

Parser.java

```
1 // Copyright 2013 Bill Campbell, Swami Iyer and Bahar Akbal-Delibas
2
3 package jminusminus;
4
5 import java.util.ArrayList;
6 import static jminusminus.TokenKind.*;
7
8 /**
9  * A recursive descent parser that, given a lexical analyzer (a
10  * LookaheadScanner), parses a Java compilation unit (program file), taking
11  * tokens from the LookaheadScanner, and produces an abstract syntax tree (AST)
12  * for it.
13  */
14
15 public class Parser {
16
17     /** The lexical analyzer with which tokens are scanned. */
18     private LookaheadScanner scanner;
19
20     /** Whether a parser error has been found. */
21     private boolean isInError;
22
23     /** Whether we have recovered from a parser error. */
24     private boolean isRecovered;
25
26     /**
27      * Construct a parser from the given lexical analyzer.
28      * @param scanner
29      *     the lexical analyzer with which tokens are scanned.
30      */
31
32     public Parser(LookaheadScanner scanner) {
33         this.scanner = scanner;
34         isInError = false;
35         isRecovered = true;
36         scanner.next(); // Prime the pump
37     }
38
39     /**
40      * Has a parser error occurred up to now?
41      *
42      * @return true or false.
43      */
44
45     public boolean errorHasOccurred() {
46         return isInError;
47     }
48
49     // //////////////////////////////////////
50     // Parsing Support //////////////////////////////////////
51     // //////////////////////////////////////
52
53     /**
54      * Is the current token this one?
55      *
56      * @param sought
57      *     the token we're looking for.
58      * @return true iff they match; false otherwise.
59      */
60
61     private boolean see(TokenKind sought) {
62         return (sought == scanner.token().kind());
63     }
64
65     /**
66
```

```

67     * Look at the current (unscanned) token to see if it's one we're looking
68     * for. If so, scan it and return true; otherwise return false (without
69     * scanning a thing).
70     *
71     * @param sought
72     *         the token we're looking for.
73     * @return true iff they match; false otherwise.
74     */
75
76     private boolean have(TokenKind sought) {
77         if (see(sought)) {
78             scanner.next();
79             return true;
80         } else {
81             return false;
82         }
83     }
84
85     /**
86     * Attempt to match a token we're looking for with the current input token.
87     * If we succeed, scan the token and go into a "isRecovered" state. If we
88     * fail, then what we do next depends on whether or not we're currently in a
89     * "isRecovered" state: if so, we report the error and go into an
90     * "Unrecovered" state; if not, we repeatedly scan tokens until we find the
91     * one we're looking for (or EOF) and then return to a "isRecovered" state.
92     * This gives us a kind of poor man's syntactic error recovery. The strategy
93     * is due to David Turner and Ron Morrison.
94     *
95     * @param sought
96     *         the token we're looking for.
97     */
98
99     private void mustBe(TokenKind sought) {
100         if (scanner.token().kind() == sought) {
101             scanner.next();
102             isRecovered = true;
103         } else if (isRecovered) {
104             isRecovered = false;
105             reportParserError("%s found where %s sought", scanner.token().
106                 .image(), sought.image());
107         } else {
108             // Do not report the (possibly spurious) error,
109             // but rather attempt to recover by forcing a match.
110             while (!see(sought) && !see(EOF)) {
111                 scanner.next();
112             }
113             if (see(sought)) {
114                 scanner.next();
115                 isRecovered = true;
116             }
117         }
118     }
119
120     /**
121     * Pull out the ambiguous part of a name and return it.
122     *
123     * @param name
124     *         with an ambiguous part (possibly).
125     * @return ambiguous part or null.
126     */
127
128     private AmbiguousName ambiguousPart(TypeName name) {
129         String qualifiedName = name.toString();
130         int lastDotIndex = qualifiedName.lastIndexOf('.');
131         return lastDotIndex == -1 ? null // It was a simple
132             // name
133             : new AmbiguousName(name.line(), qualifiedName.substring(0,
134                 lastDotIndex));
135     }

```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

```

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;

