
1259     * Parses a multiplicative expression and returns an AST for it.
1260     *
1261     * <pre>
1262     *   multiplicativeExpression ::= unaryExpression { ( DIV | REM | STAR ) unaryExpression }
1263     * </pre>
1264     *
1265     * @return an AST for a multiplicative expression.
1266     */
1267    private JExpression multiplicativeExpression() {
1268        int line = scanner.token().line();
1269        boolean more = true;
1270        JExpression lhs = unaryExpression();
1271        while (more) {
```

```
1272            if (have(DIV)) {
1273                lhs = new JDivideOp(line, lhs, unaryExpression());
1274            } else if (have(REM)) {
1275                lhs = new JRemainderOp(line, lhs, unaryExpression());
1276            } else if (have(STAR)) {
1277                lhs = new JMultiplyOp(line, lhs, unaryExpression());
1278            } else {
1279                more = false;
1280            }
1281        }
1282        return lhs;
1283    }
1284
1285    /**
1286     * Parses an unary expression and returns an AST for it.
1287     *
1288     * <pre>
1289     *   unaryExpression ::= DEC unaryExpression
1290     *                     | INC unaryExpression
1291     *                     | ( MINUS | PLUS ) unaryExpression
1292     *                     | simpleUnaryExpression
1293     * </pre>
1294     *
1295     * @return an AST for an unary expression.
1296     */
1297    private JExpression unaryExpression() {
1298        int line = scanner.token().line();
1299        if (have(DEC)) {
1300            return new JPreDecrementOp(line, unaryExpression());
1301        } else if (have(INC)) {
1302            return new JPreIncrementOp(line, unaryExpression());
1303        } else if (have(MINUS)) {
1304            return new JNegateOp(line, unaryExpression());
1305        } else if (have(PLUS)) {
1306            return new JUnaryPlusOp(line, unaryExpression());
1307        } else {
1308            return simpleUnaryExpression();
1309        }
1310    }
1311
1312    /**
1313     * Parses a simple unary expression and returns an AST for it.
1314     *
1315     * <pre>
1316     *   simpleUnaryExpression ::= LNOT unaryExpression
1317     *                           | NOT unaryExpression
1318     *                           | LPAREN basicType RPAREN unaryExpression
1319     *                           | LPAREN referenceType RPAREN simpleUnaryExpression
1320     *                           | postfixExpression
```

```
1321      * </pre>
1322      *
1323      * @return an AST for a simple unary expression.
1324      */
1325     private JExpression simpleUnaryExpression() {
1326         int line = scanner.token().line();
1327         if (have(LNOT)) {
1328             return new JLogicalNotOp(line, unaryExpression());
1329         } else if (have(NOT)) {
1330             return new JComplementOp(line, unaryExpression());
1331         } else if (seeCast()) {
1332             mustBe(LPAREN);
1333             boolean isBasicType = seeBasicType();
1334             Type type = type();
1335             mustBe(RPAREN);
1336             JExpression expr = isBasicType ? unaryExpression() : simpleUnaryExpression();
1337             return new JCastOp(line, type, expr);
1338         } else {
1339             return postfixExpression();
1340         }
1341     }
1342
1343     /**
1344      * Parses a postfix expression and returns an AST for it.
1345      *
1346      * <pre>
1347      *   postfixExpression ::= primary { selector } { DEC | INC }
1348      * </pre>
1349      *
1350      * @return an AST for a postfix expression.
1351      */
1352     private JExpression postfixExpression() {
1353         int line = scanner.token().line();
1354         JExpression primaryExpr = primary();
1355         while (see(DOT) || see(LBRACK)) {
1356             primaryExpr = selector(primaryExpr);
1357         }
1358         boolean more = true;
1359         while (more) {
1360             if (have(DEC)) {
1361                 primaryExpr = new JPostDecrementOp(line, primaryExpr);
1362             } else if (have(INC)) {
1363                 primaryExpr = new JPostIncrementOp(line, primaryExpr);
1364             } else {
1365                 more = false;
1366             }
1367         }
1368         return primaryExpr;
1369     }
```

```
1370
1371    /**
1372     * Parses a selector and returns an AST for it.
1373     *
1374     * <pre>
1375     *   selector ::= DOT qualifiedIdentifier [ arguments ]
1376     *              | LBRACK expression RBRACK
1377     * </pre>
1378     *
1379     * @param target the target expression for this selector.
1380     * @return an AST for a selector.
1381     */
1382    private JExpression selector(JExpression target) {
1383      int line = scanner.token().line();
1384      if (have(DOT)) {
1385        // target.selector.
1386        mustBe(IDENTIFIER);
1387        String name = scanner.previousToken().image();
1388        if (see(LPAREN)) {
1389          ArrayList<JExpression> args = arguments();
1390          return new JMessageExpression(line, target, name, args);
1391        } else {
1392          return new JFieldSelection(line, target, name);
1393        }
1394      } else {
1395        mustBe(LBRACK);
1396        JExpression index = expression();
1397        mustBe(RBRACK);
1398        return new JArrayExpression(line, target, index);
1399      }
1400    }
1401
1402    /**
1403     * Parses a primary expression and returns an AST for it.
1404     *
1405     * <pre>
1406     *   primary ::= parExpression
1407     *             | NEW creator
1408     *             | THIS [ arguments ]
1409     *             | SUPER ( arguments | DOT IDENTIFIER [ arguments ] )
1410     *             | qualifiedIdentifier [ arguments ]
1411     *             | literal
1412     * </pre>
1413     *
1414     * @return an AST for a primary expression.
1415     */
1416    private JExpression primary() {
1417      int line = scanner.token().line();
1418      if (see(LPAREN)) {
```

```
1419            return parExpression();
1420        } else if (have(NEW)) {
1421            return creator();
1422        } else if (have(THIS)) {
1423            if (see(LPAREN)) {
1424                ArrayList<JExpression> args = arguments();
1425                return new JThisConstruction(line, args);
1426            } else {
1427                return new JThis(line);
1428            }
1429        } else if (have(SUPER)) {
1430            if (!have(DOT)) {
1431                ArrayList<JExpression> args = arguments();
1432                return new JSuperConstruction(line, args);
1433            } else {
1434                mustBe(IDENTIFIER);
1435                String name = scanner.previousToken().image();
1436                JExpression newTarget = new JSuper(line);
1437                if (see(LPAREN)) {
1438                    ArrayList<JExpression> args = arguments();
1439                    return new JMessageExpression(line, newTarget, null, name, args);
1440                } else {
1441                    return new JFieldSelection(line, newTarget, name);
1442                }
1443            }
1444        } else if (see(IDENTIFIER)) {
1445            TypeName id = qualifiedIdentifier();
1446            if (see(LPAREN)) {
1447                // ambiguousPart.messageName(...).
1448                ArrayList<JExpression> args = arguments();
1449                return new JMessageExpression(line, null, ambiguousPart(id), id.simpleName(), args);
1450            } else if (ambiguousPart(id) == null) {
1451                // A simple name.
1452                return new JVariable(line, id.simpleName());
1453            } else {
1454                // ambiguousPart.fieldName.
1455                return new JFieldSelection(line, ambiguousPart(id), null, id.simpleName());
1456            }
1457        } else {
1458            return literal();
1459        }
1460    }
1461
1462    /**
1463     * Parses a creator and returns an AST for it.
1464     *
1465     * <pre>
1466     *   creator ::= ( basicType | qualifiedIdentifier )
1467     *                 ( arguments
```

```
1468      *                 | LBRACK RBRACK { LBRACK RBRACK } [ arrayInitializer ]
1469      *                 | newArrayDeclarator
1470      *                )
1471      * </pre>
1472      *
1473      * @return an AST for a creator.
1474      */
1475     private JExpression creator() {
1476       int line = scanner.token().line();
1477       Type type = seeBasicType() ? basicType() : qualifiedIdentifier();
1478       if (see(LPAREN)) {
1479         ArrayList<JExpression> args = arguments();
1480         return new JNewOp(line, type, args);
1481       } else if (see(LBRACK)) {
1482         if (seeDims()) {
1483           Type expected = type;
1484           while (have(LBRACK)) {
1485             mustBe(RBRACK);
1486             expected = new ArrayTypeName(expected);
1487           }
1488           return arrayInitializer(expected);
1489         } else {
1490           return newArrayDeclarator(line, type);
1491         }
1492       } else {
1493         reportParserError("( or [ sought where %s found", scanner.token().image());
1494         return new JWildExpression(line);
1495       }
1496     }
1497
1498     /**
1499      * Parses a new array declarator and returns an AST for it.
1500      *
1501      * <pre>
1502      *   newArrayDeclarator ::= LBRACK expression RBRACK
1503      *                          { LBRACK expression RBRACK } { LBRACK RBRACK }
1504      * </pre>
1505      *
1506      * @param line line in which the declarator occurred.
1507      * @param type type of the array.
1508      * @return an AST for a new array declarator.
1509      */
1510     private JNewArrayOp newArrayDeclarator(int line, Type type) {
1511       ArrayList<JExpression> dimensions = new ArrayList<JExpression>();
1512       mustBe(LBRACK);
1513       dimensions.add(expression());
1514       mustBe(RBRACK);
1515       type = new ArrayTypeName(type);
1516       while (have(LBRACK)) {
```

```
1517            if (have(RBRACK)) {
1518                // We're done with dimension expressions.
1519                type = new ArrayTypeName(type);
1520                while (have(LBRACK)) {
1521                    mustBe(RBRACK);
1522                    type = new ArrayTypeName(type);
1523                }
1524                return new JNewArrayOp(line, type, dimensions);
1525            } else {
1526                dimensions.add(expression());
1527                type = new ArrayTypeName(type);
1528                mustBe(RBRACK);
1529            }
1530        }
1531        return new JNewArrayOp(line, type, dimensions);
1532    }
1533
1534    /**
1535     * Parses a literal and returns an AST for it.
1536     *
1537     * <pre>
1538     *   literal ::= CHAR_LITERAL | DOUBLE_LITERAL | FALSE | INT_LITERAL | LONG_LITERAL | NULL
1539     *             | STRING_LITERAL | TRUE
1540     * </pre>
1541     *
1542     * @return an AST for a literal.
1543     */
1544    private JExpression literal() {
1545        int line = scanner.token().line();
1546        if (have(CHAR_LITERAL)) {
1547            return new JLiteralChar(line, scanner.previousToken().image());
1548        } else if (have(DOUBLE_LITERAL)) {
1549            return new JLiteralDouble(line, scanner.previousToken().image());
1550        } else if (have(FALSE)) {
1551            return new JLiteralBoolean(line, scanner.previousToken().image());
1552        } else if (have(INT_LITERAL)) {
1553            return new JLiteralInt(line, scanner.previousToken().image());
1554        } else if (have(LONG_LITERAL)) {
1555            return new JLiteralLong(line, scanner.previousToken().image());
1556        } else if (have(NULL)) {
1557            return new JLiteralNull(line);
1558        } else if (have(STRING_LITERAL)) {
1559            return new JLiteralString(line, scanner.previousToken().image());
1560        } else if (have(TRUE)) {
1561            return new JLiteralBoolean(line, scanner.previousToken().image());
1562        } else {
1563            reportParserError("Literal sought where %s found", scanner.token().image());
1564            return new JWildExpression(line);
1565        }
```

```
1566    }
1567
1568    ////////////////////////////////////////////////
1569    // Parsing Support
1570    // ////////////////////////////////////////////////
1571
1572    // Returns true if the current token equals sought, and false otherwise.
1573    private boolean see(TokenKind sought) {
1574        return (sought == scanner.token().kind());
1575    }
1576
1577    // If the current token equals sought, scans it and returns true. Otherwise, returns false
1578    // without scanning the token.
1579    private boolean have(TokenKind sought) {
1580        if (see(sought)) {
1581            scanner.next();
1582            return true;
1583        } else {
1584            return false;
1585        }
1586    }
1587
1588    // Attempts to match a token we're looking for with the current input token. On success,
1589    // scans the token and goes into a "Recovered" state. On failure, what happens next depends
1590    // on whether or not the parser is currently in a "Recovered" state: if so, it reports the
1591    // error and goes into an "Unrecovered" state; if not, it repeatedly scans tokens until it
1592    // finds the one it is looking for (or EOF) and then returns to a "Recovered" state. This
1593    // gives us a kind of poor man's syntactic error recovery, a strategy due to David Turner and
1594    // Ron Morrison.
1595    private void mustBe(TokenKind sought) {
1596        if (scanner.token().kind() == sought) {
1597            scanner.next();
1598            isRecovered = true;
1599        } else if (isRecovered) {
1600            isRecovered = false;
1601            reportParserError("%s found where %s sought", scanner.token().image(), sought.image());
1602        } else {
1603            // Do not report the (possibly spurious) error, but rather attempt to recover by
1604            // forcing a match.
1605            while (!see(sought) && !see(EOF)) {
1606                scanner.next();
1607            }
1608            if (see(sought)) {
1609                scanner.next();
1610                isRecovered = true;
1611            }
1612        }
1613    }
1614
```

```
1615    // Pulls out and returns the ambiguous part of a name.
1616    private AmbiguousName ambiguousPart(TypeName name) {
1617        String qualifiedName = name.toString();
1618        int i = qualifiedName.lastIndexOf('.');
1619        return i == -1 ? null : new AmbiguousName(name.line(), qualifiedName.substring(0, i));
1620    }
1621
1622    // Reports a syntax error.
1623    private void reportParserError(String message, Object... args) {
1624        isInError = true;
1625        isRecovered = false;
1626        System.err.printf("%s:%d: error: ", scanner.fileName(), scanner.token().line());
1627        System.err.printf(message, args);
1628        System.err.println();
1629    }
1630
1631    //////////////////////////////////////////////////
1632    // Lookahead Methods
1633    //////////////////////////////////////////////////
1634
1635    // Returns true if we are looking at an IDENTIFIER followed by a LPAREN, and false otherwise.
1636    private boolean seeIdentLParen() {
1637        scanner.recordPosition();
1638        boolean result = have(IDENTIFIER) && see(LPAREN);
1639        scanner.returnToPosition();
1640        return result;
1641    }
1642
1643    // Returns true if we are looking at a cast (basic or reference), and false otherwise.
1644    private boolean seeCast() {
1645        scanner.recordPosition();
1646        if (!have(LPAREN)) {
1647            scanner.returnToPosition();
1648            return false;
1649        }
1650        if (seeBasicType()) {
1651            scanner.returnToPosition();
1652            return true;
1653        }
1654        if (!see(IDENTIFIER)) {
1655            scanner.returnToPosition();
1656            return false;
1657        } else {
1658            scanner.next();
1659            // A qualified identifier is ok.
1660            while (have(DOT)) {
1661                if (!have(IDENTIFIER)) {
1662                    scanner.returnToPosition();
1663                    return false;
```

```
1664                }
1665            }
1666        }
1667        while (have(LBRACK)) {
1668            if (!have(RBRACK)) {
1669                scanner.returnToPosition();
1670                return false;
1671            }
1672        }
1673        if (!have(RPAREN)) {
1674            scanner.returnToPosition();
1675            return false;
1676        }
1677        scanner.returnToPosition();
1678        return true;
1679    }
1680
1681    // Returns true if we are looking at a local variable declaration, and false otherwise.
1682    private boolean seeLocalVariableDeclaration() {
1683        scanner.recordPosition();
1684        if (have(IDENTIFIER)) {
1685            // A qualified identifier is ok.
1686            while (have(DOT)) {
1687                if (!have(IDENTIFIER)) {
1688                    scanner.returnToPosition();
1689                    return false;
1690                }
1691            }
1692        } else if (seeBasicType()) {
1693            scanner.next();
1694        } else {
1695            scanner.returnToPosition();
1696            return false;
1697        }
1698        while (have(LBRACK)) {
1699            if (!have(RBRACK)) {
1700                scanner.returnToPosition();
1701                return false;
1702            }
1703        }
1704        if (!have(IDENTIFIER)) {
1705            scanner.returnToPosition();
1706            return false;
1707        }
1708        while (have(LBRACK)) {
1709            if (!have(RBRACK)) {
1710                scanner.returnToPosition();
1711                return false;
1712            }
```

```
1713        }
1714        scanner.returnToPosition();
1715        return true;
1716    }
1717
1718    // Returns true if we are looking at a basic type, and false otherwise.
1719    private boolean seeBasicType() {
1720        return (see(BOOLEAN) || see(CHAR) || see(INT) || see(LONG) || see(DOUBLE));
1721    }
1722
1723    // Returns true if we are looking at a reference type, and false otherwise.
1724    private boolean seeReferenceType() {
1725        if (see(IDENTIFIER)) {
1726            return true;
1727        } else {
1728            scanner.recordPosition();
1729            if (have(BOOLEAN) || have(CHAR) || have(INT) || have(DOUBLE) || have(LONG)) {
1730                if (have(LBRACK) && see(RBRACK)) {
1731                    scanner.returnToPosition();
1732                    return true;
1733                }
1734            }
1735            scanner.returnToPosition();
1736        }
1737        return false;
1738    }
1739
1740    // Returns true if we are looking at a [] pair, and false otherwise.
1741    private boolean seeDims() {
1742        scanner.recordPosition();
1743        boolean result = have(LBRACK) && see(RBRACK);
1744        scanner.returnToPosition();
1745        return result;
1746    }
1747 }
1748
```

JWhileStatement.java

Download

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

package jminusminus;

import static jminusminus.CLConstants.*;

/**
 * The AST node for a while-statement.
 */
class JWhileStatement extends JStatement {
    // Test expression.
    private JExpression condition;

    // Body.
    private JStatement body;

    /**
     * Constructs an AST node for a while-statement.
     *
     * @param line      line in which the while-statement occurs in the source file.
     * @param condition test expression.
     * @param body      the body.
     */
    public JWhileStatement(int line, JExpression condition, JStatement body) {
        super(line);
        this.condition = condition;
        this.body = body;
    }

    /**
     * {@inheritDoc}
     */
    public JWhileStatement analyze(Context context) {
        condition = condition.analyze(context);
        condition.type().mustMatchExpected(line(), Type.BOOLEAN);
        body = (JStatement) body.analyze(context);
        return this;
    }

    /**
     * {@inheritDoc}
     */
    public void codegen(CLEmitter output) {
        String test = output.createLabel();
        String out = output.createLabel();
        output.addLabel(test);
```

```
47          condition.codegen(output, out, false);
48          body.codegen(output);
49          output.addBranchInstruction(GOTO, test);
50          output.addLabel(out);
51      }
52
53      /**
54       * {@inheritDoc}
55       */
56      public void toJSON(JSONElement json) {
57          JSONElement e = new JSONElement();
58          json.addChild("JWhileStatement:" + line, e);
59          JSONElement e1 = new JSONElement();
60          e.addChild("Condition", e1);
61          condition.toJSON(e1);
62          JSONElement e2 = new JSONElement();
63          e.addChild("Body", e2);
64          body.toJSON(e2);
65      }
66  }
67
```

