APL Assignment:

Main class that parses test file and find complexity of the class.

Main.java

```
package javamcc;

import org.antlr.runtime.ANTLRStringStream;
import org.antlr.runtime.CommonTokenStream;
import org.antlr.runtime.TokenStream;
import java.nio.file.Files;
import java.nio.file.Paths;


public class Main {
   public static void main(String[] args) {
       try {
           String inputFilePath =
"/Users/ishanksharma/Desktop/antlr-2/src/javamcc/JavaMCC.java";
           String code = new
String(Files.readAllBytes(Paths.get(inputFilePath)));

           ANTLRStringStream input = new ANTLRStringStream(code);
           javamcc.JavaMCCLexer lexer = new javamcc.JavaMCCLexer(input);
           TokenStream tokens = new CommonTokenStream(lexer);

           javamcc.JavaMCCParser parser = new javamcc.JavaMCCParser(tokens);

           parser.compilationUnit();

           System.out.println("Parsing complete.");

       } catch (Exception e) {
           e.printStackTrace();
       }
   }
}


// Test Class
```

JavaMCC.java

```
package javamcc;
public class JavaMCC {
   public void simpleMethod() {
       System.out.println("This is a simple method.");
   }

   public int computeFactorial(int n) {
       if (n <= 1) {
           return 1;
       } else {
           return n * computeFactorial(n - 1);
```

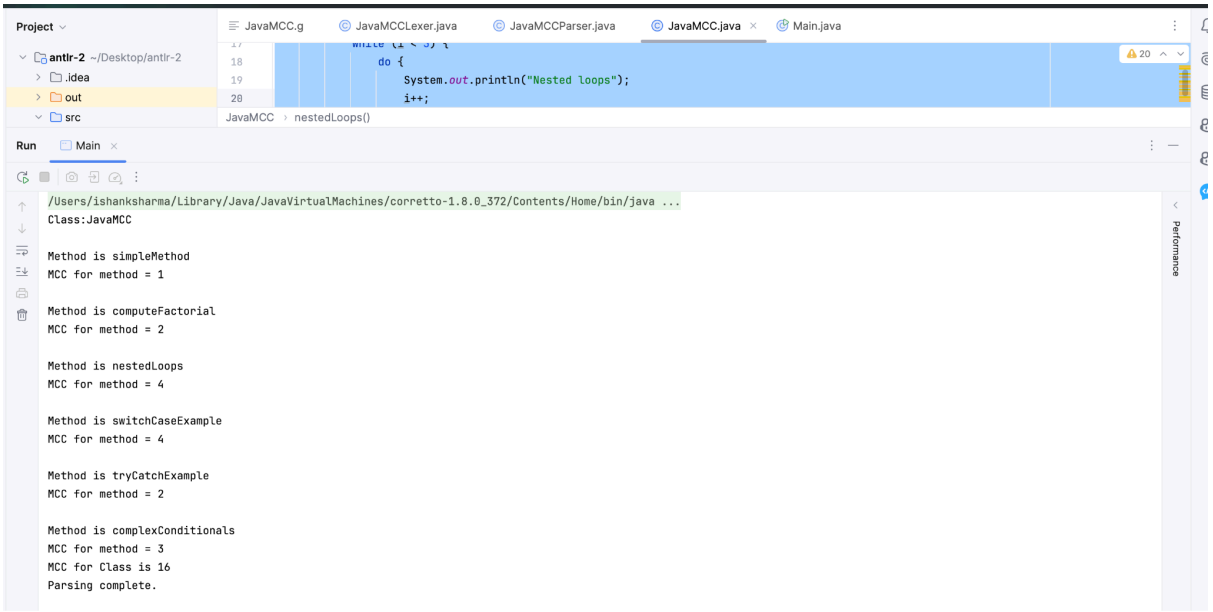
```
        }
    }

    public void nestedLoops() {
        for (int i = 0; i < 5; i++) {
            while (i < 3) {
                do {
                    System.out.println("Nested loops");
                    i++;
                } while (i < 2);
            }
        }
    }

    public void switchCaseExample(int day) {
        switch (day) {
            case 1:
                System.out.println("Monday");
                break;
            case 2:
                System.out.println("Tuesday");
                break;
            default:
                System.out.println("Other day");
        }
    }

    public void tryCatchExample() {
        try {
            int[] numbers = {1, 2, 3};
            System.out.println(numbers[5]);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Exception caught");
        } finally {
            System.out.println("Finally block");
        }
    }

    public void complexConditionals(int x) {
        if (x > 0 && x < 10) {
            System.out.println("x is between 1 and 9");
        } else if (x >= 10 || x == 0) {
            System.out.println("x is 0 or greater than or equal to 10");
        } else {
            System.out.println("x is negative");
        }
    }
}
```



Grammar File Updated

JavaMCC.g

```
grammar JavaMCC;
options {backtrack=true; memoize=true;}