```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

```

205     }
206 }
207 }
208 while (have(LBRACK)) {
209     if (!have(RBRACK)) {
210         scanner.returnToPosition();
211         return false;
212     }
213 }
214 if (!have(RPAREN)) {
215     scanner.returnToPosition();
216     return false;
217 }
218 scanner.returnToPosition();
219 return true;
220 }
221
222 /**
223  * Are we looking at a local variable declaration? ie.
224  *
225  * <pre>
226  *   type IDENTIFIER {LBRACK RBRACK} ...
227  * </pre>
228  *
229  * Look ahead to determine.
230  *
231  * @return true iff we are looking at local variable declaration; false
232  *         otherwise.
233  */
234 private boolean seeLocalVariableDeclaration() {
235     scanner.recordPosition();
236     if (have(IDENTIFIER)) {
237         // A qualified identifier is ok
238         while (have(DOT)) {
239             if (!have(IDENTIFIER)) {
240                 scanner.returnToPosition();
241                 return false;
242             }
243         }
244     } else if (seeBasicType()) {
245         scanner.next();
246     } else {
247         scanner.returnToPosition();
248         return false;
249     }
250 }
251 while (have(LBRACK)) {
252     if (!have(RBRACK)) {
253         scanner.returnToPosition();
254         return false;
255     }
256 }
257 if (!have(IDENTIFIER)) {
258     scanner.returnToPosition();
259     return false;
260 }
261 while (have(LBRACK)) {
262     if (!have(RBRACK)) {
263         scanner.returnToPosition();
264         return false;
265     }
266 }
267 scanner.returnToPosition();
268 return true;
269 }
270
271 /**
272  * Are we looking at a basic type? ie.
273  *

```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

```

274 * <pre>
275 * BOOLEAN | CHAR | INT
276 * </pre>
277 *
278 * @return true iff we're looking at a basic type; false otherwise.
279 */
280
281 private boolean seeBasicType() {
282     if (see(BOOLEAN) || see(CHAR) || see(INT)) {
283         return true;
284     } else {
285         return false;
286     }
287 }
288
289 /**
290 * Are we looking at a reference type? ie.
291 *
292 * <pre>
293 *     referenceType ::= basicType LBRACK RBRACK {LBRACK RBRACK}
294 *                     | qualifiedIdentifier {LBRACK RBRACK}
295 * </pre>
296 *
297 * @return true iff we're looking at a reference type; false otherwise.
298 */
299
300 private boolean seeReferenceType() {
301     if (see(IDENTIFIER)) {
302         return true;
303     } else {
304         scanner.recordPosition();
305         if (have(BOOLEAN) || have(CHAR) || have(INT)) {
306             if (have(LBRACK) && see(RBRACK)) {
307                 scanner.returnToPosition();
308                 return true;
309             }
310         }
311         scanner.returnToPosition();
312     }
313     return false;
314 }
315
316 /**
317 * Are we looking at []?
318 *
319 * @return true iff we're looking at a [] pair; false otherwise.
320 */
321
322 private boolean seeDims() {
323     scanner.recordPosition();
324     boolean result = have(LBRACK) && see(RBRACK);
325     scanner.returnToPosition();
326     return result;
327 }
328
329 // //////////////////////////////////////
330 // Parser Proper //////////////////////////////////////
331 // //////////////////////////////////////
332
333 /**
334 * Parse a compilation unit (a program file) and construct an AST for it.
335 * After constructing the Parser, this is its entry point.
336 *
337 * <pre>
338 *     compilationUnit ::= [PACKAGE qualifiedIdentifier SEMI]
339 *                         {IMPORT qualifiedIdentifier SEMI}
340 *                         {typeDeclaration}
341 *                         EOF
342 * </pre>

```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

```

343 *
344 * @return an AST for a compilationUnit.
345 */
346
347 public JCompilationUnit compilationUnit() {
348     int line = scanner.token().line();
349     TypeName packageName = null; // Default
350     if (have(PACKAGE)) {
351         packageName = qualifiedIdentifier();
352         mustBe(SEMI);
353     }
354     ArrayList<TypeName> imports = new ArrayList<TypeName>();
355     while (have(IMPORT)) {
356         imports.add(qualifiedIdentifier());
357         mustBe(SEMI);
358     }
359     ArrayList<JAST> typeDeclarations = new ArrayList<JAST>();
360     while (!see(EOF)) {
361         JAST typeDeclaration = typeDeclaration();
362         if (typeDeclaration != null) {
363             typeDeclarations.add(typeDeclaration);
364         }
365     }
366     mustBe(EOF);
367     return new JCompilationUnit(scanner.fileName(), line, packageName,
368         imports, typeDeclarations);
369 }
370
371 /**
372  * Parse a qualified Identifier
373  *
374  * <pre>
375  *   qualifiedIdentifier ::= IDENTIFIER {DOT IDENTIFIER}
376  * </pre>
377  *
378  * @return an instance of TypeName.
379  */
380
381 private TypeName qualifiedIdentifier() {
382     int line = scanner.token().line();
383     mustBe(IDENTIFIER);
384     String qualifiedIdentifier = scanner.previousToken().image();
385     while (have(DOT)) {
386         mustBe(IDENTIFIER);
387         qualifiedIdentifier += "." + scanner.previousToken().image();
388     }
389     return new TypeName(line, qualifiedIdentifier);
390 }
391
392 /**
393  * Parse a type declaration.
394  *
395  * <pre>
396  *   typeDeclaration ::= modifiers classDeclaration
397  * </pre>
398  *
399  * @return an AST for a typeDeclaration.
400  */
401
402 private JAST typeDeclaration() {
403     ArrayList<String> mods = modifiers();
404     return classDeclaration(mods);
405 }
406
407 /**
408  * Parse modifiers.
409  *
410  * <pre>
411  *   modifiers ::= {PUBLIC | PROTECTED | PRIVATE | STATIC |

