Paul Kummer

Dr. Hanku Lee

Compilers CSIS 455-01

11/10/2021

Lab 09: Adding Loops and Conditions

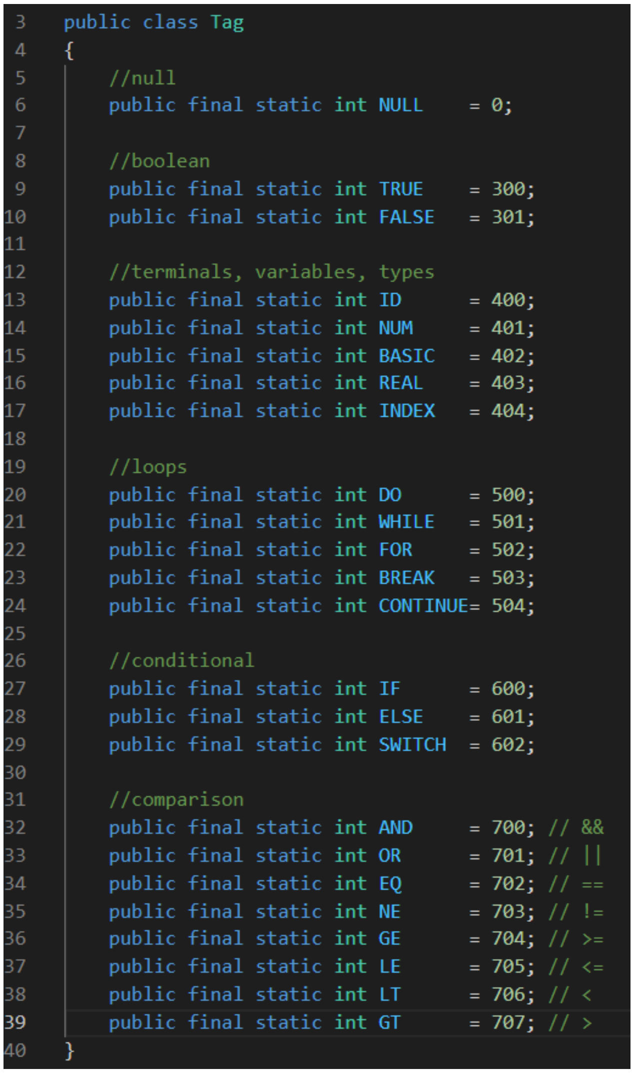
Purpose of Lab

This lab's purpose is to expand on the last version of the compiler. In this version, more operations are added like While Loops, Do While Loops, and the If clause. Also, there is added support for conditional operations.

Example Code

The lexer phase has been modified to include more keywords that will be used in future development. Now the literal to token is converted to either a Num or Real token, which will add support for floats. Furthermore, the token Type is added, which extends Word.

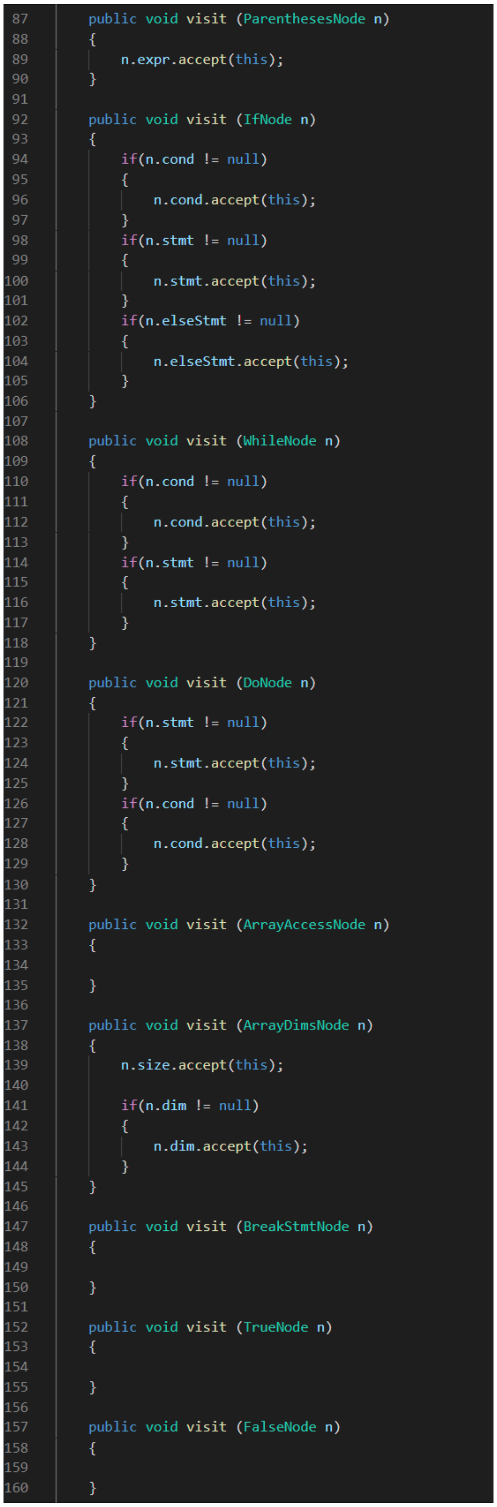
tag.java



```java
public class Tag
{
    //null
    public final static int NULL    = 0;

    //boolean
    public final static int TRUE    = 300;
    public final static int FALSE   = 301;

    //terminals, variables, types
    public final static int ID      = 400;
    public final static int NUM     = 401;
    public final static int BASIC   = 402;
    public final static int REAL    = 403;
    public final static int INDEX   = 404;

    //loops
    public final static int DO      = 500;
    public final static int WHILE   = 501;
    public final static int FOR     = 502;
    public final static int BREAK   = 503;
    public final static int CONTINUE= 504;

    //conditional
    public final static int IF      = 600;
    public final static int ELSE    = 601;
    public final static int SWITCH  = 602;

    //comparison
    public final static int AND     = 700; // &&
    public final static int OR      = 701; // ||
    public final static int EQ      = 702; // ==
    public final static int NE      = 703; // !=
    public final static int GE      = 704; // >=
    public final static int LE      = 705; // <=
    public final static int LT      = 706; // <
    public final static int GT      = 707; // >
}
```

More values were added to for detecting lexemes. Also, they were added as reserved Words in the lexer’s symbol table.

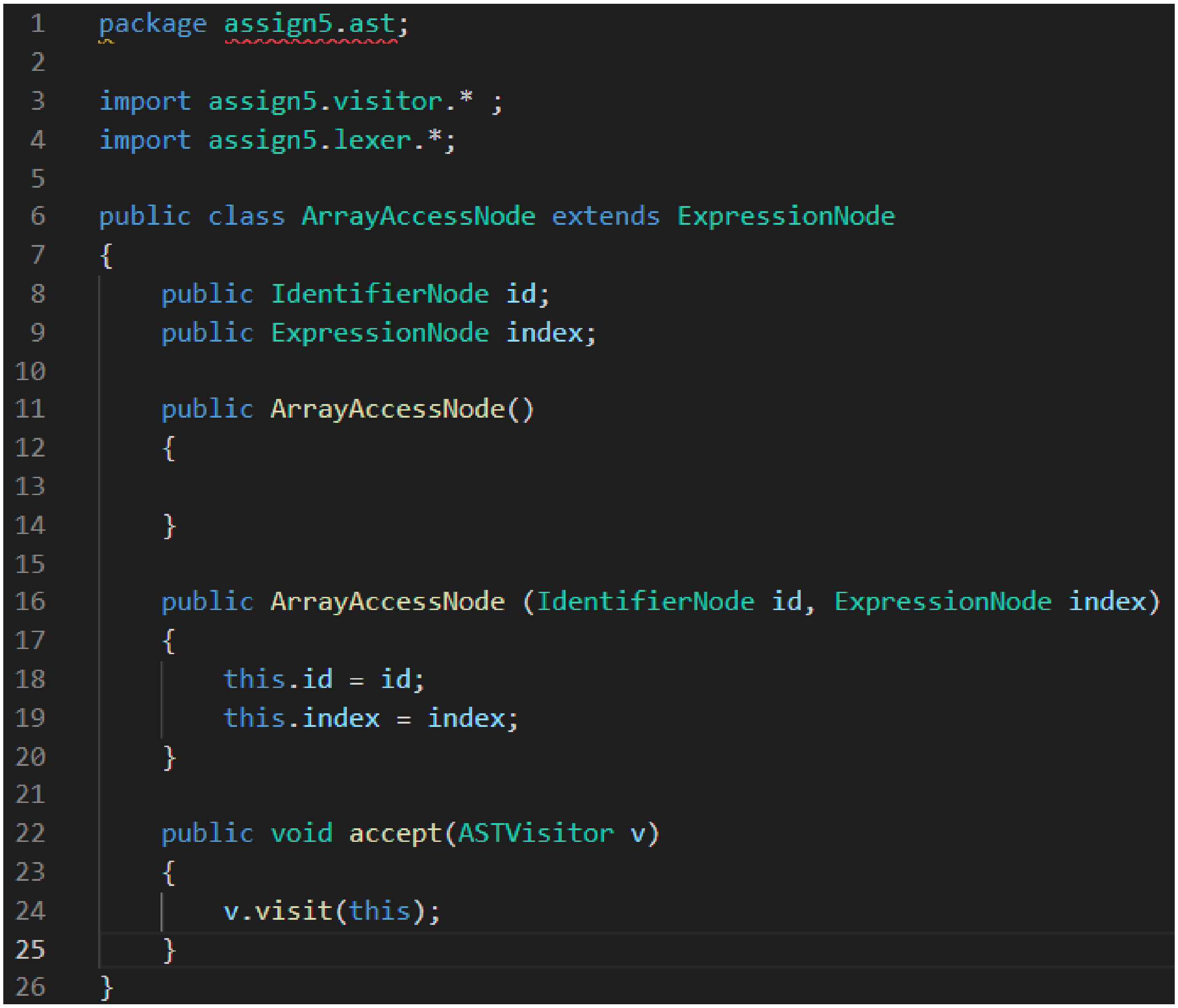
ASTVisitor.java



```
    public void visit (ParenthesesNode n)
    {
        n.expr.accept(this);
    }

    public void visit (IfNode n)
    {
        if(n.cond != null)
        {
            n.cond.accept(this);
        }
        if(n.stmt != null)
        {
            n.stmt.accept(this);
        }
        if(n.elseStmt != null)
        {
            n.elseStmt.accept(this);
        }
    }

    public void visit (WhileNode n)
    {
        if(n.cond != null)
        {
            n.cond.accept(this);
        }
        if(n.stmt != null)
        {
            n.stmt.accept(this);
        }
    }

    public void visit (DoNode n)
    {
        if(n.stmt != null)
        {
            n.stmt.accept(this);
        }
        if(n.cond != null)
        {
            n.cond.accept(this);
        }
    }

    public void visit (ArrayAccessNode n)
    {

    }

    public void visit (ArrayDimsNode n)
    {
        n.size.accept(this);

        if(n.dim != null)
        {
            n.dim.accept(this);
        }
    }

    public void visit (BreakStmtNode n)
    {

    }

    public void visit (TrueNode n)
    {

    }

    public void visit (FalseNode n)
    {

    }
```

The visitor now has the added nodes listed and functional.

