Program -> (Class)*
      Program.classes = concatAll(Class$_i$.node)

Class -> **class IDENT {** (Field | Method | MainMethod)* **}**
      Class.node = `new Class(`
            name = **IDENT**.lexval,
            fields = `concatAll(`Field$_i$.node),
            methods = `concatAll(`Method$_i$.node),
            mainMethods = `concatAll(`MainMethod$_i$.node)
)

Field -> **public** Type **IDENT ;**
      Field.node = `new Field(`type=Type.node, name=**IDENT**.lexval)

Method -> **public** Type **IDENT (** Parameters **)** Block
      Method.node = `new Method(`
            retType = Type.node,
            name = **IDENT**.lexval,
            params = Parameters.node,
            block = Block.node
)

MainMethod -> **public static void IDENT$_1$ (String [ ] IDENT$_2$)** Block
      MainMethod.node = `new MainMethod(`
            name = **IDENT$_1$**.lexval,
            argName = **IDENT$_2$**.lexval,
            block = Block.node
)

Parameters -> (Parameter$_1$ (**,** Parameter$_n$)*)?
      Parameters.node = `new Parameters(concatAll(`Parameter$_i$.node))

Parameter -> Type **IDENT**
      Parameter.node = `new Parameter(`type = Type.node, name = **IDENT**.lexval)

Type -> BasicType
      Type.node = BasicType.node

Type -> BasicType (**[ ]**)*
      Type.node = `new ArrayType(`basicType = BasicType.node, dimension = `count(`**[ ]**))

BasicType -> **int** | **boolean** | **void**
      BasicType.node = `new PrimitiveType(`**int** | **boolean** | **void**)

BasicType -> **IDENT**
      BasicType.node = `new ClassType(`**IDENT**.lexval)

Statement -> Block
      Statement.node = Block.node

Statement -> EmptyStatement
      Statement.node = `null`

Statement -> IfStatement
      Statement.node = IfStatement.node

Statement -> ExpressionStatement
      Statement.node = ExpressionStatement.node

Statement -> WhileStatement
    Statement.node = WhileStatement.node
Statement -> ReturnStatement
    Statement.node = ReturnStatement.node
Block -> **{** BlockStatement* **}**
    Block.node = concatAll(BlockStatement_i.node)
BlockStatement -> Statement
    BlockStatement.node = Statement.node
BlockStatement -> LocalVariableDeclaration
    BlockStatement.node = LocalVariableDeclaration.node

IfStatement -> **if (** Expression **)** Statement
    IfStatement.node = new IfStatement(
        cond=Expression.node,
        then=Statement.node,
        else=null
)
IfStatement -> **if (** Expression **)** Statement **else** Statement
    IfStatement.node = new IfStatement(
        cond=Expression.node,
        then=Statement.node,
        else=Statement.node
)
WhileStatement -> **while (** Expression **)** Statement
    WhileStatement.node = new WhileStatement(
        cond=Expression.node,
        expr=Statement.node
)
ExpressionStatement -> Expression **;**
    ExpressionStatement.node ->Expression.node

ReturnStatement -> **return** Expression **;**
    ReturnStatement.node = new ReturnStatement(expr=Expression.node)
ReturnStatement -> **return ;**
    ReturnStatement.node = new ReturnStatement(expr=null)

    Expression.node = ArrayAccess.node

Expression -> BinaryExpression
    Expression.node = BinaryExpression.node

BinaryExpression -> UnaryExpression
    BinaryExpression.node = UnaryExpression.node
// **BINOP** $\in$ (**=** | **||** | **&&** | **==** | **!=** | **<** | **<=** | **>** | **>=** | **+** | **-** | ***** | **/** | **%**)
BinaryExpression -> BinaryExpression_1 **BINOP** BinaryExpression_2
    BinaryExpression.node = new BinaryExpression(

```
                lhs = BinaryExpression₁.node
                operation = BINOP.lexval
                rhs = BinaryExpression₂.node
        )
```

// **UNOP** ∈ (**-** | **!**)
UnaryExpression₁ -> **UNOP** UnaryExpression₂
  UnaryExpression₁.node = new `UnaryExpression`(
    expression = UnaryExpression₂.node
    operation = **UNOP**.lexval
)
UnaryExpression -> PostfixExpression
  UnaryExpression.node = PostfixExpression.node
PostfixExpression -> PrimaryExpression
  PostfixExpression.node = PrimaryExpression.node
PostfixExpression₁ -> PostfixExpression₂ **. IDENT**
  PostfixExpression₁.node = new `FieldAccess`(
    left = PostfixExpression₂.node,
    name = **IDENT**.lexval
)
PostfixExpression₁ -> PostfixExpression₂ **. IDENT (** Arguments **)**
  PostfixExpression₁.node = new `MethodInvokation`(
    left = PostfixExpression₂.node,
    name = **IDENT**.lexval,
    args = Arguments.node
)
PostfixExpression₁ -> PostfixExpression₂ **[** Expression **]**
  PostfixExpression₁.node = new `ArrayAccess`(
    array = PostfixExpression₂.node,
    index = Expression.node
)
Arguments -> (Expression₁ (**,** Expressionₙ)*)?
  Arguments.node = new `Arguments`(concatAll(Expressionᵢ.node))

PrimaryExpression -> NewArrayExpression
  PrimaryExpression.node = NewArrayExpression.node
PrimaryExpression -> NewObjectExpression
  PrimaryExpression.node = NewObjectExpression.node
PrimaryExpression -> **INTEGER_LITERAL**
  PrimaryExpression.node = new `IntLiteral`(value = **INTEGER_LITERAL**.lexval)
PrimaryExpression -> **true**
  PrimaryExpression.node = new `BoolLiteral`(value = `true`)
PrimaryExpression -> **false**
  PrimaryExpression.node = new `BoolLiteral`(value = `false`)

PrimaryExpression -> **null**
      NullLiteral.node = `new NullLiteral()`
PrimaryExpression -> **this**
      ThisLiteral.node = `new ThisLiteral()`
PrimaryExpression -> **IDENT**
      PrimaryExpression.node = `new VarRef(name=`**IDENT**.lexval`)`
NewArrayExpression -> **new** BasicType **[** Expression **]** (**[ ]**)*
      NewArrayExpression.node = `new NewArrayExpression(`
            arrayType = `new ArrayType(`
                  basicType = BasicType.node,
                  dimension = `count(`**[ ]**`) + 1`
            `)`,
            size = Expression.node
      `)`
NewObjectExpression -> **new IDENT ( )**
      NewObjectExpression.node = `new NewObjectExpression(`name = **IDENT**.lexval`)`