```java
// Copyright 2013 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 Java 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;

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

    /**
     * Has a parser error occurred up to now?
     *
     * @return true or false.
     */

    public boolean errorHasOccurred() {
        return isInError;
    }

    // ////////////////////////////////////////////////
    // Parsing Support ///////////////////////////////
    // ////////////////////////////////////////////////

    /**
     * Is the current token this one?
     *
     * @param sought
     *            the token we're looking for.
     * @return true iff they match; false otherwise.
     */

    private boolean see(TokenKind sought) {
        return (sought == scanner.token().kind());
    }

    /**
```

```
     * Look at the current (unscanned) token to see if it's one we're looking
     * for. If so, scan it and return true; otherwise return false (without
     * scanning a thing).
     *
     * @param sought
     *            the token we're looking for.
     * @return true iff they match; false otherwise.
     */

    private boolean have(TokenKind sought) {
        if (see(sought)) {
            scanner.next();
            return true;
        } else {
            return false;
        }
    }

    /**
     * Attempt to match a token we're looking for with the current input token.
     * If we succeed, scan the token and go into a "isRecovered" state. If we
     * fail, then what we do next depends on whether or not we're currently in a
     * "isRecovered" state: if so, we report the error and go into an
     * "Unrecovered" state; if not, we repeatedly scan tokens until we find the
     * one we're looking for (or EOF) and then return to a "isRecovered" state.
     * This gives us a kind of poor man's syntactic error recovery. The strategy
     * is due to David Turner and Ron Morrison.
     *
     * @param sought
     *            the token we're looking for.
     */

    private void mustBe(TokenKind sought) {
        if (scanner.token().kind() == sought) {
            scanner.next();
            isRecovered = true;
        } else if (isRecovered) {
            isRecovered = false;
            reportParserError("%s found where %s sought", scanner.token()
                    .image(), sought.image());
        } else {
            // Do not report the (possibly spurious) error,
            // but rather attempt to recover by forcing a match.
            while (!see(sought) && !see(EOF)) {
                scanner.next();
            }
            if (see(sought)) {
                scanner.next();
                isRecovered = true;
            }
        }
    }

    /**
     * Pull out the ambiguous part of a name and return it.
     *
     * @param name
     *            with an ambiguos part (possibly).
     * @return ambiguous part or null.
     */

    private AmbiguousName ambiguousPart(TypeName name) {
        String qualifiedName = name.toString();
        int lastDotIndex = qualifiedName.lastIndexOf('.');
        return lastDotIndex == -1 ? null // It was a simple
                // name
                : new AmbiguousName(name.line(), qualifiedName.substring(0,
                        lastDotIndex));
    }
```

```
136
137     /**
138      * Report a syntax error.
139      *
140      * @param message
141      *            message identifying the error.
142      * @param args
143      *            related values.
144      */
145
146     private void reportParserError(String message, Object... args) {
147         isInError = true;
148         isRecovered = false;
149         System.err
150                 .printf("%s:%d: ", scanner.fileName(), scanner.token().line());
151         System.err.printf(message, args);
152         System.err.println();
153     }
154
155     // ////////////////////////////////////////////////
156     // Lookahead //////////////////////////////////////
157     // ////////////////////////////////////////////////
158
159     /**
160      * Are we looking at an IDENTIFIER followed by a LPAREN? Look ahead to find
161      * out.
162      *
163      * @return true iff we're looking at IDENTIFIER LPAREN; false otherwise.
164      */
165
166     private boolean seeIdentLParen() {
167         scanner.recordPosition();
168         boolean result = have(IDENTIFIER) && see(LPAREN);
169         scanner.returnToPosition();
170         return result;
171     }
172
173     /**
174      * Are we looking at a cast? ie
175      *
176      * <pre>
177      *   LPAREN type RPAREN ...
178      * </pre>
179      *
180      * Look ahead to find out.
181      *
182      * @return true iff we're looking at a cast; false otherwise.
183      */
184
185     private boolean seeCast() {
186         scanner.recordPosition();
187         if (!have(LPAREN)) {
188             scanner.returnToPosition();
189             return false;
190         }
191         if (seeBasicType()) {
192             scanner.returnToPosition();
193             return true;
194         }
195         if (!see(IDENTIFIER)) {
196             scanner.returnToPosition();
197             return false;
198         } else {
199             scanner.next(); // Scan the IDENTIFIER
200             // A qualified identifier is ok
201             while (have(DOT)) {
202                 if (!have(IDENTIFIER)) {
203                     scanner.returnToPosition();
204                     return false;
```

```
                }
            }
        }
        while (have(LBRACK)) {
            if (!have(RBRACK)) {
                scanner.returnToPosition();
                return false;
            }
        }
        if (!have(RPAREN)) {
            scanner.returnToPosition();
            return false;
        }
        scanner.returnToPosition();
        return true;
    }

    /**
     * Are we looking at a local variable declaration? ie.
