Project 2 (Scanning)

● Graded

Student

Giancarlos Marte

Total Points

97.5 / 100 pts

Autograder Score

80.0 / 80.0

Passed Tests

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

Problem 1. Multiline Comment (15/15)

Problem 2. Operators (15/15)

Problem 3. Reserved Words (15/15)

Problem 4. Literals (15/15)

Question 2

Code Clarity and Efficiency

7.5 / 10 pts

Operators

- ✔ **+ 0.5 pts** Passed all tests
- ✔ **+ 1 pt** Followed good programming practices
- ✔ **+ 1 pt** Changes to scanner commented adequately

Multiline Comment

- ✔ **+ 0.5 pts** Passed all tests
- ✔ **+ 1 pt** Followed good programming practices
- ✔ **+ 1 pt** Changes to scanner commented adequately

Reserved Words

- ✔ **+ 0.5 pts** Passed all tests
- ✔ **+ 1 pt** Changes to scanner commented adequately
- ✔ **+ 1 pt** Followed good programming practices

Literals

- **+ 1 pt** Changes to scanner commented adequately
- **+ 1 pt** Followed good programming practices
- **+ 0.5 pts** Passed all tests

- **+ 0 pts** Does not meet expectations

Question 3

Notes File

10 / 10 pts

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

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

Autograder Results

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

ant

Problem 1. Multiline Comment (15/15)

j-- -t tests/MultiLineComment.java

Problem 2. Operators (15/15)

j-- -t tests/Operators.java

Problem 3. Reserved Words (15/15)

j-- -t tests/Keywords.java

Problem 4. Literals (15/15)

j-- -t tests/IntLiterals.java
j-- -t tests/LongLiterals.java
j-- -t tests/DoubleLiterals.java
j-- -t tests/MalformedLiterals.java □
'1\t [16 chars] = 1l1L\n2\t : <DOUBLE_LITERAL> = 9-L\n3\t : <[1070 chars]F> =' != '1\t [16 chars] = 1l\n1\t : <LON
Diff is 3686 characters long. Set self.maxDiff to None to see it.

Submitted Files

TokenInfo.java Download

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

package jminusminus;

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

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

    // Operators.
    ASSIGN("="), DEC("--"), EQUAL("=="), GT(">"), INC("++"), LAND("&&"), LE("<="), LNOT("!"),
    MINUS("-"), PLUS("+"), PLUS_ASSIGN("+="), STAR("*"), DIV("/"), REM("%"), LSHIFT("<<"),
    RSHIFT(">>"), LRSHIFT(">>>"), COMP("~"), AND("&"), XOR("^"), OR("|"),
    // New operators
    QUEST("?"), NOTEQ("!="), DIVEQ("/="), MINUSEQ("-="), STAREQ("*="), REMEQ("%="),
    RSHIFTEQ(">>="), LRSHIFTEQ(">>>="), GTEQ(">="), LSHIFTEQ("<<="), LT("<"),
    XOREQ("^="), OREQ("|="), LOR("||"), ANDEQ("&="),


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


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

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

```
47      LONG_LITERAL("<LONG_LITERAL>"), DOUBLE_LITERAL("<DOUBLE_LITERAL>");
48
49      // The token kind's string representation.
50      private String image;
51
52      /**
53       * Constructs an instance of TokenKind given its string representation.
54       *
55       * @param image string representation of the token kind.
56       */
57      private TokenKind(String image) {
58          this.image = image;
59      }
60
61      /**
62       * Returns the token kind's string representation.
63       *
64       * @return the token kind's string representation.
65       */
66      public String tokenRep() {
67          if (this == EOF) {
68              return "<EOF>";
69          }
70          if (image.startsWith("<") && image.endsWith(">")) {
71              return image;
72          }
73          return "\"" + image + "\"";
74      }
75
76      /**
77       * Returns the token kind's image.
78       *
79       * @return the token kind's image.
80       */
81      public String image() {
82          return image;
83      }
84  }
85
86  /**
87   * A representation of tokens returned by the Scanner method getNextToken(). A token has a kind
88   * identifying what kind of token it is, an image for providing any semantic text, and the line in
89   * which it occurred in the source file.
90   */
91  public class TokenInfo {
92      // Token kind.
93      private TokenKind kind;
94
95      // Semantic text (if any). For example, the identifier name when the token kind is IDENTIFIER
```

```
96      // . For tokens without a semantic text, it is simply its string representation. For example,
97      // "+=" when the token kind is PLUS_ASSIGN.
98      private String image;
99
100     // Line in which the token occurs in the source file.
101     private int line;
102
103     /**
104      * Constructs a TokenInfo object given its kind, the semantic text forming the token, and its
105      * line number.
106      *
107      * @param kind  the token's kind.
108      * @param image the semantic text forming the token.
109      * @param line  the line in which the token occurs in the source file.
110      */
111     public TokenInfo(TokenKind kind, String image, int line) {
112        this.kind = kind;
113        this.image = image;
114        this.line = line;
115     }
116
117     /**
118      * Constructs a TokenInfo object given its kind and its line number. Its image is simply the
119      * token kind's string representation.
120      *
121      * @param kind the token's identifying number.
122      * @param line the line in which the token occurs in the source file.
123      */
124     public TokenInfo(TokenKind kind, int line) {
125        this(kind, kind.image(), line);
126     }
127
128     /**
129      * Returns the token's kind.
130      *
131      * @return the token's kind.
132      */
133     public TokenKind kind() {
134        return kind;
135     }
136
137     /**
138      * Returns the line number associated with the token.
139      *
140      * @return the line number associated with the token.
141      */
142     public int line() {
143        return line;
144     }
```

```
145
146     /**
147      * Returns the token's string representation.
148      *
149      * @return the token's string representation.
150      */
151     public String tokenRep() {
152         return kind.tokenRep();
153     }
154
155     /**
156      * Returns the token's image.
157      *
158      * @return the token's image.
159      */
160     public String image() {
161         return image;
162     }
163 }
164
```