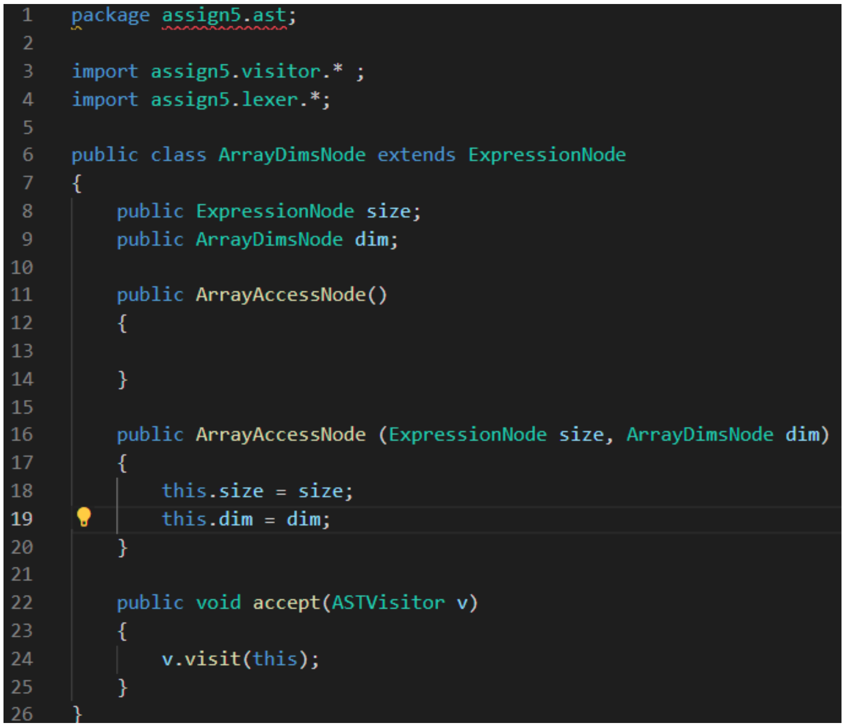
ArrayAccessNode.java.



```
package assign5.ast;

import assign5.visitor.* ;
import assign5.lexer.*;

public class ArrayAccessNode extends ExpressionNode
{
    public IdentifierNode id;
    public ExpressionNode index;

    public ArrayAccessNode()
    {

    }

    public ArrayAccessNode (IdentifierNode id, ExpressionNode index)
    {
        this.id = id;
        this.index = index;
    }

    public void accept(ASTVisitor v)
    {
        v.visit(this);
    }
}
```

Newly added node for accessing an array.

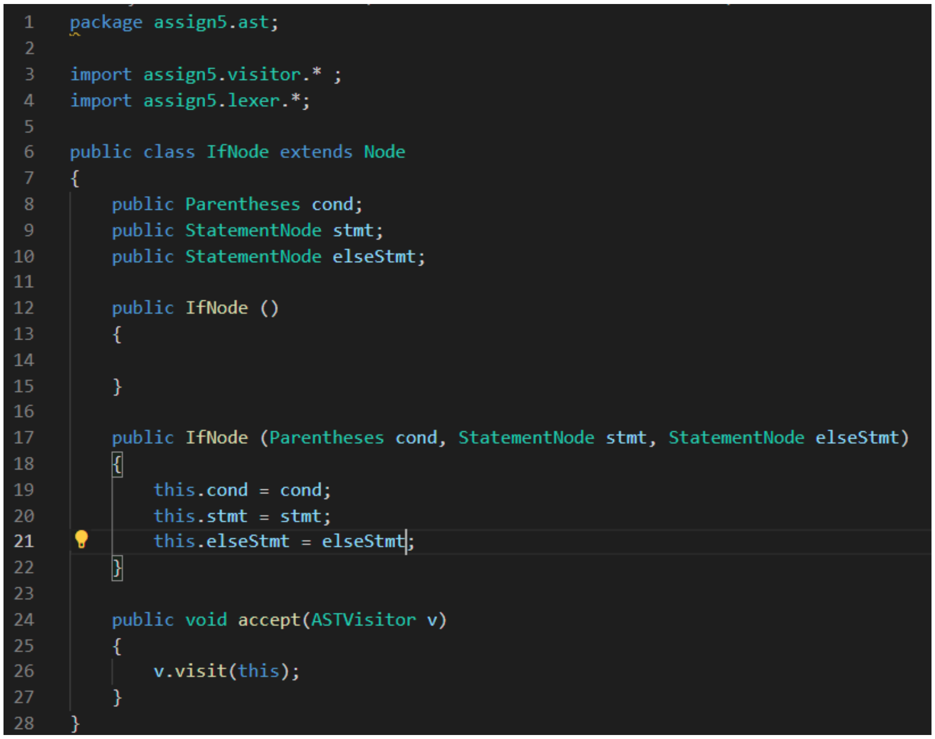
ArrayDimsNode.java



```java
package assign5.ast;

import assign5.visitor.* ;
import assign5.lexer.*;

public class ArrayDimsNode extends ExpressionNode
{
    public ExpressionNode size;
    public ArrayDimsNode dim;

    public ArrayAccessNode()
    {

    }

    public ArrayAccessNode (ExpressionNode size, ArrayDimsNode dim)
    {
        this.size = size;
        this.dim = dim;
    }

    public void accept(ASTVisitor v)
    {
        v.visit(this);
    }
}
```

Node used for traversing within the dimensions of an array.

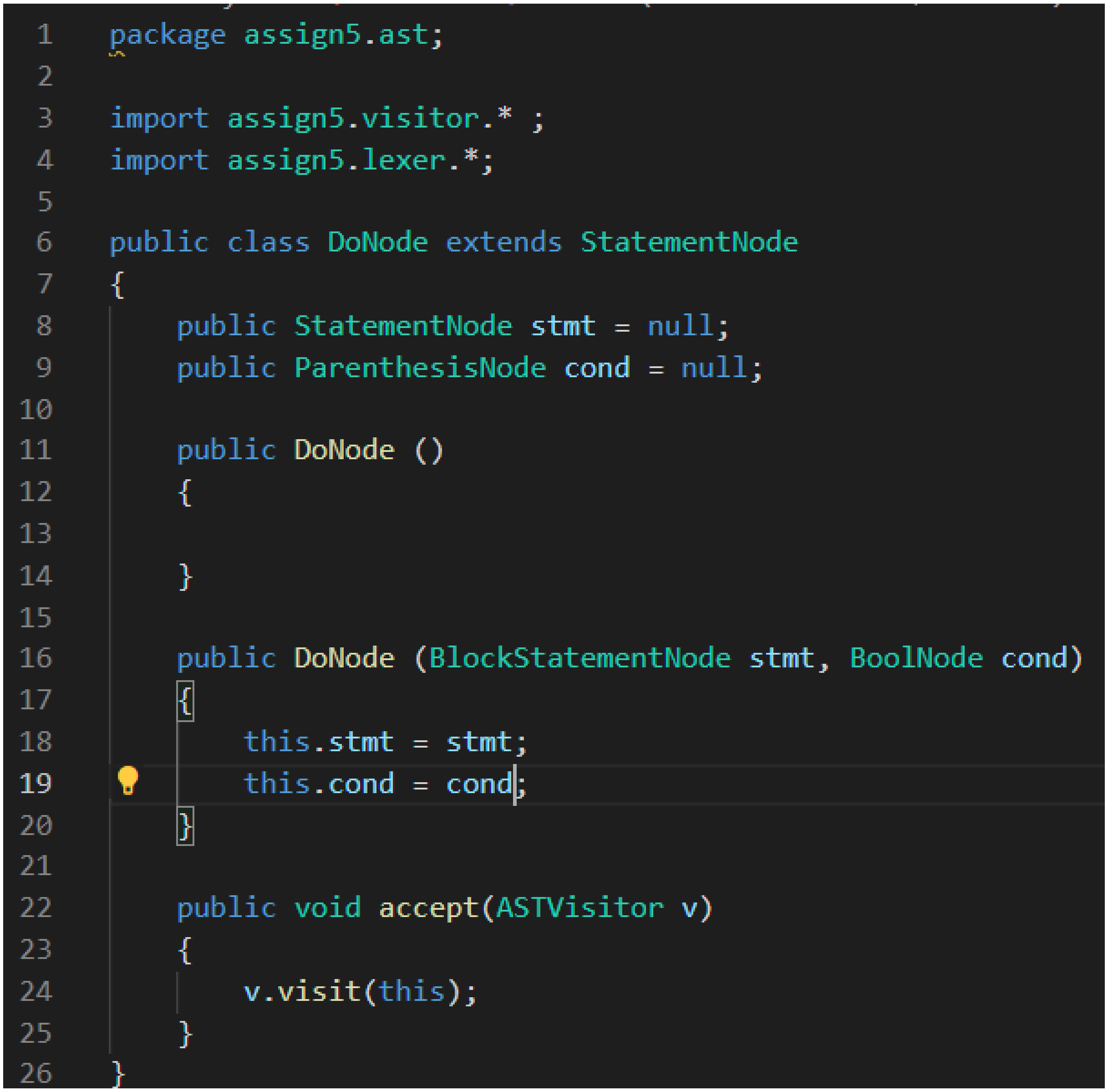
IfNode.java



```java
package assign5.ast;

import assign5.visitor.* ;
import assign5.lexer.*;

public class IfNode extends Node
{
    public Parentheses cond;
    public StatementNode stmt;
    public StatementNode elseStmt;

    public IfNode ()
    {

    }

    public IfNode (Parentheses cond, StatementNode stmt, StatementNode elseStmt)
    {
        this.cond = cond;
        this.stmt = stmt;
        this.elseStmt = elseStmt;
    }

    public void accept(ASTVisitor v)
    {
        v.visit(this);
    }
}
```

The if node is used for conditional execution of a block statement when the condition is true.