     *
     * <pre>
     *   type IDENTIFIER {LBRACK RBRACK} ...
     * </pre>
     *
     * Look ahead to determine.
     *
     * @return true iff we are looking at local variable declaration; false
     *         otherwise.
     */

    private boolean seeLocalVariableDeclaration() {
        scanner.recordPosition();
        if (have(IDENTIFIER)) {
            // A qualified identifier is ok
            while (have(DOT)) {
                if (!have(IDENTIFIER)) {
                    scanner.returnToPosition();
                    return false;
                }
            }
        } else if (seeBasicType()) {
            scanner.next();
        } else {
            scanner.returnToPosition();
            return false;
        }
        while (have(LBRACK)) {
            if (!have(RBRACK)) {
                scanner.returnToPosition();
                return false;
            }
        }
        if (!have(IDENTIFIER)) {
            scanner.returnToPosition();
            return false;
        }
        while (have(LBRACK)) {
            if (!have(RBRACK)) {
                scanner.returnToPosition();
                return false;
            }
        }
        scanner.returnToPosition();
        return true;
    }

    /**
     * Are we looking at a basic type? ie.
     *
```

```
     * <pre>
     * BOOLEAN | CHAR | INT
     * </pre>
     *
     * @return true iff we're looking at a basic type; false otherwise.
     */

    private boolean seeBasicType() {
        if (see(BOOLEAN) || see(CHAR) || see(INT)) {
            return true;
        } else {
            return false;
        }
    }

    /**
     * Are we looking at a reference type? ie.
     *
     * <pre>
     *   referenceType ::= basicType LBRACK RBRACK {LBRACK RBRACK}
     *                   | qualifiedIdentifier {LBRACK RBRACK}
     * </pre>
     *
     * @return true iff we're looking at a reference type; false otherwise.
     */

    private boolean seeReferenceType() {
        if (see(IDENTIFIER)) {
            return true;
        } else {
            scanner.recordPosition();
            if (have(BOOLEAN) || have(CHAR) || have(INT)) {
                if (have(LBRACK) && see(RBRACK)) {
                    scanner.returnToPosition();
                    return true;
                }
            }
            scanner.returnToPosition();
        }
        return false;
    }

    /**
     * Are we looking at []?
     *
     * @return true iff we're looking at a [] pair; false otherwise.
     */

    private boolean seeDims() {
        scanner.recordPosition();
        boolean result = have(LBRACK) && see(RBRACK);
        scanner.returnToPosition();
        return result;
    }

    // ////////////////////////////////////////////////
    // Parser Proper ///////////////////////////////////
    // ////////////////////////////////////////////////

    /**
     * Parse a compilation unit (a program file) and construct an AST for it.
     * After constructing the Parser, this is its entry point.
     *
     * <pre>
     *   compilationUnit ::= [PACKAGE qualifiedIdentifier SEMI]
     *                       {IMPORT  qualifiedIdentifier SEMI}
     *                       {typeDeclaration}
     *                       EOF
     * </pre>
```

```
     *
     * @return an AST for a compilationUnit.
     */

    public JCompilationUnit compilationUnit() {
        int line = scanner.token().line();
        TypeName packageName = null; // Default
        if (have(PACKAGE)) {
            packageName = qualifiedIdentifier();
            mustBe(SEMI);
        }
        ArrayList<TypeName> imports = new ArrayList<TypeName>();
        while (have(IMPORT)) {
            imports.add(qualifiedIdentifier());
            mustBe(SEMI);
        }
        ArrayList<JAST> typeDeclarations = new ArrayList<JAST>();
        while (!see(EOF)) {
            JAST typeDeclaration = typeDeclaration();
            if (typeDeclaration != null) {
                typeDeclarations.add(typeDeclaration);
            }
        }
        mustBe(EOF);
        return new JCompilationUnit(scanner.fileName(), line, packageName,
                imports, typeDeclarations);
    }

    /**
     * Parse a qualified identifier.
     *
     * <pre>
     *   qualifiedIdentifier ::= IDENTIFIER {DOT IDENTIFIER}
     * </pre>
     *
     * @return an instance of TypeName.
     */

    private TypeName qualifiedIdentifier() {
        int line = scanner.token().line();
        mustBe(IDENTIFIER);
        String qualifiedIdentifier = scanner.previousToken().image();
        while (have(DOT)) {
            mustBe(IDENTIFIER);
            qualifiedIdentifier += "." + scanner.previousToken().image();
        }
        return new TypeName(line, qualifiedIdentifier);
    }

    /**
     * Parse a type declaration.