```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

```

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]

```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

```

481         *                               classBody
482     * </pre>
483     *
484     * A class which doesn't explicitly extend another (super) class implicitly
485     * extends the superclass java.lang.Object.
486     *
487     * @param mods
488     *         the class modifiers.
489     * @return an AST for a classDeclaration.
490     */
491
492     private JClassDeclaration classDeclaration(ArrayList<String> mods) {
493         int line = scanner.token().line();
494         mustBe(CLASS);
495         mustBe(IDENTIFIER);
496         String name = scanner.previousToken().image();
497         Type superClass;
498         if (have(EXTENDS)) {
499             superClass = qualifiedIdentifier();
500         } else {
501             superClass = Type.OBJECT;
502         }
503         return new JClassDeclaration(line, mods, name, superClass, classBody());
504     }
505
506     /**
507     * Parse a class body.
508     *
509     * <pre>
510     * classBody ::= LCURLY
511     *               {modifiers memberDecl}
512     *               RCURLY
513     * </pre>
514     *
515     * @return list of members in the class body.
516     */
517
518     private ArrayList<JMember> classBody() {
519         ArrayList<JMember> members = new ArrayList<JMember>();
520         mustBe(LCURLY);
521         while (!see(RCURLY) && !see(EOF)) {
522             members.add(memberDecl(modifiers()));
523         }
524         mustBe(RCURLY);
525         return members;
526     }
527
528     /**
529     * Parse a member declaration.
530     *
531     * <pre>
532     * memberDecl ::= IDENTIFIER           // constructor
533     *                formalParameters
534     *                block
535     *                | (VOID | type) IDENTIFIER // method
536     *                formalParameters
537     *                (block | SEMI)
538     *                | type variableDeclarators SEMI
539     * </pre>
540     *
541     * @param mods
542     *         the class member modifiers.
543     * @return an AST for a memberDecl.
544     */
545
546     private JMember memberDecl(ArrayList<String> mods) {
547         int line = scanner.token().line();
548         JMember memberDecl = null;
549         if (seeIdentLParen()) {

```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder


```

550         // A constructor
551         mustBe(IDENTIFIER);
552         String name = scanner.previousToken().image();
553         ArrayList<JFormalParameter> params = formalParameters();
554         JBlock body = block();
555         memberDecl = new JConstructorDeclaration(line, mods, name, params,
556             body);
557     } else {
558         Type type = null;
559         if (have(VOID)) {
560             // void method
561             type = Type.VOID;
562             mustBe(IDENTIFIER);
563             String name = scanner.previousToken().image();
564             ArrayList<JFormalParameter> params = formalParameters();
565             JBlock body = have(SEMI) ? null : block();
566             memberDecl = new JMethodDeclaration(line, mods, name, type,
567                 params, body);
568         } else {
569             type = type();
570             if (seeIdentLParen()) {
571                 // Non void method
572                 mustBe(IDENTIFIER);
573                 String name = scanner.previousToken().image();
574                 ArrayList<JFormalParameter> params = formalParameters();
575                 JBlock body = have(SEMI) ? null : block();
576                 memberDecl = new JMethodDeclaration(line, mods, name, type,
577                     params, body);
578             } else {
579                 // Field
580                 memberDecl = new JFieldDeclaration(line, mods,
581                     variableDeclarators(type));
582                 mustBe(SEMI);
583             }
584         }
585     }
586     return memberDecl;
587 }

588 /**
589  * Parse a block.
590  *
591  * <pre>
592  *   block ::= LCURLY {blockStatement} RCURLY
593  * </pre>
594  *
595  * @return an AST for a block.
596  */
597
598 private JBlock block() {
599     int line = scanner.token().line();
600     ArrayList<JStatement> statements = new ArrayList<JStatement>();
601     mustBe(LCURLY);
602     while (!see(RCURLY) && !see(EOF)) {
603         statements.add(blockStatement());
604     }
605     mustBe(RCURLY);
606     return new JBlock(line, statements);
607 }

608 /**
609  * Parse a block statement.
610  *
611  * <pre>
612  *   blockStatement ::= localVariableDeclarationStatement
613  *                       | statement
614  * </pre>
615  *
616  * @return an AST for a blockStatement.