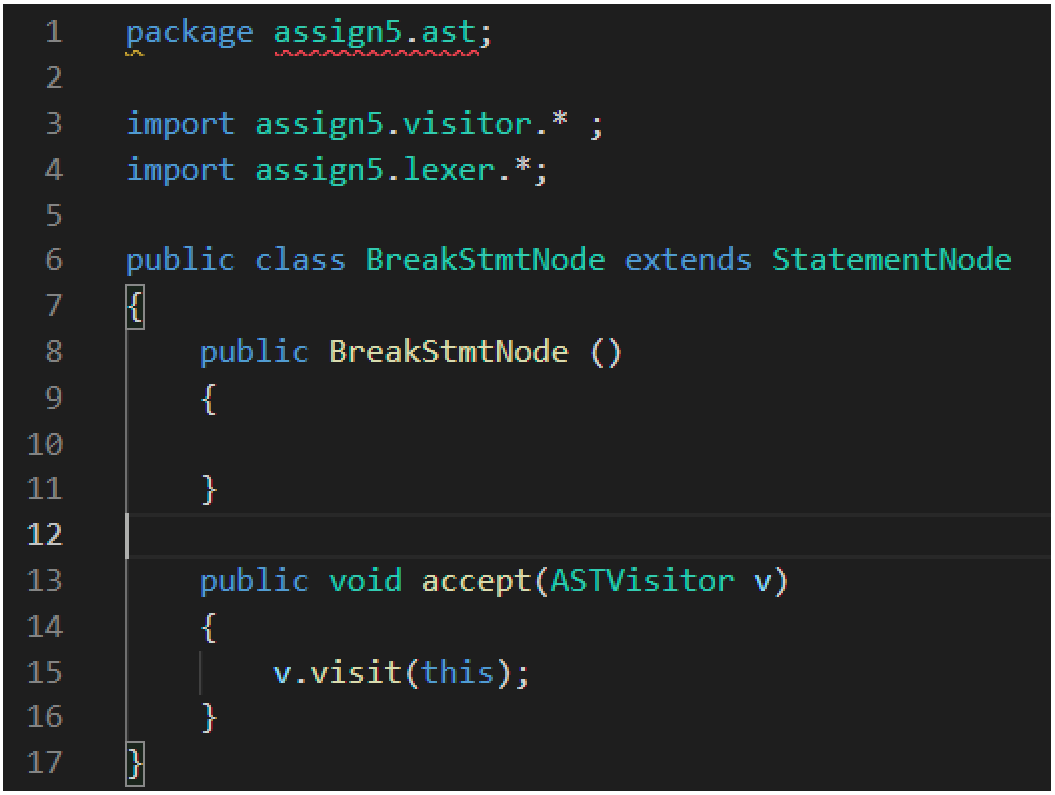
DoNode.java



```
package assign5.ast;

import assign5.visitor.* ;
import assign5.lexer.*;

public class DoNode extends StatementNode
{
    public StatementNode stmt = null;
    public ParenthesisNode cond = null;

    public DoNode ()
    {

    }

    public DoNode (BlockStatementNode stmt, BoolNode cond)
    {
        this.stmt = stmt;
        this.cond = cond;
    }

    public void accept(ASTVisitor v)
    {
        v.visit(this);
    }
}
```

The do node will first execute a block statement and continue executing the block statement as long as the condition is true.

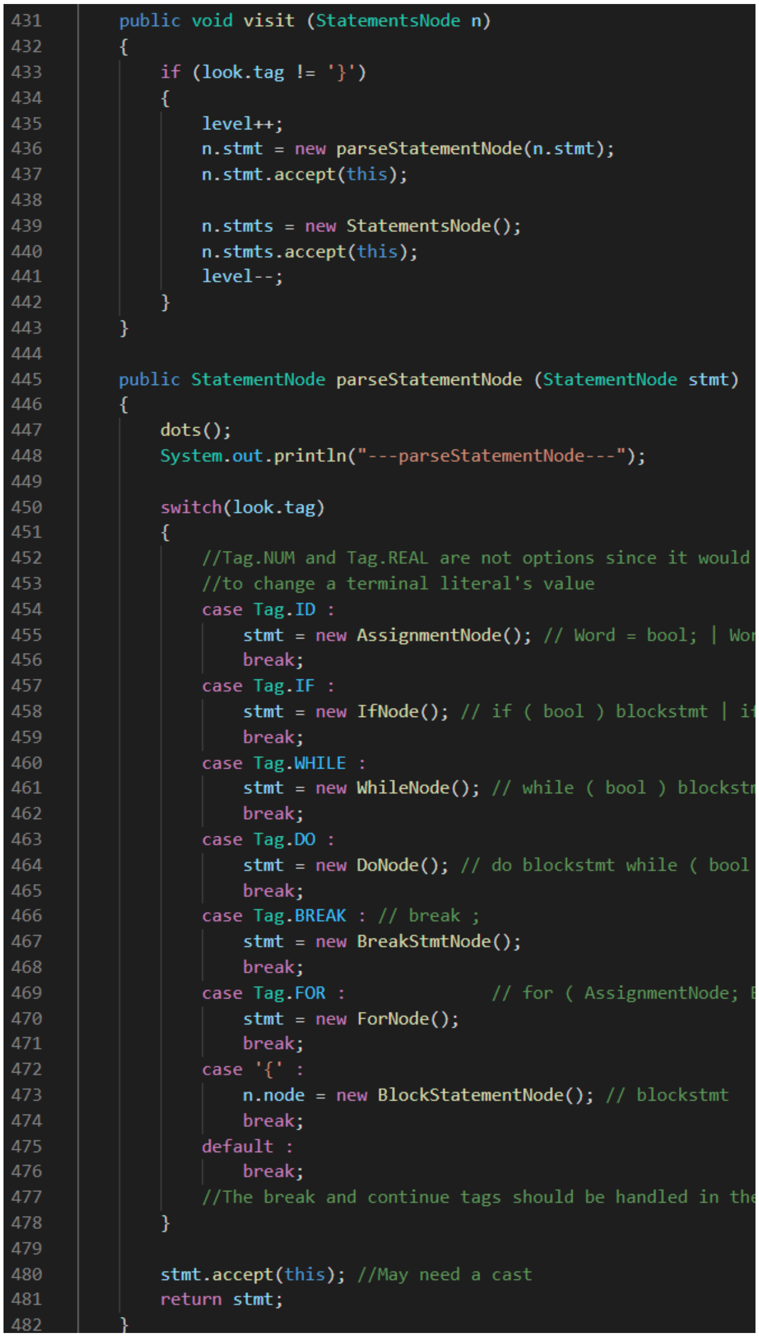
BreakStmtNode.java



```java
package assign5.ast;

import assign5.visitor.* ;
import assign5.lexer.*;

public class BreakStmtNode extends StatementNode
{
    public BreakStmtNode ()
    {

    }

    public void accept(ASTVisitor v)
    {
        v.visit(this);
    }
}
```

Used to exit a statement early.

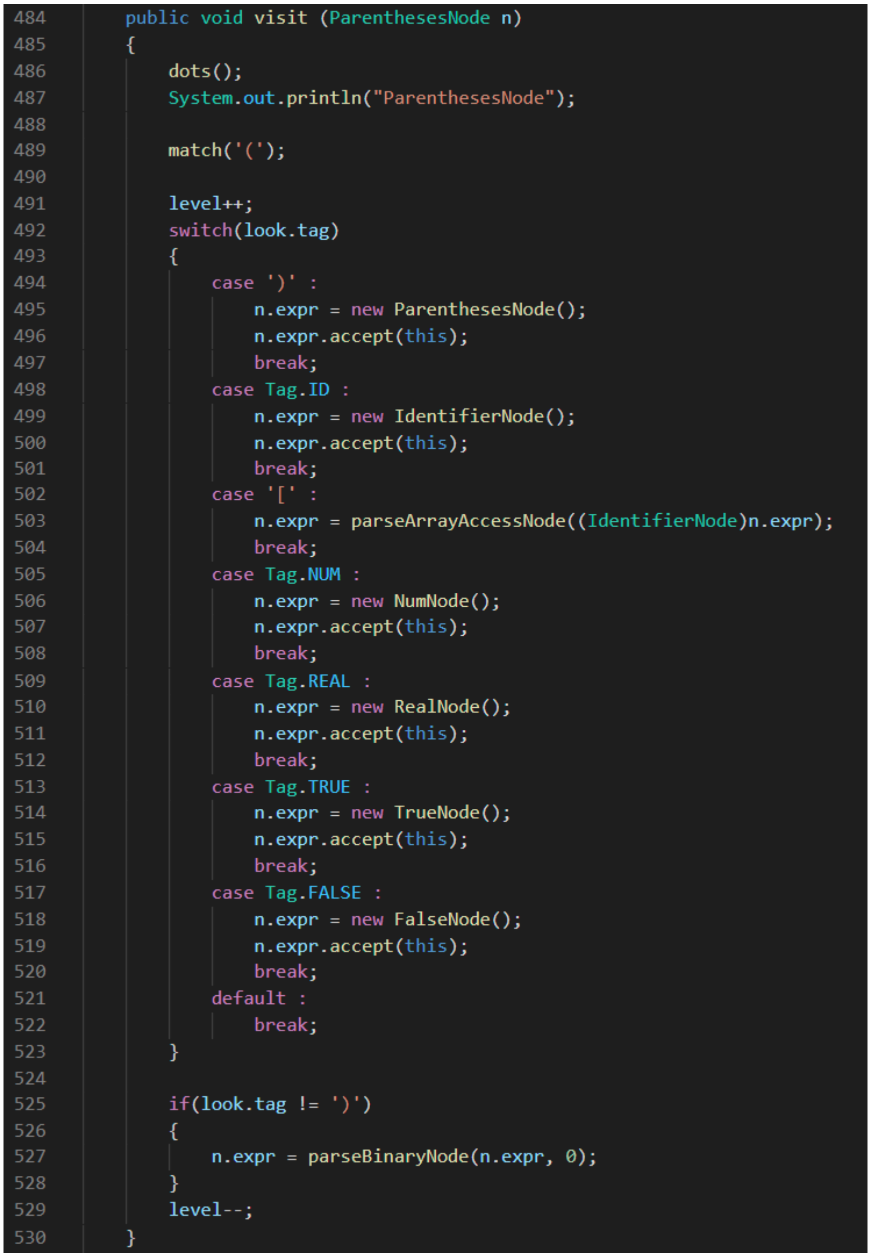
Parser.java: StatementsNode and parseStatementsNode



```java
    public void visit (StatementsNode n)
    {
        if (look.tag != '}')
        {
            level++;
            n.stmt = new parseStatementNode(n.stmt);
            n.stmt.accept(this);

            n.stmts = new StatementsNode();
            n.stmts.accept(this);
            level--;
        }
    }

    public StatementNode parseStatementNode (StatementNode stmt)
    {
        dots();
        System.out.println("---parseStatementNode---");

        switch(look.tag)
        {
            //Tag.NUM and Tag.REAL are not options since it would
            //to change a terminal literal's value
            case Tag.ID :
                stmt = new AssignmentNode(); // Word = bool; | Wor
                break;
            case Tag.IF :
                stmt = new IfNode(); // if ( bool ) blockstmt | i
                break;
            case Tag.WHILE :
                stmt = new WhileNode(); // while ( bool ) blockstm
                break;
            case Tag.DO :
                stmt = new DoNode(); // do blockstmt while ( bool
                break;
            case Tag.BREAK : // break ;
                stmt = new BreakStmtNode();
                break;
            case Tag.FOR :              // for ( AssignmentNode; 
                stmt = new ForNode();
                break;
            case '{' :
                n.node = new BlockStatementNode(); // blockstmt
                break;
            default :
                break;
            //The break and continue tags should be handled in the
        }

        stmt.accept(this); //May need a cast
        return stmt;
    }
```

The new parseStatementNode now determines what type of node to branch too when a statement is executed.