Scanner.java

Download

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

package jminusminus;

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

import static jminusminus.TokenKind.*;

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

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

    // Source characters.
    private CharReader input;

    // Next unscanned character.
    private char ch;

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

    // Source file name.
    private String fileName;

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

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

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

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

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

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

```
241             nextCh();
242             if (ch == '=') {
243                 nextCh();
244                 return new TokenInfo(MINUSEQ, line);
245             }
246             if (ch == '-') {
247                 nextCh();
248                 return new TokenInfo(DEC, line);
249             } else {
250                 return new TokenInfo(MINUS, line);
251             }
252         case '=':
253             nextCh();
254             if (ch == '=') {
255                 nextCh();
256                 return new TokenInfo(EQUAL, line);
257             }
258             else if (ch == '+') {
259                 nextCh();
260                 return new TokenInfo(PLUS, line);
261             }
262             else {
263                 return new TokenInfo(ASSIGN, line);
264             }
265         case '~':
266             nextCh();
267             return new TokenInfo(COMP, line);
268         case '>':
269             nextCh();
270             if (ch == '>') {
271                 nextCh();
272                 if (ch == '>') {
273                     nextCh();
274                     if (ch == '=') {
275                         nextCh();
276                         return new TokenInfo(LRSHIFTEQ, line);
277                     }
278                     else {
279                         return new TokenInfo(LRSHIFT, line);
280                     }
281                 }
282                 else if (ch == '=') {
283                     nextCh();
284                     return new TokenInfo(RSHIFTEQ, line);
285                 }
286                 else {
287                     return new TokenInfo(RSHIFT, line);
288                 }
289             }
```

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

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

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

```
436             else {
437                 return new TokenInfo(INT_LITERAL, buffer.toString(), line);
438             }
439         default:
440             if (isIdentifierStart(ch)) {
441                 buffer = new StringBuffer();
442                 while (isIdentifierPart(ch)) {
443                     buffer.append(ch);
444                     nextCh();
445                 }
446                 String identifier = buffer.toString();
447                 if (reserved.containsKey(identifier)) {
448                     return new TokenInfo(reserved.get(identifier), line);
449                 } else {
450                     return new TokenInfo(IDENTIFIER, identifier, line);
451                 }
452             } else {
453                 reportScannerError("Unidentified input token: '%c'", ch);
454                 nextCh();
455                 return getNextToken();
456             }
457     }
458 }
459
460 /**
461  * Returns true if an error has occurred, and false otherwise.
462  *
463  * @return true if an error has occurred, and false otherwise.
464  */
465 public boolean errorHasOccurred() {
466     return isInError;
467 }
468
469 /**
470  * Returns the name of the source file.
471  *
472  * @return the name of the source file.
473  */
474 public String fileName() {
475     return fileName;
476 }
477
478 // Scans and returns an escaped character.
479 private String escape() {
480     switch (ch) {
481         case 'b':
482             nextCh();
483             return "\\b";
484         case 't':
```

```
485                 nextCh();
486                 return "\\t";
487             case 'n':
488                 nextCh();
489                 return "\\n";
490             case 'f':
491                 nextCh();
492                 return "\\f";
493             case 'r':
494                 nextCh();
495                 return "\\r";
496             case "":
497                 nextCh();
498                 return "\\\"";
499             case '\':
500                 nextCh();
501                 return "\\'";
502             case '\\':
503                 nextCh();
504                 return "\\\\";
505             default:
506                 reportScannerError("Badly formed escape: \\%c", ch);
507                 nextCh();
508                 return "";
509         }
510     }
511
512     // Advances ch to the next character from input, and updates the line number.
513     private void nextCh() {
514         line = input.line();
515         try {
516             ch = input.nextChar();
517         } catch (Exception e) {
518             reportScannerError("Unable to read characters from input");
519         }
520     }
521
522     // Reports a lexical error and records the fact that an error has occurred. This fact can be
523     // ascertained from the Scanner by sending it an errorHasOccurred message.
524     private void reportScannerError(String message, Object... args) {
525         isInError = true;
526         System.err.printf("%s:%d: error: ", fileName, line);
527         System.err.printf(message, args);
528         System.err.println();
529     }
530
531     // Returns true if the specified character is a digit (0-9), and false otherwise.
532     private boolean isDigit(char c) {
533         return (c >= '0' && c <= '9');
```

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

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

Parser.java

Download

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

package jminusminus;

import java.util.ArrayList;

import static jminusminus.TokenKind.*;