```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

```

619     */
620
621     private JStatement blockStatement() {
622         if (seeLocalVariableDeclaration()) {
623             return localVariableDeclarationStatement();
624         } else {
625             return statement();
626         }
627     }
628
629     /**
630     * Parse a statement.
631     *
632     * <pre>
633     *     statement ::= block
634     *                  | IF parExpression statement [ELSE statement]
635     *                  | WHILE parExpression statement
636     *                  | RETURN [expression] SEMI
637     *                  | SEMI
638     *                  | statementExpression SEMI
639     * </pre>
640     *
641     * @return an AST for a statement.
642     */
643
644     private JStatement statement() {
645         int line = scanner.token().line();
646         if (see(LCURLY)) {
647             return block();
648         } else if (have(LEFT_PAREN)) {
649             JExpression test = parExpression();
650             JStatement consequent = statement();
651             JStatement alternate = have(ELSE) ? statement() : null;
652             return new JIfStatement(line, test, consequent, alternate);
653         } else if (have(WHILE)) {
654             JExpression test = parExpression();
655             JStatement statement = statement();
656             return new JWhileStatement(line, test, statement);
657         } else if (have(RETURN)) {
658             if (have(SEMI)) {
659                 return new JReturnStatement(line, null);
660             } else {
661                 JExpression expr = expression();
662                 mustBe(SEMI);
663                 return new JReturnStatement(line, expr);
664             }
665         } else if (have(SEMI)) {
666             return new JEmptyStatement(line);
667         } else { // Must be a statementExpression
668             JStatement statement = statementExpression();
669             mustBe(SEMI);
670             return statement;
671         }
672     }
673
674     /**
675     * Parse formal parameters.
676     *
677     * <pre>
678     *     formalParameters ::= LPAREN
679     *                          [formalParameter
680     *                           {COMMA formalParameter}]
681     *                          RPAREN
682     * </pre>
683     *
684     * @return a list of formal parameters.
685     */
686
687     private ArrayList<JFormalParameter> formalParameters() {

```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

```

688     ArrayList<JFormalParameter> parameters = new
ArrayList<JFormalParameter>();
689     mustBe(LPAREN);
690     if (have(RPAREN))
691         return parameters; // ()
692     do {
693         parameters.add(formalParameter());
694     } while (have(COMMA));
695     mustBe(RPAREN);
696     return parameters;
697 }
698
699 /**
700  * Parse a formal parameter.
701  *
702  * <pre>
703  *   formalParameter ::= type IDENTIFIER
704  * </pre>
705  *
706  * @return an AST for a formalParameter.
707  */
708
709 private JFormalParameter formalParameter() {
710     int line = scanner.token().line();
711     Type type = type();
712     mustBe(IDENTIFIER);
713     String name = scanner.previousToken().image();
714     return new JFormalParameter(line, name, type);
715 }
716
717 /**
718  * Parse a parenthesized expression.
719  *
720  * <pre>
721  *   parExpression ::= LPAREN expression RPAREN
722  * </pre>
723  *
724  * @return an AST for a parExpression.
725  */
726
727 private JExpression parExpression() {
728     mustBe(LPAREN);
729     JExpression expr = expression();
730     mustBe(RPAREN);
731     return expr;
732 }
733
734 /**
735  * Parse a local variable declaration statement.
736  *
737  * <pre>
738  *   localVariableDeclarationStatement ::= type
739  *                                           variableDeclarators
740  *                                           SEMI
741  * </pre>
742  *
743  * @return an AST for a variableDeclaration.
744  */
745
746 private JVariableDeclaration localVariableDeclarationStatement() {
747     int line = scanner.token().line();
748     ArrayList<String> mods = new ArrayList<String>();
749     ArrayList<JVariableDeclarator> vdecls = variableDeclarators(type());
750     mustBe(SEMI);
751     return new JVariableDeclaration(line, mods, vdecls);
752 }
753
754 /**
755  * Parse variable declarators.