     *
     * <pre>
     *   typeDeclaration ::= modifiers classDeclaration
     * </pre>
     *
     * @return an AST for a typeDeclaration.
     */

    private JAST typeDeclaration() {
        ArrayList<String> mods = modifiers();
        return classDeclaration(mods);
    }

    /**
     * Parse modifiers.
     *
     * <pre>
     *   modifiers ::= {PUBLIC | PROTECTED | PRIVATE | STATIC |
```

```
412      *                   ABSTRACT}
413      * </pre>
414      *
415      * Check for duplicates, and conflicts among access modifiers (public,
416      * protected, and private). Otherwise, no checks.
417      *
418      * @return a list of modifiers.
419      */
420
421     private ArrayList<String> modifiers() {
422         ArrayList<String> mods = new ArrayList<String>();
423         boolean scannedPUBLIC = false;
424         boolean scannedPROTECTED = false;
425         boolean scannedPRIVATE = false;
426         boolean scannedSTATIC = false;
427         boolean scannedABSTRACT = false;
428         boolean more = true;
429         while (more)
430             if (have(PUBLIC)) {
431                 mods.add("public");
432                 if (scannedPUBLIC) {
433                     reportParserError("Repeated modifier: public");
434                 }
435                 if (scannedPROTECTED || scannedPRIVATE) {
436                     reportParserError("Access conflict in modifiers");
437                 }
438                 scannedPUBLIC = true;
439             } else if (have(PROTECTED)) {
440                 mods.add("protected");
441                 if (scannedPROTECTED) {
442                     reportParserError("Repeated modifier: protected");
443                 }
444                 if (scannedPUBLIC || scannedPRIVATE) {
445                     reportParserError("Access conflict in modifiers");
446                 }
447                 scannedPROTECTED = true;
448             } else if (have(PRIVATE)) {
449                 mods.add("private");
450                 if (scannedPRIVATE) {
451                     reportParserError("Repeated modifier: private");
452                 }
453                 if (scannedPUBLIC || scannedPROTECTED) {
454                     reportParserError("Access conflict in modifiers");
455                 }
456                 scannedPRIVATE = true;
457             } else if (have(STATIC)) {
458                 mods.add("static");
459                 if (scannedSTATIC) {
460                     reportParserError("Repeated modifier: static");
461                 }
462                 scannedSTATIC = true;
463             } else if (have(ABSTRACT)) {
464                 mods.add("abstract");
465                 if (scannedABSTRACT) {
466                     reportParserError("Repeated modifier: abstract");
467                 }
468                 scannedABSTRACT = true;
469             } else {
470                 more = false;
471             }
472         return mods;
473     }
474
475     /**
476      * Parse a class declaration.
477      *
478      * <pre>
479      *   classDeclaration ::= CLASS IDENTIFIER
480      *                        [EXTENDS qualifiedIdentifier]
```

```
     *                       classBody
     * </pre>
     *
     * A class which doesn't explicitly extend another (super) class implicitly
     * extends the superclass java.lang.Object.
     *
     * @param mods
     *            the class modifiers.
     * @return an AST for a classDeclaration.
     */

    private JClassDeclaration classDeclaration(ArrayList<String> mods) {
        int line = scanner.token().line();
        mustBe(CLASS);
        mustBe(IDENTIFIER);
        String name = scanner.previousToken().image();
        Type superClass;
        if (have(EXTENDS)) {
            superClass = qualifiedIdentifier();
        } else {
            superClass = Type.OBJECT;
        }
        return new JClassDeclaration(line, mods, name, superClass, classBody());
    }

    /**
     * Parse a class body.
     *
     * <pre>
     *  classBody ::= LCURLY
     *                  {modifiers memberDecl}
     *                RCURLY
     * </pre>
     *
     * @return list of members in the class body.
     */

    private ArrayList<JMember> classBody() {
        ArrayList<JMember> members = new ArrayList<JMember>();
        mustBe(LCURLY);
        while (!see(RCURLY) && !see(EOF)) {
            members.add(memberDecl(modifiers()));
        }
        mustBe(RCURLY);
        return members;
    }

    /**
     * Parse a member declaration.
     *
     * <pre>
     *   memberDecl ::= IDENTIFIER            // constructor
     *                    formalParameters
     *                    block
     *                | (VOID | type) IDENTIFIER  // method
     *                    formalParameters
     *                    (block | SEMI)
     *                | type variableDeclarators SEMI
     * </pre>
     *
     * @param mods
     *            the class member modifiers.
     * @return an AST for a memberDecl.
