```
1  // Copyright 2013 Bill Campbell, Swami Iyer and Bahar Akbal-Delibas
2
3  package jminusminus;
4
5  import java.io.FileNotFoundException;
6  import java.io.FileReader;
7  import java.io.IOException;
8  import java.io.LineNumberReader;
9  import java.util.Hashtable;
10 import static jminusminus.TokenKind.*;
11
12 /**
13  * A lexical analyzer for j--, that has no backtracking mechanism.
14  *
15  * When you add a new token to the scanner, you must also add an entry in the
16  * TokenKind enum in TokenInfo.java specifying the kind and image of the new
17  * token.
18  */
19
20 class Scanner {
21
22     /** End of file character. */
23     public final static char EOFCH = CharReader.EOFCH;
24
25     /** Keywords in j--. */
26     private Hashtable<String, TokenKind> reserved;
27
28     /** Source characters. */
29     private CharReader input;
30
31     /** Next unscanned character. */
32     private char ch;
33
34     /** Whether a scanner error has been found. */
35     private boolean isInError;
36
37     /** Source file name. */
38     private String fileName;
39
40     /** Line number of current token. */
41     private int line;
42
43     /**
44      * Construct a Scanner object.
45      *
46      * @param fileName
47      *            the name of the file containing the source.
48      * @exception FileNotFoundException
49      *                when the named file cannot be found.
50      */
51
52     public Scanner(String fileName) throws FileNotFoundException {
53         this.input = new CharReader(fileName);
54         this.fileName = fileName;
55         isInError = false;
56
57         // Keywords in j--
58         reserved = new Hashtable<String, TokenKind>();
59         reserved.put(ABSTRACT.image(), ABSTRACT);
60         reserved.put(BOOLEAN.image(), BOOLEAN);
61         reserved.put(CHAR.image(), CHAR);
62         reserved.put(CLASS.image(), CLASS);
63         reserved.put(ELSE.image(), ELSE);
64         reserved.put(EXTENDS.image(), EXTENDS);
65         reserved.put(FALSE.image(), FALSE);
66         reserved.put(IF.image(), IF);
```

```
        reserved.put(IMPORT.image(), IMPORT);
        reserved.put(INSTANCEOF.image(), INSTANCEOF);
        reserved.put(INT.image(), INT);
        reserved.put(NEW.image(), NEW);
        reserved.put(NULL.image(), NULL);
        reserved.put(PACKAGE.image(), PACKAGE);
        reserved.put(PRIVATE.image(), PRIVATE);
        reserved.put(PROTECTED.image(), PROTECTED);
        reserved.put(PUBLIC.image(), PUBLIC);
        reserved.put(RETURN.image(), RETURN);
        reserved.put(STATIC.image(), STATIC);
        reserved.put(SUPER.image(), SUPER);
        reserved.put(THIS.image(), THIS);
        reserved.put(TRUE.image(), TRUE);
        reserved.put(VOID.image(), VOID);
        reserved.put(WHILE.image(), WHILE);

        // Prime the pump.
        nextCh();
    }

    /**
     * Scan the next token from input.
     *
     * @return the the next scanned token.