JVariableDeclaration.java

Download

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

package jminusminus;

import java.util.ArrayList;

/**
 * The AST node for a local variable declaration. Local variables are declared by its analyze()
 * method, which also re-writes any initializations as assignment statements, in turn generated
 * by its codegen() method.
 */
class JVariableDeclaration extends JStatement {
    // Variable declarators.
    private ArrayList<JVariableDeclarator> decls;

    // Variable initializers.
    private ArrayList<JStatement> initializations;

    /**
     * Constructs an AST node for a variable declaration.
     *
     * @param line  line in which the variable declaration occurs in the source file.
     * @param decls variable declarators.
     */
    public JVariableDeclaration(int line, ArrayList<JVariableDeclarator> decls) {
        super(line);
        this.decls = decls;
        initializations = new ArrayList<JStatement>();
    }

    /**
     * {@inheritDoc}
     */
    public JStatement analyze(Context context) {
        for (JVariableDeclarator decl : decls) {
            // Local variables are declared here (fields are declared in preAnalyze()).
            int offset = ((LocalContext) context).nextOffset();

            LocalVariableDefn defn = new LocalVariableDefn(decl.type().resolve(context), offset);

            // First, check for shadowing.
            IDefn previousDefn = context.lookup(decl.name());
            if (previousDefn != null && previousDefn instanceof LocalVariableDefn) {
                JAST.compilationUnit.reportSemanticError(decl.line(),
                        "The name " + decl.name() + " overshadows another local variable");
            }
```

```
47
48          // Then declare it in the local context.
49          context.addEntry(decl.line(), decl.name(), defn);
50
51          if (decl.type() == Type.LONG || decl.type() == Type.DOUBLE) {
52              ((LocalContext) context).nextOffset();
53          }
54
55          // All initializations must be turned into assignment statements and analyzed.
56          if (decl.initializer() != null) {
57              defn.initialize();
58              JAssignOp assignOp = new JAssignOp(decl.line(), new JVariable(decl.line(),
59                      decl.name()), decl.initializer());
60              assignOp.isStatementExpression = true;
61              initializations.add(new JStatementExpression(decl.line(),
62                      assignOp).analyze(context));
63          }
64      }
65      return this;
66  }
67
68  /**
69   * {@inheritDoc}
70   */
71  public void codegen(CLEmitter output) {
72      for (JStatement initialization : initializations) {
73          initialization.codegen(output);
74      }
75  }
76
77  /**
78   * {@inheritDoc}
79   */
80  public void toJSON(JSONElement json) {
81      JSONElement e = new JSONElement();
82      json.addChild("JVariableDeclaration:" + line, e);
83      if (decls != null) {
84          for (JVariableDeclarator decl : decls) {
85              decl.toJSON(e);
86          }
87      }
88  }
89  }
90
```

JVariable.java

Download

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

package jminusminus;

import static jminusminus.CLConstants.*;

/**
 * The AST node for an identifier used as a primary expression.
 */
class JVariable extends JExpression implements JLhs {
    // The variable's name.
    private String name;

    // The variable's definition.
    private IDefn iDefn;

    // Was analyzeLhs() done?
    private boolean analyzeLhs;

    /**
     * Constructs the AST node for a variable.
     *
     * @param line line in which the variable occurs in the source file.
     * @param name the name.
     */
    public JVariable(int line, String name) {
        super(line);
        this.name = name;
    }

    /**
     * Returns the identifier name.
     *
     * @return the identifier name.
     */
    public String name() {
        return name;
    }

    /**
     * Returns the identifier's definition.
     *
     * @return the identifier's definition.
     */
    public IDefn iDefn() {
        return iDefn;
```

```
47      }
48
49      /**
50       * {@inheritDoc}
51       */
52      public JExpression analyze(Context context) {
53          iDefn = context.lookup(name);
54          if (iDefn == null) {
55              // Not a local, but is it a field?
56              Type definingType = context.definingType();
57              Field field = definingType.fieldFor(name);
58              if (field == null) {
59                  type = Type.ANY;
60                  JAST.compilationUnit.reportSemanticError(line, "Cannot find name: " + name);
61              } else {
62                  // Rewrite a variable denoting a field as an explicit field selection.
63                  type = field.type();
64                  JExpression newTree = new JFieldSelection(line(),
65                          field.isStatic() || (context.methodContext() != null &&
66                                  context.methodContext().isStatic()) ?
67                                  new JVariable(line(), definingType.toString()) : new JThis(line),
68                          name);
69                  return (JExpression) newTree.analyze(context);
70              }
71          } else {
72              if (!analyzeLhs && iDefn instanceof LocalVariableDefn &&
73                      !((LocalVariableDefn) iDefn).isInitialized()) {
74                  JAST.compilationUnit.reportSemanticError(line, "Variable " + name +
75                          " might not have been initialized");
76              }
77              type = iDefn.type();
78          }
79          return this;
80      }
81
82      /**
83       * {@inheritDoc}
84       */
85      public JExpression analyzeLhs(Context context) {
86          analyzeLhs = true;
87          JExpression newTree = analyze(context);
88          if (newTree instanceof JVariable) {
89              // Could (now) be a JFieldSelection, but if it's (still) a JVariable...
90              if (iDefn != null && !(iDefn instanceof LocalVariableDefn)) {
91                  JAST.compilationUnit.reportSemanticError(line(), name + " is a bad LHS to a =");
92              }
93          }
94          return newTree;
95      }
```

```
96
97      /**
98       * {@inheritDoc}
99       */
100     public void codegen(CLEmitter output) {
101         if (iDefn instanceof LocalVariableDefn) {
102             int offset = ((LocalVariableDefn) iDefn).offset();
103             if (type.isReference()) {
104                 switch (offset) {
105                     case 0:
106                         output.addNoArgInstruction(ALOAD_0);
107                         break;
108                     case 1:
109                         output.addNoArgInstruction(ALOAD_1);
110                         break;
111                     case 2:
112                         output.addNoArgInstruction(ALOAD_2);
113                         break;
114                     case 3:
115                         output.addNoArgInstruction(ALOAD_3);
116                         break;
117                     default:
118                         output.addOneArgInstruction(ALOAD, offset);
119                         break;
120                 }
121             } else if (type == Type.INT || type == Type.BOOLEAN || type == Type.CHAR) {
122                 switch (offset) {
123                     case 0:
124                         output.addNoArgInstruction(ILOAD_0);
125                         break;
126                     case 1:
127                         output.addNoArgInstruction(ILOAD_1);
128                         break;
129                     case 2:
130                         output.addNoArgInstruction(ILOAD_2);
131                         break;
132                     case 3:
133                         output.addNoArgInstruction(ILOAD_3);
134                         break;
135                     default:
136                         output.addOneArgInstruction(ILOAD, offset);
137                         break;
138                 }
139             } else if (type == Type.LONG) {
140                 switch (offset) {
141                     case 0:
142                         output.addNoArgInstruction(LLOAD_0);
143                         break;
144                     case 1:
```

```
                    output.addNoArgInstruction(LLOAD_1);
                    break;
                case 2:
                    output.addNoArgInstruction(LLOAD_2);
                    break;
                case 3:
                    output.addNoArgInstruction(LLOAD_3);
                    break;
                default:
                    output.addOneArgInstruction(LLOAD, offset);
                    break;
            }
        } else if (type == Type.DOUBLE) {
            switch (offset) {
                case 0:
                    output.addNoArgInstruction(DLOAD_0);
                    break;
                case 1:
                    output.addNoArgInstruction(DLOAD_1);
                    break;
                case 2:
                    output.addNoArgInstruction(DLOAD_2);
                    break;
                case 3:
                    output.addNoArgInstruction(DLOAD_3);
                    break;
                default:
                    output.addOneArgInstruction(DLOAD, offset);
                    break;
            }
        }
    }
}

/**
 * {@inheritDoc}
 */
public void codegen(CLEmitter output, String targetLabel, boolean onTrue) {
    if (iDefn instanceof LocalVariableDefn) {
        codegen(output);
        if (onTrue) {
            output.addBranchInstruction(IFNE, targetLabel);
        } else {
            output.addBranchInstruction(IFEQ, targetLabel);
        }
    }
}

/**
```

```
194     * {@inheritDoc}
195     */
196    public void codegenLoadLhsLvalue(CLEmitter output) {
197        // Nothing here.
198    }
199
200    /**
201     * {@inheritDoc}
202     */
203    public void codegenLoadLhsRvalue(CLEmitter output) {
204        codegen(output);
205    }
206
207    /**
208     * {@inheritDoc}
209     */
210    public void codegenDuplicateRvalue(CLEmitter output) {
211        if (iDefn instanceof LocalVariableDefn) {
212            // It's copied atop the stack.
213            output.addNoArgInstruction(DUP);
214        }
215    }
216
217    /**
218     * {@inheritDoc}
219     */
220    public void codegenStore(CLEmitter output) {
221        if (iDefn instanceof LocalVariableDefn) {
222            int offset = ((LocalVariableDefn) iDefn).offset();
223            if (type.isReference()) {
224                switch (offset) {
225                    case 0:
226                        output.addNoArgInstruction(ASTORE_0);
227                        break;
228                    case 1:
229                        output.addNoArgInstruction(ASTORE_1);
230                        break;
231                    case 2:
232                        output.addNoArgInstruction(ASTORE_2);
233                        break;
234                    case 3:
235                        output.addNoArgInstruction(ASTORE_3);
236                        break;
237                    default:
238                        output.addOneArgInstruction(ASTORE, offset);
239                        break;
240                }
241            } else if (type == Type.INT || type == Type.BOOLEAN || type == Type.CHAR) {
242                switch (offset) {
```

```
243                 case 0:
244                     output.addNoArgInstruction(ISTORE_0);
245                     break;
246                 case 1:
247                     output.addNoArgInstruction(ISTORE_1);
248                     break;
249                 case 2:
250                     output.addNoArgInstruction(ISTORE_2);
251                     break;
252                 case 3:
253                     output.addNoArgInstruction(ISTORE_3);
254                     break;
255                 default:
256                     output.addOneArgInstruction(ISTORE, offset);
257                     break;
258             }
259         } else if (type == Type.LONG) {
260             switch (offset) {
261                 case 0:
262                     output.addNoArgInstruction(LSTORE_0);
263                     break;
264                 case 1:
265                     output.addNoArgInstruction(LSTORE_1);
266                     break;
267                 case 2:
268                     output.addNoArgInstruction(LSTORE_2);
269                     break;
270                 case 3:
271                     output.addNoArgInstruction(LSTORE_3);
272                     break;
273                 default:
274                     output.addOneArgInstruction(LSTORE, offset);
275                     break;
276                 }
277         } else if (type == Type.DOUBLE) {
278             switch (offset) {
279                 case 0:
280                     output.addNoArgInstruction(DSTORE_0);
281                     break;
282                 case 1:
283                     output.addNoArgInstruction(DSTORE_1);
284                     break;
285                 case 2:
286                     output.addNoArgInstruction(DSTORE_2);
287                     break;
288                 case 3:
289                     output.addNoArgInstruction(DSTORE_3);
290                     break;
291                 default:
```

```
292                     output.addOneArgInstruction(DSTORE, offset);
293                     break;
294                 }
295             }
296         }
297     }
298 
299     /**
300      * {@inheritDoc}
301      */
302     public void toJSON(JSONElement json) {
303         JSONElement e = new JSONElement();
304         json.addChild("JVariable:" + line, e);
305         e.addAttribute("name", name());
306     }
307 }
308 
```

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);
        if (operand.type() == Type.INT) {
            type = Type.INT;
        } else if (operand.type() == Type.LONG) {
            type = Type.LONG;
        } else if (operand.type() == Type.DOUBLE) {
            type = Type.DOUBLE;
        }
        return this;
    }

    /**
     * {@inheritDoc}
     */
    public void codegen(CLEmitter output) {
        operand.codegen(output);
        if (operand.type() == Type.INT) {
            output.addNoArgInstruction(INEG);
        } else if (operand.type == Type.LONG) {
            output.addNoArgInstruction(LNEG);
        } else if (operand.type == Type.DOUBLE){
            output.addNoArgInstruction(DNEG);
        }
    }
}

/**
 * The AST node for a post-decrement (--) expression.
 */
class JPostDecrementOp extends JUnaryExpression {
    /**
     * Constructs an AST node for a post-decrement expression.
```

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

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

```
243             // Loading its original rvalue.
244             ((JLhs) operand).codegenDuplicateRvalue(output);
245           }
246           ((JLhs) operand).codegenStore(output);
247         }
248       }
249     }
250
251     /**
252      * The AST node for a unary plus (+) expression.
253      */
254     class JUnaryPlusOp extends JUnaryExpression {
255       /**
256        * Constructs an AST node for a unary plus expression.
257        *
258        * @param line    line in which the unary plus expression occurs in the source file.
259        * @param operand the operand.
260        */
261       public JUnaryPlusOp(int line, JExpression operand) {
262         super(line, "+", operand);
263       }
264
265       /**
266        * {@inheritDoc}
267        */
268       public JExpression analyze(Context context) {
269         operand = operand.analyze(context);
270         if (operand.type() == Type.INT) {
271           type = Type.INT;
272         } else if (operand.type() == Type.LONG) {
273           type = Type.LONG;
274         } else if (operand.type() == Type.DOUBLE) {
275           type = Type.DOUBLE;
276         }
277         return this;
278       }
279
280       /**
281        * {@inheritDoc}
282        */
283       public void codegen(CLEmitter output) {
284         operand.codegen(output);
285         if (operand.type() == Type.INT) {
286           output.addNoArgInstruction(ICONST_0);
287           output.addNoArgInstruction(IADD);
288         } else if (operand.type == Type.LONG) {
289           output.addNoArgInstruction(LCONST_0);
290           output.addNoArgInstruction(LADD);
291         } else if (operand.type() == Type.DOUBLE){
```

```
292             output.addNoArgInstruction(DCONST_0);
293             output.addNoArgInstruction(DADD);
294         }
295     }
296 }
297 
298 /**
299  * The AST node for a unary complement (~) expression.
300  */
301 class JComplementOp extends JUnaryExpression {
302     /**
303      * Constructs an AST node for a unary complement expression.
304      *
305      * @param line    line in which the unary complement expression occurs in the source file.
306      * @param operand the operand.
307      */
308     public JComplementOp(int line, JExpression operand) {
309         super(line, "~", operand);
310     }
311 
312     /**
313      * {@inheritDoc}
314      */
315     public JExpression analyze(Context context) {
316         operand = operand.analyze(context);
317         operand.type().mustMatchExpected(line(), Type.INT);
318         type = Type.INT;
319         return this;
320     }
321 
322     /**
323      * {@inheritDoc}
324      */
325     public void codegen(CLEmitter output) {
326         operand.codegen(output);
327         output.addLDCInstruction(-1);
328         output.addNoArgInstruction(IXOR);
329     }
330 }
331 
332 /**
333  * The AST node for post-increment (++) expression.
334  */
335 class JPostIncrementOp extends JUnaryExpression {
336     /**
337      * Constructs an AST node for a post-increment expression.
338      *
339      * @param line    line in which the expression occurs in the source file.
340      * @param operand the operand.
```

```
341      */
342     public JPostIncrementOp(int line, JExpression operand) {
343         super(line, "++ (post)", operand);
344     }
345 
346     /**
347      * {@inheritDoc}
348      */
349     public JExpression analyze(Context context) {
350         // TODO
351         if (!(operand instanceof JLhs)) {
352             JAST.compilationUnit.reportSemanticError(line, "Operand to ++ must have an LValue.");
353             type = Type.ANY;
354         } else {
355             operand = (JExpression) operand.analyze(context);
356             operand.type().mustMatchExpected(line(), Type.INT);
357             type = Type.INT;
358         }
359         return this;
360     }
361 
362     /**
363      * {@inheritDoc}
364      */
365     public void codegen(CLEmitter output) {
366         // TODO
367         if (operand instanceof JVariable) {
368             // A local variable; otherwise analyze() would have replaced it with an explicit
369             // field selection.
370             int offset = ((LocalVariableDefn) ((JVariable) operand).iDefn()).offset();
371             if (!isStatementExpression) {
372                 // Loading its original rvalue.
373                 operand.codegen(output);
374             }
375             output.addIINCInstruction(offset, 1);
376         } else {
377             ((JLhs) operand).codegenLoadLhsLvalue(output);
378             ((JLhs) operand).codegenLoadLhsRvalue(output);
379             if (!isStatementExpression) {
380                 // Loading its original rvalue.
381                 ((JLhs) operand).codegenDuplicateRvalue(output);
382             }
383             output.addNoArgInstruction(ICONST_1);
384             output.addNoArgInstruction(IADD);
385             ((JLhs) operand).codegenStore(output);
386         }
387     }
388 }
389 
```