     */

    private JMember memberDecl(ArrayList<String> mods) {
        int line = scanner.token().line();
        JMember memberDecl = null;
        if (seeIdentLParen()) {
```

```
            // A constructor
            mustBe(IDENTIFIER);
            String name = scanner.previousToken().image();
            ArrayList<JFormalParameter> params = formalParameters();
            JBlock body = block();
            memberDecl = new JConstructorDeclaration(line, mods, name, params,
                    body);
        } else {
            Type type = null;
            if (have(VOID)) {
                // void method
                type = Type.VOID;
                mustBe(IDENTIFIER);
                String name = scanner.previousToken().image();
                ArrayList<JFormalParameter> params = formalParameters();
                JBlock body = have(SEMI) ? null : block();
                memberDecl = new JMethodDeclaration(line, mods, name, type,
                        params, body);
            } else {
                type = type();
                if (seeIdentLParen()) {
                    // Non void method
                    mustBe(IDENTIFIER);
                    String name = scanner.previousToken().image();
                    ArrayList<JFormalParameter> params = formalParameters();
                    JBlock body = have(SEMI) ? null : block();
                    memberDecl = new JMethodDeclaration(line, mods, name, type,
                            params, body);
                } else {
                    // Field
                    memberDecl = new JFieldDeclaration(line, mods,
                            variableDeclarators(type));
                    mustBe(SEMI);
                }
            }
        }
        return memberDecl;
    }

    /**
     * Parse a block.
     *
     * <pre>
     *   block ::= LCURLY {blockStatement} RCURLY
     * </pre>
     *
     * @return an AST for a block.
     */

    private JBlock block() {
        int line = scanner.token().line();
        ArrayList<JStatement> statements = new ArrayList<JStatement>();
        mustBe(LCURLY);
        while (!see(RCURLY) && !see(EOF)) {
            statements.add(blockStatement());
        }
        mustBe(RCURLY);
        return new JBlock(line, statements);
    }

    /**
     * Parse a block statement.
     *
     * <pre>
     *   blockStatement ::= localVariableDeclarationStatement
     *                    | statement
     * </pre>
     *
     * @return an AST for a blockStatement.
```

```
     */

    private JStatement blockStatement() {
        if (seeLocalVariableDeclaration()) {
            return localVariableDeclarationStatement();
        } else {
            return statement();
        }
    }

    /**
     * Parse a statement.
     *
     * <pre>
     *   statement ::= block
     *               | IF parExpression statement [ELSE statement]
     *               | WHILE parExpression statement
     *               | RETURN [expression] SEMI
     *               | SEMI
     *               | statementExpression SEMI
     * </pre>
     *
     * @return an AST for a statement.
     */

    private JStatement statement() {
        int line = scanner.token().line();
        if (see(LCURLY)) {
            return block();
        } else if (have(IF)) {
            JExpression test = parExpression();
            JStatement consequent = statement();
            JStatement alternate = have(ELSE) ? statement() : null;
            return new JIfStatement(line, test, consequent, alternate);
        } else if (have(WHILE)) {
            JExpression test = parExpression();
            JStatement statement = statement();
            return new JWhileStatement(line, test, statement);
        } else if (have(RETURN)) {
            if (have(SEMI)) {
                return new JReturnStatement(line, null);
            } else {
                JExpression expr = expression();
                mustBe(SEMI);
                return new JReturnStatement(line, expr);
            }
        } else if (have(SEMI)) {
            return new JEmptyStatement(line);
        } else { // Must be a statementExpression
            JStatement statement = statementExpression();
            mustBe(SEMI);
            return statement;
        }
    }

    /**
     * Parse formal parameters.
     *
     * <pre>
     *   formalParameters ::= LPAREN
     *                          [formalParameter
     *                            {COMMA  formalParameter}]
     *                        RPAREN
     * </pre>
     *
     * @return a list of formal parameters.
     */

    private ArrayList<JFormalParameter> formalParameters() {
```

```
        ArrayList<JFormalParameter> parameters = new
ArrayList<JFormalParameter>();
        mustBe(LPAREN);
        if (have(RPAREN))
            return parameters; // ()
        do {
            parameters.add(formalParameter());
        } while (have(COMMA));
        mustBe(RPAREN);
        return parameters;
    }

    /**
     * Parse a formal parameter.
     *
     * <pre>
     *   formalParameter ::= type IDENTIFIER
     * </pre>
     *
     * @return an AST for a formalParameter.
     */

    private JFormalParameter formalParameter() {
        int line = scanner.token().line();
        Type type = type();
        mustBe(IDENTIFIER);
        String name = scanner.previousToken().image();
        return new JFormalParameter(line, name, type);
    }

    /**
     * Parse a parenthesized expression.
     *
     * <pre>
     *   parExpression ::= LPAREN expression RPAREN
     * </pre>
     *
     * @return an AST for a parExpression.
     */

    private JExpression parExpression() {
        mustBe(LPAREN);
        JExpression expr = expression();
        mustBe(RPAREN);
        return expr;
    }

    /**
     * Parse a local variable declaration statement.