     */

    public TokenInfo getNextToken() {
        StringBuffer buffer;
        boolean moreWhiteSpace = true;
        while (moreWhiteSpace) {
            while (isWhitespace(ch)) {
                nextCh();
            }
            if (ch == '/') {
                nextCh();
                if (ch == '/') {
                    // CharReader maps all new lines to '\n'
                    while (ch != '\n' && ch != EOFCH) {
                        nextCh();
                    }
                } else {
                    reportScannerError("Operator / is not supported in j--.");
                }
            } else {
                moreWhiteSpace = false;
            }
        }
        line = input.line();
        switch (ch) {
        case '(':
            nextCh();
            return new TokenInfo(LPAREN, line);
        case ')':
            nextCh();
            return new TokenInfo(RPAREN, line);
        case '{':
            nextCh();
            return new TokenInfo(LCURLY, line);
        case '}':
            nextCh();
            return new TokenInfo(RCURLY, line);
        case '[':
            nextCh();
            return new TokenInfo(LBRACK, line);
        case ']':
            nextCh();
            return new TokenInfo(RBRACK, line);
        case ';':
```

```
            nextCh();
            return new TokenInfo(SEMI, line);
        case ',':
            nextCh();
            return new TokenInfo(COMMA, line);
        case '=':
            nextCh();
            if (ch == '=') {
                nextCh();
                return new TokenInfo(EQUAL, line);
            } else {
                return new TokenInfo(ASSIGN, line);
            }
        case '!':
            nextCh();
            return new TokenInfo(LNOT, line);
        case '*':
            nextCh();
            return new TokenInfo(STAR, line);
        case '+':
            nextCh();
            if (ch == '=') {
                nextCh();
                return new TokenInfo(PLUS_ASSIGN, line);
            } else if (ch == '+') {
                nextCh();
                return new TokenInfo(INC, line);
            } else {
                return new TokenInfo(PLUS, line);
            }
        case '-':
            nextCh();
            if (ch == '-') {
                nextCh();
                return new TokenInfo(DEC, line);
            } else {
                return new TokenInfo(MINUS, line);
            }
        case '&':
            nextCh();
            if (ch == '&') {
                nextCh();
                return new TokenInfo(LAND, line);
            } else {
                reportScannerError("Operator & is not supported in j--.");
                return getNextToken();
            }
        case '>':
            nextCh();
            return new TokenInfo(GT, line);
        case '<':
            nextCh();
            if (ch == '=') {
                nextCh();
                return new TokenInfo(LE, line);
            } else {
                reportScannerError("Operator < is not supported in j--.");
                return getNextToken();
            }
        case '\'':
            buffer = new StringBuffer();
            buffer.append('\'');
            nextCh();
            if (ch == '\\') {
                nextCh();
                buffer.append(escape());
            } else {
                buffer.append(ch);
                nextCh();
```

```
205             }
206             if (ch == '\'') {
207                 buffer.append('\'');
208                 nextCh();
209                 return new TokenInfo(CHAR_LITERAL, buffer.toString(), line);
210             } else {
211                 // Expected a ' ; report error and try to
212                 // recover.
213                 reportScannerError(ch
214                         + " found by scanner where closing ' was expected.");
215                 while (ch != '\'' && ch != ';' && ch != '\n') {
216                     nextCh();
217                 }
218                 return new TokenInfo(CHAR_LITERAL, buffer.toString(), line);
219             }
220         case '"':
221             buffer = new StringBuffer();
222             buffer.append("\"");
223             nextCh();
224             while (ch != '"' && ch != '\n' && ch != EOFCH) {
225                 if (ch == '\\') {
226                     nextCh();
227                     buffer.append(escape());
228                 } else {
229                     buffer.append(ch);
230                     nextCh();
231                 }
232             }
233             if (ch == '\n') {
234                 reportScannerError("Unexpected end of line found in String");
235             } else if (ch == EOFCH) {
236                 reportScannerError("Unexpected end of file found in String");
237             } else {
238                 // Scan the closing "
239                 nextCh();
240                 buffer.append("\"");
241             }
242             return new TokenInfo(STRING_LITERAL, buffer.toString(), line);
243         case '.':
244             nextCh();
245             return new TokenInfo(DOT, line);
246         case EOFCH:
247             return new TokenInfo(EOF, line);
248         case '0':
249             // Handle only simple decimal integers for now.
250             nextCh();
251             return new TokenInfo(INT_LITERAL, "0", line);
252         case '1':
253         case '2':
254         case '3':
255         case '4':
256         case '5':
257         case '6':
258         case '7':
259         case '8':
260         case '9':
261             buffer = new StringBuffer();
262             while (isDigit(ch)) {
263                 buffer.append(ch);
264                 nextCh();
265             }
266             return new TokenInfo(INT_LITERAL, buffer.toString(), line);
267         default:
268             if (isIdentifierStart(ch)) {
269                 buffer = new StringBuffer();
270                 while (isIdentifierPart(ch)) {
271                     buffer.append(ch);
272                     nextCh();
273                 }
```

```
                String identifier = buffer.toString();
                if (reserved.containsKey(identifier)) {
                    return new TokenInfo(reserved.get(identifier), line);
                } else {
                    return new TokenInfo(IDENTIFIER, identifier, line);
                }
            } else {
                reportScannerError("Unidentified input token: '%c'", ch);
                nextCh();
                return getNextToken();
            }
        }
    }

    /**
     * Scan and return an escaped character.
     *
     * @return escaped character.