```
390 /**
391  * The AST node for a pre-decrement (--) expression.
392  */
393 class JPreDecrementOp extends JUnaryExpression {
394     /**
395      * Constructs an AST node for a pre-decrement expression.
396      *
397      * @param line    line in which the expression occurs in the source file.
398      * @param operand the operand.
399      */
400     public JPreDecrementOp(int line, JExpression operand) {
401         super(line, "-- (pre)", operand);
402     }
403 
404     /**
405      * {@inheritDoc}
406      */
407     public JExpression analyze(Context context) {
408         // TODO
409         if (!(operand instanceof JLhs)) {
410             JAST.compilationUnit.reportSemanticError(line, "Operand to -- must have an LValue.");
411             type = Type.ANY;
412         } else {
413             operand = (JExpression) operand.analyze(context);
414             operand.type().mustMatchExpected(line(), Type.INT);
415             type = Type.INT;
416         }
417         return this;
418     }
419 
420     /**
421      * {@inheritDoc}
422      */
423     public void codegen(CLEmitter output) {
424         // TODO
425         if (operand instanceof JVariable) {
426             // A local variable; otherwise analyze() would have replaced it with an explicit
427             // field selection.
428             int offset = ((LocalVariableDefn) ((JVariable) operand).iDefn()).offset();
429             output.addIINCInstruction(offset, -1);
430             if (!isStatementExpression) {
431                 // Loading its original rvalue.
432                 operand.codegen(output);
433             }
434         } else {
435             ((JLhs) operand).codegenLoadLhsLvalue(output);
436             ((JLhs) operand).codegenLoadLhsRvalue(output);
437             output.addNoArgInstruction(ICONST_1);
438             output.addNoArgInstruction(ISUB);
```

```
439             if (!isStatementExpression) {
440                 // Loading its original rvalue.
441                 ((JLhs) operand).codegenDuplicateRvalue(output);
442             }
443             ((JLhs) operand).codegenStore(output);
444         }
445     }
446 }
```

JTryStatement.java

Download

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

package jminusminus;

import java.util.ArrayList;

import static jminusminus.CLConstants.*;

/**
 * The AST node for a try-catch-finally statement.
 */
class JTryStatement extends JStatement {
    // The try block.
    private JBlock tryBlock;

    // The catch parameters.
    private ArrayList<JFormalParameter> parameters;

    // The catch blocks.
    private ArrayList<JBlock> catchBlocks;

    // The finally block.
    private JBlock finallyBlock;

    /**
     * Constructs an AST node for a try-statement.
     *
     * @param line         line in which the while-statement occurs in the source file.
     * @param tryBlock     the try block.
     * @param parameters   the catch parameters.
     * @param catchBlocks  the catch blocks.
     * @param finallyBlock the finally block.
     */
    public JTryStatement(int line, JBlock tryBlock, ArrayList<JFormalParameter> parameters,
                         ArrayList<JBlock> catchBlocks, JBlock finallyBlock) {
        super(line);
        this.tryBlock = tryBlock;
        this.parameters = parameters;
        this.catchBlocks = catchBlocks;
        this.finallyBlock = finallyBlock;
    }

    /**
     * {@inheritDoc}
     */
    public JTryStatement analyze(Context context) {
```

```
47          // TODO
48          return this;
49      }
50
51      /**
52       * {@inheritDoc}
53       */
54      public void codegen(CLEmitter output) {
55          // TODO
56      }
57
58      /**
59       * {@inheritDoc}
60       */
61      public void toJSON(JSONElement json) {
62          JSONElement e = new JSONElement();
63          json.addChild("JTryStatement:" + line, e);
64          JSONElement e1 = new JSONElement();
65          e.addChild("TryBlock", e1);
66          tryBlock.toJSON(e1);
67          if (catchBlocks != null) {
68              for (int i = 0; i < catchBlocks.size(); i++) {
69                  JFormalParameter param = parameters.get(i);
70                  JBlock catchBlock = catchBlocks.get(i);
71                  JSONElement e2 = new JSONElement();
72                  e.addChild("CatchBlock", e2);
73                  String s = String.format("[\"%s\", \"%s\"]", param.name(), param.type() == null ?
74                          "" : param.type().toString());
75                  e2.addAttribute("parameter", s);
76                  catchBlock.toJSON(e2);
77              }
78          }
79          if (finallyBlock != null) {
80              JSONElement e2 = new JSONElement();
81              e.addChild("FinallyBlock", e2);
82              finallyBlock.toJSON(e2);
83          }
84      }
85  }
86
```

JThrowStatement.java

Download

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

package jminusminus;

import static jminusminus.CLConstants.*;

/**
 * An AST node for a throw-statement.
 */
class JThrowStatement extends JStatement {
    // The thrown exception.
    private JExpression expr;

    /**
     * Constructs an AST node for a throw-statement.
     *
     * @param line line in which the throw-statement appears in the source file.
     * @param expr the returned expression.
     */
    public JThrowStatement(int line, JExpression expr) {
        super(line);
        this.expr = expr;
    }

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

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

    /**
     * {@inheritDoc}
     */
    public void toJSON(JSONElement json) {
        JSONElement e = new JSONElement();
        json.addChild("JThrowStatement:" + line, e);
        JSONElement e1 = new JSONElement();
```

```
47          e.addChild("Expression", e1);
48          expr.toJSON(e1);
49      }
50  }
51
```

JSwitchStatement.java

Download

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

package jminusminus;

import java.util.ArrayList;

import static jminusminus.CLConstants.*;

/**
 * The AST node for a switch-statement.
 */
public class JSwitchStatement extends JStatement {
    // Test expression.
    private JExpression condition;

    // List of switch-statement groups.
    private ArrayList<SwitchStatementGroup> stmtGroup;

    /**
     * Constructs an AST node for a switch-statement.
     *
     * @param line      line in which the switch-statement occurs in the source file.
     * @param condition test expression.
     * @param stmtGroup list of statement groups.
     */
    public JSwitchStatement(int line, JExpression condition,
                            ArrayList<SwitchStatementGroup> stmtGroup) {
        super(line);
        this.condition = condition;
        this.stmtGroup = stmtGroup;
    }

    /**
     * {@inheritDoc}
     */
    public JStatement analyze(Context context) {
        // TODO
        condition = condition.analyze(context);
        condition.type.mustMatchExpected(line(), Type.INT);
        for (SwitchStatementGroup ssg : stmtGroup) {
            for (JExpression label : ssg.getSwitchLabels()) {
                label.type.mustMatchExpected(line(), Type.INT);
            }
        }
        LocalContext localContext = new LocalContext(context);
        for (SwitchStatementGroup ssg : stmtGroup) {
```

```
            for (JStatement statement : ssg.getBlock()) {
                statement = (JStatement) statement.analyze(localContext);
            }
        }

        return this;
    }

    /**
     * {@inheritDoc}
     */
    public void codegen(CLEmitter output) {
        // TODO
        int hi = 0;
        int lo = 0;


    }

    /**
     * {@inheritDoc}
     */
    public void toJSON(JSONElement json) {
        JSONElement e = new JSONElement();
        json.addChild("JSwitchStatement:" + line, e);
        JSONElement e1 = new JSONElement();
        e.addChild("Condition", e1);
        condition.toJSON(e1);
        for (SwitchStatementGroup group : stmtGroup) {
            group.toJSON(e);
        }
    }
}

/**
 * A switch statement group consists of case labels and a block of statements.
 */
class SwitchStatementGroup {
    // Case labels.
    private ArrayList<JExpression> switchLabels;

    // Block of statements.
    private ArrayList<JStatement> block;

    /**
     * Constructs a switch-statement group.
     *
     * @param switchLabels case labels.
     * @param block        block of statements.
```

```
96      */
97     public SwitchStatementGroup(ArrayList<JExpression> switchLabels, ArrayList<JStatement> block) {
98         this.switchLabels = switchLabels;
99         this.block = block;
100    }
101
102    public ArrayList<JExpression> getSwitchLabels() {
103        return switchLabels;
104    }
105    public ArrayList<JStatement> getBlock() {
106        return block;
107    }
108
109    /**
110     * Stores information about this switch statement group in JSON format.
111     *
112     * @param json the JSON emitter.
113     */
114    public void toJSON(JSONElement json) {
115        JSONElement e = new JSONElement();
116        json.addChild("SwitchStatementGroup", e);
117        for (JExpression label : switchLabels) {
118            JSONElement e1 = new JSONElement();
119            if (label != null) {
120                e.addChild("Case", e1);
121                label.toJSON(e1);
122            } else {
123                e.addChild("Default", e1);
124            }
125        }
126        if (block != null) {
127            for (JStatement stmt : block) {
128                stmt.toJSON(e);
129            }
130        }
131    }
132 }
133
```

JReturnStatement.java

Download

```
1  // Copyright 2012- Bill Campbell, Swami Iyer and Bahar Akbal-Delibas
2
3  package jminusminus;
4
5  import static jminusminus.CLConstants.*;
6
7  /**
8   * The AST node for a return-statement. If the enclosing method is non-void, then there is a
9   * value to return, so we keep track of the expression denoting that value and its type.
10  */
11 class JReturnStatement extends JStatement {
12     // The returned expression.
13     private JExpression expr;
14
15     /**
16      * Constructs an AST node for a return-statement.
17      *
18      * @param line line in which the return-statement appears in the source file.
19      * @param expr the returned expression.
20      */
21     public JReturnStatement(int line, JExpression expr) {
22         super(line);
23         this.expr = expr;
24     }
25
26     /**
27      * {@inheritDoc}
28      */
29     public JStatement analyze(Context context) {
30         MethodContext methodContext = context.methodContext();
31
32         // The methodContext can be null if return statement occurs in a block that is not within
33         // a method. For example, in the Java grammar, return statement, at least syntactically,
34         // can occur in a static block. But since j-- does not allow a block to occur outside of a
35         // method, we don't check for methodContext being null.
36         if (methodContext.methodReturnType() == Type.CONSTRUCTOR) {
37             if (expr != null) {
38                 // Can't return a value from a constructor.
39                 JAST.compilationUnit.reportSemanticError(line(),
40                         "Cannot return a value from a constructor");
41             }
42         } else {
43             // Must be a method.
44             Type returnType = methodContext.methodReturnType();
45             methodContext.confirmMethodHasReturn();
46             if (expr != null) {
```

```
47              if (returnType == Type.VOID) {
48                  // Can't return a value from void method.
49                  JAST.compilationUnit.reportSemanticError(line(),
50                          "Cannot return a value from a void method");
51              } else {
52                  // There's a (non-void) return value. Its type must match the return type of
53                  // the method.
54                  expr = expr.analyze(context);
55                  expr.type().mustMatchExpected(line(), returnType);
56              }
57          } else {
58              // The method better have void as return type.
59              if (returnType != Type.VOID) {
60                  JAST.compilationUnit.reportSemanticError(line(), "Missing return value");
61              }
62          }
63      }
64      return this;
65  }
66
67  /**
68   * {@inheritDoc}
69   */
70  public void codegen(CLEmitter output) {
71      if (expr == null) {
72          output.addNoArgInstruction(RETURN);
73      } else {
74          expr.codegen(output);
75          if (expr.type() == Type.INT || expr.type() == Type.BOOLEAN ||
76                  expr.type() == Type.CHAR) {
77              output.addNoArgInstruction(IRETURN);
78          } else if (expr.type() == Type.LONG) {
79              output.addNoArgInstruction(LRETURN);
80          } else if (expr.type == Type.DOUBLE) {
81              output.addNoArgInstruction(DRETURN);
82          } else {
83              output.addNoArgInstruction(ARETURN);
84          }
85      }
86  }
87
88  /**
89   * {@inheritDoc}
90   */
91  public void toJSON(JSONElement json) {
92      JSONElement e = new JSONElement();
93      json.addChild("JReturnStatement:" + line, e);
94      if (expr != null) {
95          JSONElement e1 = new JSONElement();
```

```
96              e.addChild("Expression", e1);
97              expr.toJSON(e1);
98          }
99      }
100 }
101
```

JMethodDeclaration.java

Download

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

package jminusminus;

import java.util.ArrayList;

import static jminusminus.CLConstants.*;

/**
 * The AST node for a method declaration.
 */
class JMethodDeclaration extends JAST implements JMember {
    /**
     * Method modifiers.
     */
    protected ArrayList<String> mods;

    /**
     * Method name.
     */
    protected String name;

    /**
     * Return type.
     */
    protected Type returnType;

    /**
     * The formal parameters.
     */
    protected ArrayList<JFormalParameter> params;

    /**
     * Exceptions thrown.
     */
    protected ArrayList<TypeName> exceptions;

    /**
     * Method body.
     */
    protected JBlock body;

    /**
     * Method context (built in analyze()).
     */
    protected MethodContext context;
```

```
47
48      /**
49       * Method descriptor (computed in preAnalyze()).
50       */
51      protected String descriptor;
52
53      /**
54       * Is this method abstract?
55       */
56      protected boolean isAbstract;
57
58      /**
59       * Is this method static?
60       */
61      protected boolean isStatic;
62
63      /**
64       * Is this method private?
65       */
66      protected boolean isPrivate;
67
68      /**
69       * Constructs an AST node for a method declaration.
70       *
71       * @param line       line in which the method declaration occurs in the source file.
72       * @param mods       modifiers.
73       * @param name       method name.
74       * @param returnType return type.
75       * @param params     the formal parameters.
76       * @param exceptions exceptions thrown.
77       * @param body       method body.
78       */
79      public JMethodDeclaration(int line, ArrayList<String> mods, String name, Type returnType,
80                                ArrayList<JFormalParameter> params,
81                                ArrayList<TypeName> exceptions, JBlock body) {
82          super(line);
83          this.mods = mods;
84          this.name = name;
85          this.returnType = returnType;
86          this.params = params;
87          this.exceptions = exceptions;
88          this.body = body;
89          isAbstract = mods.contains("abstract");
90          isStatic = mods.contains("static");
91          isPrivate = mods.contains("private");
92      }
93
94      /**
95       * {@inheritDoc}
```

```
96      */
97     public void preAnalyze(Context context, CLEmitter partial) {
98         // Resolve types of the formal parameters.
99         for (JFormalParameter param : params) {
100            param.setType(param.type().resolve(context));
101        }
102
103        // Resolve return type.
104        returnType = returnType.resolve(context);
105
106        // Check proper local use of abstract
107        if (isAbstract && body != null) {
108            JAST.compilationUnit.reportSemanticError(line(), "abstract method cannot have a body");
109        } else if (body == null && !isAbstract) {
110            JAST.compilationUnit.reportSemanticError(line(),
111                    "Method without body must be abstract");
112        } else if (isAbstract && isPrivate) {
113            JAST.compilationUnit.reportSemanticError(line(), "private method cannot be abstract");
114        } else if (isAbstract && isStatic) {
115            JAST.compilationUnit.reportSemanticError(line(), "static method cannot be abstract");
116        }
117
118        // Compute descriptor.
119        descriptor = "(";
120        for (JFormalParameter param : params) {
121            descriptor += param.type().toDescriptor();
122        }
123        descriptor += ")" + returnType.toDescriptor();
124
125        // Generate the method with an empty body (for now).
126        partialCodegen(context, partial);
127    }
128
129    /**
130     * {@inheritDoc}
131     */
132    public JAST analyze(Context context) {
133        MethodContext methodContext = new MethodContext(context, isStatic, returnType);
134        this.context = methodContext;
135
136        if (!isStatic) {
137            // Offset 0 is used to address "this".
138            this.context.nextOffset();
139        }
140
141        // Declare the parameters. We consider a formal parameter to be always initialized, via a
142        // method call.
143        for (JFormalParameter param : params) {
144            LocalVariableDefn defn = new LocalVariableDefn(param.type(), this.context.nextOffset());
```

```
145             defn.initialize();
146             this.context.addEntry(param.line(), param.name(), defn);
147 
148             if (param.type() == Type.LONG || param.type() == Type.DOUBLE) {
149                 this.context.nextOffset();
150             }
151         }
152 
153         if (body != null) {
154             body = body.analyze(this.context);
155             if (returnType != Type.VOID && !methodContext.methodHasReturn()) {
156                 JAST.compilationUnit.reportSemanticError(line(),
157                         "Non-void method must have a return statement");
158             }
159         }
160         return this;
161     }
162 
163     /**
164      * {@inheritDoc}
165      */
166     public void partialCodegen(Context context, CLEmitter partial) {
167         partial.addMethod(mods, name, descriptor, null, false);
168         if (returnType == Type.VOID) {
169             partial.addNoArgInstruction(RETURN);
170         } else if (returnType == Type.INT || returnType == Type.BOOLEAN ||
171                 returnType == Type.CHAR) {
172             partial.addNoArgInstruction(ICONST_0);
173             partial.addNoArgInstruction(IRETURN);
174         } else if (returnType == Type.LONG) {
175             partial.addNoArgInstruction(LCONST_0);
176             partial.addNoArgInstruction(LRETURN);
177         } else if (returnType == Type.DOUBLE) {
178             partial.addNoArgInstruction(DCONST_0);
179             partial.addNoArgInstruction(DRETURN);
180         } else {
181             partial.addNoArgInstruction(ACONST_NULL);
182             partial.addNoArgInstruction(ARETURN);
183         }
184     }
185 
186     /**
187      * {@inheritDoc}
188      */
189     public void codegen(CLEmitter output) {
190         output.addMethod(mods, name, descriptor, null, false);
191         if (body != null) {
192             body.codegen(output);
193         }
```