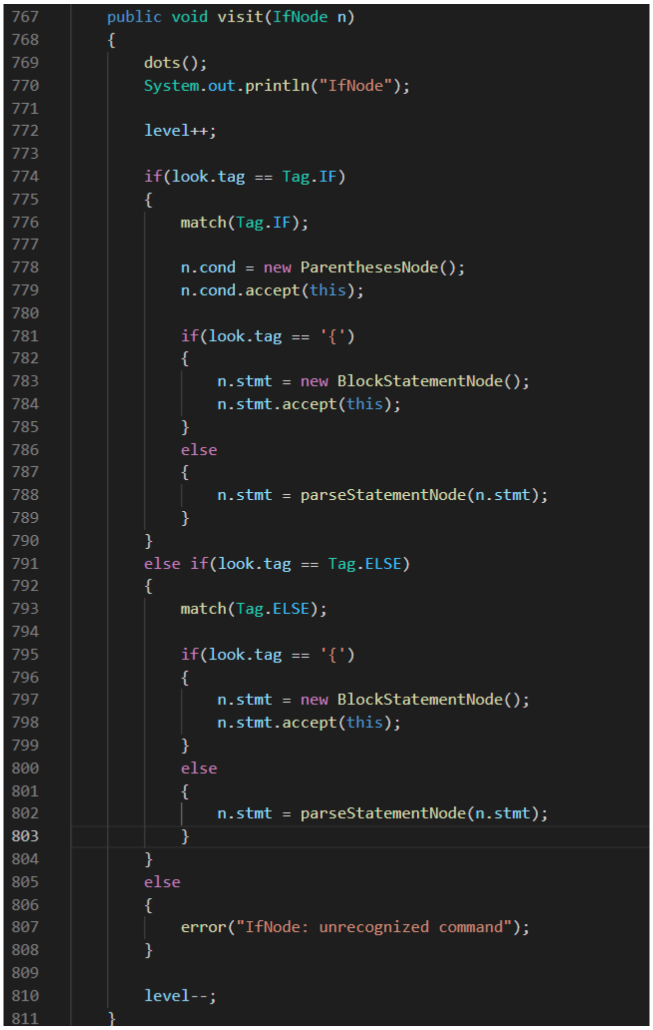
parser.java: ParenthesesNode



```java
public void visit (ParenthesesNode n)
{
    dots();
    System.out.println("ParenthesesNode");

    match('(');

    level++;
    switch(look.tag)
    {
        case ')' :
            n.expr = new ParenthesesNode();
            n.expr.accept(this);
            break;
        case Tag.ID :
            n.expr = new IdentifierNode();
            n.expr.accept(this);
            break;
        case '[' :
            n.expr = parseArrayAccessNode((IdentifierNode)n.expr);
            break;
        case Tag.NUM :
            n.expr = new NumNode();
            n.expr.accept(this);
            break;
        case Tag.REAL :
            n.expr = new RealNode();
            n.expr.accept(this);
            break;
        case Tag.TRUE :
            n.expr = new TrueNode();
            n.expr.accept(this);
            break;
        case Tag.FALSE :
            n.expr = new FalseNode();
            n.expr.accept(this);
            break;
        default :
            break;
    }

    if(look.tag != ')')
    {
        n.expr = parseBinaryNode(n.expr, 0);
    }
    level--;
}
```

This node will execute whatever is in a parenthesized statement.

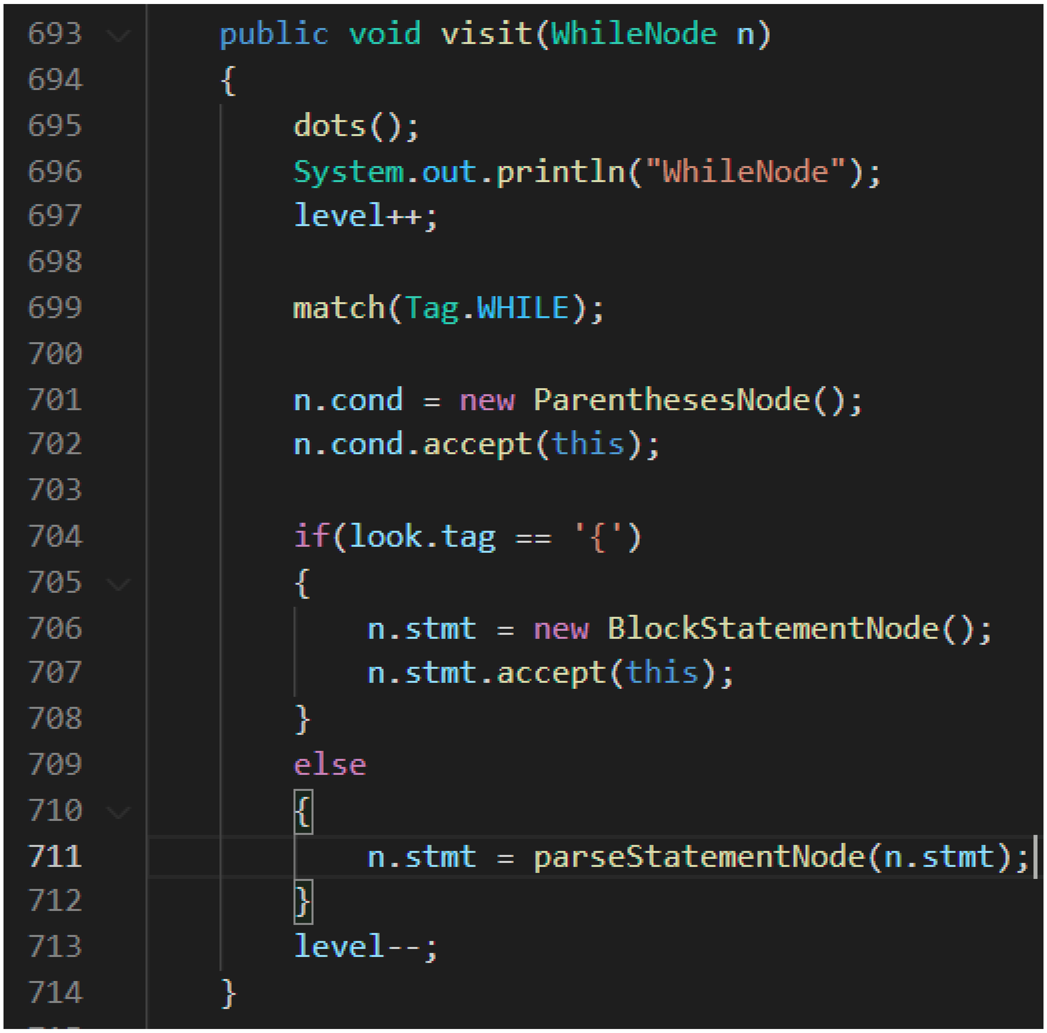
Parser.java: IfNode



```java
    public void visit(IfNode n)
    {
        dots();
        System.out.println("IfNode");

        level++;

        if(look.tag == Tag.IF)
        {
            match(Tag.IF);

            n.cond = new ParenthesesNode();
            n.cond.accept(this);

            if(look.tag == '{')
            {
                n.stmt = new BlockStatementNode();
                n.stmt.accept(this);
            }
            else
            {
                n.stmt = parseStatementNode(n.stmt);
            }
        }
        else if(look.tag == Tag.ELSE)
        {
            match(Tag.ELSE);

            if(look.tag == '{')
            {
                n.stmt = new BlockStatementNode();
                n.stmt.accept(this);
            }
            else
            {
                n.stmt = parseStatementNode(n.stmt);
            }
        }
        else
        {
            error("IfNode: unrecognized command");
        }

        level--;
    }
```

The if node will execute possibly one of many block statements if one of the conditions is true, or a default else statement is encountered.