```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

```

756 *
757 * <pre>
758 *   variableDeclarators ::= variableDeclarator
759 *                           {COMMA variableDeclarator}
760 * </pre>
761 *
762 * @param type
763 *       type of the variables.
764 * @return a list of variable declarators.
765 */
766
767 private ArrayList<JVariableDeclarator> variableDeclarators(Type type) {
768     ArrayList<JVariableDeclarator> variableDeclarators = new
ArrayList<JVariableDeclarator>();
769     do {
770         variableDeclarators.add(variableDeclarator(type));
771     } while (have(COMMA));
772     return variableDeclarators;
773 }
774
775 /**
776 * Parse a variable declarator.
777 *
778 * <pre>
779 *   variableDeclarator ::= IDENTIFIER
780 *                       [ASSIGN variableInitializer]
781 * </pre>
782 *
783 * @param type
784 *       type of the variable.
785 * @return an AST for a variable declarator.
786 */
787
788 private JVariableDeclarator variableDeclarator(Type type) {
789     int line = scanner.token().line();
790     mustBe(IDENTIFIER);
791     String name = scanner.previousToken().image();
792     JExpression initial = have(ASSIGN) ? variableInitializer(type) : null;
793     return new JVariableDeclarator(line, name, type, initial);
794 }
795
796 /**
797 * Parse a variable initializer.
798 *
799 * <pre>
800 *   variableInitializer ::= arrayInitializer
801 *                       | expression
802 * </pre>
803 *
804 * @param type
805 *       type of the variable.
806 * @return an AST for a variable initializer.
807 */
808
809 private JExpression variableInitializer(Type type) {
810     if (see(LCURLY)) {
811         return arrayInitializer(type);
812     }
813     return expression();
814 }
815
816 /**
817 * Parse an array initializer.
818 *
819 * <pre>
820 *   arrayInitializer ::= LCURLY
821 *                       [variableInitializer
822 *                       {COMMA variableInitializer} [COMMA]]
823 *                       RCURLY

```

```

824 * </pre>
825 *
826 * @param type
827 *     type of the array.
828 * @return an AST for an arrayInitializer.
829 */
830
831 private JArrayInitializer arrayInitializer(Type type) {
832     int line = scanner.token().line();
833     ArrayList<JExpression> initials = new ArrayList<JExpression>();
834     mustBe(LCURLY);
835     if (have(RCURLY)) {
836         return new JArrayInitializer(line, type, initials);
837     }
838     initials.add(variableInitializer(type.componentType()));
839     while (have(COMMA)) {
840         initials.add(see(RCURLY) ? null : variableInitializer(type
841             .componentType()));
842     }
843     mustBe(RCURLY);
844     return new JArrayInitializer(line, type, initials);
845 }
846
847 /**
848 * Parse arguments.
849 *
850 * <pre>
851 * arguments ::= LPAREN [expression {COMMA expression}] RPAREN
852 * </pre>
853 * @return a list of expressions
854 */
855
856 private ArrayList<JExpression> arguments() {
857     ArrayList<JExpression> args = new ArrayList<JExpression>();
858     mustBe(LPAREN);
859     if (have(RPAREN)) {
860         return args;
861     }
862     do {
863         args.add(expression());
864     } while (have(COMMA));
865     mustBe(RPAREN);
866     return args;
867 }
868
869 /**
870 * Parse a type.
871 *
872 * <pre>
873 * type ::= referenceType
874 *       | basicType
875 * </pre>
876 *
877 * @return an instance of Type.
878 */
879
880 private Type type() {
881     if (seeReferenceType()) {
882         return referenceType();
883     }
884     return basicType();
885 }
886
887 /**
888 * Parse a basic type.
889 *
890 * <pre>
891 * basicType ::= BOOLEAN | CHAR | INT

```

```

893     * </pre>
894     *
895     * @return an instance of Type.
896     */
897
898     private Type basicType() {
899         if (have(BOOLEAN)) {
900             return Type.BOOLEAN;
901         } else if (have(CHAR)) {
902             return Type.CHAR;
903         } else if (have(INT)) {
904             return Type.INT;
905         } else {
906             reportParserError("Type sought where %s found", scanner.token()
907                 .image());
908             return Type.ANY;
909         }
910     }
911
912     /**
913     * Parse a reference type.
914     *
915     * <pre>
916     *     referenceType ::= basicType LBRACK RBRACK {LBRACK RBRACK}
917     *                       | qualifiedIdentifier {LBRACK RBRACK}
918     * </pre>
919     *
920     * @return an instance of Type.
921     */
922     private Type referenceType() {
923         Type type = null;
924         if (!see(IDENTIFIER)) {
925             type = basicType();
926             mustBe(LBRACK);
927             mustBe(RBRACK);
928             type = new ArrayTypeName(type);
929         } else {
930             type = qualifiedIdentifier();
931         }
932         while (seeDims()) {
933             mustBe(LBRACK);
934             mustBe(RBRACK);
935             type = new ArrayTypeName(type);
936         }
937         return type;
938     }
939
940
941     /**
942     * Parse a statement expression.
943     *
944     * <pre>
945     *     statementExpression ::= expression // but must have
946     *                                     // side-effect, eg i++
947     * </pre>
948     *
949     * @return an AST for a statementExpression.
950     */
951
952     private JStatement statementExpression() {
953         int line = scanner.token().line();
954         JExpression expr = expression();
955         if (expr instanceof JAssignment || expr instanceof JPreIncrementOp
956             || expr instanceof JPostDecrementOp
957             || expr instanceof JMessageExpression
958             || expr instanceof JSuperConstruction
959             || expr instanceof JThisConstruction || expr instanceof JNewOp
960             || expr instanceof JNewArrayOp) {
961             // So as not to save on stack