```
194         if (returnType == Type.VOID) {
195             output.addNoArgInstruction(RETURN);
196         }
197     }
198 
199     /**
200      * {@inheritDoc}
201      */
202     public void toJSON(JSONElement json) {
203         JSONElement e = new JSONElement();
204         json.addChild("JMethodDeclaration:" + line, e);
205         e.addAttribute("name", name);
206         e.addAttribute("returnType", returnType.toString());
207         if (mods != null) {
208             ArrayList<String> value = new ArrayList<String>();
209             for (String mod : mods) {
210                 value.add(String.format("\"%s\"", mod));
211             }
212             e.addAttribute("modifiers", value);
213         }
214         if (params != null) {
215             ArrayList<String> value = new ArrayList<String>();
216             for (JFormalParameter param : params) {
217                 value.add(String.format("[\"%s\", \"%s\"]", param.name(),
218                         param.type() == null ? "" : param.type().toString()));
219             }
220             e.addAttribute("parameters", value);
221         }
222         if (exceptions != null) {
223             ArrayList<String> value = new ArrayList<String>();
224             for (TypeName exception : exceptions) {
225                 value.add(String.format("\"%s\"", exception.toString()));
226             }
227             e.addAttribute("throws", value);
228         }
229         if (context != null) {
230             context.toJSON(e);
231         }
232         if (body != null) {
233             body.toJSON(e);
234         }
235     }
236 }
237 
```

JMember.java

Download

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

package jminusminus;

/**
 * An interface supported by all class (or later, interface) members.
 */
interface JMember {
    /**
     * Declares the member names in the specified (class) context and generates the member headers
     * in the partial class.
     *
     * @param context class context in which names are resolved.
     * @param partial the code emitter.
     */
    public void preAnalyze(Context context, CLEmitter partial);
}

```

JLiteralLong.java

Download

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

package jminusminus;

import static jminusminus.CLConstants.*;

/**
 * The AST node for a long literal.
 */
class JLiteralLong extends JExpression {
    // String representation of the literal.
    private String text;

    /**
     * Constructs an AST node for a long literal given its line number and string representation.
     *
     * @param line line in which the literal occurs in the source file.
     * @param text string representation of the literal.
     */
    public JLiteralLong(int line, String text) {
        super(line);
        this.text = text;
    }

    /**
     * Returns the literal as a long.
     *
     * @return the literal as a long.
     */
    public long toLong() {
        return Long.parseLong(text.substring(0, text.length() - 1));
    }

    /**
     * {@inheritDoc}
     */
    public JExpression analyze(Context context) {
        type = Type.LONG;
        return this;
    }

    /**
     * {@inheritDoc}
     */
    public void codegen(CLEmitter output) {
        long l = toLong();
```

```
47          output.addLDCInstruction(l);
48      }
49
50      /**
51       * {@inheritDoc}
52       */
53      public void toJSON(JSONElement json) {
54          JSONElement e = new JSONElement();
55          json.addChild("JLiteralLong:" + line, e);
56          e.addAttribute("type", type == null ? "" : type.toString());
57          e.addAttribute("value", text);
58      }
59  }
60
```

JLiteralDouble.java

Download

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

package jminusminus;

import static jminusminus.CLConstants.*;

/**
 * The AST node for a double literal.
 */
class JLiteralDouble extends JExpression {
    // String representation of the literal.
    private String text;

    /**
     * Constructs an AST node for a double literal given its line number and string representation.
     *
     * @param line line in which the literal occurs in the source file.
     * @param text string representation of the literal.
     */
    public JLiteralDouble(int line, String text) {
        super(line);
        this.text = text;
    }

    /**
     * Returns the literal as a double.
     *
     * @return the literal as a double.
     */
    public double toDouble() {
        return Double.parseDouble(text);
    }

    /**
     * {@inheritDoc}
     */
    public JExpression analyze(Context context) {
        type = Type.DOUBLE;
        return this;
    }

    /**
     * {@inheritDoc}
     */
    public void codegen(CLEmitter output) {
        double d = toDouble();
```

```
47          output.addLDCInstruction(d);
48      }
49
50      /**
51       * {@inheritDoc}
52       */
53      public void toJSON(JSONElement json) {
54          JSONElement e = new JSONElement();
55          json.addChild("JLiteralDouble:" + line, e);
56          e.addAttribute("type", type == null ? "" : type.toString());
57          e.addAttribute("value", text);
58      }
59  }
60
```

JInterfaceDeclaration.java

Download

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

package jminusminus;

import java.util.ArrayList;

/**
 * A representation of an interface declaration.
 */
class JInterfaceDeclaration extends JAST implements JTypeDecl {
    // Interface modifiers.
    private ArrayList<String> mods;

    // Interface name.
    private String name;

    // This interface type.
    private Type thisType;

    // Super class type.
    private Type superType;

    // Extended interfaces.
    private ArrayList<TypeName> superInterfaces;

    // Interface block.
    private ArrayList<JMember> interfaceBlock;

    // Context for this interface.
    private ClassContext context;

    /**
     * Constructs an AST node for an interface declaration.
     *
     * @param line           line in which the interface declaration occurs in the source file.
     * @param mods           class modifiers.
     * @param name           class name.
     * @param superInterfaces super class types.
     * @param interfaceBlock  interface block.
     */
    public JInterfaceDeclaration(int line, ArrayList<String> mods, String name,
                                 ArrayList<TypeName> superInterfaces,
                                 ArrayList<JMember> interfaceBlock) {
        super(line);
        this.mods = mods;
        this.name = name;
```

```
47          this.superType = Type.OBJECT;
48          this.superInterfaces = superInterfaces;
49          this.interfaceBlock = interfaceBlock;
50      }
51
52      /**
53       * {@inheritDoc}
54       */
55      public void declareThisType(Context context) {
56          // TODO
57      }
58
59      /**
60       * {@inheritDoc}
61       */
62      public void preAnalyze(Context context) {
63          // TODO
64      }
65
66      /**
67       * {@inheritDoc}
68       */
69      public String name() {
70          return name;
71      }
72
73      /**
74       * {@inheritDoc}
75       */
76      public Type superType() {
77          return superType;
78      }
79
80      /**
81       * {@inheritDoc}
82       */
83      public ArrayList<TypeName> superInterfaces() {
84          return superInterfaces;
85      }
86
87      /**
88       * {@inheritDoc}
89       */
90      public Type thisType() {
91          // TODO
92          return null;
93      }
94
95      /**
```

```
96      * {@inheritDoc}
97      */
98     public JAST analyze(Context context) {
99         // TODO
100        return this;
101    }
102
103    /**
104     * {@inheritDoc}
105     */
106    public void codegen(CLEmitter output) {
107        // TODO
108    }
109
110    /**
111     * {@inheritDoc}
112     */
113    public void toJSON(JSONElement json) {
114        JSONElement e = new JSONElement();
115        json.addChild("JInterfaceDeclaration:" + line, e);
116        if (mods != null) {
117            ArrayList<String> value = new ArrayList<String>();
118            for (String mod : mods) {
119                value.add(String.format("\"%s\"", mod));
120            }
121            e.addAttribute("modifiers", value);
122        }
123        e.addAttribute("name", name);
124        e.addAttribute("super", superType == null ? "" : superType.toString());
125        if (superInterfaces != null) {
126            ArrayList<String> value = new ArrayList<String>();
127            for (TypeName impl : superInterfaces) {
128                value.add(String.format("\"%s\"", impl.toString()));
129            }
130            e.addAttribute("extends", value);
131        }
132        if (context != null) {
133            context.toJSON(e);
134        }
135        if (interfaceBlock != null) {
136            for (JMember member : interfaceBlock) {
137                ((JAST) member).toJSON(e);
138            }
139        }
140    }
141}
142
```

JForStatement.java

Download

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

package jminusminus;

import java.util.ArrayList;

import static jminusminus.CLConstants.*;

/**
 * The AST node for a for-statement.
 */
class JForStatement extends JStatement {
    // Initialization.
    private ArrayList<JStatement> init;

    // Test expression
    private JExpression condition;

    // Update.
    private ArrayList<JStatement> update;

    // The body.
    private JStatement body;

    /**
     * Constructs an AST node for a for-statement.
     *
     * @param line      line in which the for-statement occurs in the source file.
     * @param init      the initialization.
     * @param condition the test expression.
     * @param update    the update.
     * @param body      the body.
     */
    public JForStatement(int line, ArrayList<JStatement> init, JExpression condition,
                         ArrayList<JStatement> update, JStatement body) {
        super(line);
        this.init = init;
        this.condition = condition;
        this.update = update;
        this.body = body;
    }

    /**
     * {@inheritDoc}
     */
    public JForStatement analyze(Context context) {
```

```
47          // TODO
48          LocalContext localContext = new LocalContext(context);
49          for (int i = 0; i < init.size(); i++) {
50              JStatement in = init.get(i);
51              init.set(i, (JStatement) in.analyze(localContext));
52          }
53          condition = condition.analyze(localContext);
54          condition.type.mustMatchExpected(line(), Type.BOOLEAN);
55          for (int i = 0; i < update.size(); i++) {
56              JStatement up = update.get(i);
57              update.set(i, (JStatement) up.analyze(localContext));
58          }
59          body = (JStatement) body.analyze(localContext);
60          return this;
61      }
62
63      /**
64       * {@inheritDoc}
65       */
66      public void codegen(CLEmitter output) {
67          // TODO
68          String test = output.createLabel();
69          String out = output.createLabel();
70          for (int i = 0; i < init.size(); i++) {
71              JStatement in = init.get(i);
72              in.codegen(output);
73          }
74          output.addLabel(test);
75          condition.codegen(output, out, false);
76          body.codegen(output);
77          for (int i = 0; i < update.size(); i++) {
78              JStatement up = update.get(i);
79              up.codegen(output);
80          }
81          output.addBranchInstruction(GOTO, test);
82          output.addLabel(out);
83      }
84
85      /**
86       * {@inheritDoc}
87       */
88      public void toJSON(JSONElement json) {
89          JSONElement e = new JSONElement();
90          json.addChild("JForStatement:" + line, e);
91          if (init != null) {
92              JSONElement e1 = new JSONElement();
93              e.addChild("Init", e1);
94              for (JStatement stmt : init) {
95                  stmt.toJSON(e1);
```

```
96              }
97          }
98          if (condition != null) {
99              JSONElement e1 = new JSONElement();
100             e.addChild("Condition", e1);
101             condition.toJSON(e1);
102         }
103         if (update != null) {
104             JSONElement e1 = new JSONElement();
105             e.addChild("Update", e1);
106             for (JStatement stmt : update) {
107                 stmt.toJSON(e1);
108             }
109         }
110         if (body != null) {
111             JSONElement e1 = new JSONElement();
112             e.addChild("Body", e1);
113             body.toJSON(e1);
114         }
115     }
116 }
117
```

JDoStatement.java

Download

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

package jminusminus;

import static jminusminus.CLConstants.*;

/**
 * The AST node for a do-statement.
 */
public class JDoStatement extends JStatement {
    // Body.
    private JStatement body;

    // Test expression.
    private JExpression condition;

    /**
     * Constructs an AST node for a do-statement.
     *
     * @param line      line in which the do-statement occurs in the source file.
     * @param body      the body.
     * @param condition test expression.
     */
    public JDoStatement(int line, JStatement body, JExpression condition) {
        super(line);
        this.body = body;
        this.condition = condition;
    }

    /**
     * {@inheritDoc}
     */
    public JStatement analyze(Context context) {
        // TODO
        condition = condition.analyze(context);
        condition.type.mustMatchExpected(line(), Type.BOOLEAN);
        body = (JStatement) body.analyze(context);
        return this;
    }

    /**
     * {@inheritDoc}
     */
    public void codegen(CLEmitter output) {
        // TODO
        String topLabel = output.createLabel();
```

```
47          output.addLabel(topLabel);
48          body.codegen(output);
49          condition.codegen(output, topLabel, true);
50      }
51
52      /**
53       * {@inheritDoc}
54       */
55      public void toJSON(JSONElement json) {
56          JSONElement e = new JSONElement();
57          json.addChild("JDoStatement:" + line, e);
58          JSONElement e1 = new JSONElement();
59          e.addChild("Body", e1);
60          body.toJSON(e1);
61          JSONElement e2 = new JSONElement();
62          e.addChild("Condition", e2);
63          condition.toJSON(e2);
64      }
65  }
66
```

JContinueStatement.java

Download

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

package jminusminus;

import static jminusminus.CLConstants.GOTO;

/**
 * An AST node for a continue-statement.
 */
public class JContinueStatement extends JStatement {
    /**
     * Constructs an AST node for a continue-statement.
     *
     * @param line line in which the continue-statement occurs in the source file.
     */
    public JContinueStatement(int line) {
        super(line);
    }

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

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

    /**
     * {@inheritDoc}
     */
    public void toJSON(JSONElement json) {
        JSONElement e = new JSONElement();
        json.addChild("JContinueStatement:" + line, e);
    }
}

```

JConstructorDeclaration.java

Download

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

package jminusminus;

import java.util.ArrayList;

import static jminusminus.CLConstants.*;

/**
 * The AST node for a constructor declaration.
 */
class JConstructorDeclaration extends JMethodDeclaration implements JMember {
    // Does this constructor invoke this(...) or super(...)?
    private boolean invokesConstructor;

    // Defining class
    private JClassDeclaration definingClass;

    /**
     * Constructs an AST node for a constructor declaration.
     *
     * @param line       line in which the constructor declaration occurs in the source file.
     * @param mods       modifiers.
     * @param name       constructor name.
     * @param params     the formal parameters.
     * @param exceptions exceptions thrown.
     * @param body       constructor body.
     */
    public JConstructorDeclaration(int line, ArrayList<String> mods, String name,
                                   ArrayList<JFormalParameter> params,
                                   ArrayList<TypeName> exceptions, JBlock body) {
        super(line, mods, name, Type.CONSTRUCTOR, params, exceptions, body);
    }

    /**
     * {@inheritDoc}
     */
    public void preAnalyze(Context context, CLEmitter partial) {
        super.preAnalyze(context, partial);
        if (isStatic) {
            JAST.compilationUnit.reportSemanticError(line(), "Constructor cannot be static");
        } else if (isAbstract) {
            JAST.compilationUnit.reportSemanticError(line(), "Constructor cannot be abstract");
        }
        if (body.statements().size() > 0 &&
                body.statements().get(0) instanceof JStatementExpression) {
```

```
47          JStatementExpression first = (JStatementExpression) body.statements().get(0);
48          if (first.expr instanceof JSuperConstruction) {
49              ((JSuperConstruction) first.expr).markProperUseOfConstructor();
50              invokesConstructor = true;
51          } else if (first.expr instanceof JThisConstruction) {
52              ((JThisConstruction) first.expr).markProperUseOfConstructor();
53              invokesConstructor = true;
54          }
55      }
56  }
57
58  /**
59   * {@inheritDoc}
60   */
61  public JAST analyze(Context context) {
62      // Record the defining class declaration.
63      definingClass = (JClassDeclaration) (context.classContext().definition());
64
65      MethodContext methodContext = new MethodContext(context, isStatic, returnType);
66      this.context = methodContext;
67
68      if (!isStatic) {
69          // Offset 0 is used to address "this".
70          this.context.nextOffset();
71      }
72
73      // Declare the parameters. We consider a formal parameter to be always initialized, via a
74      // method call.
75      for (JFormalParameter param : params) {
76          LocalVariableDefn defn = new LocalVariableDefn(param.type(), this.context.nextOffset());
77          defn.initialize();
78          this.context.addEntry(param.line(), param.name(), defn);
79
80          if (param.type() == Type.LONG || param.type() == Type.DOUBLE) {
81              this.context.nextOffset();
82          }
83      }
84
85      if (body != null) {
86          body = body.analyze(this.context);
87      }
88      return this;
89  }
90
91  /**
92   * {@inheritDoc}
93   */
94  public void partialCodegen(Context context, CLEmitter partial) {
95      partial.addMethod(mods, "<init>", descriptor, null, false);
```

```
96          if (!invokesConstructor) {
97              partial.addNoArgInstruction(ALOAD_0);
98              partial.addMemberAccessInstruction(INVOKESPECIAL,
99                      ((JClassDeclaration) context.classContext().definition()).superType().jvmName(),
100                     "<init>", "()V");
101         }
102         partial.addNoArgInstruction(RETURN);
103     }
104
105     /**
106      * {@inheritDoc}
107      */
108     public void codegen(CLEmitter output) {
109         output.addMethod(mods, "<init>", descriptor, null, false);
110         if (!invokesConstructor) {
111             output.addNoArgInstruction(ALOAD_0);
112             output.addMemberAccessInstruction(INVOKESPECIAL, definingClass.superType().jvmName(),
113                     "<init>", "()V");
114         }
115
116         // Field initializations.
117         for (JFieldDeclaration field : definingClass.instanceFieldInitializations()) {
118             field.codegenInitializations(output);
119         }
120
121         // And then the body.
122         body.codegen(output);
123
124         output.addNoArgInstruction(RETURN);
125     }
126
127     /**
128      * {@inheritDoc}
129      */
130     public void toJSON(JSONElement json) {
131         JSONElement e = new JSONElement();
132         json.addChild("JConstructorDeclaration:" + line, e);
133         e.addAttribute("name", name);
134         if (mods != null) {
135             ArrayList<String> value = new ArrayList<String>();
136             for (String mod : mods) {
137                 value.add(String.format("\"%s\"", mod));
138             }
139             e.addAttribute("modifiers", value);
140         }
141         if (params != null) {
142             ArrayList<String> value = new ArrayList<String>();
143             for (JFormalParameter param : params) {
144                 value.add(String.format("[\"%s\", \"%s\"]", param.name(),
```