     */

    private String escape() {
        switch (ch) {
        case 'b':
            nextCh();
            return "\\b";
        case 't':
            nextCh();
            return "\\t";
        case 'n':
            nextCh();
            return "\\n";
        case 'f':
            nextCh();
            return "\\f";
        case 'r':
            nextCh();
            return "\\r";
        case '"':
            nextCh();
            return "\"";
        case '\'':
            nextCh();
            return "\\'";
        case '\\':
            nextCh();
            return "\\\\";
        default:
            reportScannerError("Badly formed escape: \\%c", ch);
            nextCh();
            return "";
        }
    }

    /**
     * Advance ch to the next character from input, and update the line number.
     */

    private void nextCh() {
        line = input.line();
        try {
            ch = input.nextChar();
        } catch (Exception e) {
            reportScannerError("Unable to read characters from input");
        }
    }

    /**
     * Report a lexcial error and record the fact that an error has occured.
     * This fact can be ascertained from the Scanner by sending it an
```

```
     * errorHasOccurred() message.
     *
     * @param message
     *            message identifying the error.
     * @param args
     *            related values.
     */

    private void reportScannerError(String message, Object... args) {
        isInError = true;
        System.err.printf("%s:%d: ", fileName, line);
        System.err.printf(message, args);
        System.err.println();
    }

    /**
     * Return true if the specified character is a digit (0-9); false otherwise.
     *
     * @param c
     *            character.
     * @return true or false.
     */

    private boolean isDigit(char c) {
        return (c >= '0' && c <= '9');
    }

    /**
     * Return true if the specified character is a whitespace; false otherwise.
     *
     * @param c
     *            character.
     * @return true or false.
     */

    private boolean isWhitespace(char c) {
        switch (c) {
        case ' ':
        case '\t':
        case '\n': // CharReader maps all new lines to '\n'
        case '\f':
            return true;
        }
        return false;
    }

    /**
     * Return true if the specified character can start an identifier name;
     * false otherwise.
     *
     * @param c
     *            character.
     * @return true or false.
     */

    private boolean isIdentifierStart(char c) {
        return (c >= 'a' && c <= 'z' || c >= 'A' && c <= 'Z' || c == '_' || c ==
'$');
    }

    /**
     * Return true if the specified character can be part of an identifier name;
     * false otherwise.
     *
     * @param c
     *            character.
     * @return true or false.
     */

```

```
411     private boolean isIdentifierPart(char c) {
412         return (isIdentifierStart(c) || isDigit(c));
413     }
414 
415     /**
416      * Has an error occurred up to now in lexical analysis?
417      *
418      * @return true or false.
419      */
420 
421     public boolean errorHasOccurred() {
422         return isInError;
423     }
424 
425     /**
426      * The name of the source file.
427      *
428      * @return name of the source file.
429      */
430 
431     public String fileName() {
432         return fileName;
433     }
434 
435 }
436 
437 /**
438  * A buffered character reader. Abstracts out differences between platforms,
439  * mapping all new lines to '\n'. Also, keeps track of line numbers where the
440  * first line is numbered 1.
441  */
442 
443 class CharReader {
444 
445     /** A representation of the end of file as a character. */
446     public final static char EOFCH = (char) -1;
447 
448     /** The underlying reader records line numbers. */
449     private LineNumberReader lineNumberReader;
450 
451     /** Name of the file that is being read. */
452     private String fileName;
453 
454     /**
455      * Construct a CharReader from a file name.
456      *
457      * @param fileName
458      *            the name of the input file.
459      * @exception FileNotFoundException
460      *                if the file is not found.
461      */
462 
463     public CharReader(String fileName) throws FileNotFoundException {
464         lineNumberReader = new LineNumberReader(new FileReader(fileName));
465         this.fileName = fileName;
466     }
467 
468     /**
469      * Scan the next character.
470      *
471      * @return the character scanned.
472      * @exception IOException
473      *                if an I/O error occurs.
474      */
475 
476     public char nextChar() throws IOException {
477         return (char) lineNumberReader.read();
478     }
479 
```

```
480     /**
481      * The current line number in the source file, starting at 1.
482      *
483      * @return the current line number.
484      */
485
486     public int line() {
487         // LineNumberReader counts lines from 0.
488         return lineNumberReader.getLineNumber() + 1;
489     }
490
491     /**
492      * Return the file name.
493      *
494      * @return the file name.
495      */
496
497     public String fileName() {
498         return fileName;
499     }
500
501     /**
502      * Close the file.
503      *
504      * @exception IOException
505      *                if an I/O error occurs.
506      */
507
508     public void close() throws IOException {
509         lineNumberReader.close();
510     }
511
512 }
513
```