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

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

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

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

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

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

```

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

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

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

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

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

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

```
341             JStatement alternate = have(ELSE) ? statement() : null;
342             return new JIfStatement(line, test, consequent, alternate);
343         } else if (have(RETURN)) {
344             if (have(SEMI)) {
345                 return new JReturnStatement(line, null);
346             } else {
347                 JExpression expr = expression();
348                 mustBe(SEMI);
349                 return new JReturnStatement(line, expr);
350             }
351         } else if (have(SEMI)) {
352             return new JEmptyStatement(line);
353         } else if (have(WHILE)) {
354             JExpression test = parExpression();
355             JStatement statement = statement();
356             return new JWhileStatement(line, test, statement);
357         } else {
358             // Must be a statementExpression.
359             JStatement statement = statementExpression();
360             mustBe(SEMI);
361             return statement;
362         }
363     }
364 
365     /**
366      * Parses and returns a list of formal parameters.
367      *
368      * <pre>
369      *   formalParameters ::= LPAREN [ formalParameter { COMMA formalParameter } ] RPAREN
370      * </pre>
371      *
372      * @return a list of formal parameters.
373      */
374     private ArrayList<JFormalParameter> formalParameters() {
375         ArrayList<JFormalParameter> parameters = new ArrayList<JFormalParameter>();
376         mustBe(LPAREN);
377         if (have(RPAREN)) {
378             return parameters;
379         }
380         do {
381             parameters.add(formalParameter());
382         } while (have(COMMA));
383         mustBe(RPAREN);
384         return parameters;
385     }
386 
387     /**
388      * Parses a formal parameter and returns an AST for it.
389      *
```

```
390     * <pre>
391     *   formalParameter ::= type IDENTIFIER
392     * </pre>
393     *
394     * @return an AST for a formal parameter.
395     */
396    private JFormalParameter formalParameter() {
397      int line = scanner.token().line();
398      Type type = type();
399      mustBe(IDENTIFIER);
400      String name = scanner.previousToken().image();
401      return new JFormalParameter(line, name, type);
402    }
403
404    /**
405     * Parses a parenthesized expression and returns an AST for it.
406     *
407     * <pre>
408     *   parExpression ::= LPAREN expression RPAREN
409     * </pre>
410     *
411     * @return an AST for a parenthesized expression.
412     */
413    private JExpression parExpression() {
414      mustBe(LPAREN);
415      JExpression expr = expression();
416      mustBe(RPAREN);
417      return expr;
418    }
419
420    /**
421     * Parses a local variable declaration statement and returns an AST for it.
422     *
423     * <pre>
424     *   localVariableDeclarationStatement ::= type variableDeclarators SEMI
425     * </pre>
426     *
427     * @return an AST for a local variable declaration statement.
428     */
429    private JVariableDeclaration localVariableDeclarationStatement() {
430      int line = scanner.token().line();
431      Type type = type();
432      ArrayList<JVariableDeclarator> vdecls = variableDeclarators(type);
433      mustBe(SEMI);
434      return new JVariableDeclaration(line, vdecls);
435    }
436
437    /**
438     * Parses and returns a list of variable declarators.
```

```
439      *
440      * <pre>
441      *   variableDeclarators ::= variableDeclarator { COMMA variableDeclarator }
442      * </pre>
443      *
444      * @param type type of the variables.
445      * @return a list of variable declarators.
446      */
447     private ArrayList<JVariableDeclarator> variableDeclarators(Type type) {
448       ArrayList<JVariableDeclarator> variableDeclarators = new ArrayList<JVariableDeclarator>();
449       do {
450         variableDeclarators.add(variableDeclarator(type));
451       } while (have(COMMA));
452       return variableDeclarators;
453     }
454
455     /**
456      * Parses a variable declarator and returns an AST for it.
457      *
458      * <pre>
459      *   variableDeclarator ::= IDENTIFIER [ ASSIGN variableInitializer ]
460      * </pre>
461      *
462      * @param type type of the variable.
463      * @return an AST for a variable declarator.
464      */
465     private JVariableDeclarator variableDeclarator(Type type) {
466       int line = scanner.token().line();
467       mustBe(IDENTIFIER);
468       String name = scanner.previousToken().image();
469       JExpression initial = have(ASSIGN) ? variableInitializer(type) : null;
470       return new JVariableDeclarator(line, name, type, initial);
471     }
472
473     /**
474      * Parses a variable initializer and returns an AST for it.
475      *
476      * <pre>
477      *   variableInitializer ::= arrayInitializer | expression
478      * </pre>
479      *
480      * @param type type of the variable.
481      * @return an AST for a variable initializer.
482      */
483     private JExpression variableInitializer(Type type) {
484       if (see(LCURLY)) {
485         return arrayInitializer(type);
486       }
487       return expression();
```