```
                    param.type() == null ? "" : param.type().toString()));
            }
            e.addAttribute("parameters", value);
        }
        if (exceptions != null) {
            ArrayList<String> value = new ArrayList<String>();
            for (TypeName exception : exceptions) {
                value.add(String.format("\"%s\"", exception.toString()));
            }
            e.addAttribute("throws", value);
        }
        if (context != null) {
            context.toJSON(e);
        }
        if (body != null) {
            body.toJSON(e);
        }
    }
}

```

JConditionalExpression.java

Download

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

package jminusminus;

import static jminusminus.CLConstants.*;

/**
 * The AST node for a conditional expression.
 */
class JConditionalExpression extends JExpression {
    // Test expression.
    private JExpression condition;

    // Then part.
    private JExpression thenPart;

    // Else part.
    private JExpression elsePart;

    /**
     * Constructs an AST node for a conditional expression.
     *
     * @param line      line in which the conditional expression occurs in the source file.
     * @param condition test expression.
     * @param thenPart  then part.
     * @param elsePart  else part.
     */
    public JConditionalExpression(int line, JExpression condition, JExpression thenPart,
                                  JExpression elsePart) {
        super(line);
        this.condition = condition;
        this.thenPart = thenPart;
        this.elsePart = elsePart;
    }

    /**
     * {@inheritDoc}
     */
    public JExpression analyze(Context context) {
        // TODO
        condition = condition.analyze(context);
        thenPart = thenPart.analyze(context);
        elsePart = elsePart.analyze(context);
        thenPart.type().mustMatchExpected(line(), elsePart.type());
        type = thenPart.type();
        return this;
```

```
47      }
48
49      /**
50       * {@inheritDoc}
51       */
52      public void codegen(CLEmitter output) {
53          // TODO
54          String elseLabel = output.createLabel();
55          String endLabel = output.createLabel();
56          condition.codegen(output, elseLabel, false);
57          thenPart.codegen(output);
58          if (elsePart != null) {
59              output.addBranchInstruction(GOTO, endLabel);
60          }
61          output.addLabel(elseLabel);
62          if (elsePart != null) {
63              elsePart.codegen(output);
64              output.addLabel(endLabel);
65          }
66      }
67
68      /**
69       * {@inheritDoc}
70       */
71      public void toJSON(JSONElement json) {
72          JSONElement e = new JSONElement();
73          json.addChild("JConditionalExpression:" + line, e);
74          JSONElement e1 = new JSONElement();
75          e.addChild("Condition", e1);
76          condition.toJSON(e1);
77          JSONElement e2 = new JSONElement();
78          e.addChild("ThenPart", e2);
79          thenPart.toJSON(e2);
80          JSONElement e3 = new JSONElement();
81          e.addChild("ElsePart", e3);
82          elsePart.toJSON(e3);
83      }
84  }
85
```

JComparisonExpression.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 comparison expression.
 */
abstract class JComparisonExpression extends JBooleanBinaryExpression {
    /**
     * Constructs an AST node for a comparison expression.
     *
     * @param line     line in which the expression occurs in the source file.
     * @param operator the comparison operator.
     * @param lhs      the lhs operand.
     * @param rhs      the rhs operand.
     */
    protected JComparisonExpression(int line, String operator, JExpression lhs, JExpression rhs) {
        super(line, operator, lhs, rhs);
    }

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

/**
 * The AST node for a greater-than (&gt;) expression.
 */
class JGreaterThanOp extends JComparisonExpression {
    /**
     * Constructs an AST node for a greater-than expression.
     *
     * @param line line in which the greater-than expression occurs in the source file.
     * @param lhs  lhs operand.
     * @param rhs  rhs operand.
     */
```

```
47      public JGreaterThanOp(int line, JExpression lhs, JExpression rhs) {
48          super(line, ">", lhs, rhs);
49      }
50
51      /**
52       * {@inheritDoc}
53       */
54      public void codegen(CLEmitter output, String targetLabel, boolean onTrue) {
55          lhs.codegen(output);
56          rhs.codegen(output);
57          output.addBranchInstruction(onTrue ? IF_ICMPGT : IF_ICMPLE, targetLabel);
58      }
59  }
60
61  /**
62   * The AST node for a less-than-or-equal-to (&lt;=) expression.
63   */
64  class JLessEqualOp extends JComparisonExpression {
65
66      /**
67       * Constructs an AST node for a less-than-or-equal-to expression.
68       *
69       * @param line line in which the less-than-or-equal-to expression occurs in the source file.
70       * @param lhs  lhs operand.
71       * @param rhs  rhs operand.
72       */
73      public JLessEqualOp(int line, JExpression lhs, JExpression rhs) {
74          super(line, "<=", lhs, rhs);
75      }
76
77      /**
78       * {@inheritDoc}
79       */
80      public void codegen(CLEmitter output, String targetLabel, boolean onTrue) {
81          lhs.codegen(output);
82          rhs.codegen(output);
83          output.addBranchInstruction(onTrue ? IF_ICMPLE : IF_ICMPGT, targetLabel);
84      }
85  }
86
87  /**
88   * The AST node for a greater-than-or-equal-to (&gt;=) expression.
89   */
90  class JGreaterEqualOp extends JComparisonExpression {
91
92      /**
93       * Constructs an AST node for a greater-than-or-equal-to expression.
94       *
95       * @param line line in which the greater-than-or-equal-to expression occurs in the source file.
```

```
96      * @param lhs  lhs operand.
97      * @param rhs  rhs operand.
98      */
99     public JGreaterEqualOp(int line, JExpression lhs, JExpression rhs) {
100        super(line, ">=", lhs, rhs);
101    }
102
103    /**
104     * {@inheritDoc}
105     */
106    public void codegen(CLEmitter output, String targetLabel, boolean onTrue) {
107        // TODO
108        if (lhs.type() == Type.INT && rhs.type() == Type.INT) {
109            lhs.codegen(output);
110            rhs.codegen(output);
111            output.addBranchInstruction(onTrue ? IF_ICMPGE : IF_ICMPLT, targetLabel);
112        } else if (lhs.type() == Type.LONG && rhs.type() == Type.LONG) {
113            lhs.codegen(output);
114            output.addNoArgInstruction(L2I);
115            rhs.codegen(output);
116            output.addNoArgInstruction(L2I);
117            output.addBranchInstruction(onTrue ? IF_ICMPGE : IF_ICMPLT, targetLabel);
118        } else if (lhs.type() == Type.DOUBLE && rhs.type() == Type.DOUBLE) {
119            lhs.codegen(output);
120            output.addNoArgInstruction(D2I);
121            rhs.codegen(output);
122            output.addNoArgInstruction(D2I);
123            output.addBranchInstruction(onTrue ? IF_ICMPGE : IF_ICMPLT, targetLabel);
124        }
125    }
126}
127
128/**
129 * The AST node for a less-than (&lt;) expression.
130 */
131class JLessThanOp extends JComparisonExpression {
132    /**
133     * Constructs an AST node for a less-than expression.
134     *
135     * @param line line in which the less-than expression occurs in the source file.
136     * @param lhs  lhs operand.
137     * @param rhs  rhs operand.
138     */
139    public JLessThanOp(int line, JExpression lhs, JExpression rhs) {
140        super(line, "<", lhs, rhs);
141    }
142
143    /**
144     * {@inheritDoc}
```

```
145     */
146     public void codegen(CLEmitter output, String targetLabel, boolean onTrue) {
147         // TODO
148         if (lhs.type() == Type.INT && rhs.type() == Type.INT) {
149             lhs.codegen(output);
150             rhs.codegen(output);
151             output.addBranchInstruction(onTrue ? IF_ICMPLT : IF_ICMPGE, targetLabel);
152         } else if (lhs.type() == Type.LONG && rhs.type() == Type.LONG) {
153             lhs.codegen(output);
154             output.addNoArgInstruction(L2I);
155             rhs.codegen(output);
156             output.addNoArgInstruction(L2I);
157             output.addBranchInstruction(onTrue ? IF_ICMPLT : IF_ICMPGE, targetLabel);
158         } else if (lhs.type() == Type.DOUBLE && rhs.type() == Type.DOUBLE) {
159             lhs.codegen(output);
160             output.addNoArgInstruction(D2I);
161             rhs.codegen(output);
162             output.addNoArgInstruction(D2I);
163             output.addBranchInstruction(onTrue ? IF_ICMPLT : IF_ICMPGE, targetLabel);
164         }
165     }
166 }
```

JClassDeclaration.java

Download

```
1  // Copyright 2012- Bill Campbell, Swami Iyer and Bahar Akbal-Delibas
2
3  package jminusminus;
4
5  import java.util.ArrayList;
6
7  import static jminusminus.CLConstants.*;
8
9  /**
10  * A representation of a class declaration.
11  */
12 class JClassDeclaration extends JAST implements JTypeDecl {
13     // Class modifiers.
14     private ArrayList<String> mods;
15
16     // Class name.
17     private String name;
18
19     // This class type.
20     private Type thisType;
21
22     // Super class type.
23     private Type superType;
24
25     // Implemented interfaces.
26     private ArrayList<TypeName> superInterfaces;
27
28     // Class block.
29     private ArrayList<JMember> classBlock;
30
31     // Context for this class.
32     private ClassContext context;
33
34     // Whether this class has an explicit constructor.
35     private boolean hasExplicitConstructor;
36
37     // Instance fields of this class.
38     private ArrayList<JFieldDeclaration> instanceFieldInitializations;
39
40     // Static (class) fields of this class.
41     private ArrayList<JFieldDeclaration> staticFieldInitializations;
42
43     /**
44      * Constructs an AST node for a class declaration.
45      *
46      * @param line          line in which the class declaration occurs in the source file.
```

```
47      * @param mods           class modifiers.
48      * @param name           class name.
49      * @param superType      super class type.
50      * @param superInterfaces implemented interfaces.
51      * @param classBlock     class block.
52      */
53     public JClassDeclaration(int line, ArrayList<String> mods, String name, Type superType,
54                              ArrayList<TypeName> superInterfaces, ArrayList<JMember> classBlock) {
55       super(line);
56       this.mods = mods;
57       this.name = name;
58       this.superType = superType;
59       this.superInterfaces = superInterfaces;
60       this.classBlock = classBlock;
61       hasExplicitConstructor = false;
62       instanceFieldInitializations = new ArrayList<JFieldDeclaration>();
63       staticFieldInitializations = new ArrayList<JFieldDeclaration>();
64     }
65
66     /**
67      * Returns the initializations for instance fields (expressed as assignment statements).
68      *
69      * @return the initializations for instance fields (expressed as assignment statements).
70      */
71     public ArrayList<JFieldDeclaration> instanceFieldInitializations() {
72       return instanceFieldInitializations;
73     }
74
75     /**
76      * {@inheritDoc}
77      */
78     public void declareThisType(Context context) {
79       String qualifiedName = JAST.compilationUnit.packageName() == "" ?
80             name : JAST.compilationUnit.packageName() + "/" + name;
81       CLEmitter partial = new CLEmitter(false);
82       partial.addClass(mods, qualifiedName, Type.OBJECT.jvmName(), null, false);
83       thisType = Type.typeFor(partial.toClass());
84       context.addType(line, thisType);
85     }
86
87     /**
88      * {@inheritDoc}
89      */
90     public void preAnalyze(Context context) {
91       // Construct a class context.
92       this.context = new ClassContext(this, context);
93
94       // Resolve superclass.
95       superType = superType.resolve(this.context);
```

```
96
97          // Creating a partial class in memory can result in a java.lang.VerifyError if the
98          // semantics below are violated, so we can't defer these checks to analyze().
99          thisType.checkAccess(line, superType);
100         if (superType.isFinal()) {
101             JAST.compilationUnit.reportSemanticError(line, "Cannot extend a final type: %s",
102                     superType.toString());
103         }
104
105         // Create the (partial) class.
106         CLEmitter partial = new CLEmitter(false);
107
108         // Add the class header to the partial class
109         String qualifiedName = JAST.compilationUnit.packageName() == "" ?
110                 name : JAST.compilationUnit.packageName() + "/" + name;
111         partial.addClass(mods, qualifiedName, superType.jvmName(), null, false);
112
113         // Pre-analyze the members and add them to the partial class.
114         for (JMember member : classBlock) {
115             member.preAnalyze(this.context, partial);
116             hasExplicitConstructor =
117                     hasExplicitConstructor || member instanceof JConstructorDeclaration;
118         }
119
120         // Add the implicit empty constructor?
121         if (!hasExplicitConstructor) {
122             codegenPartialImplicitConstructor(partial);
123         }
124
125         // Get the ClassRep for the (partial) class and make it the representation for this type.
126         Type id = this.context.lookupType(name);
127         if (id != null && !JAST.compilationUnit.errorHasOccurred()) {
128             id.setClassRep(partial.toClass());
129         }
130     }
131
132     /**
133      * {@inheritDoc}
134      */
135     public String name() {
136         return name;
137     }
138
139     /**
140      * {@inheritDoc}
141      */
142     public Type thisType() {
143         return thisType;
144     }
```

```
145
146     /**
147      * {@inheritDoc}
148      */
149     public Type superType() {
150         return superType;
151     }
152
153     /**
154      * {@inheritDoc}
155      */
156     public ArrayList<TypeName> superInterfaces() {
157         return superInterfaces;
158     }
159
160     /**
161      * {@inheritDoc}
162      */
163     public JAST analyze(Context context) {
164         // Analyze all members.
165         for (JMember member : classBlock) {
166             ((JAST) member).analyze(this.context);
167         }
168
169         // Separate declared fields for purposes of initialization.
170         for (JMember member : classBlock) {
171             if (member instanceof JFieldDeclaration) {
172                 JFieldDeclaration fieldDecl = (JFieldDeclaration) member;
173                 if (fieldDecl.mods().contains("static")) {
174                     staticFieldInitializations.add(fieldDecl);
175                 } else {
176                     instanceFieldInitializations.add(fieldDecl);
177                 }
178             }
179         }
180
181         // Finally, ensure that a non-abstract class has no abstract methods.
182         if (!thisType.isAbstract() && thisType.abstractMethods().size() > 0) {
183             String methods = "";
184             for (Method method : thisType.abstractMethods()) {
185                 methods += "\n" + method;
186             }
187             JAST.compilationUnit.reportSemanticError(line,
188                     "Class must be abstract since it defines abstract methods: %s", methods);
189         }
190         return this;
191     }
192
193     /**
```

```
194     * {@inheritDoc}
195     */
196    public void codegen(CLEmitter output) {
197        // The class header.
198        String qualifiedName = JAST.compilationUnit.packageName() == "" ?
199                name : JAST.compilationUnit.packageName() + "/" + name;
200        output.addClass(mods, qualifiedName, superType.jvmName(), null, false);
201
202        // The implicit empty constructor?
203        if (!hasExplicitConstructor) {
204            codegenImplicitConstructor(output);
205        }
206
207        // The members.
208        for (JMember member : classBlock) {
209            ((JAST) member).codegen(output);
210        }
211
212        // Generate a class initialization method?
213        if (staticFieldInitializations.size() > 0) {
214            codegenClassInit(output);
215        }
216    }
217
218    /**
219     * {@inheritDoc}
220     */
221    public void toJSON(JSONElement json) {
222        JSONElement e = new JSONElement();
223        json.addChild("JClassDeclaration:" + line, e);
224        if (mods != null) {
225            ArrayList<String> value = new ArrayList<String>();
226            for (String mod : mods) {
227                value.add(String.format("\"%s\"", mod));
228            }
229            e.addAttribute("modifiers", value);
230        }
231        e.addAttribute("name", name);
232        e.addAttribute("super", superType == null ? "" : superType.toString());
233        if (superInterfaces != null) {
234            ArrayList<String> value = new ArrayList<String>();
235            for (TypeName impl : superInterfaces) {
236                value.add(String.format("\"%s\"", impl.toString()));
237            }
238            e.addAttribute("implements", value);
239        }
240        if (context != null) {
241            context.toJSON(e);
242        }
```

```
243         if (classBlock != null) {
244             for (JMember member : classBlock) {
245                 ((JAST) member).toJSON(e);
246             }
247         }
248     }
249 
250     // Generates code for an implicit empty constructor (necessary only if there is not already
251     // an explicit one).
252     private void codegenPartialImplicitConstructor(CLEmitter partial) {
253         ArrayList<String> mods = new ArrayList<String>();
254         mods.add("public");
255         partial.addMethod(mods, "<init>", "()V", null, false);
256         partial.addNoArgInstruction(ALOAD_0);
257         partial.addMemberAccessInstruction(INVOKESPECIAL, superType.jvmName(), "<init>", "()V");
258         partial.addNoArgInstruction(RETURN);
259     }
260 
261     // Generates code for an implicit empty constructor (necessary only if there is not already
262     // an explicit one).
263     private void codegenImplicitConstructor(CLEmitter output) {
264         ArrayList<String> mods = new ArrayList<String>();
265         mods.add("public");
266         output.addMethod(mods, "<init>", "()V", null, false);
267         output.addNoArgInstruction(ALOAD_0);
268         output.addMemberAccessInstruction(INVOKESPECIAL, superType.jvmName(), "<init>", "()V");
269 
270         // If there are instance field initializations, generate code for them.
271         for (JFieldDeclaration instanceField : instanceFieldInitializations) {
272             instanceField.codegenInitializations(output);
273         }
274 
275         output.addNoArgInstruction(RETURN);
276     }
277 
278     // Generates code for class initialization (in j-- this means static field initializations.
279     private void codegenClassInit(CLEmitter output) {
280         ArrayList<String> mods = new ArrayList<String>();
281         mods.add("public");
282         mods.add("static");
283         output.addMethod(mods, "<clinit>", "()V", null, false);
284 
285         // If there are static field initializations, generate code for them.
286         for (JFieldDeclaration staticField : staticFieldInitializations) {
287             staticField.codegenInitializations(output);
288         }
289 
290         output.addNoArgInstruction(RETURN);
291     }
```