```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

```

962         expr.isStatementExpression = true;
963     } else {
964         reportParserError("Invalid statement expression; "
965             + "it does not have a side-effect");
966     }
967     return new JStatementExpression(line, expr);
968 }
969
970 /**
971  * An expression.
972  *
973  * <pre>
974  *     expression ::= assignmentExpression
975  * </pre>
976  *
977  * @return an AST for an expression.
978  */
979
980 private JExpression expression() {
981     return assignmentExpression();
982 }
983
984 /**
985  * Parse an assignment expression.
986  *
987  * <pre>
988  *     assignmentExpression ::=
989  *         conditionalAndExpression // level 13
990  *         [( ASSIGN // conditionalExpression
991  *           PLUS_ASSIGN // must be valid lhs
992  *           )
993  *         assignmentExpression]
994  * </pre>
995  *
996  * @return an AST for an assignmentExpression.
997  */
998
999 private JExpression assignmentExpression() {
1000     int line = scanner.token().line();
1001     JExpression lhs = conditionalAndExpression();
1002     if (have(ASSIGN)) {
1003         return new JAssignOp(line, lhs, assignmentExpression());
1004     } else if (have(PLUS_ASSIGN)) {
1005         return new JPlusAssignOp(line, lhs, assignmentExpression());
1006     } else {
1007         return lhs;
1008     }
1009 }
1010
1011 /**
1012  * Parse a conditional-and expression.
1013  *
1014  * <pre>
1015  *     conditionalAndExpression ::= equalityExpression // level 10
1016  *                                     {LAND equalityExpression}
1017  * </pre>
1018  *
1019  * @return an AST for a conditionalExpression.
1020  */
1021
1022 private JExpression conditionalAndExpression() {
1023     int line = scanner.token().line();
1024     boolean more = true;
1025     JExpression lhs = equalityExpression();
1026     while (more) {
1027         if (have(LAND)) {
1028             lhs = new JLogicalAndOp(line, lhs, equalityExpression());
1029         } else {
1030             more = false;

```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

```

1031     }
1032 }
1033 return lhs;
1034 }
1035
1036 /**
1037  * Parse an equality expression.
1038  *
1039  * <pre>
1040  *   equalityExpression ::= relationalExpression // level 6
1041  *                           {EQUAL relationalExpression}
1042  * </pre>
1043  *
1044  * @return an AST for an equalityExpression.
1045  */
1046
1047 private JExpression equalityExpression() {
1048     int line = scanner.token().line();
1049     boolean more = true;
1050     JExpression lhs = relationalExpression();
1051     while (more) {
1052         if (have(EQUAL)) {
1053             lhs = new JEqualOp(line, lhs, relationalExpression());
1054         } else {
1055             more = false;
1056         }
1057     }
1058     return lhs;
1059 }
1060
1061 /**
1062  * Parse a relational expression.
1063  *
1064  * <pre>
1065  *   relationalExpression ::= additiveExpression // level 5
1066  *                           [(GT | LE) additiveExpression
1067  *                           | INSTANCEOF referenceType]
1068  * </pre>
1069  *
1070  * @return an AST for a relationalExpression.
1071  */
1072
1073 private JExpression relationalExpression() {
1074     int line = scanner.token().line();
1075     JExpression lhs = additiveExpression();
1076     if (have(GT)) {
1077         return new JGreaterThanOp(line, lhs, additiveExpression());
1078     } else if (have(LE)) {
1079         return new JLessEqualOp(line, lhs, additiveExpression());
1080     } else if (have(INSTANCEOF)) {
1081         return new JInstanceOfOp(line, lhs, referenceType());
1082     } else {
1083         return lhs;
1084     }
1085 }
1086
1087 /**
1088  * Parse an additive expression.
1089  *
1090  * <pre>
1091  *   additiveExpression ::= multiplicativeExpression // level 3
1092  *                           {MINUS multiplicativeExpression}
1093  * </pre>
1094  *
1095  * @return an AST for an additiveExpression.
1096  */
1097
1098 private JExpression additiveExpression() {
1099     int line = scanner.token().line();