     *
     * <pre>
     *   localVariableDeclarationStatement ::= type
     *                                         variableDeclarators
     *                                           SEMI
     * </pre>
     *
     * @return an AST for a variableDeclaration.
     */

    private JVariableDeclaration localVariableDeclarationStatement() {
        int line = scanner.token().line();
        ArrayList<String> mods = new ArrayList<String>();
        ArrayList<JVariableDeclarator> vdecls = variableDeclarators(type());
        mustBe(SEMI);
        return new JVariableDeclaration(line, mods, vdecls);
    }

    /**
     * Parse variable declarators.
```

```
     *
     * <pre>
     *   variableDeclarators ::= variableDeclarator
     *                             {COMMA variableDeclarator}
     * </pre>
     *
     * @param type
     *            type of the variables.
     * @return a list of variable declarators.
     */

    private ArrayList<JVariableDeclarator> variableDeclarators(Type type) {
        ArrayList<JVariableDeclarator> variableDeclarators = new
ArrayList<JVariableDeclarator>();
        do {
            variableDeclarators.add(variableDeclarator(type));
        } while (have(COMMA));
        return variableDeclarators;
    }

    /**
     * Parse a variable declarator.
     *
     * <pre>
     *   variableDeclarator ::= IDENTIFIER
     *                          [ASSIGN variableInitializer]
     * </pre>
     *
     * @param type
     *            type of the variable.
     * @return an AST for a variableDeclarator.
     */

    private JVariableDeclarator variableDeclarator(Type type) {
        int line = scanner.token().line();
        mustBe(IDENTIFIER);
        String name = scanner.previousToken().image();
        JExpression initial = have(ASSIGN) ? variableInitializer(type) : null;
        return new JVariableDeclarator(line, name, type, initial);
    }

    /**
     * Parse a variable initializer.
     *
     * <pre>
     *   variableInitializer ::= arrayInitializer
     *                         | expression
     * </pre>
     *
     * @param type
     *            type of the variable.
     * @return an AST for a variableInitializer.
     */

    private JExpression variableInitializer(Type type) {
        if (see(LCURLY)) {
            return arrayInitializer(type);
        }
        return expression();
    }

    /**
     * Parse an array initializer.
     *
     * <pre>
     *   arrayInitializer ::= LCURLY
     *                          [variableInitializer
     *                            {COMMA variableInitializer} [COMMA]]
     *                        RCURLY
```

```
     * </pre>
     *
     * @param type
     *            type of the array.
     * @return an AST for an arrayInitializer.
     */

    private JArrayInitializer arrayInitializer(Type type) {
        int line = scanner.token().line();
        ArrayList<JExpression> initials = new ArrayList<JExpression>();
        mustBe(LCURLY);
        if (have(RCURLY)) {
            return new JArrayInitializer(line, type, initials);
        }
        initials.add(variableInitializer(type.componentType()));
        while (have(COMMA)) {
            initials.add(see(RCURLY) ? null : variableInitializer(type
                    .componentType()));
        }
        mustBe(RCURLY);
        return new JArrayInitializer(line, type, initials);
    }

    /**
     * Parse arguments.
     *
     * <pre>
     *   arguments ::= LPAREN [expression {COMMA expression}] RPAREN
     * </pre>
     *
     * @return a list of expressions
     */

    private ArrayList<JExpression> arguments() {
        ArrayList<JExpression> args = new ArrayList<JExpression>();
        mustBe(LPAREN);
        if (have(RPAREN)) {
            return args;
        }
        do {
            args.add(expression());
        } while (have(COMMA));
        mustBe(RPAREN);
        return args;
    }

    /**
     * Parse a type.
     *
     * <pre>
     *   type ::= referenceType
     *          | basicType
     * </pre>
     *
     * @return an instance of Type.
     */

    private Type type() {
        if (seeReferenceType()) {
            return referenceType();
        }
        return basicType();
    }

    /**
     * Parse a basic type.
     *
     * <pre>
     *   basicType ::= BOOLEAN | CHAR | INT
```

```
     * </pre>
     *
     * @return an instance of Type.
     */

    private Type basicType() {
        if (have(BOOLEAN)) {
            return Type.BOOLEAN;
        } else if (have(CHAR)) {
            return Type.CHAR;
        } else if (have(INT)) {
            return Type.INT;
        } else {
            reportParserError("Type sought where %s found", scanner.token()
                    .image());
            return Type.ANY;
        }
    }

    /**
     * Parse a reference type.
     *
     * <pre>
     *   referenceType ::= basicType LBRACK RBRACK {LBRACK RBRACK}
     *                   | qualifiedIdentifier {LBRACK RBRACK}
     * </pre>
     *
     * @return an instance of Type.