```
292 }
293
```

JCastOp.java

Download

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

package jminusminus;

import java.util.Hashtable;

import static jminusminus.CLConstants.*;

/**
 * The AST for an cast expression, which has both a cast (a type) and the expression to be cast.
 */
class JCastOp extends JExpression {
    // The cast.
    private Type cast;

    // The expression we're casting.
    private JExpression expr;

    // The conversions table.
    private static Conversions conversions;

    // The converter to use for this cast.
    private Converter converter;

    /**
     * Constructs an AST node for a cast expression.
     *
     * @param line the line in which the expression occurs in the source file.
     * @param cast the type we're casting our expression as.
     * @param expr the expression we're casting.
     */
    public JCastOp(int line, Type cast, JExpression expr) {
        super(line);
        this.cast = cast;
        this.expr = expr;
        conversions = new Conversions();
    }

    /**
     * {@inheritDoc}
     */
    public JExpression analyze(Context context) {
        expr = (JExpression) expr.analyze(context);
        type = cast = cast.resolve(context);
        if (cast.equals(expr.type())) {
            converter = Converter.Identity;
```

```
47          } else if (cast.isJavaAssignableFrom(expr.type())) {
48              converter = Converter.WidenReference;
49          } else if (expr.type().isJavaAssignableFrom(cast)) {
50              converter = new NarrowReference(cast);
51          } else if (conversions.get(expr.type(), cast) != null) {
52              converter = conversions.get(expr.type(), cast);
53          } else {
54              JAST.compilationUnit.reportSemanticError(line,
55                      "Cannot cast a " + expr.type().toString() + " to a " + cast.toString());
56          }
57          return this;
58      }
59
60      /**
61       * {@inheritDoc}
62       */
63      public void codegen(CLEmitter output) {
64          expr.codegen(output);
65          converter.codegen(output);
66      }
67
68      /**
69       * {@inheritDoc}
70       */
71      public void toJSON(JSONElement json) {
72          JSONElement e = new JSONElement();
73          json.addChild("JCastOp:" + line, e);
74          e.addAttribute("type", cast == null ? "" : cast.toString());
75          JSONElement e1 = new JSONElement();
76          e.addChild("Expression", e1);
77          expr.toJSON(e1);
78      }
79  }
80
81  /**
82   * A 2D table of conversions, from one type to another.
83   */
84  class Conversions {
85      // Table of conversions; maps a source and target type pair to its converter.
86      private Hashtable<String, Converter> table;
87
88      /**
89       * Constructs a table of conversions and populates it.
90       */
91      public Conversions() {
92          table = new Hashtable<String, Converter>();
93
94          // Populate the table.
95          put(Type.CHAR, Type.INT, Converter.Identity);   // Widening
```

```
96          put(Type.INT, Type.CHAR, new I2C());          // Narrowing
97
98          // Widening
99          put(Type.INT, Type.LONG, new I2L());
100         put(Type.INT, Type.DOUBLE, new I2D());
101         put(Type.LONG, Type.DOUBLE, new L2D());
102
103         // Narrowing
104         put(Type.LONG, Type.INT, new L2I());
105         put(Type.DOUBLE, Type.INT, new D2I());
106         put(Type.DOUBLE, Type.LONG, new D2L());
107
108         // Boxing.
109         put(Type.CHAR, Type.BOXED_CHAR, new Boxing(Type.CHAR, Type.BOXED_CHAR));
110         put(Type.INT, Type.BOXED_INT, new Boxing(Type.INT, Type.BOXED_INT));
111         put(Type.BOOLEAN, Type.BOXED_BOOLEAN, new Boxing(Type.BOOLEAN, Type.BOXED_BOOLEAN));
112
113         put(Type.LONG, Type.BOXED_LONG, new Boxing(Type.LONG, Type.BOXED_LONG));
114         put(Type.DOUBLE, Type.BOXED_DOUBLE, new Boxing(Type.DOUBLE, Type.BOXED_DOUBLE));
115
116         // Un-boxing.
117         put(Type.BOXED_CHAR, Type.CHAR, new UnBoxing(Type.BOXED_CHAR, Type.CHAR, "charValue"));
118         put(Type.BOXED_INT, Type.INT, new UnBoxing(Type.BOXED_INT, Type.INT, "intValue"));
119         put(Type.BOXED_BOOLEAN, Type.BOOLEAN, new UnBoxing(Type.BOXED_BOOLEAN,
    Type.BOOLEAN,
120                 "booleanValue"));
121
122         put(Type.BOXED_LONG, Type.LONG, new UnBoxing(Type.BOXED_LONG, Type.LONG, "longValue"));
123         put(Type.BOXED_DOUBLE, Type.DOUBLE, new UnBoxing(Type.BOXED_DOUBLE, Type.DOUBLE,
    "doubleValue"));
124
125     }
126
127     /**
128      * Retrieves and returns a converter for converting from some original type to a target type.
129      *
130      * @param source the source type.
131      * @param target the target type.
132      * @return the converter.
133      */
134     public Converter get(Type source, Type target) {
135         return table.get(source.toDescriptor() + "2" + target.toDescriptor());
136     }
137
138     // Defines a conversion. This is used for populating the conversions table.
139     private void put(Type source, Type target, Converter c) {
140         table.put(source.toDescriptor() + "2" + target.toDescriptor(), c);
141     }
142 }
```

```
143
144 /**
145  * A Converter encapsulates any (possibly none) code necessary to perform a cast operation.
146  */
147 interface Converter {
148     /**
149      * For identity conversion (no run-time code needed).
150      */
151     public static Converter Identity = new Identity();
152
153     /**
154      * For widening conversion (no run-time code needed).
155      */
156     public static Converter WidenReference = Identity;
157
158     /**
159      * Emits code necessary to convert (cast) a source type to a target type.
160      *
161      * @param output the code emitter.
162      */
163     public void codegen(CLEmitter output);
164 }
165
166 /**
167  * An identity converter.
168  */
169 class Identity implements Converter {
170     /**
171      * {@inheritDoc}
172      */
173     public void codegen(CLEmitter output) {
174         // Nothing here.
175     }
176 }
177
178 /**
179  * A narrowing reference converter.
180  */
181 class NarrowReference implements Converter {
182     // The target type.
183     private Type target;
184
185     /**
186      * Constructs a narrowing reference converter.
187      *
188      * @param target the target type.
189      */
190     public NarrowReference(Type target) {
191         this.target = target;
```

```
    }

    /**
     * {@inheritDoc}
     */
    public void codegen(CLEmitter output) {
        output.addReferenceInstruction(CHECKCAST, target.jvmName());
    }
}

/**
 * A boxing converter.
 */
class Boxing implements Converter {
    // The source type.
    private Type source;

    // The target type.
    private Type target;

    /**
     * Constructs a Boxing converter.
     *
     * @param source the source type.
     * @param target the target type.
     */
    public Boxing(Type source, Type target) {
        this.source = source;
        this.target = target;
    }

    /**
     * {@inheritDoc}
     */
    public void codegen(CLEmitter output) {
        output.addMemberAccessInstruction(INVOKESTATIC, target.jvmName(), "valueOf",
                "(" + source.toDescriptor() + ")" + target.toDescriptor());
    }
}

/**
 * An un-boxing converter.
 */
class UnBoxing implements Converter {
    // The source type.
    private Type source;

    // The target type.
    private Type target;
```

```
241
242     // The Java method to invoke for the conversion.
243     private String methodName;
244
245     /**
246      * Constructs an UnBoxing converter.
247      *
248      * @param source     the source type.
249      * @param target     the target type.
250      * @param methodName the Java method to invoke for the conversion.
251      */
252     public UnBoxing(Type source, Type target, String methodName) {
253         this.source = source;
254         this.target = target;
255         this.methodName = methodName;
256     }
257
258     /**
259      * {@inheritDoc}
260      */
261     public void codegen(CLEmitter output) {
262         output.addMemberAccessInstruction(INVOKEVIRTUAL, source.jvmName(), methodName,
263                 "()" + target.toDescriptor());
264     }
265 }
266
267 /**
268  * An int to char converter.
269  */
270 class I2C implements Converter {
271     /**
272      * {@inheritDoc}
273      */
274     public void codegen(CLEmitter output) {
275         output.addNoArgInstruction(I2C);
276     }
277 }
278
279 /**
280  * A long to int converter.
281  */
282 class L2I implements Converter {
283     /**
284      * {@inheritDoc}
285      */
286     public void codegen(CLEmitter output) {
287         output.addNoArgInstruction(L2I);
288     }
289 }
```

```
290
291 /**
292  * A double to int converter.
293  */
294 class D2I implements Converter {
295     /**
296      * {@inheritDoc}
297      */
298     public void codegen(CLEmitter output) {
299         output.addNoArgInstruction(D2I);
300     }
301 }
302
303 /**
304  * A double to long converter.
305  */
306 class D2L implements Converter {
307     /**
308      * {@inheritDoc}
309      */
310     public void codegen(CLEmitter output) {
311         output.addNoArgInstruction(D2L);
312     }
313 }
314
315 /**
316  * An int to long converter.
317  */
318 class I2L implements Converter {
319     /**
320      * {@inheritDoc}
321      */
322     public void codegen(CLEmitter output) {
323         output.addNoArgInstruction(I2L);
324     }
325 }
326
327 /**
328  * An int to double converter.
329  */
330 class I2D implements Converter {
331     /**
332      * {@inheritDoc}
333      */
334     public void codegen(CLEmitter output) {
335         output.addNoArgInstruction(I2D);
336     }
337 }
338
```

```
/**
 * A long to double converter.
 */
class L2D implements Converter {
    /**
     * {@inheritDoc}
     */
    public void codegen(CLEmitter output) {
        output.addNoArgInstruction(L2D);
    }
}
```

JBreakStatement.java

Download

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

package jminusminus;

import static jminusminus.CLConstants.*;

/**
 * An AST node for a break-statement.
 */
public class JBreakStatement extends JStatement {
    /**
     * Constructs an AST node for a break-statement.
     *
     * @param line line in which the break-statement occurs in the source file.
     */
    public JBreakStatement(int line) {
        super(line);
    }

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

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

    /**
     * {@inheritDoc}
     */
    public void toJSON(JSONElement json) {
        JSONElement e = new JSONElement();
        json.addChild("JBreakStatement:" + line, e);
    }
}

```

JBooleanBinaryExpression.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 binary expressions that return booleans.
 */
abstract class JBooleanBinaryExpression extends JBinaryExpression {
    /**
     * Constructs an AST node for a boolean binary expression.
     *
     * @param line     line in which the boolean binary expression occurs in the source file.
     * @param operator the boolean binary operator.
     * @param lhs      lhs operand.
     * @param rhs      rhs operand.
     */

    protected JBooleanBinaryExpression(int line, String operator, JExpression lhs,
                                       JExpression rhs) {
        super(line, operator, lhs, rhs);
    }

    /**
     * {@inheritDoc}
     */
    public void codegen(CLEmitter output) {
        String falseLabel = output.createLabel();
        String trueLabel = output.createLabel();
        this.codegen(output, falseLabel, false);
        output.addNoArgInstruction(ICONST_1); // true
        output.addBranchInstruction(GOTO, trueLabel);
        output.addLabel(falseLabel);
        output.addNoArgInstruction(ICONST_0); // false
        output.addLabel(trueLabel);
    }
}

/**
 * The AST node for an equality (==) expression.
 */
class JEqualOp extends JBooleanBinaryExpression {
    /**
     * Constructs an AST node for an equality expression.
     *
```

```
47      * @param line line number in which the equality expression occurs in the source file.
48      * @param lhs  lhs operand.
49      * @param rhs  rhs operand.
50      */
51
52     public JEqualOp(int line, JExpression lhs, JExpression rhs) {
53         super(line, "==", lhs, rhs);
54     }
55
56     /**
57      * {@inheritDoc}
58      */
59     public JExpression analyze(Context context) {
60         lhs = (JExpression) lhs.analyze(context);
61         rhs = (JExpression) rhs.analyze(context);
62         lhs.type().mustMatchExpected(line(), rhs.type());
63         type = Type.BOOLEAN;
64         return this;
65     }
66
67     /**
68      * {@inheritDoc}
69      */
70     public void codegen(CLEmitter output, String targetLabel, boolean onTrue) {
71         lhs.codegen(output);
72         rhs.codegen(output);
73         if (lhs.type().isReference()) {
74             output.addBranchInstruction(onTrue ? IF_ACMPEQ : IF_ACMPNE, targetLabel);
75         } else {
76             output.addBranchInstruction(onTrue ? IF_ICMPEQ : IF_ICMPNE, targetLabel);
77         }
78     }
79 }
80
81 /**
82  * The AST node for a logical-and (&amp;&amp;) expression.
83  */
84 class JLogicalAndOp extends JBooleanBinaryExpression {
85     /**
86      * Constructs an AST node for a logical-and expression.
87      *
88      * @param line line in which the logical-and expression occurs in the source file.
89      * @param lhs  lhs operand.
90      * @param rhs  rhs operand.
91      */
92     public JLogicalAndOp(int line, JExpression lhs, JExpression rhs) {
93         super(line, "&&", lhs, rhs);
94     }
95
```

```
96      /**
97       * {@inheritDoc}
98       */
99      public JExpression analyze(Context context) {
100         lhs = (JExpression) lhs.analyze(context);
101         rhs = (JExpression) rhs.analyze(context);
102         lhs.type().mustMatchExpected(line(), Type.BOOLEAN);
103         rhs.type().mustMatchExpected(line(), Type.BOOLEAN);
104         type = Type.BOOLEAN;
105         return this;
106     }
107
108     /**
109      * {@inheritDoc}
110      */
111     public void codegen(CLEmitter output, String targetLabel, boolean onTrue) {
112         if (onTrue) {
113             String falseLabel = output.createLabel();
114             lhs.codegen(output, falseLabel, false);
115             rhs.codegen(output, targetLabel, true);
116             output.addLabel(falseLabel);
117         } else {
118             lhs.codegen(output, targetLabel, false);
119             rhs.codegen(output, targetLabel, false);
120         }
121     }
122 }
123
124 /**
125  * The AST node for a logical-or (||) expression.
126  */
127 class JLogicalOrOp extends JBooleanBinaryExpression {
128     /**
129      * Constructs an AST node for a logical-or expression.
130      *
131      * @param line line in which the logical-or expression occurs in the source file.
132      * @param lhs  lhs operand.
133      * @param rhs  rhs operand.
134      */
135     public JLogicalOrOp(int line, JExpression lhs, JExpression rhs) {
136         super(line, "||", lhs, rhs);
137     }
138
139     /**
140      * {@inheritDoc}
141      */
142     public JExpression analyze(Context context) {
143         // TODO
144         lhs = (JExpression) lhs.analyze(context);
```

```
145         rhs = (JExpression) rhs.analyze(context);
146         lhs.type().mustMatchExpected(line(), Type.BOOLEAN);
147         rhs.type().mustMatchExpected(line(), Type.BOOLEAN);
148         type = Type.BOOLEAN;
149         return this;
150     }
151 
152     /**
153      * {@inheritDoc}
154      */
155     public void codegen(CLEmitter output, String targetLabel, boolean onTrue) {
156         // TODO
157         if (onTrue) {
158             lhs.codegen(output, targetLabel, true);
159             rhs.codegen(output, targetLabel, true);
160         } else {
161             String falseLabel = output.createLabel();
162             lhs.codegen(output, falseLabel, true);
163             rhs.codegen(output, targetLabel, false);
164             output.addLabel(falseLabel);
165         }
166     }
167 }
168 
169 /**
170  * The AST node for a not-equal-to (!=) expression.
171  */
172 class JNotEqualOp extends JBooleanBinaryExpression {
173     /**
174      * Constructs an AST node for not-equal-to (!=) expression.
175      *
176      * @param line line number in which the not-equal-to (!=) expression occurs in the source file.
177      * @param lhs  lhs operand.
178      * @param rhs  rhs operand.
179      */
180 
181     public JNotEqualOp(int line, JExpression lhs, JExpression rhs) {
182         super(line, "!=", lhs, rhs);
183     }
184 
185     /**
186      * {@inheritDoc}
187      */
188     public JExpression analyze(Context context) {
189         // TODO
190         lhs = (JExpression) lhs.analyze(context);
191         rhs = (JExpression) rhs.analyze(context);
192         lhs.type().mustMatchExpected(line(), rhs.type());
193         type = Type.BOOLEAN;
```

```
194         return this;
195     }
196 
197     /**
198      * {@inheritDoc}
199      */
200     public void codegen(CLEmitter output, String targetLabel, boolean onTrue) {
201         // TODO
202         lhs.codegen(output);
203         rhs.codegen(output);
204         if (lhs.type().isReference()) {
205             output.addBranchInstruction(onTrue ? IF_ACMPNE : IF_ACMPEQ, targetLabel);
206         } else {
207             output.addBranchInstruction(onTrue ? IF_ICMPNE : IF_ICMPEQ, targetLabel);
208         }
209     }
210 }
211 
```

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          if (lhs.type() == Type.INT && rhs.type() == Type.INT) {
81              type = Type.INT;
82          } else if (lhs.type() == Type.LONG && rhs.type() == Type.LONG) {
83              type = Type.LONG;
84          } else if (lhs.type() == Type.DOUBLE && rhs.type == Type.DOUBLE) {
85              type = Type.DOUBLE;
86          }
87          return this;
88      }
89
90      /**
91       * {@inheritDoc}
92       */
93      public void codegen(CLEmitter output) {
94          lhs.codegen(output);
95          rhs.codegen(output);
```

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

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

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

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

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

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

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