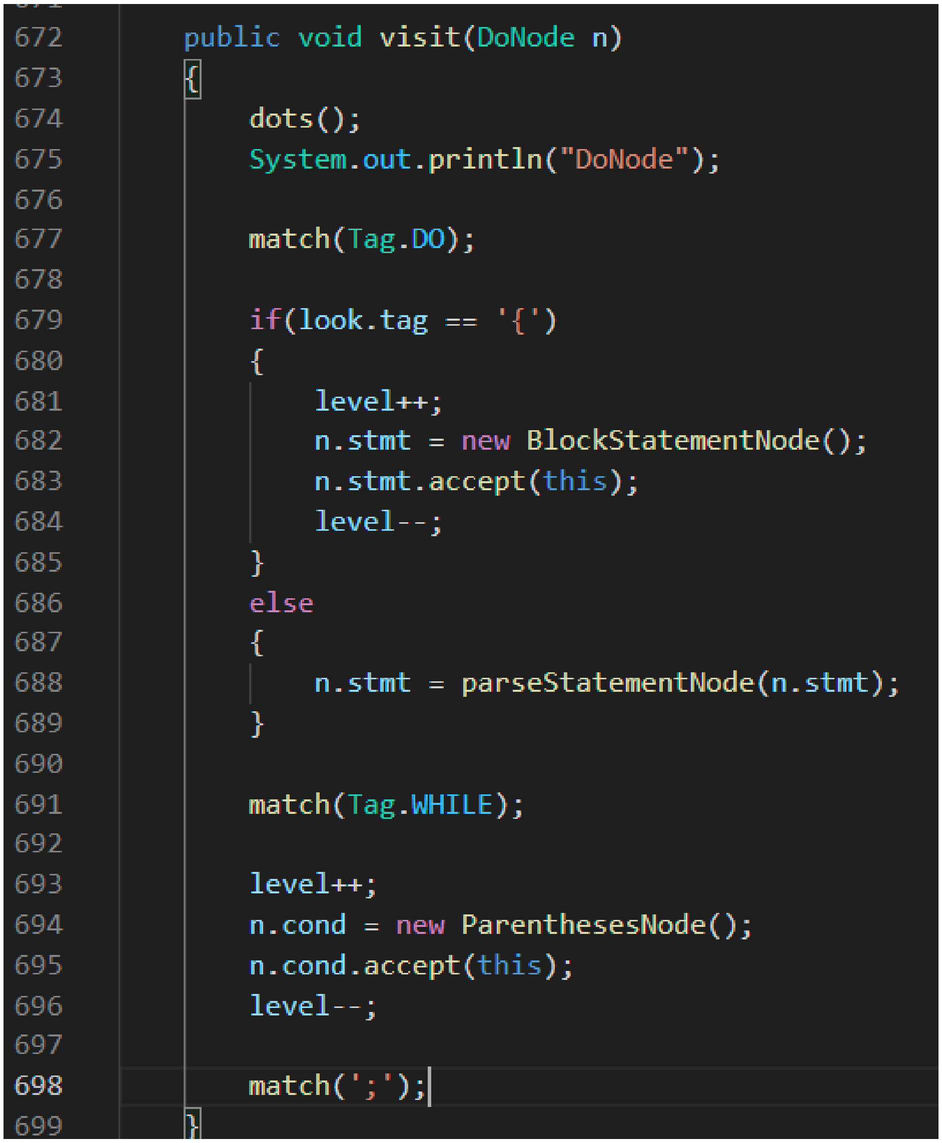
Parser.java: WhileNode



```
    public void visit(WhileNode n)
    {
        dots();
        System.out.println("WhileNode");
        level++;

        match(Tag.WHILE);

        n.cond = new ParenthesesNode();
        n.cond.accept(this);

        if(look.tag == '{')
        {
            n.stmt = new BlockStatementNode();
            n.stmt.accept(this);
        }
        else
        {
            n.stmt = parseStatementNode(n.stmt);
        }
        level--;
    }
```

The while node will execute a block statement only if a condition is true.

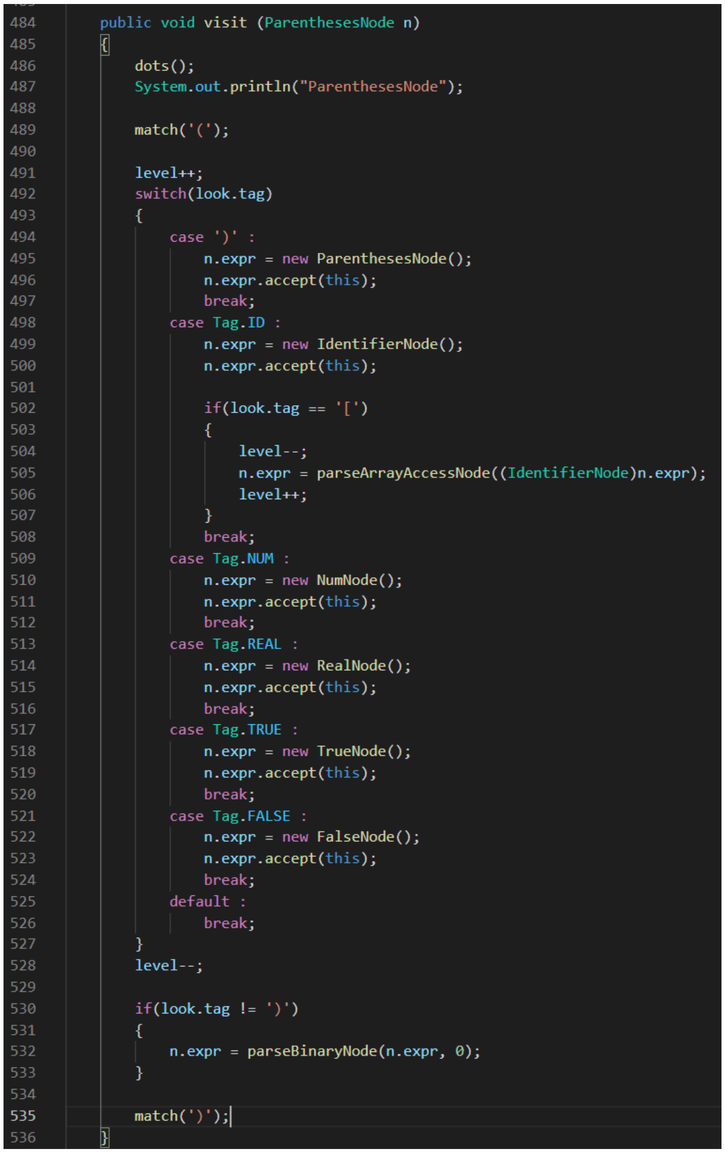
Parser.java: DoNode



```
    public void visit(DoNode n)
    {
        dots();
        System.out.println("DoNode");

        match(Tag.DO);

        if(look.tag == '{')
        {
            level++;
            n.stmt = new BlockStatementNode();
            n.stmt.accept(this);
            level--;
        }
        else
        {
            n.stmt = parseStatementNode(n.stmt);
        }

        match(Tag.WHILE);

        level++;
        n.cond = new ParenthesesNode();
        n.cond.accept(this);
        level--;

        match(';');
    }
```

The Do node is almost identical to the while node. However, it will always execute the block statement once, and continue to loop if the parenthesis statement is true.

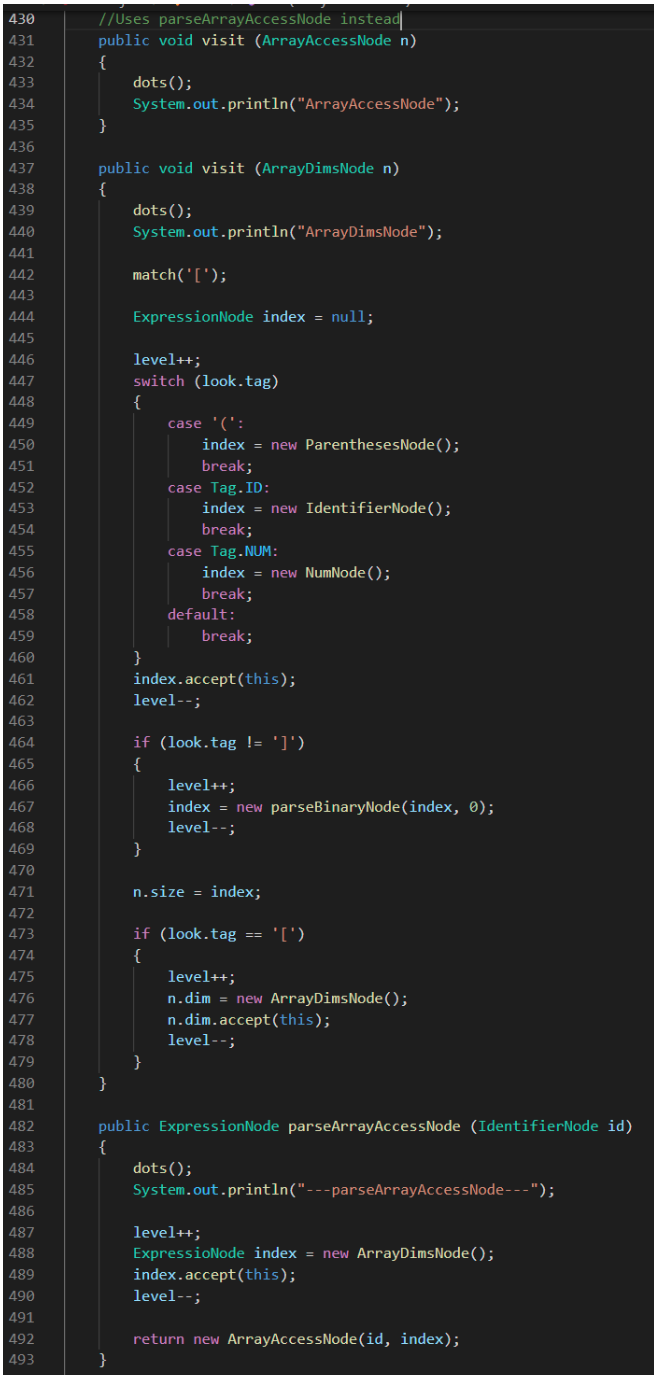
Parser.java: ParenthesesNode



```
    public void visit (ParenthesesNode n)
    {
        dots();
        System.out.println("ParenthesesNode");

        match('(');

        level++;
        switch(look.tag)
        {
            case ')' :
                n.expr = new ParenthesesNode();
                n.expr.accept(this);
                break;
            case Tag.ID :
                n.expr = new IdentifierNode();
                n.expr.accept(this);

                if(look.tag == '[')
                {
                    level--;
                    n.expr = parseArrayAccessNode((IdentifierNode)n.expr);
                    level++;
                }
                break;
            case Tag.NUM :
                n.expr = new NumNode();
                n.expr.accept(this);
                break;
            case Tag.REAL :
                n.expr = new RealNode();
                n.expr.accept(this);
                break;
            case Tag.TRUE :
                n.expr = new TrueNode();
                n.expr.accept(this);
                break;
            case Tag.FALSE :
                n.expr = new FalseNode();
                n.expr.accept(this);
                break;
            default :
                break;
        }
        level--;

        if(look.tag != ')')
        {
            n.expr = parseBinaryNode(n.expr, 0);
        }

        match(')');
    }
```

The parenthesesNode will execute whatever type of expression is within a set of parentheses. These can be nested.

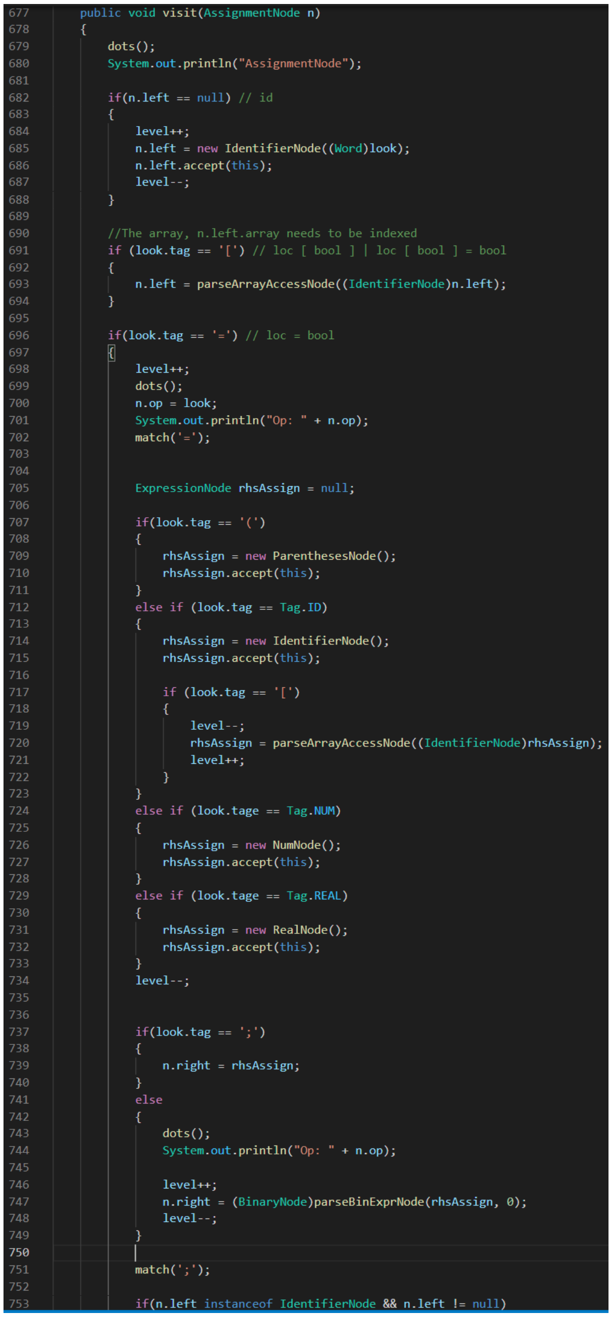
Parser.java: ArrayElements



```
    //Uses parseArrayAccessNode instead
    public void visit (ArrayAccessNode n)
    {
        dots();
        System.out.println("ArrayAccessNode");
    }

    public void visit (ArrayDimsNode n)
    {
        dots();
        System.out.println("ArrayDimsNode");

        match('[');

        ExpressionNode index = null;

        level++;
        switch (look.tag)
        {
            case '(':
                index = new ParenthesesNode();
                break;
            case Tag.ID:
                index = new IdentifierNode();
                break;
            case Tag.NUM:
                index = new NumNode();
                break;
            default:
                break;
        }
        index.accept(this);
        level--;

        if (look.tag != ']')
        {
            level++;
            index = new parseBinaryNode(index, 0);
            level--;
        }

        n.size = index;

        if (look.tag == '[')
        {
            level++;
            n.dim = new ArrayDimsNode();
            n.dim.accept(this);
            level--;
        }
    }

    public ExpressionNode parseArrayAccessNode (IdentifierNode id)
    {
        dots();
        System.out.println("---parseArrayAccessNode---");

        level++;
        ExpressioNode index = new ArrayDimsNode();
        index.accept(this);
        level--;

        return new ArrayAccessNode(id, index);
    }
```

This addes the functionality of making multidimensional arrays and creating a way of accessing each element in the array without using a stack.

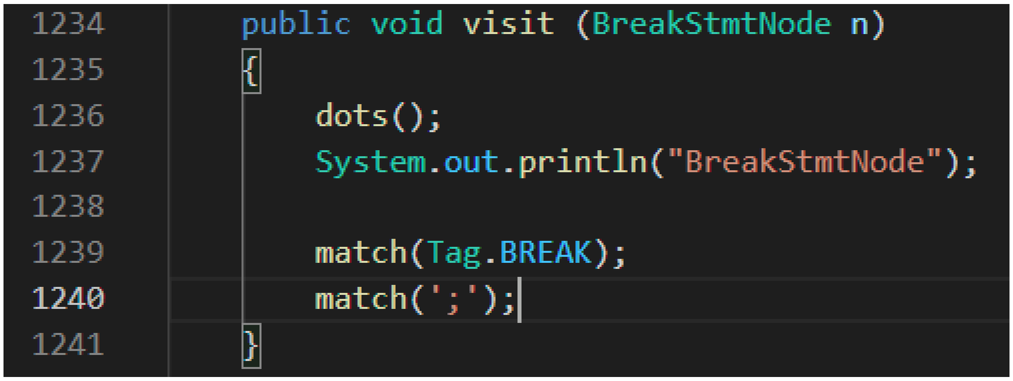
Parser.java: AssignmentNode



```java
    public void visit(AssignmentNode n)
    {
        dots();
        System.out.println("AssignmentNode");

        if(n.left == null) // id
        {
            level++;
            n.left = new IdentifierNode((Word)look);
            n.left.accept(this);
            level--;
        }

        //The array, n.left.array needs to be indexed
        if (look.tag == '[') // loc [ bool ] | loc [ bool ] = bool
        {
            n.left = parseArrayAccessNode((IdentifierNode)n.left);
        }

        if(look.tag == '=') // loc = bool
        {
            level++;
            dots();
            n.op = look;
            System.out.println("Op: " + n.op);
            match('=');


            ExpressionNode rhsAssign = null;

            if(look.tag == '(')
            {
                rhsAssign = new ParenthesesNode();
                rhsAssign.accept(this);
            }
            else if (look.tag == Tag.ID)
            {
                rhsAssign = new IdentifierNode();
                rhsAssign.accept(this);

                if (look.tag == '[')
                {
                    level--;
                    rhsAssign = parseArrayAccessNode((IdentifierNode)rhsAssign);
                    level++;
                }
            }
            else if (look.tage == Tag.NUM)
            {
                rhsAssign = new NumNode();
                rhsAssign.accept(this);
            }
            else if (look.tage == Tag.REAL)
            {
                rhsAssign = new RealNode();
                rhsAssign.accept(this);
            }
            level--;


            if(look.tag == ';')
            {
                n.right = rhsAssign;
            }
            else
            {
                dots();
                System.out.println("Op: " + n.op);

                level++;
                n.right = (BinaryNode)parseBinExprNode(rhsAssign, 0);
                level--;
            }

            match(';');

            if(n.left instanceof IdentifierNode && n.left != null)
```

The assignment node will give a value of an expression node to an identifier node. The expression Node can become many different types of nodes and will commonly become a binary node with nested binary nodes.

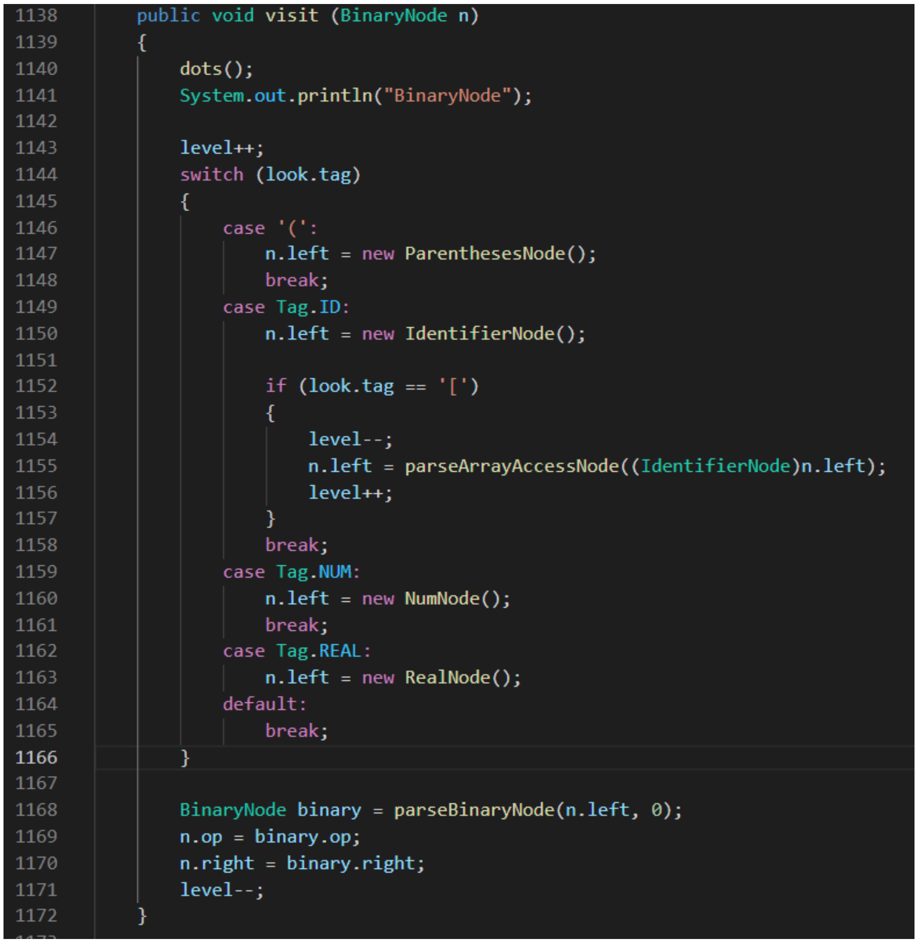
Parser.java: BreakNode



```java
    public void visit (BreakStmtNode n)
    {
        dots();
        System.out.println("BreakStmtNode");

        match(Tag.BREAK);
        match(';');
    }
```

The break node simply allows a block to stop executing immediately when it is encounted.

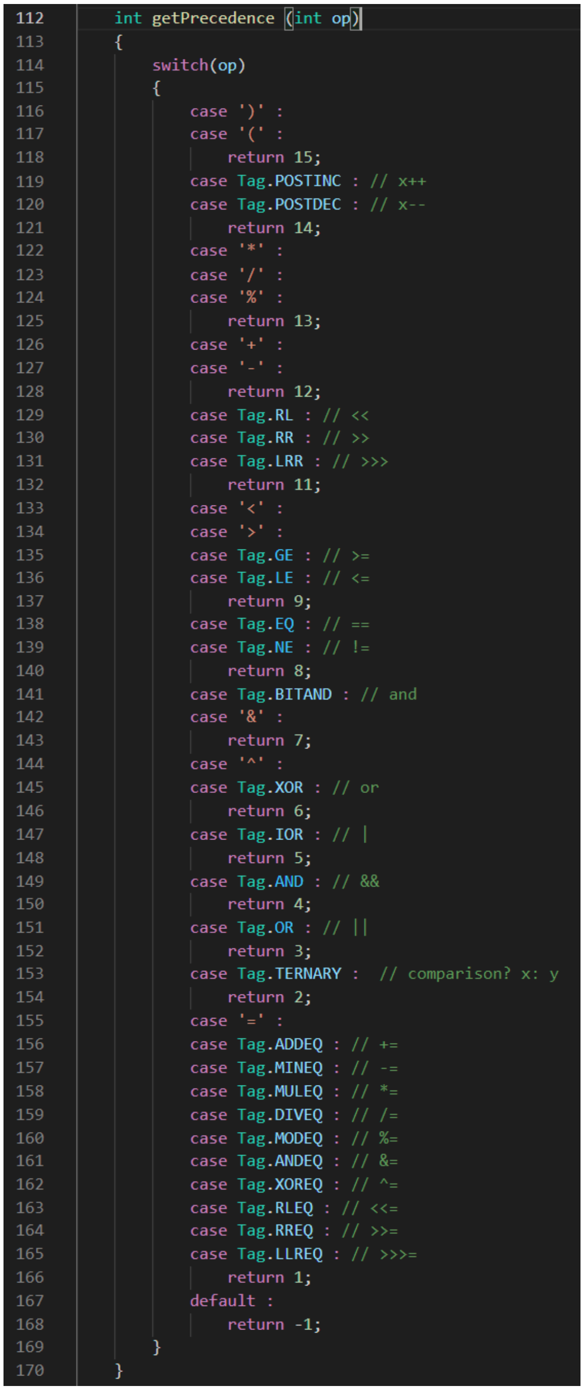
Parser.java: BinaryNode



```java
    public void visit (BinaryNode n)
    {
        dots();
        System.out.println("BinaryNode");

        level++;
        switch (look.tag)
        {
            case '(':
                n.left = new ParenthesesNode();
                break;
            case Tag.ID:
                n.left = new IdentifierNode();

                if (look.tag == '[')
                {
                    level--;
                    n.left = parseArrayAccessNode((IdentifierNode)n.left);
                    level++;
                }
                break;
            case Tag.NUM:
                n.left = new NumNode();
                break;
            case Tag.REAL:
                n.left = new RealNode();
            default:
                break;
        }

        BinaryNode binary = parseBinaryNode(n.left, 0);
        n.op = binary.op;
        n.right = binary.right;
        level--;
    }
```

The binarynode will take on a left-hand-side and a right-hand-side and have an operator between them. The parse binary node will be used to populate the binary nodes.

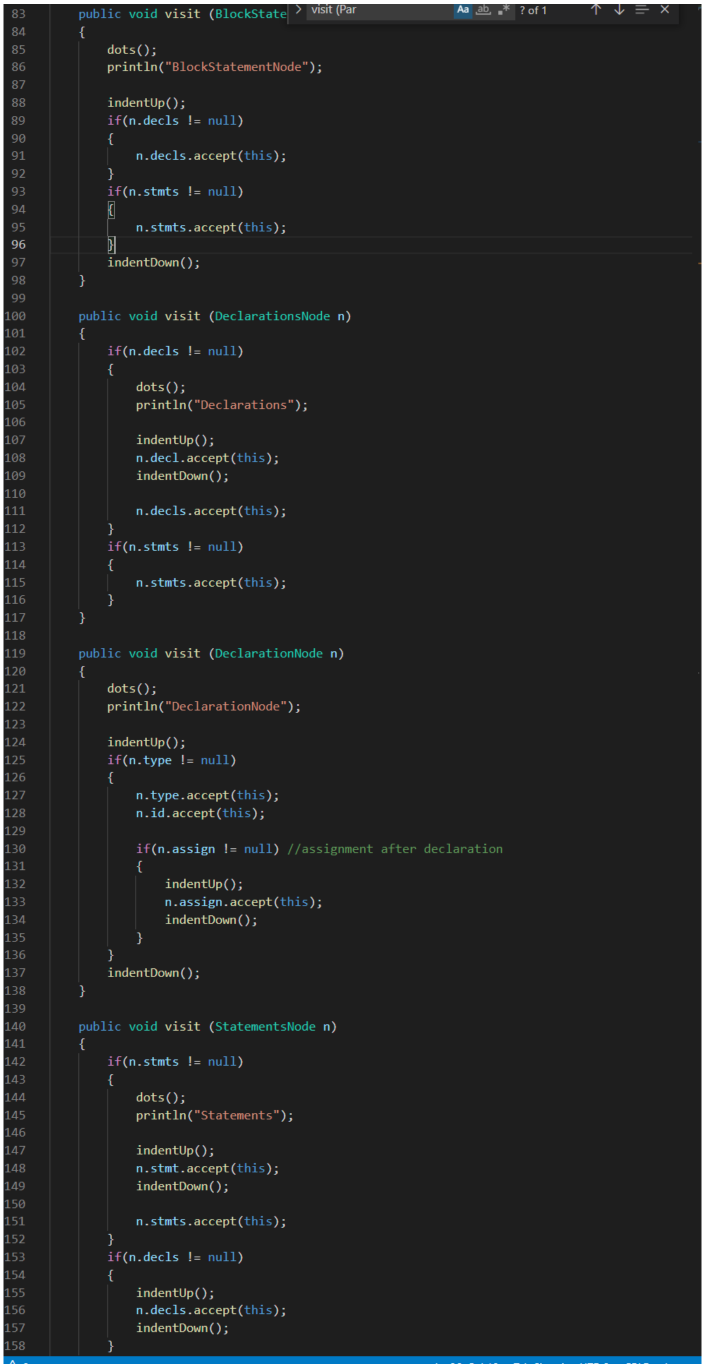
Parser.java: getPrecedence



```
int getPrecedence (int op)
{
    switch(op)
    {
        case ')' :
        case '(' :
            return 15;
        case Tag.POSTINC : // x++
        case Tag.POSTDEC : // x--
            return 14;
        case '*' :
        case '/' :
        case '%' :
            return 13;
        case '+' :
        case '-' :
            return 12;
        case Tag.RL : // <<
        case Tag.RR : // >>
        case Tag.LRR : // >>>
            return 11;
        case '<' :
        case '>' :
        case Tag.GE : // >=
        case Tag.LE : // <=
            return 9;
        case Tag.EQ : // ==
        case Tag.NE : // !=
            return 8;
        case Tag.BITAND : // and
        case '&' :
            return 7;
        case '^' :
        case Tag.XOR : // or
            return 6;
        case Tag.IOR : // |
            return 5;
        case Tag.AND : // &&
            return 4;
        case Tag.OR : // ||
            return 3;
        case Tag.TERNARY :  // comparison? x: y
            return 2;
        case '=' :
        case Tag.ADDEQ : // +=
        case Tag.MINEQ : // -=
        case Tag.MULEQ : // *=
        case Tag.DIVEQ : // /=
        case Tag.MODEQ : // %=
        case Tag.ANDEQ : // &=
        case Tag.XOREQ : // ^=
        case Tag.RLEQ : // <<=
        case Tag.RREQ : // >>=
        case Tag.LLREQ : // >>>=
            return 1;
        default :
            return -1;
    }
}
```

Get precedence has had many new operators added to it to allow the parser binary expression to correctly build an AST based on the precedence of the operator encountered.

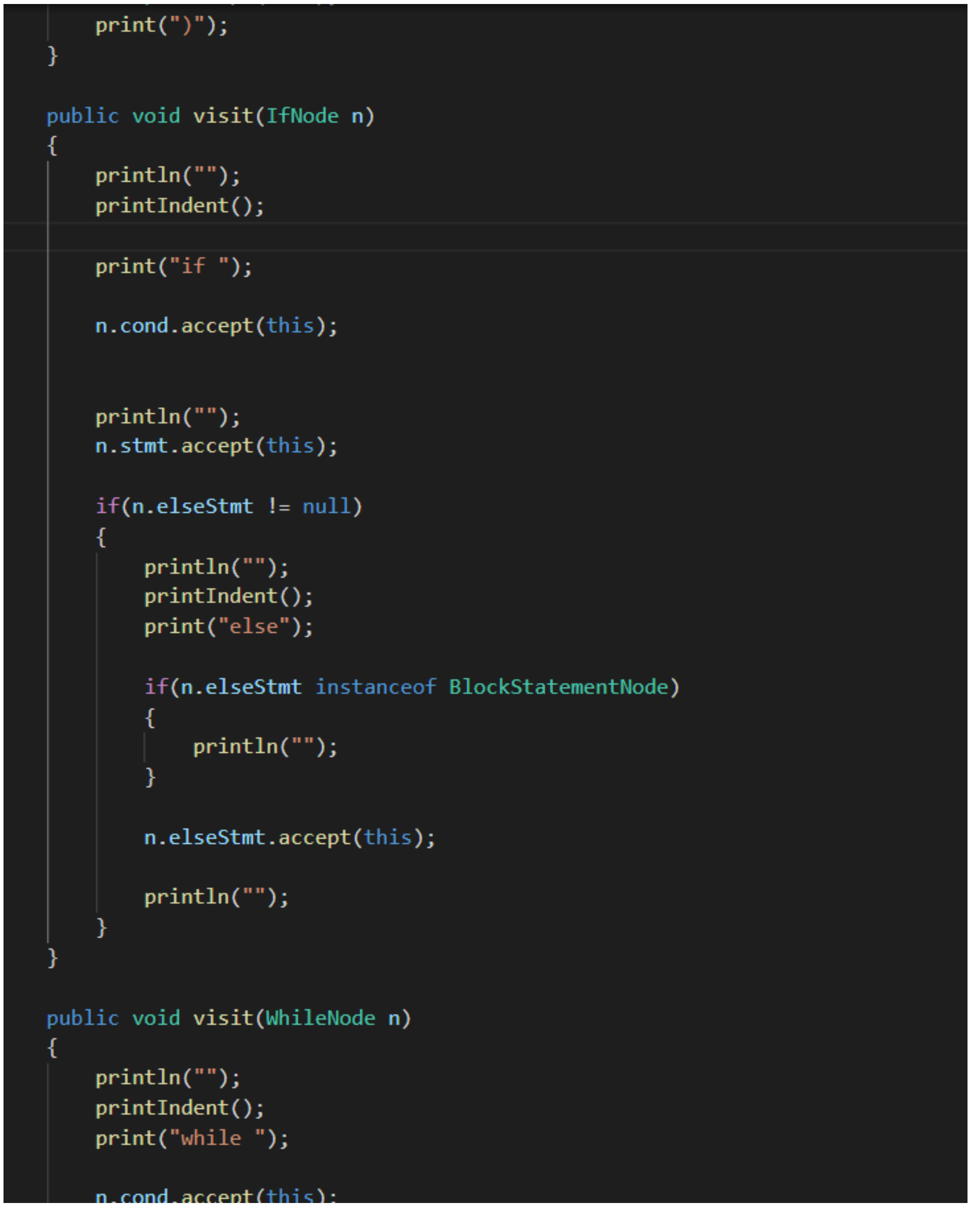
TreePrinter.java



```java
    public void visit (BlockState
    {
        dots();
        println("BlockStatementNode");

        indentUp();
        if(n.decls != null)
        {
            n.decls.accept(this);
        }
        if(n.stmts != null)
        {
            n.stmts.accept(this);
        }
        indentDown();
    }

    public void visit (DeclarationsNode n)
    {
        if(n.decls != null)
        {
            dots();
            println("Declarations");

            indentUp();
            n.decl.accept(this);
            indentDown();

            n.decls.accept(this);
        }
        if(n.stmts != null)
        {
            n.stmts.accept(this);
        }
    }

    public void visit (DeclarationNode n)
    {
        dots();
        println("DeclarationNode");

        indentUp();
        if(n.type != null)
        {
            n.type.accept(this);
            n.id.accept(this);

            if(n.assign != null) //assignment after declaration
            {
                indentUp();
                n.assign.accept(this);
                indentDown();
            }
        }
        indentDown();
    }

    public void visit (StatementsNode n)
    {
        if(n.stmts != null)
        {
            dots();
            println("Statements");

            indentUp();
            n.stmt.accept(this);
            indentDown();

            n.stmts.accept(this);
        }
        if(n.decls != null)
        {
            indentUp();
            n.decls.accept(this);
            indentDown();
        }
```

Tree Printer does the same thing as before, but with added support for the new nodes.