```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder


```

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

```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

```

1169      *                               | LPAREN
1170      *                               referenceType
1171      *                               RPAREN simpleUnaryExpression
1172      *                               | postfixExpression
1173      * </pre>
1174      *
1175      * @return an AST for a simpleUnaryExpression.
1176      */
1177
1178  private JExpression simpleUnaryExpression() {
1179      int line = scanner.token().line();
1180      if (have(LNOT)) {
1181          return new JLogicalNotOp(line, unaryExpression());
1182      } else if (seeCast()) {
1183          mustBe(LPAREN);
1184          boolean isBasicType = seeBasicType();
1185          Type type = type();
1186          mustBe(RPAREN);
1187          JExpression expr = isBasicType ? unaryExpression()
1188                                  : simpleUnaryExpression();
1189          return new JCastOp(line, type, expr);
1190      } else {
1191          return postfixExpression();
1192      }
1193  }
1194
1195  /**
1196   * Parse a postfix expression.
1197   *
1198   * <pre>
1199   * postfixExpression ::= primary {selector} {DEC}
1200   * </pre>
1201   *
1202   * @return an AST for a postfixExpression.
1203   */
1204
1205  private JExpression postfixExpression() {
1206      int line = scanner.token().line();
1207      JExpression primaryExpr = primary();
1208      while (see(DOT) || see(LBRACK)) {
1209          primaryExpr = selector(primaryExpr);
1210      }
1211      while (have(DEC)) {
1212          primaryExpr = new JPostDecrementOp(line, primaryExpr);
1213      }
1214      return primaryExpr;
1215  }
1216
1217  /**
1218   * Parse a selector.
1219   *
1220   * <pre>
1221   * selector ::= DOT qualifiedIdentifier [arguments]
1222   *              | LBRACK expression RBRACK
1223   * </pre>
1224   *
1225   * @param target
1226   *         the target expression for this selector.
1227   * @return an AST for a selector.
1228   */
1229
1230  private JExpression selector(JExpression target) {
1231      int line = scanner.token().line();
1232      if (have(DOT)) {
1233          // Target . selector
1234          mustBe(IDENTIFIER);
1235          String name = scanner.previousToken().image();
1236          if (see(LPAREN)) {
1237              ArrayList<JExpression> args = arguments();

```

```

1238         return new JMessageExpression(line, target, name, args);
1239     } else {
1240         return new JFieldSelection(line, target, name);
1241     }
1242 } else {
1243     mustBe(LBRACK);
1244     JExpression index = expression();
1245     mustBe(RBRACK);
1246     return new JArrayExpression(line, target, index);
1247 }
1248 }
1249
1250 /**
1251  * Parse a primary expression.
1252  *
1253  * <pre>
1254  * primary ::= parExpression
1255  *           | THIS [arguments]
1256  *           | SUPER ( arguments
1257  *                 | DOT IDENTIFIER [arguments]
1258  *                 )
1259  *           | literal
1260  *           | NEW creator
1261  *           | qualifiedIdentifier [arguments]
1262  * </pre>
1263  *
1264  * @return an AST for a primary.
1265  */
1266
1267 private JExpression primary() {
1268     int line = scanner.token().line();
1269     if (see(LPAREN)) {
1270         return parExpression();
1271     } else if (have(THIS)) {
1272         if (see(LPAREN)) {
1273             return new JThisConstruction(line, arguments());
1274         } else {
1275             return new JThis(line);
1276         }
1277     } else if (have(SUPER)) {
1278         if (!have(DOT)) {
1279             return new JSuperConstruction(line, arguments());
1280         } else {
1281             mustBe(IDENTIFIER);
1282             String name = scanner.previousToken().image();
1283             JExpression newTarget = new JSuper(line);
1284             if (see(LPAREN)) {
1285                 return new JMessageExpression(line, newTarget, null, name,
1286                     arguments());
1287             } else {
1288                 return new JFieldSelection(line, newTarget, name);
1289             }
1290         }
1291     } else if (have(NEW)) {
1292         return creator();
1293     } else if (see(IDENTIFIER)) {
1294         TypeName id = qualifiedIdentifier();
1295         if (see(LPAREN)) {
1296             return new JMessageExpression(line, null, ambiguousPart(id), id
1297                 .simpleName(), arguments());
1298         } else if (ambiguousPart(id) == null) {
1299             // A simple name
1300             return new JVariable(line, id.simpleName());
1301         } else {
1302             // ambiguousPart.fieldName
1303             return new JFieldSelection(line, ambiguousPart(id), null, id
1304                 .simpleName());
1305         }
1306     } else {