```
439     * {@inheritDoc}
440     */
441    public void codegen(CLEmitter output) {
442        lhs.codegen(output);
443        rhs.codegen(output);
444        output.addNoArgInstruction(ISHL);
445    }
446 }
447 
448 /**
449  * The AST node for an arithmetic right shift (&rt;&rt;) expression.
450  */
451 class JARightShiftOp extends JBinaryExpression {
452    /**
453     * Constructs an AST node for an arithmetic right shift expression.
454     *
455     * @param line line in which the arithmetic right shift expression occurs in the source file.
456     * @param lhs  the lhs operand.
457     * @param rhs  the rhs operand.
458     */
459    public JARightShiftOp(int line, JExpression lhs, JExpression rhs) {
460        super(line, ">>", lhs, rhs);
461    }
462 
463    /**
464     * {@inheritDoc}
465     */
466    public JExpression analyze(Context context) {
467        lhs = (JExpression) lhs.analyze(context);
468        rhs = (JExpression) rhs.analyze(context);
469        lhs.type().mustMatchExpected(line(), Type.INT);
470        rhs.type().mustMatchExpected(line(), Type.INT);
471        type = Type.INT;
472        return this;
473    }
474 
475    /**
476     * {@inheritDoc}
477     */
478    public void codegen(CLEmitter output) {
479        lhs.codegen(output);
480        rhs.codegen(output);
481        output.addNoArgInstruction(ISHR);
482    }
483 }
484 
485 /**
486  * The AST node for a logical right shift (&rt;&rt;&rt;) expression.
487  */
```

```
class JLRightShiftOp extends JBinaryExpression {
    /**
     * Constructs an AST node for a logical right shift expression.
     *
     * @param line line in which the logical right shift expression occurs in the source file.
     * @param lhs  the lhs operand.
     * @param rhs  the rhs operand.
     */
    public JLRightShiftOp(int line, JExpression lhs, JExpression rhs) {
        super(line, ">>>", lhs, rhs);
    }

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

    /**
     * {@inheritDoc}
     */
    public void codegen(CLEmitter output) {
        lhs.codegen(output);
        rhs.codegen(output);
        output.addNoArgInstruction(IUSHR);
    }
}

```

JAssignment.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 assignment operation.
 */
abstract class JAssignment extends JBinaryExpression {
    /**
     * Constructs an AST node for an assignment operation.
     *
     * @param line     line in which the assignment operation occurs in the source file.
     * @param operator the assignment operator.
     * @param lhs      the lhs operand.
     * @param rhs      the rhs operand.
     */
    public JAssignment(int line, String operator, JExpression lhs, JExpression rhs) {
        super(line, operator, lhs, rhs);
    }
}

/**
 * The AST node for an assignment (=) operation.
 */
class JAssignOp extends JAssignment {
    /**
     * Constructs the AST node for an assignment (=) operation..
     *
     * @param line line in which the assignment operation occurs in the source file.
     * @param lhs  lhs operand.
     * @param rhs  rhs operand.
     */
    public JAssignOp(int line, JExpression lhs, JExpression rhs) {
        super(line, "=", lhs, rhs);
    }

    /**
     * {@inheritDoc}
     */
    public JExpression analyze(Context context) {
        if (!(lhs instanceof JLhs)) {
            JAST.compilationUnit.reportSemanticError(line(), "Illegal lhs for assignment");
        } else {
            lhs = (JExpression) ((JLhs) lhs).analyzeLhs(context);
```

```
47          }
48          rhs = (JExpression) rhs.analyze(context);
49          rhs.type().mustMatchExpected(line(), lhs.type());
50          type = rhs.type();
51          if (lhs instanceof JVariable) {
52              IDefn defn = ((JVariable) lhs).iDefn();
53              if (defn != null) {
54                  // Local variable; consider it to be initialized now.
55                  ((LocalVariableDefn) defn).initialize();
56              }
57          }
58          return this;
59      }
60
61      /**
62       * {@inheritDoc}
63       */
64      public void codegen(CLEmitter output) {
65          ((JLhs) lhs).codegenLoadLhsLvalue(output);
66          rhs.codegen(output);
67          if (!isStatementExpression) {
68              ((JLhs) lhs).codegenDuplicateRvalue(output);
69          }
70          ((JLhs) lhs).codegenStore(output);
71      }
72  }
73
74  /**
75   * The AST node for a plus-assign (+=) operation.
76   */
77  class JPlusAssignOp extends JAssignment {
78      /**
79       * Constructs the AST node for a plus-assign (+=) operation.
80       *
81       * @param line line in which the assignment operation occurs in the source file.
82       * @param lhs  the lhs operand.
83       * @param rhs  the rhs operand.
84       */
85      public JPlusAssignOp(int line, JExpression lhs, JExpression rhs) {
86          super(line, "+=", lhs, rhs);
87      }
88
89      /**
90       * {@inheritDoc}
91       */
92      public JExpression analyze(Context context) {
93          if (!(lhs instanceof JLhs)) {
94              JAST.compilationUnit.reportSemanticError(line(), "Illegal lhs for assignment");
95              return this;
```

```
96          } else {
97              lhs = (JExpression) ((JLhs) lhs).analyzeLhs(context);
98          }
99          rhs = (JExpression) rhs.analyze(context);
100         if (lhs.type().equals(Type.INT)) {
101             rhs.type().mustMatchExpected(line(), Type.INT);
102             type = Type.INT;
103         } else if (lhs.type().equals(Type.LONG)) {
104             rhs.type().mustMatchExpected(line(), Type.LONG);
105             type = Type.LONG;
106         } else if(lhs.type().equals(Type.DOUBLE)) {
107             rhs.type().mustMatchExpected(line(), Type.DOUBLE);
108             type = Type.DOUBLE;
109         } else if (lhs.type().equals(Type.STRING)) {
110             rhs = (new JStringConcatenationOp(line, lhs, rhs)).analyze(context);
111             type = Type.STRING;
112         } else {
113             JAST.compilationUnit.reportSemanticError(line(),
114                     "Invalid lhs type for +=: " + lhs.type());
115         }
116         return this;
117     }
118 
119     /**
120      * {@inheritDoc}
121      */
122     public void codegen(CLEmitter output) {
123         ((JLhs) lhs).codegenLoadLhsLvalue(output);
124         if (lhs.type().equals(Type.STRING)) {
125             rhs.codegen(output);
126         } else if (lhs.type().equals(Type.LONG)) {
127             ((JLhs) lhs).codegenLoadLhsRvalue(output);
128             rhs.codegen(output);
129             output.addNoArgInstruction(LADD);
130         } else if (lhs.type().equals(Type.DOUBLE)) {
131             ((JLhs) lhs).codegenLoadLhsRvalue(output);
132             rhs.codegen(output);
133             output.addNoArgInstruction(DADD);
134         } else if (lhs.type().equals(Type.INT)){
135             ((JLhs) lhs).codegenLoadLhsRvalue(output);
136             rhs.codegen(output);
137             output.addNoArgInstruction(IADD);
138         }
139         if (!isStatementExpression) {
140             ((JLhs) lhs).codegenDuplicateRvalue(output);
141         }
142         ((JLhs) lhs).codegenStore(output);
143     }
144 }
```

```
145
146 /**
147  * The AST node for a minus-assign (-=) operation.
148  */
149 class JMinusAssignOp extends JAssignment {
150     /**
151      * Constructs the AST node for a minus-assign operation.
152      *
153      * @param line line in which the assignment operation occurs in the source file.
154      * @param lhs  the lhs operand.
155      * @param rhs  the rhs operand.
156      */
157     public JMinusAssignOp(int line, JExpression lhs, JExpression rhs) {
158         super(line, "-=", lhs, rhs);
159     }
160
161     /**
162      * {@inheritDoc}
163      */
164     public JExpression analyze(Context context) {
165         if (!(lhs instanceof JLhs)) {
166             JAST.compilationUnit.reportSemanticError(line(), "Illegal lhs for assignment");
167             return this;
168         } else {
169             lhs = (JExpression) ((JLhs) lhs).analyzeLhs(context);
170         }
171         rhs = (JExpression) rhs.analyze(context);
172         if (lhs.type().equals(Type.INT)) {
173             rhs.type().mustMatchExpected(line(), Type.INT);
174             type = Type.INT;
175         } else if (lhs.type().equals(Type.LONG)) {
176             rhs.type().mustMatchExpected(line(), Type.LONG);
177             type = Type.LONG;
178         } else if(lhs.type().equals(Type.DOUBLE)) {
179             rhs.type().mustMatchExpected(line(), Type.DOUBLE);
180             type = Type.DOUBLE;
181         } else {
182             JAST.compilationUnit.reportSemanticError(line(),
183                     "Invalid lhs type for -=: " + lhs.type());
184         }
185         return this;
186     }
187
188     /**
189      * {@inheritDoc}
190      */
191     public void codegen(CLEmitter output) {
192         // TODO
193         ((JLhs) lhs).codegenLoadLhsLvalue(output);
```

```
194         if (lhs.type().equals(Type.LONG)) {
195             ((JLhs) lhs).codegenLoadLhsRvalue(output);
196             rhs.codegen(output);
197             output.addNoArgInstruction(LSUB);
198         } else if (lhs.type().equals(Type.DOUBLE)) {
199             ((JLhs) lhs).codegenLoadLhsRvalue(output);
200             rhs.codegen(output);
201             output.addNoArgInstruction(DSUB);
202         } else if (lhs.type().equals(Type.INT)){
203             ((JLhs) lhs).codegenLoadLhsRvalue(output);
204             rhs.codegen(output);
205             output.addNoArgInstruction(ISUB);
206         }
207         if (!isStatementExpression) {
208             ((JLhs) lhs).codegenDuplicateRvalue(output);
209         }
210         ((JLhs) lhs).codegenStore(output);
211     }
212 }
213 
214 /**
215  * The AST node for a star-assign (*=) operation.
216  */
217 class JStarAssignOp extends JAssignment {
218     /**
219      * Constructs the AST node for a star-assign operation.
220      *
221      * @param line line in which the assignment operation occurs in the source file.
222      * @param lhs  the lhs operand.
223      * @param rhs  the rhs operand.
224      */
225     public JStarAssignOp(int line, JExpression lhs, JExpression rhs) {
226         super(line, "*=", lhs, rhs);
227     }
228 
229     /**
230      * {@inheritDoc}
231      */
232     public JExpression analyze(Context context) {
233         // TODO
234         if (!(lhs instanceof JLhs)) {
235             JAST.compilationUnit.reportSemanticError(line(), "Illegal lhs for assignment");
236             return this;
237         } else {
238             lhs = (JExpression) ((JLhs) lhs).analyzeLhs(context);
239         }
240         rhs = (JExpression) rhs.analyze(context);
241         if (lhs.type().equals(Type.INT)) {
242             rhs.type().mustMatchExpected(line(), Type.INT);
```

```
243             type = Type.INT;
244         } else if (lhs.type().equals(Type.LONG)) {
245             rhs.type().mustMatchExpected(line(), Type.LONG);
246             type = Type.LONG;
247         } else if(lhs.type().equals(Type.DOUBLE)) {
248             rhs.type().mustMatchExpected(line(), Type.DOUBLE);
249             type = Type.DOUBLE;
250         } else {
251             JAST.compilationUnit.reportSemanticError(line(),
252                     "Invalid lhs type for *=: " + lhs.type());
253         }
254         return this;
255     }
256 
257     /**
258      * {@inheritDoc}
259      */
260     public void codegen(CLEmitter output) {
261         // TODO
262         ((JLhs) lhs).codegenLoadLhsLvalue(output);
263         if (lhs.type().equals(Type.LONG)) {
264             ((JLhs) lhs).codegenLoadLhsRvalue(output);
265             rhs.codegen(output);
266             output.addNoArgInstruction(LMUL);
267         } else if (lhs.type().equals(Type.DOUBLE)) {
268             ((JLhs) lhs).codegenLoadLhsRvalue(output);
269             rhs.codegen(output);
270             output.addNoArgInstruction(DMUL);
271         } else if (lhs.type().equals(Type.INT)){
272             ((JLhs) lhs).codegenLoadLhsRvalue(output);
273             rhs.codegen(output);
274             output.addNoArgInstruction(IMUL);
275         }
276         if (!isStatementExpression) {
277             ((JLhs) lhs).codegenDuplicateRvalue(output);
278         }
279         ((JLhs) lhs).codegenStore(output);
280     }
281 }
282 
283 /**
284  * The AST node for a div-assign (/=) operation.
285  */
286 class JDivAssignOp extends JAssignment {
287     /**
288      * Constructs the AST node for a div-assign operation.
289      *
290      * @param line line in which the assignment operation occurs in the source file.
291      * @param lhs  the lhs operand.
```

```
292      * @param rhs  the rhs operand.
293      */
294     public JDivAssignOp(int line, JExpression lhs, JExpression rhs) {
295         super(line, "/=", lhs, rhs);
296     }
297 
298     /**
299      * {@inheritDoc}
300      */
301     public JExpression analyze(Context context) {
302         // TODO
303         if (!(lhs instanceof JLhs)) {
304             JAST.compilationUnit.reportSemanticError(line(), "Illegal lhs for assignment");
305             return this;
306         } else {
307             lhs = (JExpression) ((JLhs) lhs).analyzeLhs(context);
308         }
309         rhs = (JExpression) rhs.analyze(context);
310         if (lhs.type().equals(Type.INT)) {
311             rhs.type().mustMatchExpected(line(), Type.INT);
312             type = Type.INT;
313         } else if (lhs.type().equals(Type.LONG)) {
314             rhs.type().mustMatchExpected(line(), Type.LONG);
315             type = Type.LONG;
316         } else if(lhs.type().equals(Type.DOUBLE)) {
317             rhs.type().mustMatchExpected(line(), Type.DOUBLE);
318             type = Type.DOUBLE;
319         } else {
320             JAST.compilationUnit.reportSemanticError(line(),
321                     "Invalid lhs type for /=: " + lhs.type());
322         }
323         return this;
324     }
325 
326     /**
327      * {@inheritDoc}
328      */
329     public void codegen(CLEmitter output) {
330         // TODO
331         ((JLhs) lhs).codegenLoadLhsLvalue(output);
332         if (lhs.type().equals(Type.LONG)) {
333             ((JLhs) lhs).codegenLoadLhsRvalue(output);
334             rhs.codegen(output);
335             output.addNoArgInstruction(LDIV);
336         } else if (lhs.type().equals(Type.DOUBLE)) {
337             ((JLhs) lhs).codegenLoadLhsRvalue(output);
338             rhs.codegen(output);
339             output.addNoArgInstruction(DDIV);
340         } else if (lhs.type().equals(Type.INT)){
```

```
341             ((JLhs) lhs).codegenLoadLhsRvalue(output);
342             rhs.codegen(output);
343             output.addNoArgInstruction(IDIV);
344         }
345         if (!isStatementExpression) {
346             ((JLhs) lhs).codegenDuplicateRvalue(output);
347         }
348         ((JLhs) lhs).codegenStore(output);
349     }
350 }
351 
352 /**
353  * The AST node for a rem-assign (%=) operation.
354  */
355 class JRemAssignOp extends JAssignment {
356     /**
357      * Constructs the AST node for a rem-assign operation.
358      *
359      * @param line line in which the assignment operation occurs in the source file.
360      * @param lhs  the lhs operand.
361      * @param rhs  the rhs operand.
362      */
363     public JRemAssignOp(int line, JExpression lhs, JExpression rhs) {
364         super(line, "%=", lhs, rhs);
365     }
366 
367     /**
368      * {@inheritDoc}
369      */
370     public JExpression analyze(Context context) {
371         // TODO
372         if (!(lhs instanceof JLhs)) {
373             JAST.compilationUnit.reportSemanticError(line(), "Illegal lhs for assignment");
374             return this;
375         } else {
376             lhs = (JExpression) ((JLhs) lhs).analyzeLhs(context);
377         }
378         rhs = (JExpression) rhs.analyze(context);
379         if (lhs.type().equals(Type.INT)) {
380             rhs.type().mustMatchExpected(line(), Type.INT);
381             type = Type.INT;
382         } else if (lhs.type().equals(Type.LONG)) {
383             rhs.type().mustMatchExpected(line(), Type.LONG);
384             type = Type.LONG;
385         } else if(lhs.type().equals(Type.DOUBLE)) {
386             rhs.type().mustMatchExpected(line(), Type.DOUBLE);
387             type = Type.DOUBLE;
388         } else {
389             JAST.compilationUnit.reportSemanticError(line(),
```

```
                "Invalid lhs type for %=: " + lhs.type());
        }
        return this;
    }

    /**
     * {@inheritDoc}
     */
    public void codegen(CLEmitter output) {
        // TODO
        ((JLhs) lhs).codegenLoadLhsLvalue(output);
        if (lhs.type().equals(Type.LONG)) {
            ((JLhs) lhs).codegenLoadLhsRvalue(output);
            rhs.codegen(output);
            output.addNoArgInstruction(LREM);
        } else if (lhs.type().equals(Type.DOUBLE)) {
            ((JLhs) lhs).codegenLoadLhsRvalue(output);
            rhs.codegen(output);
            output.addNoArgInstruction(DREM);
        } else if (lhs.type().equals(Type.INT)){
            ((JLhs) lhs).codegenLoadLhsRvalue(output);
            rhs.codegen(output);
            output.addNoArgInstruction(IREM);
        }
        if (!isStatementExpression) {
            ((JLhs) lhs).codegenDuplicateRvalue(output);
        }
        ((JLhs) lhs).codegenStore(output);
    }
}

/**
 * The AST node for an or-assign (|=) operation.
 */
class JOrAssignOp extends JAssignment {
    /**
     * Constructs the AST node for an or-assign operation.
     *
     * @param line line in which the assignment operation occurs in the source file.
     * @param lhs  the lhs operand.
     * @param rhs  the rhs operand.
     */
    public JOrAssignOp(int line, JExpression lhs, JExpression rhs) {
        super(line, "|=", lhs, rhs);
    }

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

```
439     public JExpression analyze(Context context) {
440         // TODO
441         if (!(lhs instanceof JLhs)) {
442             JAST.compilationUnit.reportSemanticError(line(), "Illegal lhs for assignment");
443             return this;
444         } else {
445             lhs = (JExpression) ((JLhs) lhs).analyzeLhs(context);
446         }
447         rhs = (JExpression) rhs.analyze(context);
448         if (lhs.type().equals(Type.INT)) {
449             rhs.type().mustMatchExpected(line(), Type.INT);
450             type = Type.INT;
451         } else {
452             JAST.compilationUnit.reportSemanticError(line(),
453                     "Invalid lhs type for |=: " + lhs.type());
454         }
455         return this;
456     }
457 
458     /**
459      * {@inheritDoc}
460      */
461     public void codegen(CLEmitter output) {
462         // TODO
463         ((JLhs) lhs).codegenLoadLhsRvalue(output);
464         rhs.codegen(output);
465         output.addNoArgInstruction(IOR);
466 
467         if (!isStatementExpression) {
468             ((JLhs) lhs).codegenDuplicateRvalue(output);
469         }
470         ((JLhs) lhs).codegenStore(output);
471     }
472 }
473 
474 /**
475  * The AST node for an and-assign (&amp;amp;=) operation.
476  */
477 class JAndAssignOp extends JAssignment {
478     /**
479      * Constructs the AST node for an and-assign operation.
480      *
481      * @param line line in which the assignment operation occurs in the source file.
482      * @param lhs  the lhs operand.
483      * @param rhs  the rhs operand.
484      */
485     public JAndAssignOp(int line, JExpression lhs, JExpression rhs) {
486         super(line, "&=", lhs, rhs);
487     }
```

```
488
489     /**
490      * {@inheritDoc}
491      */
492     public JExpression analyze(Context context) {
493         // TODO
494         if (!(lhs instanceof JLhs)) {
495             JAST.compilationUnit.reportSemanticError(line(), "Illegal lhs for assignment");
496             return this;
497         } else {
498             lhs = (JExpression) ((JLhs) lhs).analyzeLhs(context);
499         }
500         rhs = (JExpression) rhs.analyze(context);
501         if (lhs.type().equals(Type.INT)) {
502             rhs.type().mustMatchExpected(line(), Type.INT);
503             type = Type.INT;
504         } else {
505             JAST.compilationUnit.reportSemanticError(line(),
506                     "Invalid lhs type for &=: " + lhs.type());
507         }
508         return this;
509     }
510
511     /**
512      * {@inheritDoc}
513      */
514     public void codegen(CLEmitter output) {
515         // TODO
516         ((JLhs) lhs).codegenLoadLhsRvalue(output);
517         rhs.codegen(output);
518         output.addNoArgInstruction(IAND);
519
520         if (!isStatementExpression) {
521             ((JLhs) lhs).codegenDuplicateRvalue(output);
522         }
523         ((JLhs) lhs).codegenStore(output);
524     }
525 }
526
527 /**
528  * The AST node for an xor-assign (^=) operation.
529  */
530 class JXorAssignOp extends JAssignment {
531     /**
532      * Constructs the AST node for an xor-assign operation.
533      *
534      * @param line line in which the assignment operation occurs in the source file.
535      * @param lhs  the lhs operand.
536      * @param rhs  the rhs operand.
```

```
537      */
538     public JXorAssignOp(int line, JExpression lhs, JExpression rhs) {
539         super(line, "^=", lhs, rhs);
540     }
541 
542     /**
543      * {@inheritDoc}
544      */
545     public JExpression analyze(Context context) {
546         // TODO
547         if (!(lhs instanceof JLhs)) {
548             JAST.compilationUnit.reportSemanticError(line(), "Illegal lhs for assignment");
549             return this;
550         } else {
551             lhs = (JExpression) ((JLhs) lhs).analyzeLhs(context);
552         }
553         rhs = (JExpression) rhs.analyze(context);
554         if (lhs.type().equals(Type.INT)) {
555             rhs.type().mustMatchExpected(line(), Type.INT);
556             type = Type.INT;
557         } else {
558             JAST.compilationUnit.reportSemanticError(line(),
559                     "Invalid lhs type for ^=: " + lhs.type());
560         }
561         return this;
562     }
563 
564     /**
565      * {@inheritDoc}
566      */
567     public void codegen(CLEmitter output) {
568         // TODO
569         ((JLhs) lhs).codegenLoadLhsRvalue(output);
570         rhs.codegen(output);
571         output.addNoArgInstruction(IXOR);
572 
573         if (!isStatementExpression) {
574             ((JLhs) lhs).codegenDuplicateRvalue(output);
575         }
576         ((JLhs) lhs).codegenStore(output);
577     }
578 }
579 
580 /**
581  * The AST node for an arithmetic-left-shift-assign (&amp;lt;&amp;lt;=) operation.
582  */
583 class JALeftShiftAssignOp extends JAssignment {
584     /**
585      * Constructs the AST node for an arithmetic-left-shift-assign operation.
```

```
586      *
587      * @param line line in which the assignment operation occurs in the source file.
588      * @param lhs  the lhs operand.
589      * @param rhs  the rhs operand.
590      */
591     public JALeftShiftAssignOp(int line, JExpression lhs, JExpression rhs) {
592         super(line, "<<=", lhs, rhs);
593     }
594 
595     /**
596      * {@inheritDoc}
597      */
598     public JExpression analyze(Context context) {
599         // TODO
600         if (!(lhs instanceof JLhs)) {
601             JAST.compilationUnit.reportSemanticError(line(), "Illegal lhs for assignment");
602             return this;
603         } else {
604             lhs = (JExpression) ((JLhs) lhs).analyzeLhs(context);
605         }
606         rhs = (JExpression) rhs.analyze(context);
607         if (lhs.type().equals(Type.INT)) {
608             rhs.type().mustMatchExpected(line(), Type.INT);
609             type = Type.INT;
610         } else {
611             JAST.compilationUnit.reportSemanticError(line(),
612                     "Invalid lhs type for <<=: " + lhs.type());
613         }
614         return this;
615     }
616 
617     /**
618      * {@inheritDoc}
619      */
620     public void codegen(CLEmitter output) {
621         // TODO
622         ((JLhs) lhs).codegenLoadLhsRvalue(output);
623         rhs.codegen(output);
624         output.addNoArgInstruction(ISHL);
625 
626         if (!isStatementExpression) {
627             ((JLhs) lhs).codegenDuplicateRvalue(output);
628         }
629         ((JLhs) lhs).codegenStore(output);
630     }
631 }
632 
633 /**
634  * The AST node for an arithmetic-right-shift-assign (&gt;&gt;=) operation.
```

```
635  */
636 class JARightShiftAssignOp extends JAssignment {
637     /**
638      * Constructs the AST node for an arithmetic-right-shift-assign operation.
639      *
640      * @param line line in which the assignment operation occurs in the source file.
641      * @param lhs  the lhs operand.
642      * @param rhs  the rhs operand.
643      */
644     public JARightShiftAssignOp(int line, JExpression lhs, JExpression rhs) {
645         super(line, ">>=", lhs, rhs);
646     }
647 
648     /**
649      * {@inheritDoc}
650      */
651     public JExpression analyze(Context context) {
652         // TODO
653         if (!(lhs instanceof JLhs)) {
654             JAST.compilationUnit.reportSemanticError(line(), "Illegal lhs for assignment");
655             return this;
656         } else {
657             lhs = (JExpression) ((JLhs) lhs).analyzeLhs(context);
658         }
659         rhs = (JExpression) rhs.analyze(context);
660         if (lhs.type().equals(Type.INT)) {
661             rhs.type().mustMatchExpected(line(), Type.INT);
662             type = Type.INT;
663         } else {
664             JAST.compilationUnit.reportSemanticError(line(),
665                     "Invalid lhs type for >>=: " + lhs.type());
666         }
667         return this;
668     }
669 
670     /**
671      * {@inheritDoc}
672      */
673     public void codegen(CLEmitter output) {
674         // TODO
675         ((JLhs) lhs).codegenLoadLhsRvalue(output);
676         rhs.codegen(output);
677         output.addNoArgInstruction(ISHR);
678 
679         if (!isStatementExpression) {
680             ((JLhs) lhs).codegenDuplicateRvalue(output);
681         }
682         ((JLhs) lhs).codegenStore(output);
683     }
```

```
684 }
685
686 /**
687  * The AST node for an logical-right-shift-assign (&gt;&gt;&gt;=) operation.
688  */
689 class JLRightShiftAssignOp extends JAssignment {
690     /**
691      * Constructs the AST node for an logical-right-shift-assign operation.
692      *
693      * @param line line in which the assignment operation occurs in the source file.
694      * @param lhs  the lhs operand.
695      * @param rhs  the rhs operand.
696      */
697     public JLRightShiftAssignOp(int line, JExpression lhs, JExpression rhs) {
698         super(line, ">>>=", lhs, rhs);
699     }
700
701     /**
702      * {@inheritDoc}
703      */
704     public JExpression analyze(Context context) {
705         // TODO
706         if (!(lhs instanceof JLhs)) {
707             JAST.compilationUnit.reportSemanticError(line(), "Illegal lhs for assignment");
708             return this;
709         } else {
710             lhs = (JExpression) ((JLhs) lhs).analyzeLhs(context);
711         }
712         rhs = (JExpression) rhs.analyze(context);
713         if (lhs.type().equals(Type.INT)) {
714             rhs.type().mustMatchExpected(line(), Type.INT);
715             type = Type.INT;
716         } else {
717             JAST.compilationUnit.reportSemanticError(line(),
718                     "Invalid lhs type for >>>=: " + lhs.type());
719         }
720         return this;
721     }
722
723     /**
724      * {@inheritDoc}
725      */
726     public void codegen(CLEmitter output) {
727         // TODO
728         ((JLhs) lhs).codegenLoadLhsRvalue(output);
729         rhs.codegen(output);
730         output.addNoArgInstruction(IUSHR);
731
732         if (!isStatementExpression) {
```

```
733             ((JLhs) lhs).codegenDuplicateRvalue(output);
734         }
735         ((JLhs) lhs).codegenStore(output);
736     }
737 }
738 
```

JArrayInitializer.java

Download

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

package jminusminus;

import java.util.ArrayList;

import static jminusminus.CLConstants.*;

/**
 * The AST node for an array initializer.
 */
class JArrayInitializer extends JExpression {
    // The initializations.
    private ArrayList<JExpression> initials;

    /**
     * Constructs an AST node for an array initializer.
     *
     * @param line     line in which this array initializer occurs in the source file.
     * @param type     the type of the array we're initializing.
     * @param initials initializations.
     */
    public JArrayInitializer(int line, Type type, ArrayList<JExpression> initials) {
        super(line);
        this.type = type;
        this.initials = initials;
    }

    /**
     * {@inheritDoc}
     */
    public JExpression analyze(Context context) {
        type = type.resolve(context);
        if (!type.isArray()) {
            JAST.compilationUnit.reportSemanticError(line, "Cannot initialize a " + type.toString()
                    + " with an array sequence {...}");
            return this;
        }
        Type componentType = type.componentType();
        for (int i = 0; i < initials.size(); i++) {
            JExpression initial = initials.get(i);
            initials.set(i, initial = initial.analyze(context));
            if (!(initial instanceof JArrayInitializer)) {
                initial.type().mustMatchExpected(line, componentType);
            }
        }
```

```
47        return this;
48    }
49
50    /**
51     * {@inheritDoc}
52     */
53    public void codegen(CLEmitter output) {
54        Type componentType = type.componentType();
55
56        // Code to push array length.
57        (new JLiteralInt(line, String.valueOf(initials.size()))).codegen(output);
58
59        // Code to create the (empty) array.
60        output.addArrayInstruction(componentType.isReference() ? ANEWARRAY : NEWARRAY,
61                componentType.jvmName());
62
63        // Code to load initial values and store them as elements in the newly created array.
64        for (int i = 0; i < initials.size(); i++) {
65            JExpression initial = initials.get(i);
66
67            // Duplicate the array for each element store.
68            output.addNoArgInstruction(DUP);
69
70            // Code to push index for store.
71            (new JLiteralInt(line, String.valueOf(i))).codegen(output);
72
73            // Code to compute the initial value.
74            initial.codegen(output);
75
76            // Code to store the initial value in the array.
77            if (componentType == Type.INT) {
78                output.addNoArgInstruction(IASTORE);
79            } else if (componentType == Type.BOOLEAN) {
80                output.addNoArgInstruction(BASTORE);
81            } else if (componentType == Type.CHAR) {
82                output.addNoArgInstruction(CASTORE);
83            } else if (componentType == Type.LONG) {
84                output.addNoArgInstruction(LASTORE);
85            } else if (componentType == Type.DOUBLE) {
86                output.addNoArgInstruction(DASTORE);
87            } else if (!componentType.isPrimitive()) {
88                output.addNoArgInstruction(AASTORE);
89            }
90        }
91    }
92
93    /**
94     * {@inheritDoc}
95     */
```

```
96      public void toJSON(JSONElement json) {
97          JSONElement e = new JSONElement();
98          json.addChild("JArrayInitializer:" + line, e);
99          if (initials != null) {
100             for (JExpression initial : initials) {
101                 initial.toJSON(e);
102             }
103         }
104     }
105 }
106
```

JArrayExpression.java

Download

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

package jminusminus;

import static jminusminus.CLConstants.*;

/**
 * The AST for an array indexing operation. It has an expression denoting an array object and an
 * expression denoting an integer index.
 */
class JArrayExpression extends JExpression implements JLhs {
    // The array.
    private JExpression theArray;

    // The array index expression.
    private JExpression indexExpr;

    /**
     * Constructs an AST node for an array indexing operation.
     *
     * @param line      line in which the operation occurs in the source file.
     * @param theArray  the array we're indexing.
     * @param indexExpr the index expression.
     */
    public JArrayExpression(int line, JExpression theArray, JExpression indexExpr) {
        super(line);
        this.theArray = theArray;
        this.indexExpr = indexExpr;
    }

    /**
     * {@inheritDoc}
     */
    public JExpression analyze(Context context) {
        theArray = (JExpression) theArray.analyze(context);
        indexExpr = (JExpression) indexExpr.analyze(context);
        if (!(theArray.type().isArray())) {
            JAST.compilationUnit.reportSemanticError(line(), "attempt to index a non-array object");
            this.type = Type.ANY;
        } else {
            this.type = theArray.type().componentType();
        }
        indexExpr.type().mustMatchExpected(line(), Type.INT);
        return this;
    }

```

```
47      /**
48       * {@inheritDoc}
49       */
50      public JExpression analyzeLhs(Context context) {
51          analyze(context);
52          return this;
53      }
54
55      /**
56       * {@inheritDoc}
57       */
58      public void codegen(CLEmitter output) {
59          theArray.codegen(output);
60          indexExpr.codegen(output);
61          if (type == Type.INT) {
62              output.addNoArgInstruction(IALOAD);
63          } else if (type == Type.BOOLEAN) {
64              output.addNoArgInstruction(BALOAD);
65          } else if (type == Type.CHAR) {
66              output.addNoArgInstruction(CALOAD);
67          } else if (type == Type.LONG) {
68              output.addNoArgInstruction(LALOAD);
69          } else if (type == Type.DOUBLE) {
70              output.addNoArgInstruction(DALOAD);
71          } else if (!type.isPrimitive()) {
72              output.addNoArgInstruction(AALOAD);
73          }
74      }
75
76      /**
77       * {@inheritDoc}
78       */
79      public void codegen(CLEmitter output, String targetLabel, boolean onTrue) {
80          codegen(output);
81          if (onTrue) {
82              output.addBranchInstruction(IFNE, targetLabel);
83          } else {
84              output.addBranchInstruction(IFEQ, targetLabel);
85          }
86      }
87
88      /**
89       * {@inheritDoc}
90       */
91      public void codegenLoadLhsLvalue(CLEmitter output) {
92          theArray.codegen(output);
93          indexExpr.codegen(output);
94      }
95
```

```
96      /**
97       * {@inheritDoc}
98       */
99      public void codegenLoadLhsRvalue(CLEmitter output) {
100         if (type == Type.STRING) {
101             output.addNoArgInstruction(DUP2_X1);
102         } else {
103             output.addNoArgInstruction(DUP2);
104         }
105         if (type == Type.INT) {
106             output.addNoArgInstruction(IALOAD);
107         } else if (type == Type.BOOLEAN) {
108             output.addNoArgInstruction(BALOAD);
109         } else if (type == Type.CHAR) {
110             output.addNoArgInstruction(CALOAD);
111         } else if (type == Type.LONG) {
112             output.addNoArgInstruction(LALOAD);
113         } else if (type == Type.DOUBLE) {
114             output.addNoArgInstruction(DALOAD);
115         } else if (!type.isPrimitive()) {
116             output.addNoArgInstruction(AALOAD);
117         }
118     }
119
120     /**
121      * {@inheritDoc}
122      */
123     public void codegenDuplicateRvalue(CLEmitter output) {
124         output.addNoArgInstruction(DUP_X2);
125     }
126
127     /**
128      * {@inheritDoc}
129      */
130     public void codegenStore(CLEmitter output) {
131         if (type == Type.INT) {
132             output.addNoArgInstruction(IASTORE);
133         } else if (type == Type.BOOLEAN) {
134             output.addNoArgInstruction(BASTORE);
135         } else if (type == Type.CHAR) {
136             output.addNoArgInstruction(CASTORE);
137         } else if (type == Type.LONG) {
138             output.addNoArgInstruction(LASTORE);
139         } else if (type == Type.DOUBLE) {
140             output.addNoArgInstruction(DASTORE);
141         } else if (!type.isPrimitive()) {
142             output.addNoArgInstruction(AASTORE);
143         }
144     }
```

```
145
146     /**
147      * {@inheritDoc}
148      */
149     public void toJSON(JSONElement json) {
150         JSONElement e = new JSONElement();
151         json.addChild("JArrayExpression:" + line, e);
152         JSONElement e1 = new JSONElement();
153         e.addChild("TheArray", e1);
154         theArray.toJSON(e1);
155         JSONElement e2 = new JSONElement();
156         e.addChild("TheIndex", e2);
157         indexExpr.toJSON(e2);
158     }
159 }
160
```