@header{
package javamcc;
}
@members{
String MethodName;      //I left these declarations to help you start
int MCCClass = 0;
int MCCMethod = 0;
}
@lexer::header{
package javamcc;
}
@lexer::members {


}

// starting point for parsing a java file
/* The annotations are separated out to make parsing faster, but must be
associated with
  a packageDeclaration or a typeDeclaration (and not an empty one). */
compilationUnit
    :    annotations
        (    packageDeclaration importDeclaration* typeDeclaration*
        |    classOrInterfaceDeclaration typeDeclaration*
```

```
        )
    |   packageDeclaration? importDeclaration* typeDeclaration*
    ;

ifStatement
    :   'if' parExpression statement ( options {greedy=true;} : 'else'
statement )?
        {
            System.out.println("Found an if statement");
            MCCMethod++;
        }
;

packageDeclaration
    :   'package' qualifiedName ';'
    ;

importDeclaration
    :   'import' 'static'? qualifiedName ('.' '*')? ';'
    ;

typeDeclaration
    :   classOrInterfaceDeclaration
    |   ';'
    ;

classOrInterfaceDeclaration
    :   classOrInterfaceModifiers (classDeclaration |
interfaceDeclaration)
    ;

classOrInterfaceModifiers
    :   classOrInterfaceModifier*
    ;

classOrInterfaceModifier
    :   annotation   // class or interface
    |   'public'     // class or interface
    |   'protected'  // class or interface
    |   'private'     // class or interface
    |   'abstract'   // class or interface
    |   'static'     // class or interface
    |   'final'      // class only -- does not apply to interfaces
    |   'strictfp'   // class or interface
    ;
```

```
modifiers
   :   modifier*
   ;

classDeclaration
   :   normalClassDeclaration
   |   enumDeclaration
   ;

normalClassDeclaration
   : 'class' Identifier { MCCClass = 0; System.out.println("Class:" +
$Identifier.text); } typeParameters?
     ('extends' type)?
     ('implements' typeList)?
     classBody
;

typeParameters
   :   '<' typeParameter (',' typeParameter)* '>'
   ;

typeParameter
   :   Identifier ('extends' typeBound)?
   ;

typeBound
   :   type ('&' type)*
   ;

enumDeclaration
   :   ENUM Identifier ('implements' typeList)? enumBody
   ;

enumBody
   :   '{' enumConstants? ','? enumBodyDeclarations? '}'
   ;

enumConstants
   :   enumConstant (',' enumConstant)*
   ;

enumConstant
   :   annotations? Identifier arguments? classBody?
   ;

enumBodyDeclarations
```

```
   :   ';' (classBodyDeclaration)*
   ;

interfaceDeclaration
   :   normalInterfaceDeclaration
   |   annotationTypeDeclaration
   ;

normalInterfaceDeclaration
   :   'interface' Identifier typeParameters? ('extends' typeList)?
interfaceBody
   ;

typeList
   :   type (',' type)*
   ;

classBody
   : '{' classBodyDeclaration* '}' { System.out.println("MCC for Class
is " + MCCClass); }
;

interfaceBody
   :   '{' interfaceBodyDeclaration* '}'
   ;

classBodyDeclaration
   :   ';'
   |   'static'? block
   |   modifiers? memberDecl
   ;

memberDecl
   :   methodDeclaration
   |   fieldDeclaration
   |   constructorDeclaration
   |   interfaceDeclaration
   |   classDeclaration
   ;


constructorDeclaration
   :   modifiers? Identifier formalParameters ( 'throws'
qualifiedNameList )? constructorBody
       {
           System.out.println("\nConstructor is " + $Identifier.text);
```

```
            MCCMethod = 1;
        }
;

memberDeclaration
    :   type (methodDeclaration | fieldDeclaration)
    ;
genericMethodOrConstructorDecl
    :   typeParameters  genericMethodOrConstructorRest
    ;

genericMethodOrConstructorRest
    :   (type | 'void') Identifier  methodDeclaratorRest
    |   Identifier   constructorDeclaratorRest
    ;

methodDeclaration
    :   modifiers? (type | 'void') Identifier
        {
            System.out.println("\nMethod is " + $Identifier.text);
            MCCMethod = 1;
        }
        formalParameters ( 'throws' qualifiedNameList )?
        ( block | ';' )
        {
            System.out.println("MCC for method = " + MCCMethod);
            MCCClass += MCCMethod;
        }
;




fieldDeclaration
    :   variableDeclarators ';'
    ;

interfaceBodyDeclaration
    :   modifiers interfaceMemberDecl
    |   ';'
    ;

interfaceMemberDecl
    :   interfaceMethodOrFieldDecl
    |   interfaceGenericMethodDecl
    |   'void' Identifier voidInterfaceMethodDeclaratorRest
```

```
    |   interfaceDeclaration
    |   classDeclaration
    ;

interfaceMethodOrFieldDecl
    :   type Identifier {System.out.println($Identifier.text);}
interfaceMethodOrFieldRest
    ;

interfaceMethodOrFieldRest
    :   constantDeclaratorsRest ';'
    |   interfaceMethodDeclaratorRest
    ;
methodDeclaratorRest
    : formalParameters ('throws' qualifiedNameList)?
      (methodBody { System.out.println("MCC for method = " + MCCMethod);
MCCClass += MCCMethod; }
     | ';')
;

voidMethodDeclaratorRest
    :   formalParameters ('throws' qualifiedNameList)?
        (   methodBody
        |   ';'
        )
    ;

interfaceMethodDeclaratorRest
    :   formalParameters ('[' ']')* ('throws' qualifiedNameList)? ';'
    ;

interfaceGenericMethodDecl
    :   typeParameters (type | 'void') Identifier
        interfaceMethodDeclaratorRest
    ;

voidInterfaceMethodDeclaratorRest
    :   formalParameters ('throws' qualifiedNameList)? ';'
    ;

constructorDeclaratorRest
    :   formalParameters ('throws' qualifiedNameList)? constructorBody
    ;

constantDeclarator
    :   Identifier constantDeclaratorRest
```

```
    ;

variableDeclarators
    :   variableDeclarator (',' variableDeclarator)*
    ;

variableDeclarator
    :   variableDeclaratorId ('=' variableInitializer)?
    ;

constantDeclaratorsRest
    :   constantDeclaratorRest (',' constantDeclarator)*
    ;

constantDeclaratorRest
    :   ('[' ']')* '=' variableInitializer
    ;

variableDeclaratorId
    :   Identifier ('[' ']')*
    ;

variableInitializer
    :   arrayInitializer
    |   expression
    ;

arrayInitializer
    :   '{' (variableInitializer (',' variableInitializer)* (',')? )? '}'
    ;

modifier
    :   annotation
    |   'public'
    |   'protected'
    |   'private'
    |   'static'
    |   'abstract'
    |   'final'
    |   'native'
    |   'synchronized'
    |   'transient'
    |   'volatile'
    |   'strictfp'
    ;
```

```
packageOrTypeName
    :   qualifiedName
    ;

enumConstantName
    :   Identifier
    ;

typeName
    :   qualifiedName
    ;

type
    :  classOrInterfaceType ('[' ']')*
    |  primitiveType ('[' ']')*
    ;

classOrInterfaceType
    :  Identifier
           typeArguments? ('.' Identifier typeArguments? )*
    ;

primitiveType
    :   'boolean'
    |   'char'
    |   'byte'
    |   'short'
    |   'int'
    |   'long'
    |   'float'
    |   'double'
    ;

variableModifier
    :   'final'
    |   annotation
    ;

typeArguments
    :   '<' typeArgument (',' typeArgument)* '>'
    ;

typeArgument
    :   type
    |   '?' (('extends' | 'super') type)?
    ;
```

```
qualifiedNameList
   :   qualifiedName (',' qualifiedName)*
   ;

formalParameters
   :   '(' formalParameterDecls? ')'
   ;

formalParameterDecls
   :   variableModifiers type formalParameterDeclsRest
   ;

formalParameterDeclsRest
   :   variableDeclaratorId (',' formalParameterDecls)?
   |   '...' variableDeclaratorId
   ;

methodBody
   : '{' blockStatement* '}'
   ;

constructorBody
   :   '{' explicitConstructorInvocation? blockStatement* '}'
   ;

explicitConstructorInvocation
   :   nonWildcardTypeArguments? ('this' | 'super') arguments ';'
   |   primary '.' nonWildcardTypeArguments? 'super' arguments ';'
   ;


qualifiedName
   :   Identifier ('.' Identifier)*
   ;

literal
   :   integerLiteral
   |   FloatingPointLiteral
   |   CharacterLiteral
   |   StringLiteral
   |   booleanLiteral
   |   'null'
   ;

integerLiteral
```

```
    :   HexLiteral
    |   OctalLiteral
    |   DecimalLiteral
    ;

booleanLiteral
    :   'true'
    |   'false'
    ;

// ANNOTATIONS

annotations
    :   annotation+
    ;

annotation
    :   '@' annotationName ( '(' ( elementValuePairs | elementValue )?
')' )?
    ;

annotationName
    : Identifier ('.' Identifier)*
    ;

elementValuePairs
    :   elementValuePair (',' elementValuePair)*
    ;

elementValuePair
    :   Identifier '=' elementValue
    ;

elementValue
    :   conditionalExpression
    |   annotation
    |   elementValueArrayInitializer
    ;

elementValueArrayInitializer
    :   '{' (elementValue (',' elementValue)*)? (',')? '}'
    ;

annotationTypeDeclaration
    :   '@' 'interface' Identifier annotationTypeBody
    ;
```

```
annotationTypeBody
   :   '{' (annotationTypeElementDeclaration)* '}'
   ;

annotationTypeElementDeclaration
   :   modifiers annotationTypeElementRest
   ;

annotationTypeElementRest
   :   type annotationMethodOrConstantRest ';'
   |   normalClassDeclaration ';'?
   |   normalInterfaceDeclaration ';'?
   |   enumDeclaration ';'?
   |   annotationTypeDeclaration ';'?
   ;

annotationMethodOrConstantRest
   :   annotationMethodRest
   |   annotationConstantRest
   ;

annotationMethodRest
   :   Identifier '(' ')' defaultValue?
   ;

annotationConstantRest
   :   variableDeclarators
   ;

defaultValue
   :   'default' elementValue
   ;

// STATEMENTS / BLOCKS

block
   :   '{' blockStatement* '}'
   ;

blockStatement
   :   localVariableDeclarationStatement
   |   classOrInterfaceDeclaration
   |   statement
   ;
```

```
localVariableDeclarationStatement
   :    localVariableDeclaration ';'
   ;

localVariableDeclaration
   :   variableModifiers type variableDeclarators
   ;

variableModifiers
   :   variableModifier*
   ;

statement
   : block
   | ASSERT expression (':' expression)? ';'
   | 'if' parExpression statement (options {k=1;}:'else' statement)? {
MCCMethod++; }
   | 'for' '(' forControl ')' statement { MCCMethod++; }
   | 'while' parExpression statement { MCCMethod++; }
   | 'do' statement 'while' parExpression ';' { MCCMethod++; }
   | 'try' block
       ( catches { MCCMethod++; } 'finally' block
       | catches { MCCMethod++; }
       | 'finally' block
       )
   | 'switch' parExpression '{' switchBlockStatementGroups '}' {
MCCMethod++; }
   | 'synchronized' parExpression block
   | 'return' expression? ';' // Handle return with or without
expression
   | 'throw' expression ';'
   | 'break' Identifier? ';'
   | 'continue' Identifier? ';'
   | ';' // Empty statement
   | statementExpression ';'
   | Identifier ':' statement
   ;



catches
   :   catchClause (catchClause)*
   ;

catchClause
   :   'catch' '(' formalParameter ')' block
```

```
    ;

formalParameter
    :   variableModifiers type variableDeclaratorId
    ;

switchBlockStatementGroups
    :   (switchBlockStatementGroup)*
    ;

/* The change here (switchLabel -> switchLabel+) technically makes this
grammar
  ambiguous; but with appropriately greedy parsing it yields the most
  appropriate AST, one in which each group, except possibly the last
one, has
  labels and statements. */
switchBlockStatementGroup
    :   switchLabel+ blockStatement*
    ;

switchLabel
    : 'case' constantExpression ':' { MCCMethod++; }
    | 'case' enumConstantName ':' { MCCMethod++; }
    | 'default' ':'
    ;

forControl
options {k=3;} // be efficient for common case: for (ID ID : ID) ...
    :   enhancedForControl
    |   forInit? ';' expression? ';' forUpdate?
    ;

forInit
    :   localVariableDeclaration
    |   expressionList
    ;

enhancedForControl
    :   variableModifiers type Identifier ':' expression
    ;

forUpdate
    :   expressionList
    ;

// EXPRESSIONS
```

```
parExpression
   :   '(' expression ')'
   ;

expressionList
   :   expression (',' expression)*
   ;

statementExpression
   :   expression
   ;

constantExpression
   :   expression
   ;

expression
   :   conditionalExpression (assignmentOperator expression)?
   ;

assignmentOperator
   :   '='
   |   '+='
   |   '-='
   |   '*='
   |   '/='
   |   '&='
   |   '|='
   |   '^='
   |   '%='
   |   ('<' '<' '=')=> t1='<' t2='<' t3='='
       { $t1.getLine() == $t2.getLine() &&
         $t1.getCharPositionInLine() + 1 == $t2.getCharPositionInLine()
&&
         $t2.getLine() == $t3.getLine() &&
         $t2.getCharPositionInLine() + 1 == $t3.getCharPositionInLine()
}?
   |   ('>' '>' '>' '=')=> t1='>' t2='>' t3='>' t4='='
       { $t1.getLine() == $t2.getLine() &&
         $t1.getCharPositionInLine() + 1 == $t2.getCharPositionInLine()
&&
         $t2.getLine() == $t3.getLine() &&
         $t2.getCharPositionInLine() + 1 == $t3.getCharPositionInLine()
&&
         $t3.getLine() == $t4.getLine() &&
```

```
        $t3.getCharPositionInLine() + 1 == $t4.getCharPositionInLine()
}?
   |   ('>' '>' '=')=> t1='>' t2='>' t3='='
       { $t1.getLine() == $t2.getLine() &&
         $t1.getCharPositionInLine() + 1 == $t2.getCharPositionInLine()
&&
         $t2.getLine() == $t3.getLine() &&
         $t2.getCharPositionInLine() + 1 == $t3.getCharPositionInLine()
}?
   ;

conditionalExpression
   :   conditionalOrExpression ( '?' expression ':' expression )?
   ;

conditionalOrExpression
   :   conditionalAndExpression ( '||' conditionalAndExpression )*
   ;

conditionalAndExpression
   :   inclusiveOrExpression ( '&&' inclusiveOrExpression )*
   ;

inclusiveOrExpression
   :   exclusiveOrExpression ( '|' exclusiveOrExpression )*
   ;

exclusiveOrExpression
   :   andExpression ( '^' andExpression )*
   ;

andExpression
   :   equalityExpression ( '&' equalityExpression )*
   ;

equalityExpression
   :   instanceOfExpression ( ('==' | '!=') instanceOfExpression )*
   ;

instanceOfExpression
   :   relationalExpression ('instanceof' type)?
   ;

relationalExpression
   :   shiftExpression ( relationalOp shiftExpression )*
   ;
```

```
relationalOp
   :   ('<' '=')=> t1='<' t2='='
       { $t1.getLine() == $t2.getLine() &&
         $t1.getCharPositionInLine() + 1 == $t2.getCharPositionInLine()
}?
   |   ('>' '=')=> t1='>' t2='='
       { $t1.getLine() == $t2.getLine() &&
         $t1.getCharPositionInLine() + 1 == $t2.getCharPositionInLine()
}?
   |   '<'
   |   '>'
   ;

shiftExpression
   :   additiveExpression ( shiftOp additiveExpression )*
   ;

shiftOp
   :   ('<' '<')=> t1='<' t2='<'
       { $t1.getLine() == $t2.getLine() &&
         $t1.getCharPositionInLine() + 1 == $t2.getCharPositionInLine()
}?
   |   ('>' '>' '>')=> t1='>' t2='>' t3='>'
       { $t1.getLine() == $t2.getLine() &&
         $t1.getCharPositionInLine() + 1 == $t2.getCharPositionInLine()
&&
         $t2.getLine() == $t3.getLine() &&
         $t2.getCharPositionInLine() + 1 == $t3.getCharPositionInLine()
}?
   |   ('>' '>')=> t1='>' t2='>'
       { $t1.getLine() == $t2.getLine() &&
         $t1.getCharPositionInLine() + 1 == $t2.getCharPositionInLine()
}?
   ;


additiveExpression
   :   multiplicativeExpression ( ('+' | '-') multiplicativeExpression
)*
   ;

multiplicativeExpression
   :   unaryExpression ( ( '*' | '/' | '%' ) unaryExpression )*
   ;
```

```
unaryExpression
    :   '+' unaryExpression
    |   '-' unaryExpression
    |   '++' unaryExpression
    |   '--' unaryExpression
    |   unaryExpressionNotPlusMinus
    ;

unaryExpressionNotPlusMinus
    :   '~' unaryExpression
    |   '!' unaryExpression
    |   castExpression
    |   primary selector* ('++'|'--')?
    ;

castExpression
    :  '(' primitiveType ')' unaryExpression
    |  '(' (type | expression) ')' unaryExpressionNotPlusMinus
    ;

primary
    :   parExpression
    |   'this' ('.' Identifier)* identifierSuffix?
    |   'super' superSuffix
    |   literal
    |   'new' creator
    |   Identifier ('.' Identifier)* identifierSuffix?
    |   primitiveType ('[' ']')* '.' 'class'
    |   'void' '.' 'class'
    ;

identifierSuffix
    :   ('[' ']')+ '.' 'class'
    |   ('[' expression ']')+ // can also be matched by selector, but do
here
    |   arguments
    |   '.' 'class'
    |   '.' explicitGenericInvocation
    |   '.' 'this'
    |   '.' 'super' arguments
    |   '.' 'new' innerCreator
    ;

creator
    :   nonWildcardTypeArguments createdName classCreatorRest
    |   createdName (arrayCreatorRest | classCreatorRest)
```

```
    ;

createdName
    :   classOrInterfaceType
    |   primitiveType
    ;

innerCreator
    :   nonWildcardTypeArguments? Identifier classCreatorRest
    ;

arrayCreatorRest
    :   '['
        (   ']' ('[' ']')* arrayInitializer
        |   expression ']' ('[' expression ']')* ('[' ']')*
        )
    ;

classCreatorRest
    :   arguments classBody?
    ;

explicitGenericInvocation
    :   nonWildcardTypeArguments Identifier arguments
    ;

nonWildcardTypeArguments
    :   '<' typeList '>'
    ;

selector
    :   '.' Identifier arguments?
    |   '.' 'this'
    |   '.' 'super' superSuffix
    |   '.' 'new' innerCreator
    |   '[' expression ']'
    ;

superSuffix
    :   arguments
    |   '.' Identifier arguments?
    ;

arguments
    :   '(' expressionList? ')'
    ;
```

```
// LEXER

HexLiteral : '0' ('x'|'X') HexDigit+ IntegerTypeSuffix? ;

DecimalLiteral : ('0' | '1'..'9' '0'..'9'*) IntegerTypeSuffix? ;

OctalLiteral : '0' ('0'..'7')+ IntegerTypeSuffix? ;

fragment
HexDigit : ('0'..'9'|'a'..'f'|'A'..'F') ;

fragment
IntegerTypeSuffix : ('l'|'L') ;

FloatingPointLiteral
    :   ('0'..'9')+ '.' ('0'..'9')* Exponent? FloatTypeSuffix?
    |   '.' ('0'..'9')+ Exponent? FloatTypeSuffix?
    |   ('0'..'9')+ Exponent FloatTypeSuffix?
    |   ('0'..'9')+ FloatTypeSuffix
    ;

fragment
Exponent : ('e'|'E') ('+'|'-')? ('0'..'9')+ ;

fragment
FloatTypeSuffix : ('f'|'F'|'d'|'D') ;

CharacterLiteral
    :   '\'' ( EscapeSequence | ~('\''|'\\') ) '\''
    ;

StringLiteral
    :  '"' ( EscapeSequence | ~('\\'|'"') )* '"'
    ;

fragment
EscapeSequence
    :   '\\' ('b'|'t'|'n'|'f'|'r'|'\"'|'\''|'\\')
    |   UnicodeEscape
    |   OctalEscape
    ;

fragment
OctalEscape
    :   '\\' ('0'..'3') ('0'..'7') ('0'..'7')
```

```
    |   '\\' ('0'..'7') ('0'..'7')
    |   '\\' ('0'..'7')
    ;

fragment
UnicodeEscape
    :   '\\' 'u' HexDigit HexDigit HexDigit HexDigit
    ;

ENUM:   'enum'
    ;

ASSERT
    :   'assert'
    ;

Identifier
    :   Letter (Letter|JavaIDDigit)*
    ;

/**I found this char range in JavaCC's grammar, but Letter and Digit
overlap.
  Still works, but...
*/
fragment
Letter
    :  '\u0024' |
       '\u0041'..'\u005a' |
       '\u005f' |
       '\u0061'..'\u007a' |
       '\u00c0'..'\u00d6' |
       '\u00d8'..'\u00f6' |
       '\u00f8'..'\u00ff' |
       '\u0100'..'\u1fff' |
       '\u3040'..'\u318f' |
       '\u3300'..'\u337f' |
       '\u3400'..'\u3d2d' |
       '\u4e00'..'\u9fff' |
       '\uf900'..'\ufaff'
    ;

fragment
JavaIDDigit
    :  '\u0030'..'\u0039' |
       '\u0660'..'\u0669' |
       '\u06f0'..'\u06f9' |
```

```
      '\u0966'..'\u096f' |
      '\u09e6'..'\u09ef' |
      '\u0a66'..'\u0a6f' |
      '\u0ae6'..'\u0aef' |
      '\u0b66'..'\u0b6f' |
      '\u0be7'..'\u0bef' |
      '\u0c66'..'\u0c6f' |
      '\u0ce6'..'\u0cef' |
      '\u0d66'..'\u0d6f' |
      '\u0e50'..'\u0e59' |
      '\u0ed0'..'\u0ed9' |
      '\u1040'..'\u1049'
  ;

WS  :  (' '|'\r'|'\t'|'\u000C'|'\n') {$channel=HIDDEN;}
   ;

COMMENT
   :   '/*' ( options {greedy=false;} : . )* '*/' {$channel=HIDDEN;}
   ;

LINE_COMMENT
   : '//' ~('\n'|'\r')* '\r'? '\n' {$channel=HIDDEN;}
   ;
```