```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

```

1307         return literal();
1308     }
1309 }
1310
1311 /**
1312  * Parse a creator.
1313  *
1314  * <pre>
1315  *   creator ::= (basicType | qualifiedIdentifier)
1316  *               ( arguments
1317  *                 | LBRACK RBRACK {LBRACK RBRACK}
1318  *                   [arrayInitializer]
1319  *                 | newArrayDeclarator
1320  *               )
1321  * </pre>
1322  *
1323  * @return an AST for a creator.
1324  */
1325
1326 private JExpression creator() {
1327     int line = scanner.token().line();
1328     Type type = seeBasicType() ? basicType() : qualifiedIdentifier();
1329     if (see(LPAREN)) {
1330         ArrayList<JExpression> args = arguments();
1331         return new JNewOp(line, type, args);
1332     } else if (see(LBRACK)) {
1333         if (seeDims()) {
1334             Type expected = type;
1335             while (have(LBRACK)) {
1336                 mustBe(LBRACK);
1337                 expected = new ArrayTypeName(expected);
1338             }
1339             return arrayInitializer(expected);
1340         } else {
1341             return newArrayDeclarator(line, type);
1342         }
1343     } else {
1344         reportParserError("( or [ sought where %s found", scanner.token().
1345             .image());
1346         return new JInvalidExpression(line);
1347     }
1348 }
1349
1350 /**
1351  * Parse a new array declarator.
1352  *
1353  * <pre>
1354  *   newArrayDeclarator ::= LBRACK expression RBRACK
1355  *                           {LBRACK expression RBRACK}
1356  *                           {LBRACK RBRACK}
1357  * </pre>
1358  *
1359  * @param line
1360  *         line in which the declarator occurred.
1361  * @param type
1362  *         type of the array.
1363  * @return an AST for a newArrayDeclarator.
1364  */
1365
1366 private JNewArrayOp newArrayDeclarator(int line, Type type) {
1367     ArrayList<JExpression> dimensions = new ArrayList<JExpression>();
1368     mustBe(LBRACK);
1369     dimensions.add(expression());
1370     mustBe(RBRACK);
1371     type = new ArrayTypeName(type);
1372     while (have(LBRACK)) {
1373         if (have(RBRACK)) {
1374             // We're done with dimension expressions
1375             type = new ArrayTypeName(type);
1376             while (have(LBRACK)) {

```

```

1376         mustBe(RBRACK);
1377         type = new ArrayTypeName(type);
1378     }
1379     return new JNewArrayOp(line, type, dimensions);
1380 } else {
1381     dimensions.add(expression());
1382     type = new ArrayTypeName(type);
1383     mustBe(RBRACK);
1384 }
1385 }
1386 return new JNewArrayOp(line, type, dimensions);
1387 }
1388
1389 /**
1390  * Parse a literal.
1391  *
1392  * <pre>
1393  *     literal ::= INT_LITERAL | CHAR_LITERAL | STRING_LITERAL
1394  *               | TRUE      | FALSE      | NULL
1395  * </pre>
1396  *
1397  * @return an AST for a literal.
1398  */
1399
1400 private JExpression literal() {
1401     int line = scanner.token().line();
1402     if (have(INT_LITERAL)) {
1403         return new JLiteralInt(line, scanner.previousToken().image());
1404     } else if (have(CHAR_LITERAL)) {
1405         return new JLiteralChar(line, scanner.previousToken().image());
1406     } else if (have(STRING_LITERAL)) {
1407         return new JLiteralString(line, scanner.previousToken().image());
1408     } else if (have(TRUE)) {
1409         return new JLiteralTrue(line);
1410     } else if (have(FALSE)) {
1411         return new JLiteralFalse(line);
1412     } else if (have(NULL)) {
1413         return new JLiteralNull(line);
1414     } else {
1415         reportParserError("Literal sought where %s found", scanner.token().
1416             image());
1417         return new JWildExpression(line);
1418     }
1419 }
1420
1421 // A tracing aid. Invoke to debug the parser at various
1422 //
1423 // private void trace( String message )
1424 // {
1425 //     System.err.println( "["
1426 //         + scanner.token().line()
1427 //         + ": "
1428 //         + message
1429 //         + ", looking at a: "
1430 //         + scanner.token().tokenRep()
1431 //         + " = " + scanner.token().image() + "]" );
1432 // }
1433 }
1434

```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder