src\parser\CMinusParser.java

```java
package src.parser;

import java.io.IOException;
import java.io.BufferedWriter;
import java.io.FileWriter;

import java.util.ArrayList;

import src.scanner.CMinusScanner2;
import src.scanner.DFAException;
import src.scanner.Token;
import src.scanner.Token.TokenType;

import src.parser.Expression.Operator;
import src.parser.Declaration.typeSpecifier;

public class CMinusParser {
    private CMinusScanner2 scanner;
    private Token currToken;

    public CMinusParser(String filename) throws IOException, DFAException {
        scanner = new CMinusScanner2(filename);
    }

    public Program parse() throws IOException, DFAException, ParseException {
        Program head = null;
        currToken = scanner.getNextToken();
        if (currToken.type != TokenType.EOF) {
            head = parseProgram();
        } else {
            throw new ParseException("parse() Expected: something, anything\nReceived: " +
currToken.type.name());
        }

        if(currToken.type != TokenType.EOF){
            throw new ParseException("parse() did not expect any more tokens\nReceived: " +
currToken.type.name());
        }

        return head;
    }

    private Program parseProgram() throws IOException, DFAException, ParseException {
        Program program = null;
        if (currToken.type == TokenType.INT || currToken.type == TokenType.VOID) {
            ArrayList<Declaration> declList = new ArrayList<Declaration>();
            Declaration decl = parseDeclaration();
            declList.add(decl);

            while (currToken.type == TokenType.INT || currToken.type == TokenType.VOID) {
                decl = parseDeclaration();
                declList.add(decl);
            }

            program = new Program(declList);
```

```
54
55          } else {
56              throw new ParseException("ParseProgram() Expected: INT or VOID\nReceived: " +
    currToken.type.name());
57
58          }
59
60          return program;
61      }
62
63      private Declaration parseDeclaration() throws IOException, DFAException, ParseException {
64          Declaration decl = null;
65          if (currToken.type == TokenType.VOID) {
66              // Take "void"
67              typeSpecifier typeSpec = typeSpecifier.VOID;
68              currToken = scanner.getNextToken();
69
70              if (currToken.type == TokenType.ID) {
71                  // Take ID
72                  String id = (String) currToken.data;
73                  currToken = scanner.getNextToken();
74
75                  decl = parseFunctionDeclarationPrime(typeSpec, id);
76              } else {
77                  throw new ParseException("parseDeclaration() Expected: ID\nReceived: " +
    currToken.type.name());
78              }
79
80          } else if (currToken.type == TokenType.INT) {
81              // Take "int"
82              typeSpecifier typeSpec = typeSpecifier.INT;
83              currToken = scanner.getNextToken();
84
85              if (currToken.type == TokenType.ID) {
86                  // Take ID
87                  String id = (String) currToken.data;
88                  currToken = scanner.getNextToken();
89                  decl = parseDeclarationPrime(typeSpec, id);
90              } else {
91                  throw new ParseException("parseDeclaration() Expected: ID\nReceived: " +
    currToken.type.name());
92              }
93
94          } else {
95              throw new ParseException("parseDeclaration() Expected: INT or VOID\nReceived: " +
    currToken.type.name());
96          }
97          return decl;
98      }
99
100     private Declaration parseDeclarationPrime(typeSpecifier typeSpec, String id)
101             throws IOException, DFAException, ParseException {
102         Declaration decl = null;
103         if (currToken.type == TokenType.SEMI) {
104             // Take ";"
105             currToken = scanner.getNextToken();
106             decl = new VariableDeclaration(id, 0);
107         } else if (currToken.type == TokenType.L_BRACK) {
108             // Take "["
```

```
109             currToken = scanner.getNextToken();
110             int num = 0;
111 
112             if (currToken.type == TokenType.NUM) {
113                 // Take NUM
114                 num = (int) currToken.data;
115                 currToken = scanner.getNextToken();
116             } else {
117                 throw new ParseException("parseDeclarationPrime() Expected: NUM\nReceived: " +
currToken.type.name());
118             }
119 
120             if (currToken.type == TokenType.R_BRACK) {
121                 // Take ]
122                 currToken = scanner.getNextToken();
123             } else {
124                 throw new ParseException("parseDeclarationPrime() Expected: ]\nReceived: " +
currToken.type.name());
125             }
126 
127             if (currToken.type == TokenType.SEMI) {
128                 // Take ;
129                 currToken = scanner.getNextToken();
130             } else {
131                 throw new ParseException("parseDeclarationPrime() Expected: ]\nReceived: " +
currToken.type.name());
132             }
133 
134             decl = new VariableDeclaration(id, num);
135 
136         } else if (currToken.type == TokenType.L_PAREN) {
137             decl = parseFunctionDeclarationPrime(typeSpec, id);
138 
139         } else {
140             throw new ParseException(
141                     "parseDeclarationPrime() Expected: ;, [, or (\nReceived: " +
currToken.type.name());
142         }
143         return decl;
144     }
145 
146     private Declaration parseFunctionDeclarationPrime(typeSpecifier typeSpec, String id)
147             throws IOException, DFAException, ParseException {
148         Declaration funDecl = null;
149         if (currToken.type == TokenType.L_PAREN) {
150             // Take "("
151             currToken = scanner.getNextToken();
152             ArrayList<Param> paramList = parseParamList();
153 
154             if (currToken.type == TokenType.R_PAREN) {
155                 // Take )
156                 currToken = scanner.getNextToken();
157             } else {
158                 throw new ParseException(
159                         "parseFunctionDeclarationPrime() Expected: )\nReceived: " +
currToken.type.name());
160             }
161 
162             CompoundStatement compoundStatement = parseCompoundStatement();
```

```
163
164             funDecl = new FunctionDeclaration(typeSpec, id, paramList, compoundStatement);
165         } else {
166             throw new ParseException("parseFunctionDeclarationPrime() Expected: (\nReceived: " +
    currToken.type.name());
167         }
168
169         return funDecl;
170     }
171
172     private ArrayList<Param> parseParamList() throws IOException, DFAException, ParseException {
173         ArrayList<Param> paramList = new ArrayList<Param>();
174         Param param = null;
175         if (currToken.type == TokenType.INT) {
176             param = parseParam();
177             paramList.add(param);
178             while (currToken.type == TokenType.COMMA) {
179                 // take ","
180                 currToken = scanner.getNextToken();
181                 param = parseParam();
182                 paramList.add(param);
183             }
184         } else if (currToken.type == TokenType.VOID) {
185             // take "void"
186             currToken = scanner.getNextToken();
187         } else {
188             throw new ParseException("parseParamList() Expected either a list of int params or \"
    void\"");
189         }
190         return paramList;
191     }
192
193     private Param parseParam() throws IOException, DFAException, ParseException {
194         Param param = null;
195         boolean array = false;
196         String id;
197         if (currToken.type == TokenType.INT) {
198             // Take "int"
199             currToken = scanner.getNextToken();
200         } else {
201             throw new ParseException("parseParam() Expected: INT\nReceived: " +
    currToken.type.name());
202         }
203
204         if (currToken.type == TokenType.ID) {
205             // Take ID
206             id = (String) currToken.data;
207             currToken = scanner.getNextToken();
208         } else {
209             throw new ParseException("parseParam() Expected: ID\nReceived: " +
    currToken.type.name());
210         }
211
212         if (currToken.type == TokenType.L_BRACK) {
213             // Take [
214             currToken = scanner.getNextToken();
215             array = true;
216
217             if (currToken.type == TokenType.R_BRACK) {
```

```
                // Take ]
                currToken = scanner.getNextToken();
            } else {
                throw new ParseException("parseParam() Expected: ]\nReceived: " +
currToken.type.name());
            }
        }

        param = new Param(id, array);
        return param;
    }

    private CompoundStatement parseCompoundStatement() throws IOException, DFAException,
ParseException {
        if (currToken.type == TokenType.L_CURLY) {
            // Take "{"
            currToken = scanner.getNextToken();
        } else {
            throw new ParseException("parseCompoundStatement() Expected: {\nRecieved: " +
currToken.type.name());
        }
        ArrayList<VariableDeclaration> localDeclarations = parseLocalDeclarations();
        ArrayList<Statement> statementList = parseStatementList();
        if(currToken.type == TokenType.R_CURLY) {
            // Take "}"
            currToken = scanner.getNextToken();
        } else {
            throw new ParseException("parseCompoundStatement() Expected: }\nReceived: " +
currToken.type.name());
        }
        CompoundStatement compoundStatement = new CompoundStatement(localDeclarations,
statementList);
        return compoundStatement;
    }

    private ArrayList<VariableDeclaration> parseLocalDeclarations() throws IOException,
DFAException, ParseException {
        ArrayList<VariableDeclaration> localDecls = new ArrayList<VariableDeclaration>();
        String id = null;
        int num = 0;
        VariableDeclaration localDeclaration = null;
        while (currToken.type == TokenType.INT) {

            // Take INT
            currToken = scanner.getNextToken();
            if (currToken.type == TokenType.ID) {
                // Take ID
                id = (String) currToken.data;
                currToken = scanner.getNextToken();
            } else {
                throw new ParseException("parseLocalDeclarations() Expected: ID\nReceived: " +
currToken.type.name());
            }

            if (currToken.type == TokenType.L_BRACK) {
                // Take [
                currToken = scanner.getNextToken();

                if (currToken.type == TokenType.NUM) {
```

```
                    // Take NUM
                    num = (int) currToken.data;
                    currToken = scanner.getNextToken();
                } else {
                    throw new ParseException(
                            "parseLocalDeclarations() Expected: NUM\nReceived: " +
currToken.type.name());
                }

                if (currToken.type == TokenType.R_BRACK) {
                    // Take ]
                    currToken = scanner.getNextToken();
                } else {
                    throw new ParseException(
                            "parseLocalDeclarations() Expected: ]\nReceived: " +
currToken.type.name());
                }
            }
            if (currToken.type == TokenType.SEMI) {
                // Take ;
                currToken = scanner.getNextToken();
            } else {
                throw new ParseException("parseLocalDeclarations() Expected: ;\nReceived: " +
currToken.type.name());
            }
            localDeclaration = new VariableDeclaration(id, num);

            localDecls.add(localDeclaration);
        }
        return localDecls;
    }

    private ArrayList<Statement> parseStatementList() throws IOException, DFAException,
ParseException {
        ArrayList<Statement> statementList = new ArrayList<Statement>();
        while (currToken.type == TokenType.ID || currToken.type == TokenType.NUM ||
currToken.type == TokenType.L_PAREN
                || currToken.type == TokenType.L_CURLY || currToken.type == TokenType.IF
                || currToken.type == TokenType.WHILE
                || currToken.type == TokenType.RETURN) {
            statementList.add(parseStatement());
            // currToken = scanner.getNextToken(); DONT THINK WE NEED THIS HERE
        }
        return statementList;
    }

    private Statement parseStatement() throws IOException, DFAException, ParseException {
        Statement lhs = null;
        switch (currToken.type) {
            case ID:
            case NUM:
            case L_PAREN:
                lhs = parseExpressionStatement();
                break;
            case L_CURLY:
                lhs = parseCompoundStatement();
                break;
            case IF:
                lhs = parseSelectionStatement();
```

```
                break;
            case WHILE:
                lhs = parseIterationStatement();
                break;
            case RETURN:
                lhs = parseReturnStatement();
                break;
            default:
                throw new ParseException("parseStatement() Expected: ID, NUM, (, {, IF, WHILE, or RETURN\nRecieved: " + currToken.type.name());
        }
        return lhs;
    }

    private ExpressionStatement parseExpressionStatement() throws IOException, DFAException, ParseException {
        // no need to check what currToken.type is b/c we already did that to get sent
        // here
        Expression expression = parseExpression();

        if (currToken.type == TokenType.SEMI) {
            // Take ";"
            currToken = scanner.getNextToken();
        } else {
            throw new ParseException("parseExpressionStatement() Expected: ;\nRecieved: " + currToken.type.name());
        }

        return new ExpressionStatement(expression);
    }

    private SelectionStatement parseSelectionStatement() throws IOException, DFAException, ParseException {
        if(currToken.type == TokenType.IF){
            // Take "IF"
            currToken = scanner.getNextToken();
        }
        else{
            throw new ParseException("parseSelectionStatement() Expected: IF\nRecieved: " + currToken.type.name());
        }
        Expression ifExpression = null;
        Statement ifStatement = null, elseStatement = null;

        if (currToken.type == TokenType.L_PAREN) {
            // Take "("
            currToken = scanner.getNextToken();
        } else {
            throw new ParseException("parseSelectionStatement() Expected: (\nRecieved: " + currToken.type.name());
        }

        ifExpression = parseExpression();

        if (currToken.type == TokenType.R_PAREN) {
            // Take ")"
            currToken = scanner.getNextToken();
        } else {
```

```
376             throw new ParseException("parseSelectionStatement() Expected: )\nRecieved: " +
    currToken.type.name());
377         }
378 
379         ifStatement = parseStatement();
380 
381         if (currToken.type == TokenType.ELSE) {
382             // Take "ELSE"
383             currToken = scanner.getNextToken();
384             elseStatement = parseStatement();
385         }
386 
387         return new SelectionStatement(ifExpression, ifStatement, elseStatement);
388 
389     }
390 
391     private IterationStatement parseIterationStatement() throws IOException, DFAException,
    ParseException {
392         if(currToken.type == TokenType.WHILE){
393             // Take "WHILE"
394             currToken = scanner.getNextToken();
395         }
396         else{
397             throw new ParseException("parseIterationStatement() Expected: WHILE\nRecieved: " +
    currToken);
398         }
399         Expression whileExpression = null;
400         Statement whileStatement = null;
401 
402         if (currToken.type == TokenType.L_PAREN) {
403             // Take "("
404             currToken = scanner.getNextToken();
405         } else {
406             throw new ParseException("parseIterationStatement() Expected: (\nRecieved: " +
    currToken.type.name());
407         }
408 
409         whileExpression = parseExpression();
410 
411         if (currToken.type == TokenType.R_PAREN) {
412             // Take "("
413             currToken = scanner.getNextToken();
414         } else {
415             throw new ParseException("parseIterationStatement() Expected: )\nRecieved: " +
    currToken.type.name());
416         }
417 
418         whileStatement = parseStatement();
419 
420         return new IterationStatement(whileExpression, whileStatement);
421     }
422 
423     private ReturnStatement parseReturnStatement() throws IOException, DFAException,
    ParseException {
424         Expression returnExpression = null;
425         if (currToken.type == TokenType.RETURN) {
426             // Take "RETURN"
427             currToken = scanner.getNextToken();
428         } else {
```

```
429             throw new ParseException("parseReturnStatement() Expected: return\nRecieved: " +
    currToken.type.name());
430         }
431 
432         if (currToken.type == TokenType.ID || currToken.type == TokenType.NUM || currToken.type =
    = TokenType.L_PAREN) {
433             returnExpression = parseExpression();
434         }
435 
436         if (currToken.type == TokenType.SEMI) {
437             // Take ";"
438             currToken = scanner.getNextToken();
439         } else {
440             throw new ParseException("parseReturnStatement() Expected: ;\nRecieved: " +
    currToken.type.name());
441         }
442 
443         return new ReturnStatement(returnExpression);
444 
445     }
446 
447     private Expression parseExpression() throws IOException, DFAException, ParseException {
448         Expression expression = null;
449         if (currToken.type == TokenType.ID) {
450             // Take "ID"
451             String id = (String) currToken.data;
452             currToken = scanner.getNextToken();
453             expression = parseExpressionPrime(id);
454 
455         } else if (currToken.type == TokenType.NUM) {
456             // Take "NUM"
457             NUMExpression num = new NUMExpression((int) currToken.data);
458             currToken = scanner.getNextToken();
459             expression = parseSimpleExpressionPrime(num);
460 
461         } else if (currToken.type == TokenType.L_PAREN) {
462             // Take "("
463             currToken = scanner.getNextToken();
464             Expression inExpression = parseExpression();
465             if (currToken.type == TokenType.R_PAREN) {
466                 // Take ")"
467                 currToken = scanner.getNextToken();
468             } else {
469                 throw new ParseException("parseExpression() Expected: )\nRecieved: " +
    currToken.type.name());
470             }
471 
472             expression = parseSimpleExpressionPrime(inExpression);
473 
474         } else {
475             throw new ParseException(
476                     "parseExpression() Expected: ID, NUM, L_PAREN\nRecieved: " +
    currToken.type.name());
477         }
478         return expression;
479     }
480 
481     private Expression parseExpressionPrime(String id) throws IOException, DFAException,
    ParseException {
```

```
482         Expression expression = null;
483         if (currToken.type == TokenType.ASSIGN) {
484             // Take "="
485             currToken = scanner.getNextToken();
486             IDExpression idExpression = new IDExpression(id, null);
487             Expression rhs = parseExpression();
488             expression = new AssignExpression(idExpression, rhs);
489 
490         } else if (currToken.type == TokenType.L_BRACK) {
491             // Take "["
492             currToken = scanner.getNextToken();
493             Expression inExpression = parseExpression();
494 
495             if (currToken.type == TokenType.R_BRACK) {
496                 // Take "]"
497                 currToken = scanner.getNextToken();
498             } else {
499                 throw new ParseException("parseExpressionPrime() Expected: ]\nRecieved: " +
currToken.type.name());
500             }
501 
502             IDExpression idExpression = new IDExpression(id, inExpression);
503             expression = parseExpressionDoublePrime(idExpression);
504 
505         } else if (currToken.type == TokenType.L_PAREN) {
506             // Take "("
507             currToken = scanner.getNextToken();
508 
509             ArrayList<Expression> args = parseArgs();
510 
511             if (currToken.type == TokenType.R_PAREN) {
512                 // Take ")"
513                 currToken = scanner.getNextToken();
514             } else {
515                 throw new ParseException("parseExpressionPrime() Expected: )\nRecieved: " +
currToken.type.name());
516             }
517 
518             CallExpression callExpression = new CallExpression(id, args);
519             expression = parseSimpleExpressionPrime(callExpression);
520 
521         } else {
522             IDExpression idExpression = new IDExpression(id, null);
523             expression = parseSimpleExpressionPrime(idExpression);
524         }
525 
526         return expression;
527     }
528 
529     private Expression parseExpressionDoublePrime(IDExpression idExpression)
530             throws IOException, DFAException, ParseException {
531         Expression expression = null;
532         if (currToken.type == TokenType.ASSIGN) {
533             // Take "="
534             currToken = scanner.getNextToken();
535             Expression rhs = parseExpression();
536             expression = new AssignExpression(idExpression, rhs);
537         } else {
```

```
538             expression = parseSimpleExpressionPrime(idExpression);
539         }
540         return expression;
541     }
542 
543     private Expression parseSimpleExpressionPrime(Expression lhs) throws IOException,
    DFAException, ParseException {
544         Expression expression = parseAdditiveExpression(lhs);
545 
546         if (currToken.type == TokenType.LTE || currToken.type == TokenType.LT || currToken.type =
    = TokenType.GT
547                 || currToken.type == TokenType.GTE
548                 || currToken.type == TokenType.EQ || currToken.type == TokenType.NEQ) {
549             // Take relop
550             Operator relop = parseRelop();
551             currToken = scanner.getNextToken();
552             Expression rhs = parseAdditiveExpression(null);
553 
554             expression = new BinaryExpression(expression, rhs, relop);
555         }
556 
557         return expression;
558 
559     }
560 
561     private Expression parseAdditiveExpression(Expression lhs) throws IOException, DFAException,
    ParseException {
562         lhs = parseTerm(lhs); // if not null, treated as AdditiveExpressionPrime()
563         while (currToken.type == TokenType.PLUS || currToken.type == TokenType.MINUS) {
564             // Take addop
565             Operator addop = parseAddop();
566             currToken = scanner.getNextToken();
567             Expression rhs = parseTerm(null);
568 
569             lhs = new BinaryExpression(lhs, rhs, addop);
570         }
571 
572         return lhs;
573     }
574 
575     private Expression parseTerm(Expression lhs) throws IOException, DFAException, ParseException
    {
576         if (lhs == null) { // parse Term
577             lhs = parseFactor();
578         }
579         // now just do parse Term', but since we made sure we have the lhs, works for
580         // Term and Term'
581         while (currToken.type == TokenType.MULT || currToken.type == TokenType.DIV) {
582             // Take mulop
583             Operator mulop = parseMulop();
584             currToken = scanner.getNextToken();
585             Expression rhs = parseFactor();
586 
587             lhs = new BinaryExpression(lhs, rhs, mulop);
588         }
589 
590         return lhs;
591     }
592 
```

```
593     private Operator parseRelop() throws IOException, DFAException, ParseException {
594         Operator retOp = null;
595         // if (currToken.type == TokenType.LTE || currToken.type == TokenType.LT ||
596         // currToken.type == TokenType.GT || currToken.type == TokenType.GTE
597         // || currToken.type == TokenType.EQ || currToken.type == TokenType.NEQ) {
598         switch (currToken.type) {
599             case LTE:
600                 retOp = Operator.LTE;
601                 break;
602             case LT:
603                 retOp = Operator.LT;
604                 break;
605             case GT:
606                 retOp = Operator.GT;
607                 break;
608             case GTE:
609                 retOp = Operator.GTE;
610                 break;
611             case EQ:
612                 retOp = Operator.EQ;
613                 break;
614             case NEQ:
615                 retOp = Operator.NEQ;
616                 break;
617             default:
618                 throw new ParseException("parseRelop() Error: Recieved a non-Relop token");
619         }
620         return retOp;
621     }
622 
623     private Operator parseAddop() throws IOException, DFAException, ParseException {
624         Operator addOp = null;
625         switch (currToken.type) {
626             case PLUS:
627                 addOp = Operator.PLUS;
628                 break;
629             case MINUS:
630                 addOp = Operator.MINUS;
631                 break;
632             default:
633                 throw new ParseException("parseAddop() Error: Recieved a non-Addop token");
634         }
635         return addOp;
636     }
637 
638     private Operator parseMulop() throws IOException, DFAException, ParseException {
639         Operator mulOp = null;
640         switch (currToken.type) {
641             case DIV:
642                 mulOp = Operator.DIV;
643                 break;
644             case MULT:
645                 mulOp = Operator.MULT;
646                 break;
647             default:
648                 throw new ParseException("parseMulop() Error: Recieved a non-Mulop token");
649         }
650         return mulOp;
```

```
651     }
652
653     private Expression parseFactor() throws IOException, DFAException, ParseException {
654         // return parseExpression(), parseVarcall(), or NUMExpression
655         // ^^^^^^
656         // Parens are not needed in AST b/c they are just used to override precedence
657         // so just return parseExpression()
658         Expression expression = null;
659
660         if (currToken.type == TokenType.L_PAREN) {
661             // Take "("
662             currToken = scanner.getNextToken();
663             expression = parseExpression();
664
665         } else if (currToken.type == TokenType.ID) {
666             // Take ID
667             String id = (String) currToken.data;
668             currToken = scanner.getNextToken();
669             expression = parseVarcall(id);
670
671         } else if (currToken.type == TokenType.NUM) {
672             // Take NUM
673             int num = (int) currToken.data;
674             currToken = scanner.getNextToken();
675             expression = new NUMExpression(num);
676
677         } else {
678             throw new ParseException("parseFactor() Expected: (, ID, or NUM\nRecieved: " +
    currToken.type.name());
679         }
680
681         return expression;
682     }
683
684     private Expression parseVarcall(String id) throws IOException, DFAException, ParseException {
685         Expression expression = null;
686         // return IDExpression or CallExpression
687         if (currToken.type == TokenType.L_BRACK) {
688             // Take "["
689             currToken = scanner.getNextToken();
690             Expression inExpression = parseExpression();
691
692             if (currToken.type == TokenType.R_BRACK) {
693                 // Take "]"
694                 currToken = scanner.getNextToken();
695             } else {
696                 throw new ParseException("parseVarcall() Expected: ]\nRecieved: " +
    currToken.type.name());
697             }
698
699             expression = new IDExpression(id, inExpression);
700
701         } else if (currToken.type == TokenType.L_PAREN) {
702             // Take "("
703             currToken = scanner.getNextToken();
704             ArrayList<Expression> args = parseArgs();
705
706             if (currToken.type == TokenType.R_PAREN) {
```

```
707                 // Take ")"
708                 currToken = scanner.getNextToken();
709             } else {
710                 throw new ParseException("parseVarcall() Expected: )\nRecieved: " +
    currToken.type.name());
711             }
712             expression = new CallExpression(id, args);
713 
714         } else {
715             // Return ID Expression
716             expression = new IDExpression(id, null);
717         }
718         return expression;
719     }
720 
721     private ArrayList<Expression> parseArgs() throws IOException, DFAException, ParseException {
722         ArrayList<Expression> args = new ArrayList<Expression>();
723 
724         if (currToken.type == TokenType.ID || currToken.type == TokenType.NUM || currToken.type =
    = TokenType.L_PAREN) {
725             Expression expression = parseExpression();
726             args.add(expression);
727 
728             while (currToken.type == TokenType.COMMA) {
729                 // Take ","
730                 currToken = scanner.getNextToken();
731                 expression = parseExpression();
732                 args.add(expression);
733             }
734         }
735 
736         return args;
737     }
738 
739     public static void printTree(Program head) {
740         printTree(head, false, "");
741     }
742 
743     public static void printTree(Program head, String filename) {
744         printTree(head, true, filename);
745     }
746 
747     private static void printTree(Program head, boolean toFile, String filename) {
748         String output = head.print(0);
749         if (toFile) {
750             try {
751                 BufferedWriter out = new BufferedWriter(new FileWriter(filename));
752                 out.write(output);
753                 out.close();
754             } catch (IOException e) {
755                 System.err.println("COULD NOT OUTPUT TO FILE");
756                 e.printStackTrace();
757             }
758         }
759         System.out.println(output);
760     }
761 
762     public static void main(String[] args) throws Exception {
```

```
763         String filename_prefix = "code/parse3";
764         CMinusParser parser = new CMinusParser(filename_prefix + ".cm");
765         Program head = parser.parse();
766         printTree(head, filename_prefix + ".ast");
767     }
768 }
```