```
488     }
489
490     /**
491      * Parses an array initializer and returns an AST for it.
492      *
493      * <pre>
494      *   arrayInitializer ::= LCURLY [ variableInitializer { COMMA variableInitializer }
495      *                                 [ COMMA ] ] RCURLY
496      * </pre>
497      *
498      * @param type type of the array.
499      * @return an AST for an array initializer.
500      */
501     private JArrayInitializer arrayInitializer(Type type) {
502         int line = scanner.token().line();
503         ArrayList<JExpression> initials = new ArrayList<JExpression>();
504         mustBe(LCURLY);
505         if (have(RCURLY)) {
506             return new JArrayInitializer(line, type, initials);
507         }
508         initials.add(variableInitializer(type.componentType()));
509         while (have(COMMA)) {
510             initials.add(see(RCURLY) ? null : variableInitializer(type.componentType()));
511         }
512         mustBe(RCURLY);
513         return new JArrayInitializer(line, type, initials);
514     }
515
516     /**
517      * Parses and returns a list of arguments.
518      *
519      * <pre>
520      *   arguments ::= LPAREN [ expression { COMMA expression } ] RPAREN
521      * </pre>
522      *
523      * @return a list of arguments.
524      */
525     private ArrayList<JExpression> arguments() {
526         ArrayList<JExpression> args = new ArrayList<JExpression>();
527         mustBe(LPAREN);
528         if (have(RPAREN)) {
529             return args;
530         }
531         do {
532             args.add(expression());
533         } while (have(COMMA));
534         mustBe(RPAREN);
535         return args;
536     }
```

```
537
538     /**
539      * Parses and returns a type.
540      *
541      * <pre>
542      *   type ::= referenceType | basicType
543      * </pre>
544      *
545      * @return a type.
546      */
547     private Type type() {
548         if (seeReferenceType()) {
549             return referenceType();
550         }
551         return basicType();
552     }
553
554     /**
555      * Parses and returns a basic type.
556      *
557      * <pre>
558      *   basicType ::= BOOLEAN | CHAR | INT
559      * </pre>
560      *
561      * @return a basic type.
562      */
563     private Type basicType() {
564         if (have(BOOLEAN)) {
565             return Type.BOOLEAN;
566         } else if (have(CHAR)) {
567             return Type.CHAR;
568         } else if (have(INT)) {
569             return Type.INT;
570         } else {
571             reportParserError("Type sought where %s found", scanner.token().image());
572             return Type.ANY;
573         }
574     }
575
576     /**
577      * Parses and returns a reference type.
578      *
579      * <pre>
580      *   referenceType ::= basicType LBRACK RBRACK { LBRACK RBRACK }
581      *                   | qualifiedIdentifier { LBRACK RBRACK }
582      * </pre>
583      *
584      * @return a reference type.
585      */
```

```
586     private Type referenceType() {
587         Type type = null;
588         if (!see(IDENTIFIER)) {
589             type = basicType();
590             mustBe(LBRACK);
591             mustBe(RBRACK);
592             type = new ArrayTypeName(type);
593         } else {
594             type = qualifiedIdentifier();
595         }
596         while (seeDims()) {
597             mustBe(LBRACK);
598             mustBe(RBRACK);
599             type = new ArrayTypeName(type);
600         }
601         return type;
602     }
603 
604     /**
605      * Parses a statement expression and returns an AST for it.
606      *
607      * <pre>
608      *   statementExpression ::= expression
609      * </pre>
610      *
611      * @return an AST for a statement expression.
612      */
613     private JStatement statementExpression() {
614         int line = scanner.token().line();
615         JExpression expr = expression();
616         if (expr instanceof JAssignment
617                 || expr instanceof JPreIncrementOp
618                 || expr instanceof JPreDecrementOp
619                 || expr instanceof JPostIncrementOp
620                 || expr instanceof JPostDecrementOp
621                 || expr instanceof JMessageExpression
622                 || expr instanceof JSuperConstruction
623                 || expr instanceof JThisConstruction
624                 || expr instanceof JNewOp
625                 || expr instanceof JNewArrayOp) {
626             // So as not to save on stack.
627             expr.isStatementExpression = true;
628         } else {
629             reportParserError("Invalid statement expression; it does not have a side-effect");
630         }
631         return new JStatementExpression(line, expr);
632     }
633 
634     /**
```

```
635     * Parses an expression and returns an AST for it.
636     *
637     * <pre>
638     *   expression ::= assignmentExpression
639     * </pre>
640     *
641     * @return an AST for an expression.
642     */
643    private JExpression expression() {
644        return assignmentExpression();
645    }
646
647    /**
648     * Parses an assignment expression and returns an AST for it.
649     *
650     * <pre>
651     *   assignmentExpression ::= conditionalAndExpression
652     *                            [ ( ASSIGN | PLUS_ASSIGN ) assignmentExpression ]
653     * </pre>
654     *
655     * @return an AST for an assignment expression.
656     */
657    private JExpression assignmentExpression() {
658        int line = scanner.token().line();
659        JExpression lhs = conditionalAndExpression();
660        if (have(ASSIGN)) {
661            return new JAssignOp(line, lhs, assignmentExpression());
662        } else if (have(PLUS_ASSIGN)) {
663            return new JPlusAssignOp(line, lhs, assignmentExpression());
664        } else {
665            return lhs;
666        }
667    }
668
669    /**
670     * Parses a conditional-and expression and returns an AST for it.
671     *
672     * <pre>
673     *   conditionalAndExpression ::= equalityExpression { LAND equalityExpression }
674     * </pre>
675     *
676     * @return an AST for a conditional-and expression.
677     */
678    private JExpression conditionalAndExpression() {
679        int line = scanner.token().line();
680        boolean more = true;
681        JExpression lhs = equalityExpression();
682        while (more) {
683            if (have(LAND)) {
```

```
684             lhs = new JLogicalAndOp(line, lhs, equalityExpression());
685         } else {
686             more = false;
687         }
688       }
689       return lhs;
690     }
691
692     /**
693      * Parses an equality expression and returns an AST for it.
694      *
695      * <pre>
696      *   equalityExpression ::= relationalExpression { EQUAL relationalExpression }
697      * </pre>
698      *
699      * @return an AST for an equality expression.
700      */
701     private JExpression equalityExpression() {
702       int line = scanner.token().line();
703       boolean more = true;
704       JExpression lhs = relationalExpression();
705       while (more) {
706         if (have(EQUAL)) {
707             lhs = new JEqualOp(line, lhs, relationalExpression());
708         } else {
709             more = false;
710         }
711       }
712       return lhs;
713     }
714
715     /**
716      * Parses a relational expression and returns an AST for it.
717      *
718      * <pre>
719      *   relationalExpression ::= additiveExpression [ ( GT | LE ) additiveExpression
720      *                                | INSTANCEOF referenceType ]
721      * </pre>
722      *
723      * @return an AST for a relational expression.
724      */
725     private JExpression relationalExpression() {
726       int line = scanner.token().line();
727       JExpression lhs = additiveExpression();
728       if (have(GT)) {
729         return new JGreaterThanOp(line, lhs, additiveExpression());
730       } else if (have(LE)) {
731         return new JLessEqualOp(line, lhs, additiveExpression());
732       } else if (have(INSTANCEOF)) {
```

```
733             return new JInstanceOfOp(line, lhs, referenceType());
734         } else {
735             return lhs;
736         }
737     }
738 
739     /**
740      * Parses an additive expression and returns an AST for it.
741      *
742      * <pre>
743      *   additiveExpression ::= multiplicativeExpression { MINUS multiplicativeExpression }
744      * </pre>
745      *
746      * @return an AST for an additive expression.
747      */
748     private JExpression additiveExpression() {
749         int line = scanner.token().line();
750         boolean more = true;
751         JExpression lhs = multiplicativeExpression();
752         while (more) {
753             if (have(MINUS)) {
754                 lhs = new JSubtractOp(line, lhs, multiplicativeExpression());
755             } else if (have(PLUS)) {
756                 lhs = new JPlusOp(line, lhs, multiplicativeExpression());
757             } else {
758                 more = false;
759             }
760         }
761         return lhs;
762     }
763 
764     /**
765      * Parses a multiplicative expression and returns an AST for it.
766      *
767      * <pre>
768      *   multiplicativeExpression ::= unaryExpression { (STAR | DIV | REM
769      *    | LSHIFT | RSHIFT | LRSHIFT | AND | XOR | OR) unaryExpression }
770      * </pre>
771      *
772      * @return an AST for a multiplicative expression.
773      */
774     private JExpression multiplicativeExpression() {
775         int line = scanner.token().line();
776         boolean more = true;
777         JExpression lhs = unaryExpression();
778         while (more) {
779             if (have(STAR)) {
780                 lhs = new JMultiplyOp(line, lhs, unaryExpression());
781             }
```

```
782             else if (have(DIV)) {
783                 lhs = new JDivideOp(line, lhs, unaryExpression());
784             }
785             else if (have(REM)) {
786                 lhs = new JRemainderOp(line, lhs, unaryExpression());
787             }
788             else if (have(LSHIFT)) {
789                 lhs = new JALeftShiftOp(line, lhs, unaryExpression());
790             }
791             else if (have(RSHIFT)) {
792                 lhs = new JARightShiftOp(line, lhs, unaryExpression());
793             }
794             else if (have(LRSHIFT)) {
795                 lhs = new JLRightShiftOp(line, lhs, unaryExpression());
796             }
797             else if (have(AND)) {
798                 lhs = new JAndOp(line, lhs, unaryExpression());
799             }
800             else if (have(XOR)) {
801                 lhs = new JXorOp(line, lhs, unaryExpression());
802             }
803             else if (have(OR)) {
804                 lhs = new JOrOp(line, lhs, unaryExpression());
805             }
806             else {
807                 more = false;
808             }
809         }
810         return lhs;
811     }
812 
813     /**
814      * Parses an unary expression and returns an AST for it.
815      *
816      * <pre>
817      *   unaryExpression ::= INC unaryExpression
818      *                     | MINUS unaryExpression
819      *                     | simpleUnaryExpression
820      *                     | COMP unaryExpression
821      * </pre>
822      *
823      * @return an AST for an unary expression.
824      */
825     private JExpression unaryExpression() {
826         int line = scanner.token().line();
827         if (have(INC)) {
828             return new JPreIncrementOp(line, unaryExpression());
829         } else if (have(MINUS)) {
830             return new JNegateOp(line, unaryExpression());
```

```
831         }
832         else if (have(PLUS)) {
833             return new JUnaryPlusOp(line, unaryExpression());
834         }
835         else if (have(COMP)) {
836             return new JComplementOp(line, unaryExpression());
837         }
838         else {
839             return simpleUnaryExpression();
840         }
841     }
842 
843     /**
844      * Parses a simple unary expression and returns an AST for it.
845      *
846      * <pre>
847      *   simpleUnaryExpression ::= LNOT unaryExpression
848      *                           | LPAREN basicType RPAREN unaryExpression
849      *                           | LPAREN referenceType RPAREN simpleUnaryExpression
850      *                           | postfixExpression
851      * </pre>
852      *
853      * @return an AST for a simple unary expression.
854      */
855     private JExpression simpleUnaryExpression() {
856         int line = scanner.token().line();
857         if (have(LNOT)) {
858             return new JLogicalNotOp(line, unaryExpression());
859         } else if (seeCast()) {
860             mustBe(LPAREN);
861             boolean isBasicType = seeBasicType();
862             Type type = type();
863             mustBe(RPAREN);
864             JExpression expr = isBasicType ? unaryExpression() : simpleUnaryExpression();
865             return new JCastOp(line, type, expr);
866         } else {
867             return postfixExpression();
868         }
869     }
870 
871     /**
872      * Parses a postfix expression and returns an AST for it.
873      *
874      * <pre>
875      *   postfixExpression ::= primary { selector } { DEC }
876      * </pre>
877      *
878      * @return an AST for a postfix expression.
879      */
```

```
880     private JExpression postfixExpression() {
881         int line = scanner.token().line();
882         JExpression primaryExpr = primary();
883         while (see(DOT) || see(LBRACK)) {
884             primaryExpr = selector(primaryExpr);
885         }
886         while (have(DEC)) {
887             primaryExpr = new JPostDecrementOp(line, primaryExpr);
888         }
889         return primaryExpr;
890     }
891 
892     /**
893      * Parses a selector and returns an AST for it.
894      *
895      * <pre>
896      *   selector ::= DOT qualifiedIdentifier [ arguments ]
897      *              | LBRACK expression RBRACK
898      * </pre>
899      *
900      * @param target the target expression for this selector.
901      * @return an AST for a selector.
902      */
903     private JExpression selector(JExpression target) {
904         int line = scanner.token().line();
905         if (have(DOT)) {
906             // target.selector.
907             mustBe(IDENTIFIER);
908             String name = scanner.previousToken().image();
909             if (see(LPAREN)) {
910                 ArrayList<JExpression> args = arguments();
911                 return new JMessageExpression(line, target, name, args);
912             } else {
913                 return new JFieldSelection(line, target, name);
914             }
915         } else {
916             mustBe(LBRACK);
917             JExpression index = expression();
918             mustBe(RBRACK);
919             return new JArrayExpression(line, target, index);
920         }
921     }
922 
923     /**
924      * Parses a primary expression and returns an AST for it.
925      *
926      * <pre>
927      *   primary ::= parExpression
928      *             | NEW creator
```

```
929     *           | THIS [ arguments ]
930     *           | SUPER ( arguments | DOT IDENTIFIER [ arguments ] )
931     *           | qualifiedIdentifier [ arguments ]
932     *           | literal
933     * </pre>
934     *
935     * @return an AST for a primary expression.
936     */
937    private JExpression primary() {
938      int line = scanner.token().line();
939      if (see(LPAREN)) {
940        return parExpression();
941      } else if (have(NEW)) {
942        return creator();
943      } else if (have(THIS)) {
944        if (see(LPAREN)) {
945          ArrayList<JExpression> args = arguments();
946          return new JThisConstruction(line, args);
947        } else {
948          return new JThis(line);
949        }
950      } else if (have(SUPER)) {
951        if (!have(DOT)) {
952          ArrayList<JExpression> args = arguments();
953          return new JSuperConstruction(line, args);
954        } else {
955          mustBe(IDENTIFIER);
956          String name = scanner.previousToken().image();
957          JExpression newTarget = new JSuper(line);
958          if (see(LPAREN)) {
959            ArrayList<JExpression> args = arguments();
960            return new JMessageExpression(line, newTarget, null, name, args);
961          } else {
962            return new JFieldSelection(line, newTarget, name);
963          }
964        }
965      } else if (see(IDENTIFIER)) {
966        TypeName id = qualifiedIdentifier();
967        if (see(LPAREN)) {
968          // ambiguousPart.messageName(...).
969          ArrayList<JExpression> args = arguments();
970          return new JMessageExpression(line, null, ambiguousPart(id), id.simpleName(), args);
971        } else if (ambiguousPart(id) == null) {
972          // A simple name.
973          return new JVariable(line, id.simpleName());
974        } else {
975          // ambiguousPart.fieldName.
976          return new JFieldSelection(line, ambiguousPart(id), null, id.simpleName());
977        }
```

```
978         } else {
979             return literal();
980         }
981     }
982 
983     /**
984      * Parses a creator and returns an AST for it.
985      *
986      * <pre>
987      *   creator ::= ( basicType | qualifiedIdentifier )
988      *                 ( arguments
989      *                 | LBRACK RBRACK { LBRACK RBRACK } [ arrayInitializer ]
990      *                 | newArrayDeclarator
991      *                 )
992      * </pre>
993      *
994      * @return an AST for a creator.
995      */
996     private JExpression creator() {
997         int line = scanner.token().line();
998         Type type = seeBasicType() ? basicType() : qualifiedIdentifier();
999         if (see(LPAREN)) {
1000             ArrayList<JExpression> args = arguments();
1001             return new JNewOp(line, type, args);
1002         } else if (see(LBRACK)) {
1003             if (seeDims()) {
1004                 Type expected = type;
1005                 while (have(LBRACK)) {
1006                     mustBe(RBRACK);
1007                     expected = new ArrayTypeName(expected);
1008                 }
1009                 return arrayInitializer(expected);
1010             } else {
1011                 return newArrayDeclarator(line, type);
1012             }
1013         } else {
1014             reportParserError("( or [ sought where %s found", scanner.token().image());
1015             return new JWildExpression(line);
1016         }
1017     }
1018 
1019     /**
1020      * Parses a new array declarator and returns an AST for it.
1021      *
1022      * <pre>
1023      *   newArrayDeclarator ::= LBRACK expression RBRACK
1024      *                          { LBRACK expression RBRACK } { LBRACK RBRACK }
1025      * </pre>
1026      *
```

```
1027     * @param line line in which the declarator occurred.
1028     * @param type type of the array.
1029     * @return an AST for a new array declarator.
1030     */
1031    private JNewArrayOp newArrayDeclarator(int line, Type type) {
1032        ArrayList<JExpression> dimensions = new ArrayList<JExpression>();
1033        mustBe(LBRACK);
1034        dimensions.add(expression());
1035        mustBe(RBRACK);
1036        type = new ArrayTypeName(type);
1037        while (have(LBRACK)) {
1038            if (have(RBRACK)) {
1039                // We're done with dimension expressions.
1040                type = new ArrayTypeName(type);
1041                while (have(LBRACK)) {
1042                    mustBe(RBRACK);
1043                    type = new ArrayTypeName(type);
1044                }
1045                return new JNewArrayOp(line, type, dimensions);
1046            } else {
1047                dimensions.add(expression());
1048                type = new ArrayTypeName(type);
1049                mustBe(RBRACK);
1050            }
1051        }
1052        return new JNewArrayOp(line, type, dimensions);
1053    }
1054
1055    /**
1056     * Parses a literal and returns an AST for it.
1057     *
1058     * <pre>
1059     *   literal ::= CHAR_LITERAL | FALSE | INT_LITERAL | NULL | STRING_LITERAL | TRUE
1060     * </pre>
1061     *
1062     * @return an AST for a literal.
1063     */
1064    private JExpression literal() {
1065        int line = scanner.token().line();
1066        if (have(CHAR_LITERAL)) {
1067            return new JLiteralChar(line, scanner.previousToken().image());
1068        } else if (have(FALSE)) {
1069            return new JLiteralBoolean(line, scanner.previousToken().image());
1070        } else if (have(INT_LITERAL)) {
1071            return new JLiteralInt(line, scanner.previousToken().image());
1072        } else if (have(NULL)) {
1073            return new JLiteralNull(line);
1074        } else if (have(STRING_LITERAL)) {
1075            return new JLiteralString(line, scanner.previousToken().image());
```

```
1076        } else if (have(TRUE)) {
1077            return new JLiteralBoolean(line, scanner.previousToken().image());
1078        } else {
1079            reportParserError("Literal sought where %s found", scanner.token().image());
1080            return new JWildExpression(line);
1081        }
1082    }
1083
1084    ////////////////////////////////////////////////
1085    // Parsing Support
1086    // ////////////////////////////////////////////////
1087
1088    // Returns true if the current token equals sought, and false otherwise.
1089    private boolean see(TokenKind sought) {
1090        return (sought == scanner.token().kind());
1091    }
1092
1093    // If the current token equals sought, scans it and returns true. Otherwise, returns false
1094    // without scanning the token.
1095    private boolean have(TokenKind sought) {
1096        if (see(sought)) {
1097            scanner.next();
1098            return true;
1099        } else {
1100            return false;
1101        }
1102    }
1103
1104    // Attempts to match a token we're looking for with the current input token. On success,
1105    // scans the token and goes into a "Recovered" state. On failure, what happens next depends
1106    // on whether or not the parser is currently in a "Recovered" state: if so, it reports the
1107    // error and goes into an "Unrecovered" state; if not, it repeatedly scans tokens until it
1108    // finds the one it is looking for (or EOF) and then returns to a "Recovered" state. This
1109    // gives us a kind of poor man's syntactic error recovery, a strategy due to David Turner and
1110    // Ron Morrison.
1111    private void mustBe(TokenKind sought) {
1112        if (scanner.token().kind() == sought) {
1113            scanner.next();
1114            isRecovered = true;
1115        } else if (isRecovered) {
1116            isRecovered = false;
1117            reportParserError("%s found where %s sought", scanner.token().image(), sought.image());
1118        } else {
1119            // Do not report the (possibly spurious) error, but rather attempt to recover by
1120            // forcing a match.
1121            while (!see(sought) && !see(EOF)) {
1122                scanner.next();
1123            }
1124            if (see(sought)) {
```

```
1125                scanner.next();
1126                isRecovered = true;
1127            }
1128        }
1129    }
1130
1131    // Pulls out and returns the ambiguous part of a name.
1132    private AmbiguousName ambiguousPart(TypeName name) {
1133        String qualifiedName = name.toString();
1134        int i = qualifiedName.lastIndexOf('.');
1135        return i == -1 ? null : new AmbiguousName(name.line(), qualifiedName.substring(0, i));
1136    }
1137
1138    // Reports a syntax error.
1139    private void reportParserError(String message, Object... args) {
1140        isInError = true;
1141        isRecovered = false;
1142        System.err.printf("%s:%d: error: ", scanner.fileName(), scanner.token().line());
1143        System.err.printf(message, args);
1144        System.err.println();
1145    }
1146
1147    //////////////////////////////////////////////////
1148    // Lookahead Methods
1149    //////////////////////////////////////////////////
1150
1151    // Returns true if we are looking at an IDENTIFIER followed by a LPAREN, and false otherwise.
1152    private boolean seeIdentLParen() {
1153        scanner.recordPosition();
1154        boolean result = have(IDENTIFIER) && see(LPAREN);
1155        scanner.returnToPosition();
1156        return result;
1157    }
1158
1159    // Returns true if we are looking at a cast (basic or reference), and false otherwise.
1160    private boolean seeCast() {
1161        scanner.recordPosition();
1162        if (!have(LPAREN)) {
1163            scanner.returnToPosition();
1164            return false;
1165        }
1166        if (seeBasicType()) {
1167            scanner.returnToPosition();
1168            return true;
1169        }
1170        if (!see(IDENTIFIER)) {
1171            scanner.returnToPosition();
1172            return false;
1173        } else {
```

```
1174            scanner.next();
1175            // A qualified identifier is ok.
1176            while (have(DOT)) {
1177                if (!have(IDENTIFIER)) {
1178                    scanner.returnToPosition();
1179                    return false;
1180                }
1181            }
1182        }
1183        while (have(LBRACK)) {
1184            if (!have(RBRACK)) {
1185                scanner.returnToPosition();
1186                return false;
1187            }
1188        }
1189        if (!have(RPAREN)) {
1190            scanner.returnToPosition();
1191            return false;
1192        }
1193        scanner.returnToPosition();
1194        return true;
1195    }
1196
1197    // Returns true if we are looking at a local variable declaration, and false otherwise.
1198    private boolean seeLocalVariableDeclaration() {
1199        scanner.recordPosition();
1200        if (have(IDENTIFIER)) {
1201            // A qualified identifier is ok.
1202            while (have(DOT)) {
1203                if (!have(IDENTIFIER)) {
1204                    scanner.returnToPosition();
1205                    return false;
1206                }
1207            }
1208        } else if (seeBasicType()) {
1209            scanner.next();
1210        } else {
1211            scanner.returnToPosition();
1212            return false;
1213        }
1214        while (have(LBRACK)) {
1215            if (!have(RBRACK)) {
1216                scanner.returnToPosition();
1217                return false;
1218            }
1219        }
1220        if (!have(IDENTIFIER)) {
1221            scanner.returnToPosition();
1222            return false;
```

```
1223        }
1224        while (have(LBRACK)) {
1225            if (!have(RBRACK)) {
1226                scanner.returnToPosition();
1227                return false;
1228            }
1229        }
1230        scanner.returnToPosition();
1231        return true;
1232    }
1233
1234    // Returns true if we are looking at a basic type, and false otherwise.
1235    private boolean seeBasicType() {
1236        return (see(BOOLEAN) || see(CHAR) || see(INT));
1237    }
1238
1239    // Returns true if we are looking at a reference type, and false otherwise.
1240    private boolean seeReferenceType() {
1241        if (see(IDENTIFIER)) {
1242            return true;
1243        } else {
1244            scanner.recordPosition();
1245            if (have(BOOLEAN) || have(CHAR) || have(INT)) {
1246                if (have(LBRACK) && see(RBRACK)) {
1247                    scanner.returnToPosition();
1248                    return true;
1249                }
1250            }
1251            scanner.returnToPosition();
1252        }
1253        return false;
1254    }
1255
1256    // Returns true if we are looking at a [] pair, and false otherwise.
1257    private boolean seeDims() {
1258        scanner.recordPosition();
1259        boolean result = have(LBRACK) && see(RBRACK);
1260        scanner.returnToPosition();
1261        return result;
1262    }
1263 }
1264
```

JUnaryExpression.java

Download

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

package jminusminus;

import static jminusminus.CLConstants.*;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

JBinaryExpression.java

Download

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

package jminusminus;

import static jminusminus.CLConstants.*;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

notes.txt Download

1. Provide a high-level description (ie, using minimal amount of technical
jargon) of the project in no more than 200 words.

The main goal of this project was to modify and learn more about the front end of the j-- compiler, which included the TokenInfo and Scanner java files. The first problem was focused on adding multiline comments support. To do this a state diagram had to be created to base our code off of.
The next part of the project was adding support for additional operators and reserved words, which was similar to project 1. The last feature to implement was support for long and double literals.
This part of the project had us extend the existing integer state diagram in order to guide our code.
The things added to TokenInfo included new operator symbols and new reserved words.
The Scanner was changed to add the words into the hash table and to scan for the operator symbols.

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

Name	Status	Help Received
...	...	...

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

This project was interesting as it had a lot of similarities to the first one, but there was more subtle complexities involved. It was overall a good learning experience, especially when it came to making the state diagrams.