     */

    private Type referenceType() {
        Type type = null;
        if (!see(IDENTIFIER)) {
            type = basicType();
            mustBe(LBRACK);
            mustBe(RBRACK);
            type = new ArrayTypeName(type);
        } else {
            type = qualifiedIdentifier();
        }
        while (seeDims()) {
            mustBe(LBRACK);
            mustBe(RBRACK);
            type = new ArrayTypeName(type);
        }
        return type;
    }

    /**
     * Parse a statement expression.
     *
     * <pre>
     *   statementExpression ::= expression // but must have
     *                                      // side-effect, eg i++
     * </pre>
     *
     * @return an AST for a statementExpression.
     */

    private JStatement statementExpression() {
        int line = scanner.token().line();
        JExpression expr = expression();
        if (expr instanceof JAssignment || expr instanceof JPreIncrementOp
                || expr instanceof JPostDecrementOp
                || expr instanceof JMessageExpression
                || expr instanceof JSuperConstruction
                || expr instanceof JThisConstruction || expr instanceof JNewOp
                || expr instanceof JNewArrayOp) {
            // So as not to save on stack
```

```
            expr.isStatementExpression = true;
        } else {
            reportParserError("Invalid statement expression; "
                    + "it does not have a side-effect");
        }
        return new JStatementExpression(line, expr);
    }

    /**
     * An expression.
     *
     * <pre>
     *   expression ::= assignmentExpression
     * </pre>
     *
     * @return an AST for an expression.
     */

    private JExpression expression() {
        return assignmentExpression();
    }

    /**
     * Parse an assignment expression.
     *
     * <pre>
     *   assignmentExpression ::=
     *       conditionalAndExpression // level 13
     *           [( ASSIGN  // conditionalExpression
     *            | PLUS_ASSIGN ) // must be valid lhs
     *           assignmentExpression]
     * </pre>
     *
     * @return an AST for an assignmentExpression.
     */

    private JExpression assignmentExpression() {
        int line = scanner.token().line();
        JExpression lhs = conditionalAndExpression();
        if (have(ASSIGN)) {
            return new JAssignOp(line, lhs, assignmentExpression());
        } else if (have(PLUS_ASSIGN)) {
            return new JPlusAssignOp(line, lhs, assignmentExpression());
        } else {
            return lhs;
        }
    }

    /**
     * Parse a conditional-and expression.
     *
     * <pre>
     *   conditionalAndExpression ::= equalityExpression // level 10
     *                                {LAND equalityExpression}
     * </pre>
     *
     * @return an AST for a conditionalExpression.
     */

    private JExpression conditionalAndExpression() {
        int line = scanner.token().line();
        boolean more = true;
        JExpression lhs = equalityExpression();
        while (more) {
            if (have(LAND)) {
                lhs = new JLogicalAndOp(line, lhs, equalityExpression());
            } else {
                more = false;
```

```
            }
        }
        return lhs;
    }

    /**
     * Parse an equality expression.
     *
     * <pre>
     *   equalityExpression ::= relationalExpression  // level 6
     *                          {EQUAL relationalExpression}
     * </pre>
     *
     * @return an AST for an equalityExpression.
     */

    private JExpression equalityExpression() {
        int line = scanner.token().line();
        boolean more = true;
        JExpression lhs = relationalExpression();
        while (more) {
            if (have(EQUAL)) {
                lhs = new JEqualOp(line, lhs, relationalExpression());
            } else {
                more = false;
            }
        }
        return lhs;
    }

    /**
     * Parse a relational expression.
     *
     * <pre>
     *   relationalExpression ::= additiveExpression // level 5
     *                            [(GT | LE) additiveExpression
     *                            | INSTANCEOF referenceType]
     * </pre>
     *
     * @return an AST for a relationalExpression.
     */

    private JExpression relationalExpression() {
        int line = scanner.token().line();
        JExpression lhs = additiveExpression();
        if (have(GT)) {
            return new JGreaterThanOp(line, lhs, additiveExpression());
        } else if (have(LE)) {
            return new JLessEqualOp(line, lhs, additiveExpression());
        } else if (have(INSTANCEOF)) {
            return new JInstanceOfOp(line, lhs, referenceType());
        } else {
            return lhs;
        }
    }

    /**
     * Parse an additive expression.
     *
     * <pre>
     *   additiveExpression ::= multiplicativeExpression // level 3
     *                          {MINUS multiplicativeExpression}
     * </pre>
     *
     * @return an AST for an additiveExpression.
     */

    private JExpression additiveExpression() {
        int line = scanner.token().line();
```

```
1100        boolean more = true;
1101        JExpression lhs = multiplicativeExpression();
1102        while (more) {
1103            if (have(MINUS)) {
1104                lhs = new JSubtractOp(line, lhs, multiplicativeExpression());
1105            } else if (have(PLUS)) {
1106                lhs = new JPlusOp(line, lhs, multiplicativeExpression());
1107            } else {
1108                more = false;
1109            }
1110        }
1111        return lhs;
1112    }
1113
1114    /**
1115     * Parse a multiplicative expression.
1116     *
1117     * <pre>
1118     *   multiplicativeExpression ::= unaryExpression  // level 2
1119     *                                  {STAR unaryExpression}
1120     * </pre>
1121     *
1122     * @return an AST for a multiplicativeExpression.
1123     */
1124
1125    private JExpression multiplicativeExpression() {
1126        int line = scanner.token().line();
1127        boolean more = true;
1128        JExpression lhs = unaryExpression();
1129        while (more) {
1130            if (have(STAR)) {
1131                lhs = new JMultiplyOp(line, lhs, unaryExpression());
1132            } else {
1133                more = false;
1134            }
1135        }
1136        return lhs;
1137    }
1138
1139    /**
1140     * Parse an unary expression.
1141     *
1142     * <pre>
1143     *   unaryExpression ::= INC unaryExpression // level 1
1144     *                     | MINUS unaryExpression
1145     *                     | simpleUnaryExpression
1146     * </pre>
1147     *
1148     * @return an AST for an unaryExpression.
1149     */
1150
1151    private JExpression unaryExpression() {
1152        int line = scanner.token().line();
1153        if (have(INC)) {
1154            return new JPreIncrementOp(line, unaryExpression());
1155        } else if (have(MINUS)) {
1156            return new JNegateOp(line, unaryExpression());
1157        } else {
1158            return simpleUnaryExpression();
1159        }
1160    }
1161
1162    /**
1163     * Parse a simple unary expression.
1164     *
1165     * <pre>
1166     *   simpleUnaryExpression ::= LNOT unaryExpression
1167     *                           | LPAREN basicType RPAREN
1168     *                               unaryExpression
```

```
     *                             | LPAREN
     *                                referenceType
     *                               RPAREN simpleUnaryExpression
     *                             | postfixExpression
     * </pre>
     *
     * @return an AST for a simpleUnaryExpression.
     */

    private JExpression simpleUnaryExpression() {
        int line = scanner.token().line();
        if (have(LNOT)) {
            return new JLogicalNotOp(line, unaryExpression());
        } else if (seeCast()) {
            mustBe(LPAREN);
            boolean isBasicType = seeBasicType();
            Type type = type();
            mustBe(RPAREN);
            JExpression expr = isBasicType ? unaryExpression()
                    : simpleUnaryExpression();
            return new JCastOp(line, type, expr);
        } else {
            return postfixExpression();
        }
    }

    /**
     * Parse a postfix expression.
     *
     * <pre>
     *   postfixExpression ::= primary {selector} {DEC}
     * </pre>
     *
     * @return an AST for a postfixExpression.
     */

    private JExpression postfixExpression() {
        int line = scanner.token().line();
        JExpression primaryExpr = primary();
        while (see(DOT) || see(LBRACK)) {
            primaryExpr = selector(primaryExpr);
        }
        while (have(DEC)) {
            primaryExpr = new JPostDecrementOp(line, primaryExpr);
        }
        return primaryExpr;
    }

    /**
     * Parse a selector.
     *
     * <pre>
     *   selector ::= DOT qualifiedIdentifier [arguments]
     *              | LBRACK expression RBRACK
     * </pre>
     *
     * @param target
     *            the target expression for this selector.
     * @return an AST for a selector.
     */

    private JExpression selector(JExpression target) {
        int line = scanner.token().line();
        if (have(DOT)) {
            // Target . selector
            mustBe(IDENTIFIER);
            String name = scanner.previousToken().image();
            if (see(LPAREN)) {
                ArrayList<JExpression> args = arguments();
```

```
                return new JMessageExpression(line, target, name, args);
            } else {
                return new JFieldSelection(line, target, name);
            }
        } else {
            mustBe(LBRACK);
            JExpression index = expression();
            mustBe(RBRACK);
            return new JArrayExpression(line, target, index);
        }
    }

    /**
     * Parse a primary expression.
     *
     * <pre>
     *   primary ::= parExpression
     *             | THIS [arguments]
     *             | SUPER ( arguments
     *                     | DOT IDENTIFIER [arguments]
     *                     )
     *             | literal
     *             | NEW creator
     *             | qualifiedIdentifier [arguments]
     * </pre>
     *
     * @return an AST for a primary.
     */

    private JExpression primary() {
        int line = scanner.token().line();
        if (see(LPAREN)) {
            return parExpression();
        } else if (have(THIS)) {
            if (see(LPAREN)) {
                return new JThisConstruction(line, arguments());
            } else {
                return new JThis(line);
            }
        } else if (have(SUPER)) {
            if (!have(DOT)) {
                return new JSuperConstruction(line, arguments());
            } else {
                mustBe(IDENTIFIER);
                String name = scanner.previousToken().image();
                JExpression newTarget = new JSuper(line);
                if (see(LPAREN)) {
                    return new JMessageExpression(line, newTarget, null, name,
                            arguments());
                } else {
                    return new JFieldSelection(line, newTarget, name);
                }
            }
        } else if (have(NEW)) {
            return creator();
        } else if (see(IDENTIFIER)) {
            TypeName id = qualifiedIdentifier();
            if (see(LPAREN)) {
                return new JMessageExpression(line, null, ambiguousPart(id), id
                        .simpleName(), arguments());
            } else if (ambiguousPart(id) == null) {
                // A simple name
                return new JVariable(line, id.simpleName());
            } else {
                // ambiguousPart.fieldName
                return new JFieldSelection(line, ambiguousPart(id), null, id
                        .simpleName());
            }
        } else {
```

```
            return literal();
        }
    }

    /**
     * Parse a creator.
     *
     * <pre>
     *   creator ::= (basicType | qualifiedIdentifier)
     *                 ( arguments
     *                 | LBRACK RBRACK {LBRACK RBRACK}
     *                     [arrayInitializer]
     *                 | newArrayDeclarator
     *                 )
     * </pre>
     *
     * @return an AST for a creator.
     */

    private JExpression creator() {
        int line = scanner.token().line();
        Type type = seeBasicType() ? basicType() : qualifiedIdentifier();
        if (see(LPAREN)) {
            ArrayList<JExpression> args = arguments();
            return new JNewOp(line, type, args);
        } else if (see(LBRACK)) {
            if (seeDims()) {
                Type expected = type;
                while (have(LBRACK)) {
                    mustBe(RBRACK);
                    expected = new ArrayTypeName(expected);
                }
                return arrayInitializer(expected);
            } else {
                return newArrayDeclarator(line, type);
            }
        } else {
            reportParserError("( or [ sought where %s found", scanner.token()
                .image());
            return new JWildExpression(line);
        }
    }

    /**
     * Parse a new array declarator.
     *
     * <pre>
     *   newArrayDeclarator ::= LBRACK expression RBRACK
     *                          {LBRACK expression RBRACK}
     *                          {LBRACK RBRACK}
     * </pre>
     *
     * @param line
     *            line in which the declarator occurred.
     * @param type
     *            type of the array.
     * @return an AST for a newArrayDeclarator.
     */

    private JNewArrayOp newArrayDeclarator(int line, Type type) {
        ArrayList<JExpression> dimensions = new ArrayList<JExpression>();
        mustBe(LBRACK);
        dimensions.add(expression());
        mustBe(RBRACK);
        type = new ArrayTypeName(type);
        while (have(LBRACK)) {
            if (have(RBRACK)) {
                // We're done with dimension expressions
                type = new ArrayTypeName(type);
                while (have(LBRACK)) {
```

```
                    mustBe(RBRACK);
                    type = new ArrayTypeName(type);
                }
                return new JNewArrayOp(line, type, dimensions);
            } else {
                dimensions.add(expression());
                type = new ArrayTypeName(type);
                mustBe(RBRACK);
            }
        }
        return new JNewArrayOp(line, type, dimensions);
    }

    /**
     * Parse a literal.
     *
     * <pre>
     *   literal ::= INT_LITERAL | CHAR_LITERAL | STRING_LITERAL
     *             | TRUE        | FALSE        | NULL
     * </pre>
     *
     * @return an AST for a literal.
     */

    private JExpression literal() {
        int line = scanner.token().line();
        if (have(INT_LITERAL)) {
            return new JLiteralInt(line, scanner.previousToken().image());
        } else if (have(CHAR_LITERAL)) {
            return new JLiteralChar(line, scanner.previousToken().image());
        } else if (have(STRING_LITERAL)) {
            return new JLiteralString(line, scanner.previousToken().image());
        } else if (have(TRUE)) {
            return new JLiteralTrue(line);
        } else if (have(FALSE)) {
            return new JLiteralFalse(line);
        } else if (have(NULL)) {
            return new JLiteralNull(line);
        } else {
            reportParserError("Literal sought where %s found", scanner.token()
                    .image());
            return new JWildExpression(line);
        }
    }

    // A tracing aid. Invoke to debug the parser at various
    //
    // private void trace( String message )
    // {
    // System.err.println( "["
    // + scanner.token().line()
    // + ": "
    // + message
    // + ", looking at a: "
    // + scanner.token().tokenRep()
    // + " = " + scanner.token().image() + "]" );
    // }
}
```