Unparser.java

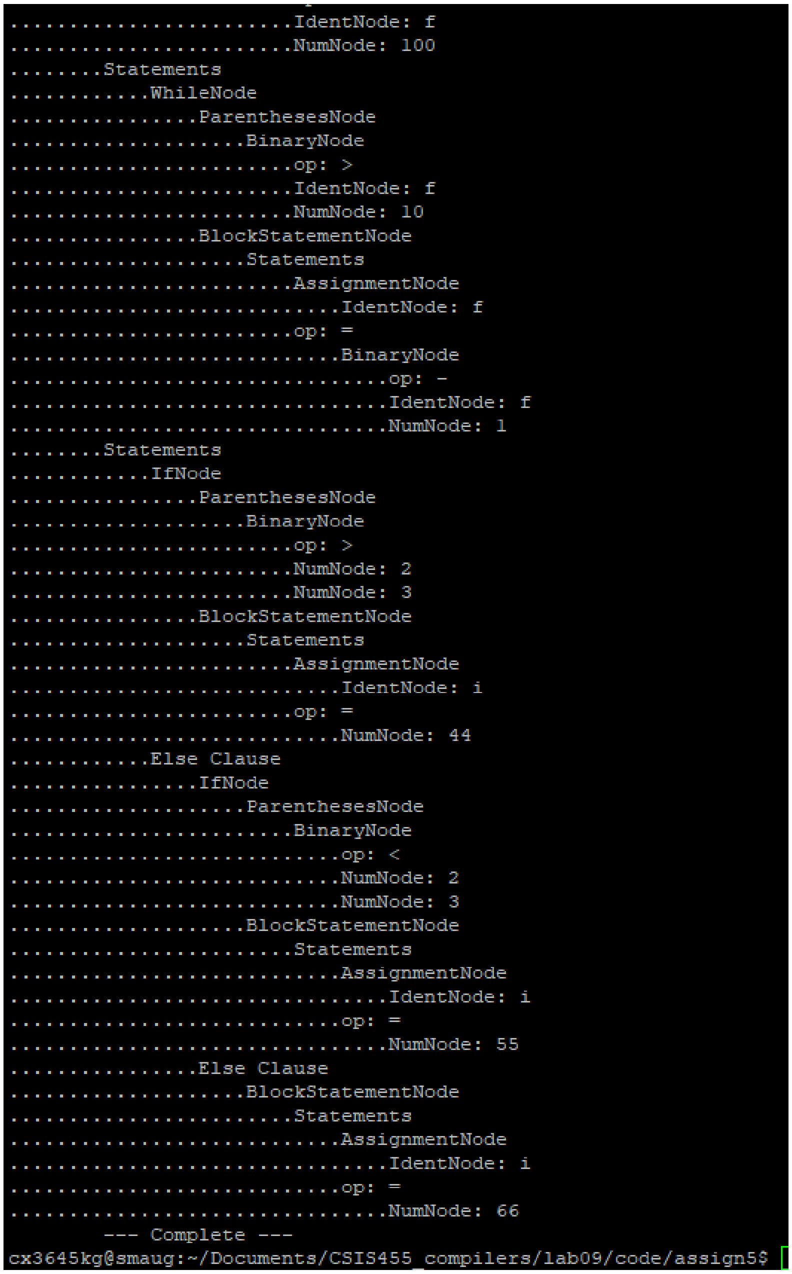


```
    print(")");
}

public void visit(IfNode n)
{
    println("");
    printIndent();

    print("if ");

    n.cond.accept(this);


    println("");
    n.stmt.accept(this);

    if(n.elseStmt != null)
    {
        println("");
        printIndent();
        print("else");

        if(n.elseStmt instanceof BlockStatementNode)
        {
            println("");
        }

        n.elseStmt.accept(this);

        println("");
    }
}

public void visit(WhileNode n)
{
    println("");
    printIndent();
    print("while ");

    n.cond.accept(this);
```

Unparser does the same thing as before, but this time will output a textfile of the code formatted that was parsed and lexically analyzed. It has added support for the many new nodes.

Execution

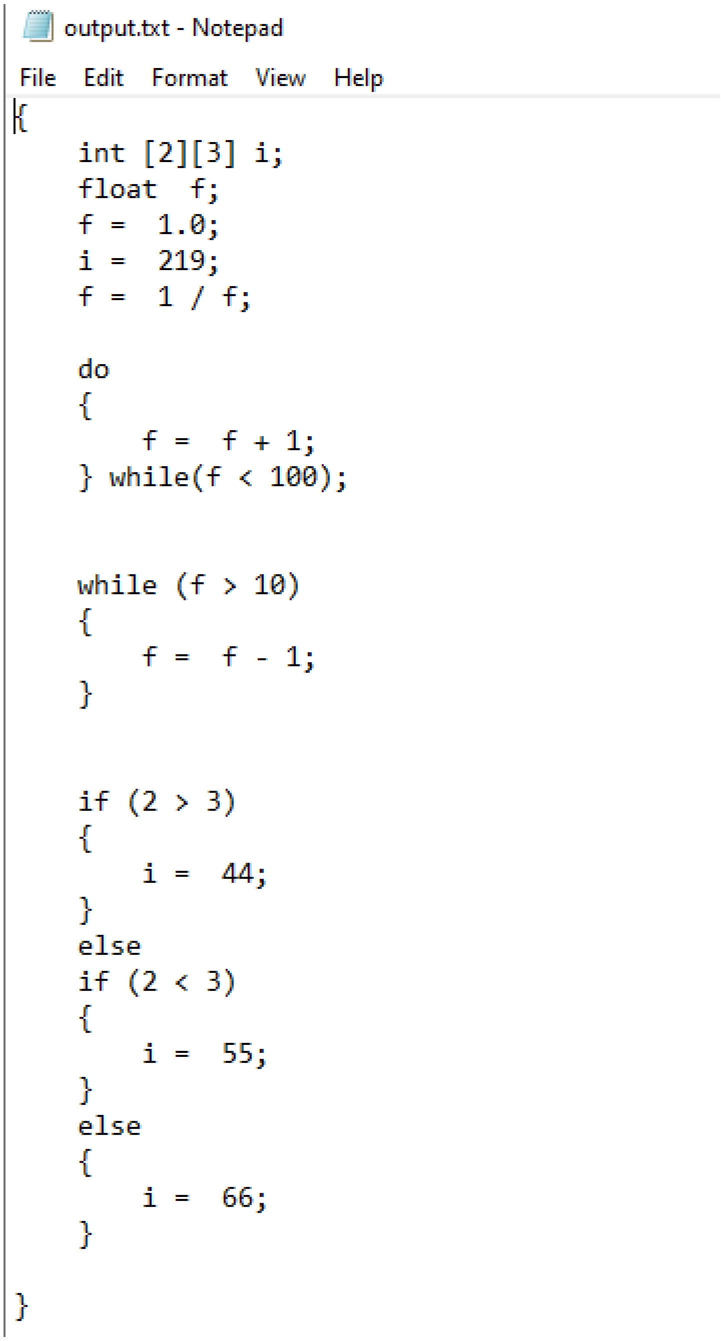
Terminal Output.



```
........................IdentNode: f
........................NumNode: 100
........Statements
............WhileNode
................ParenthesesNode
....................BinaryNode
........................op: >
........................IdentNode: f
........................NumNode: 10
................BlockStatementNode
....................Statements
........................AssignmentNode
............................IdentNode: f
........................op: =
............................BinaryNode
................................op: -
................................IdentNode: f
................................NumNode: 1
........Statements
............IfNode
................ParenthesesNode
....................BinaryNode
........................op: >
........................NumNode: 2
........................NumNode: 3
................BlockStatementNode
....................Statements
........................AssignmentNode
............................IdentNode: i
........................op: =
............................NumNode: 44
............Else Clause
................IfNode
....................ParenthesesNode
........................BinaryNode
............................op: <
............................NumNode: 2
............................NumNode: 3
....................BlockStatementNode
........................Statements
............................AssignmentNode
................................IdentNode: i
............................op: =
................................NumNode: 55
................Else Clause
....................BlockStatementNode
........................Statements
............................AssignmentNode
................................IdentNode: i
............................op: =
................................NumNode: 66
        --- Complete ---
cx3645kg@smaug:~/Documents/CSIS455_compilers/lab09/code/assign5$
```

The terminal outputted the results of parsing, unparsing, and an AST.

Output.txt



```
{
    int [2][3] i;
    float  f;
    f =  1.0;
    i =  219;
    f =  1 / f;

    do
    {
        f =  f + 1;
    } while(f < 100);


    while (f > 10)
    {
        f =  f - 1;
    }


    if (2 > 3)
    {
        i =  44;
    }
    else
    if (2 < 3)
    {
        i =  55;
    }
    else
    {
        i =  66;
    }

}
```

The unparser closely formatted the code to what it was suppose to be. However, I could not get the else if conditions to format the way I wanted and settled for what it is now because of time constraints.

Conclusion

Once again, this assignment took a lot of hours. The code from the labs is quite different from the grammar in the Dragon Book and requires drastic changes from the code being done for the other assignments. These changes cause many compilation errors and require a lot of time troubleshooting. I learned that I should not strictly follow the grammar in the book and use the code presented in the labs for future assignments. Otherwise, the code from the lab was similar to the group project except for the parsing methods within the parser.