Paul Kummer

Dr. Hanku Lee

Compilers CSIS 455-01

11/19/2021

Lab 11: Eliminate Recursion & Prepare for 3AC

Purpose of Lab

This lab's purpose is to eliminate recursion in the StatementNode, DeclarationsNode, and to prepare for breaking expressions into three-address-code(3AC) if necessary. By doing these, the compiler will not take up excessive memory with deep recursive statement and the code will begin to be ready for transformation into something that can be used in an intermediate language. Therefore, an expression like "x=1*2+3" could be broken into statements like "t=1*2" then "x=t+3". These 3AC statements can be converted into intermediate code like "MUL r0, #1, #3", or simply "MOV r0, #2", and "ADD r1, r0, #3".

Example Code

The main areas of code that were changed are within the parser. Both the DeclarationsNode and the StatementsNode are no longer used and a linked list of StatementNode or DeclarationNode are now used. To make the linked lists easier to work with, additional utility methods were added that allow adding a statement before the current statement, checking if a statement exists, replacing a statement with another statement, and getting the position of a statement in the list. Also, another utility method was added to make it easier to check if a tag is a viable candidate for a statement.

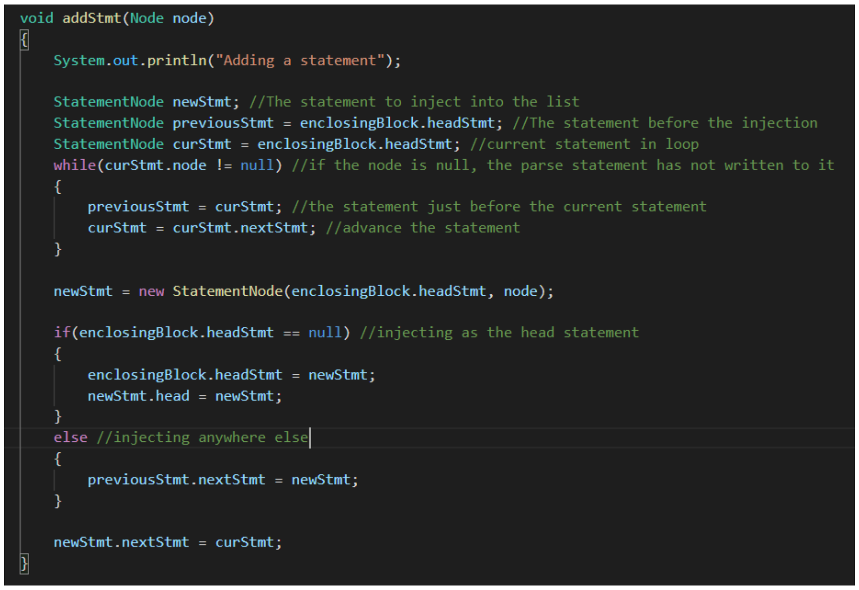
Lexer

A bug was found in the lexer that was causing operators to be read incorrectly if there wasn't a space between tokens. This was fixed by using peek instead of readch().

Parser

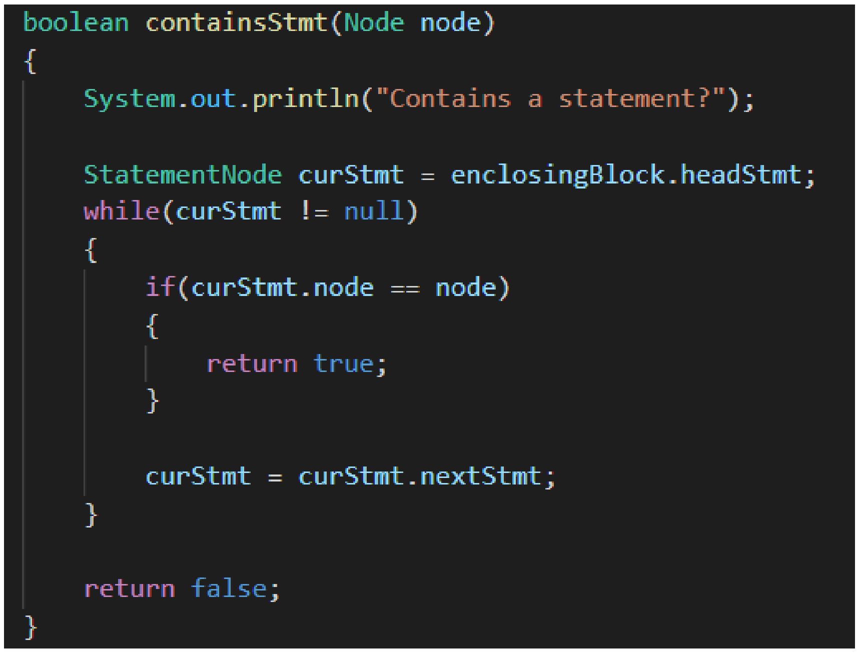
The parser no longer uses DeclarationsNode and StatementsNode. Multiple declaration and statements are now handled with a linked list of DeclarationNodes and StatementNodes. Many new utility methods are added for managing the linked lists. Also, to demonstrate the ability of the methods, they are envoked within the IfNode. This allows an example demonstration of how a statement can be added, which can be used for making 3AC.

Parser.java: new utility methods



```
void addStmt(Node node)
{
    System.out.println("Adding a statement");

    StatementNode newStmt; //The statement to inject into the list
    StatementNode previousStmt = enclosingBlock.headStmt; //The statement before the injection
    StatementNode curStmt = enclosingBlock.headStmt; //current statement in loop
    while(curStmt.node != null) //if the node is null, the parse statement has not written to it
    {
        previousStmt = curStmt; //the statement just before the current statement
        curStmt = curStmt.nextStmt; //advance the statement
    }

    newStmt = new StatementNode(enclosingBlock.headStmt, node);

    if(enclosingBlock.headStmt == null) //injecting as the head statement
    {
        enclosingBlock.headStmt = newStmt;
        newStmt.head = newStmt;
    }
    else //injecting anywhere else
    {
        previousStmt.nextStmt = newStmt;
    }

    newStmt.nextStmt = curStmt;
}
```

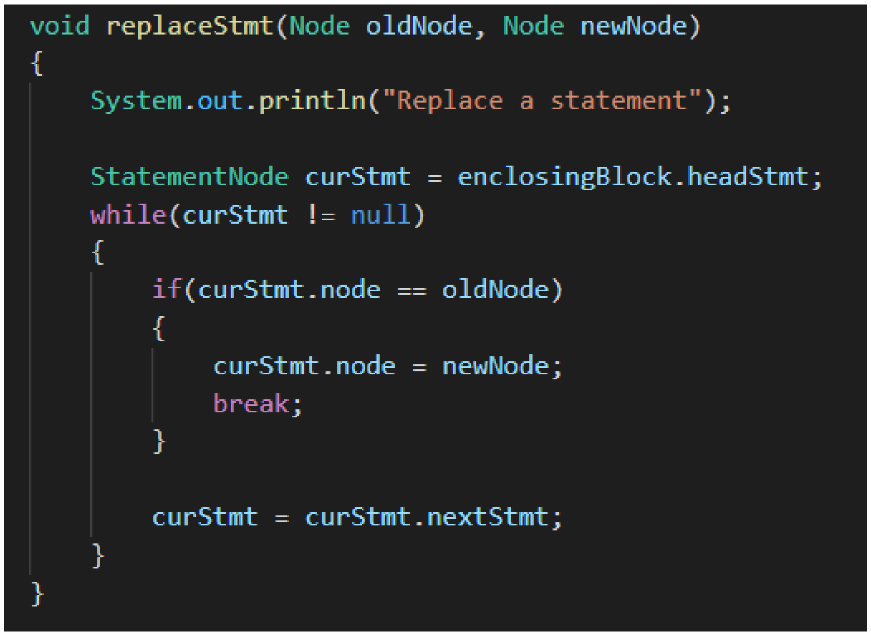
Adds a statement just before the current statement. Within each statement there is a node that holds whatever the statement is going to be. The statement.node could be an AssignmentNode, IfNode, WhileNode, DoNode, BreakStmtNode, ForNode, or BlockStatementNode.



```
boolean containsStmt(Node node)
{
    System.out.println("Contains a statement?");

    StatementNode curStmt = enclosingBlock.headStmt;
    while(curStmt != null)
    {
        if(curStmt.node == node)
        {
            return true;
        }

        curStmt = curStmt.nextStmt;
    }

    return false;
}
```

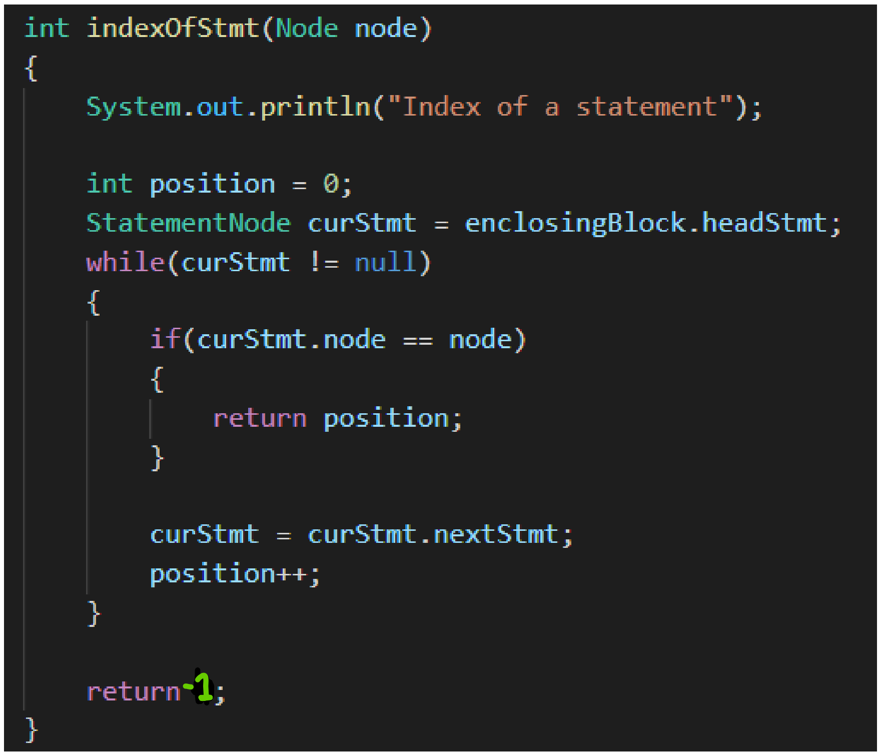
Returns a bool value of whether the enclosing block contains a specified argument.

Parser.java: new utility methods Cont.



```java
void replaceStmt(Node oldNode, Node newNode)
{
    System.out.println("Replace a statement");

    StatementNode curStmt = enclosingBlock.headStmt;
    while(curStmt != null)
    {
        if(curStmt.node == oldNode)
        {
            curStmt.node = newNode;
            break;
        }

        curStmt = curStmt.nextStmt;
    }
}
```

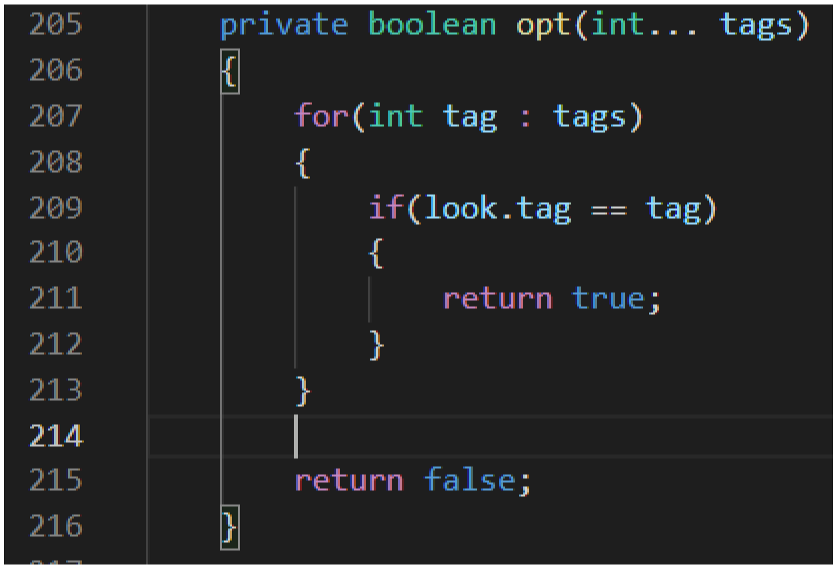
This method will search all the statements in the enclosing block for a specified statement. If the statement is found, it is replaced with a new specified statement. If it isn't found, nothing will happen with the new or old statement.



```java
int indexOfStmt(Node node)
{
    System.out.println("Index of a statement");

    int position = 0;
    StatementNode curStmt = enclosingBlock.headStmt;
    while(curStmt != null)
    {
        if(curStmt.node == node)
        {
            return position;
        }

        curStmt = curStmt.nextStmt;
        position++;
    }

    return -1;
}
```

When this method is used, the position in the linked list of statements is returned. If the statement isn't found, -1 is returned.

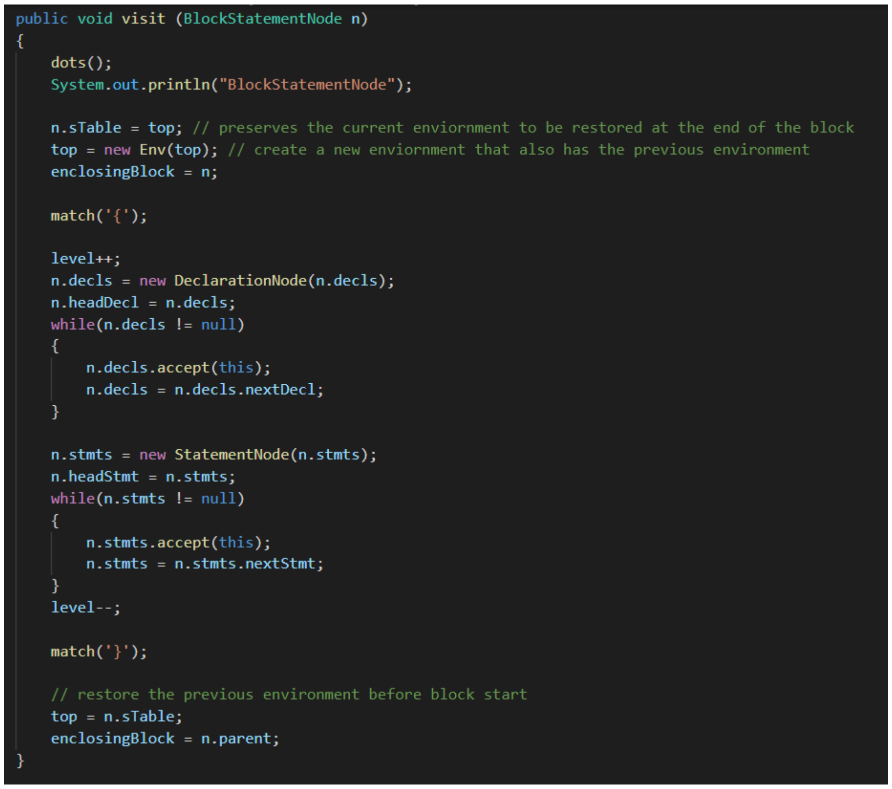
Parser.java: new utility methods Cont.



```java
    private boolean opt(int... tags)
    {
        for(int tag : tags)
        {
            if(look.tag == tag)
            {
                return true;
            }
        }

        return false;
    }
```

This opt method will return a bool value indicating if on of the integer values representing a tag is viable. A true result means one of the arguments is a matched tag value.

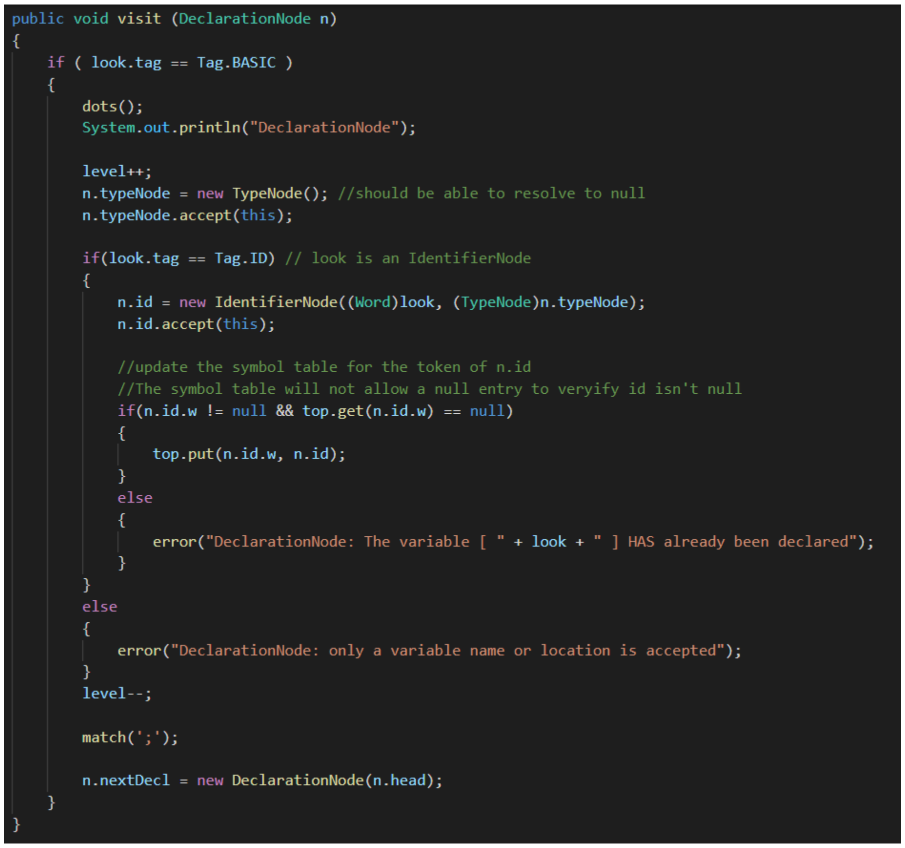
Parser.java: visit to block statement



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

    n.sTable = top; // preserves the current enviornment to be restored at the end of the block
    top = new Env(top); // create a new enviornment that also has the previous environment
    enclosingBlock = n;

    match('{');

    level++;
    n.decls = new DeclarationNode(n.decls);
    n.headDecl = n.decls;
    while(n.decls != null)
    {
        n.decls.accept(this);
        n.decls = n.decls.nextDecl;
    }

    n.stmts = new StatementNode(n.stmts);
    n.headStmt = n.stmts;
    while(n.stmts != null)
    {
        n.stmts.accept(this);
        n.stmts = n.stmts.nextStmt;
    }
    level--;

    match('}');

    // restore the previous environment before block start
    top = n.sTable;
    enclosingBlock = n.parent;
}
```

To eliminate recursion, the block statement will now loop calling new declarations or nodes. Within each BlockStatementNode there are now new attributes to help manage the linked lists.

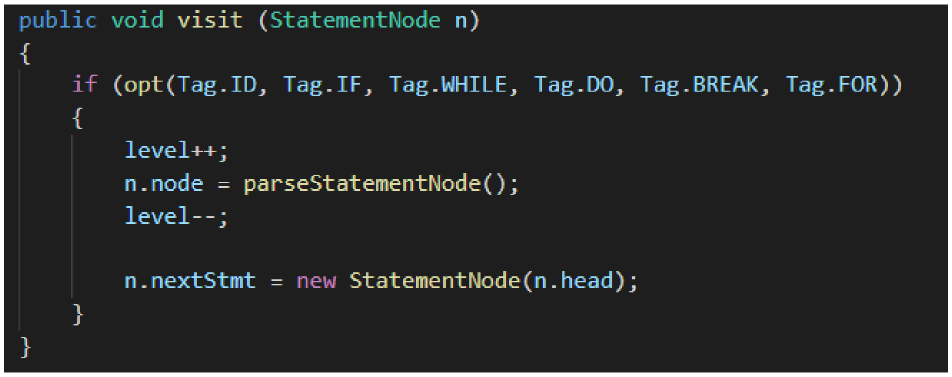
Parser.java: visit to block statement



```
public void visit (DeclarationNode n)
{
    if ( look.tag == Tag.BASIC )
    {
        dots();
        System.out.println("DeclarationNode");

        level++;
        n.typeNode = new TypeNode(); //should be able to resolve to null
        n.typeNode.accept(this);

        if(look.tag == Tag.ID) // look is an IdentifierNode
        {
            n.id = new IdentifierNode((Word)look, (TypeNode)n.typeNode);
            n.id.accept(this);

            //update the symbol table for the token of n.id
            //The symbol table will not allow a null entry to veryify id isn't null
            if(n.id.w != null && top.get(n.id.w) == null)
            {
                top.put(n.id.w, n.id);
            }
            else
            {
                error("DeclarationNode: The variable [ " + look + " ] HAS already been declared");
            }
        }
        else
        {
            error("DeclarationNode: only a variable name or location is accepted");
        }
        level--;

        match(';');

        n.nextDecl = new DeclarationNode(n.head);
    }
}
```

The functionality of the DeclarationsNode is now moved into the DeclarationNode. Also, the next DeclarationNode is created within the current declaration. This new DeclarationNode will be accepted in the BlockStatementNode.

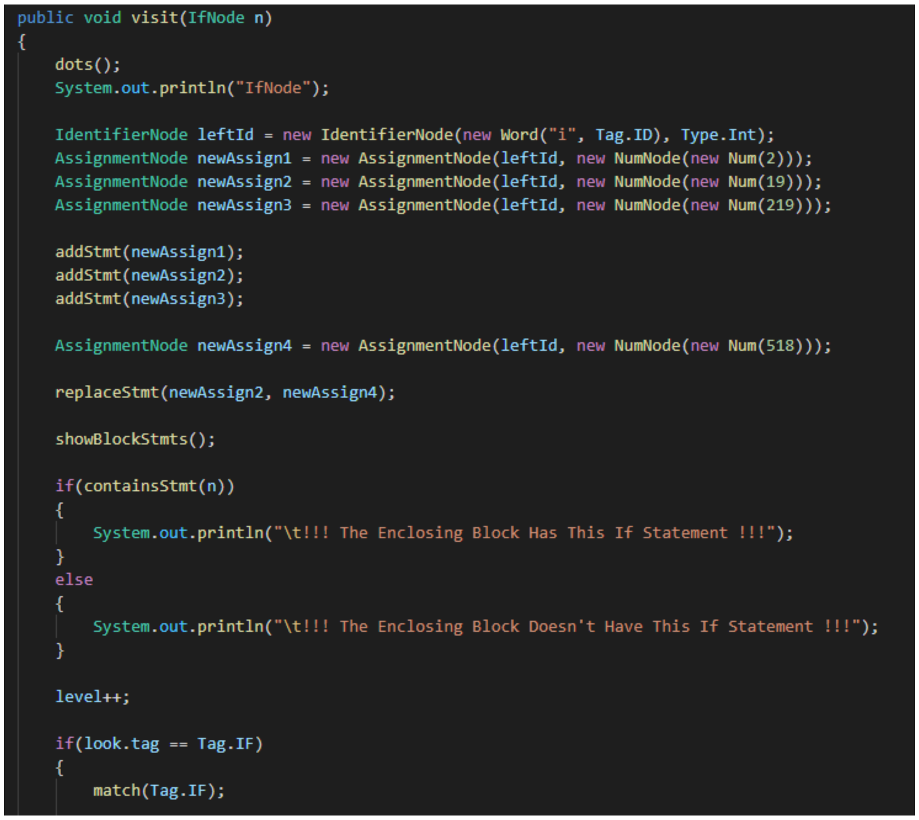
Parser.java: visit to statement



```
public void visit (StatementNode n)
{
    if (opt(Tag.ID, Tag.IF, Tag.WHILE, Tag.DO, Tag.BREAK, Tag.FOR))
    {
        level++;
        n.node = parseStatementNode();
        level--;

        n.nextStmt = new StatementNode(n.head);
    }
}
```

Much like the DeclarationNode, the functionality of the previous StatementsNode is moved into the StatementNode and the next StatementNode is initialized within the current StatementNode.

Parser.java: visit to IfNode



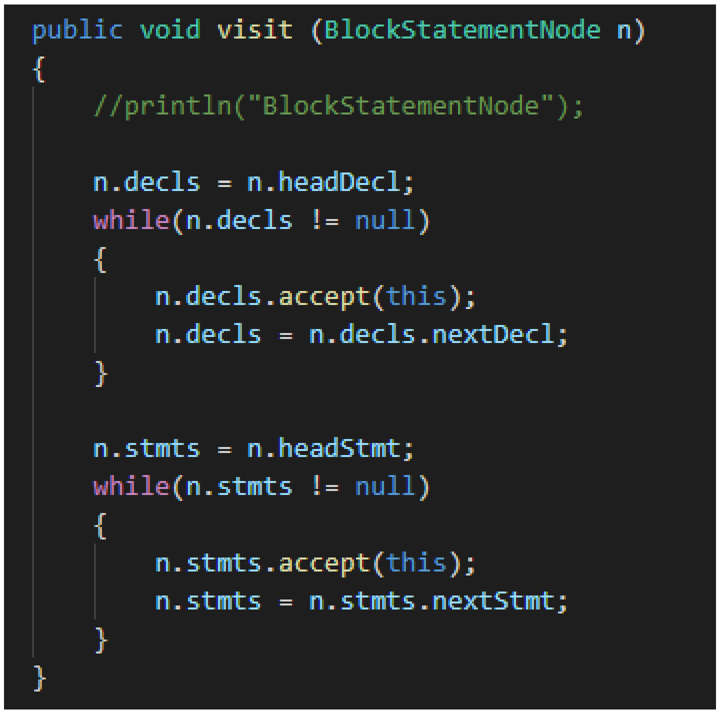
```java
public void visit(IfNode n)
{
    dots();
    System.out.println("IfNode");

    IdentifierNode leftId = new IdentifierNode(new Word("i", Tag.ID), Type.Int);
    AssignmentNode newAssign1 = new AssignmentNode(leftId, new NumNode(new Num(2)));
    AssignmentNode newAssign2 = new AssignmentNode(leftId, new NumNode(new Num(19)));
    AssignmentNode newAssign3 = new AssignmentNode(leftId, new NumNode(new Num(219)));

    addStmt(newAssign1);
    addStmt(newAssign2);
    addStmt(newAssign3);

    AssignmentNode newAssign4 = new AssignmentNode(leftId, new NumNode(new Num(518)));

    replaceStmt(newAssign2, newAssign4);

    showBlockStmts();

    if(containsStmt(n))
    {
        System.out.println("\t!!! The Enclosing Block Has This If Statement !!!");
    }
    else
    {
        System.out.println("\t!!! The Enclosing Block Doesn't Have This If Statement !!!");
    }

    level++;

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

Within the IfNode, the abilities of the new utility methods are tested. This example proves the ability to add into the linked list new statements while making a statement. When making 3AC this functionality will be important in the parsing of binary nodes because intermediate code only supports one operation at a time.

Tree Printer / Type Checker / Unparser / ASTVisitor

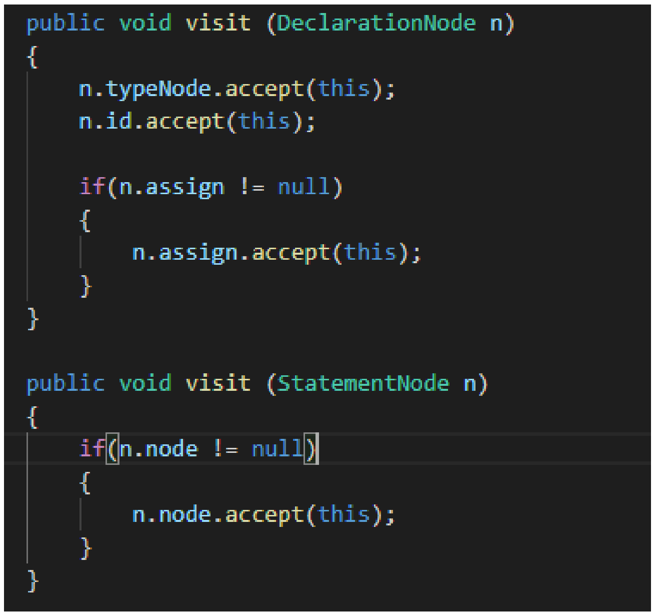
In the TreePrinter.java, TypeChecker.java, Unparser.java, and ASTVisitor.java files, the visit methods for the BlockStatementNode, DeclarationNode, and StatementNode have all been changed to remove recursion. What is done in the following example of ASTVisitor.java is done in the other files as well.

ASTVisitor.java



```java
public void visit (BlockStatementNode n)
{
    //println("BlockStatementNode");

    n.decls = n.headDecl;
    while(n.decls != null)
    {
        n.decls.accept(this);
        n.decls = n.decls.nextDecl;
    }

    n.stmts = n.headStmt;
    while(n.stmts != null)
    {
        n.stmts.accept(this);
        n.stmts = n.stmts.nextStmt;
    }
}
```

The visit method for a BlockStatementNode will now iterate over each DeclarationNode and StatementNode stored within a block statement. This is done by starting at the head of each list and then advancing to the next declaration/statement within the current declaration/statement. With every new declaration/statement that is advanced to, its visit method is called with the accept statement.



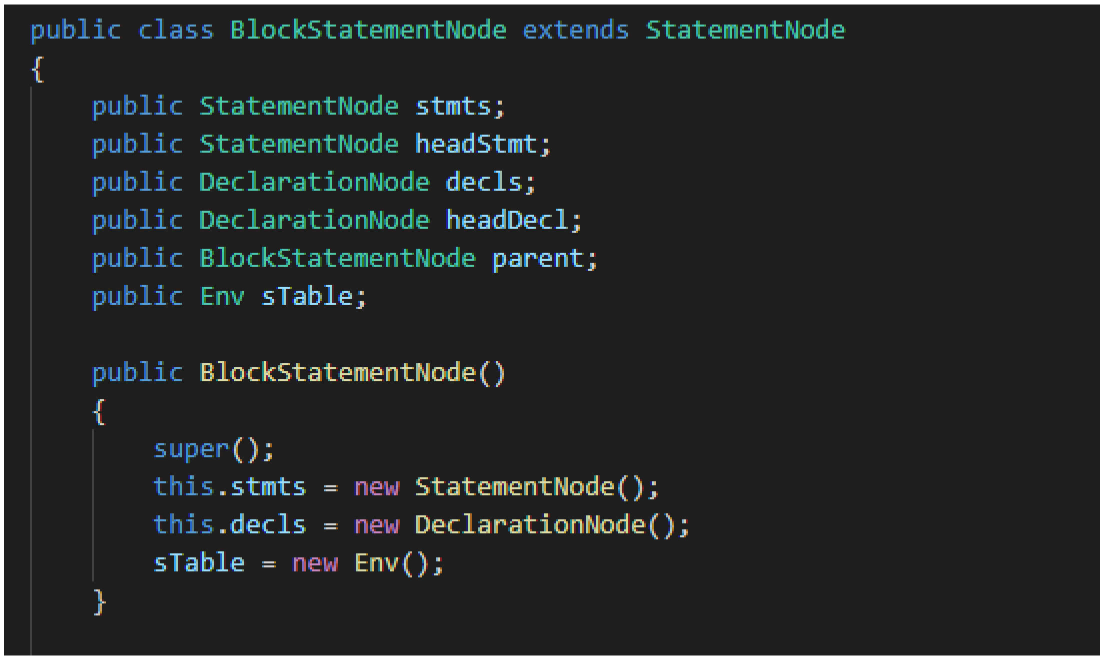
```java
public void visit (DeclarationNode n)
{
    n.typeNode.accept(this);
    n.id.accept(this);

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

public void visit (StatementNode n)
{
    if(n.node != null)
    {
        n.node.accept(this);
    }
}
```

The DeclarationNode and StatementNode’s visit method now does not recursively call its next DeclarationNode or StatementNode.

AST

The BlockStatementNode, DeclarationNode, and StatementNode have all been modified to aid in eliminating the recursive statements and store information for supporting linked lists.

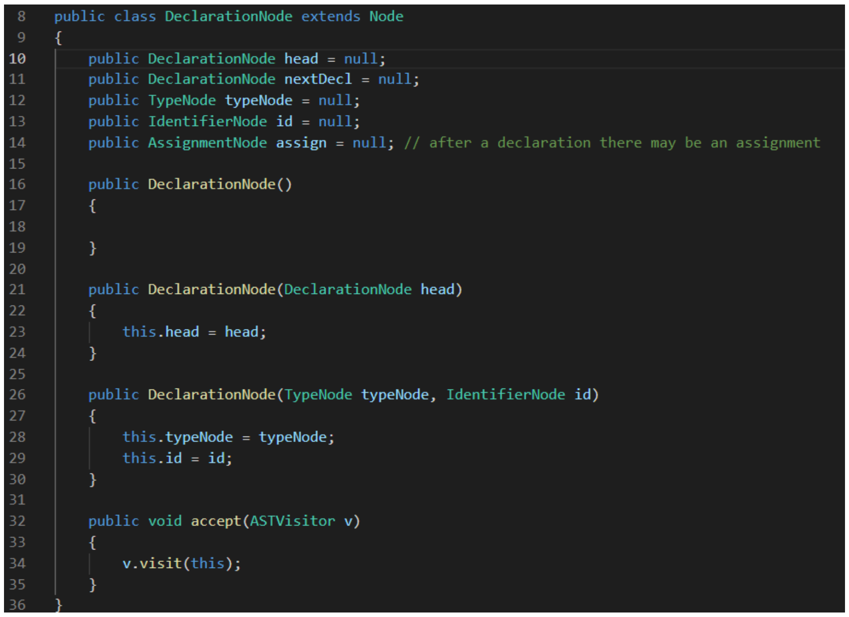
BlockStatementNode.java



```java
public class BlockStatementNode extends StatementNode
{
    public StatementNode stmts;
    public StatementNode headStmt;
    public DeclarationNode decls;
    public DeclarationNode headDecl;
    public BlockStatementNode parent;
    public Env sTable;

    public BlockStatementNode()
    {
        super();
        this.stmts = new StatementNode();
        this.decls = new DeclarationNode();
        sTable = new Env();
    }
```

When a block statement is created, its default constructor will now initialize the decls and stmts attribute. These attributes will hold the position that the current linked list is at. There is also a head attribute for declarations and statements. These head attributes will be useful for going back to the beginning of the list.

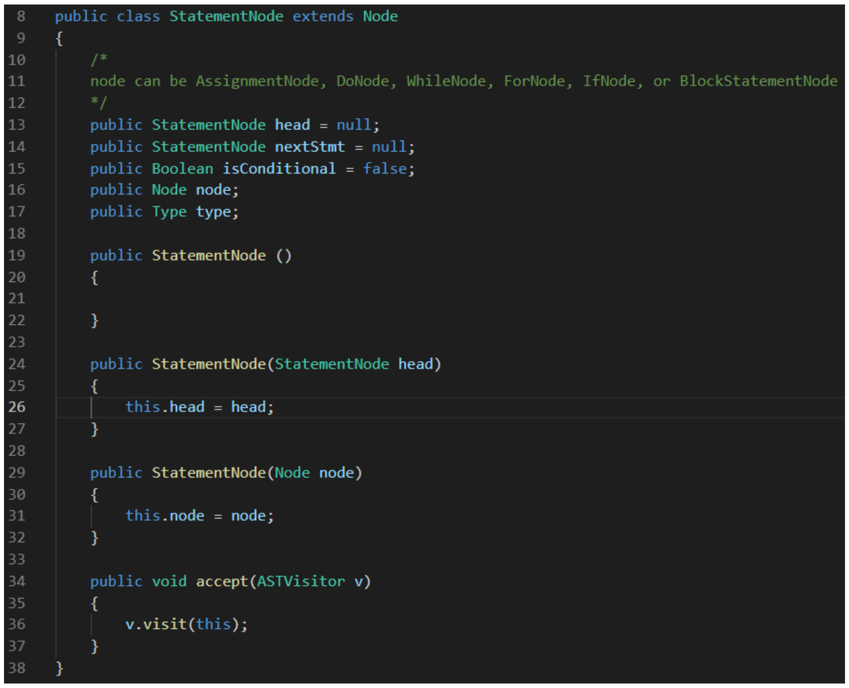
DelcarationNode.java



```
public class DeclarationNode extends Node
{
    public DeclarationNode head = null;
    public DeclarationNode nextDecl = null;
    public TypeNode typeNode = null;
    public IdentifierNode id = null;
    public AssignmentNode assign = null; // after a declaration there may be an assignment

    public DeclarationNode()
    {

    }

    public DeclarationNode(DeclarationNode head)
    {
        this.head = head;
    }

    public DeclarationNode(TypeNode typeNode, IdentifierNode id)
    {
        this.typeNode = typeNode;
        this.id = id;
    }

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

Like the BlockStatementNode, the DeclarationNode also stores an attribute head, which will allow any declaration to get back to the beginning of the list. Also, each declaration will store and attribute that holds the next declaration node. If this attribute is ever null, then the current declaration is the end of the list.

StatementNode.java

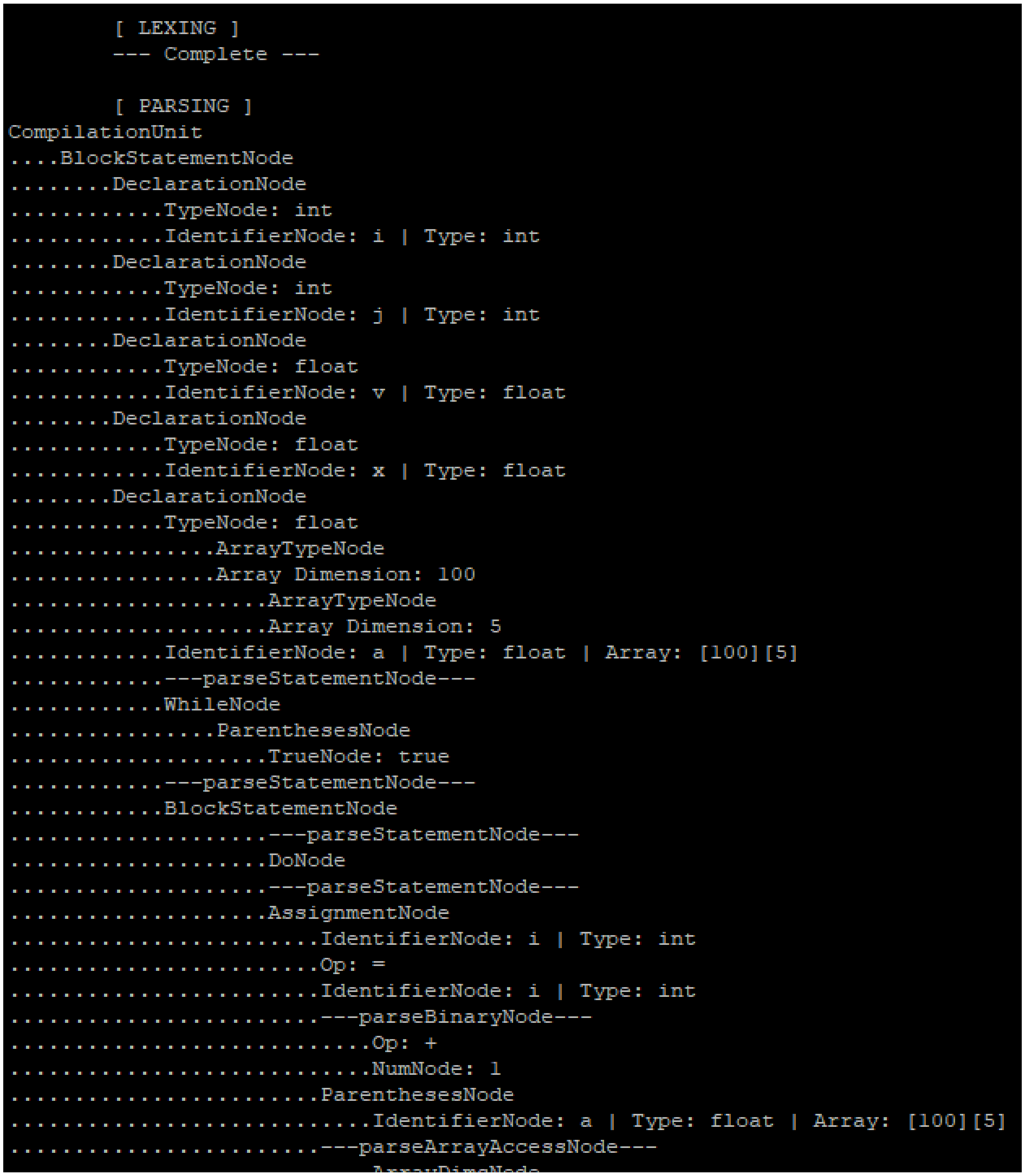


```
public class StatementNode extends Node
{
    /*
    node can be AssignmentNode, DoNode, WhileNode, ForNode, IfNode, or BlockStatementNode
    */
    public StatementNode head = null;
    public StatementNode nextStmt = null;
    public Boolean isConditional = false;
    public Node node;
    public Type type;

    public StatementNode ()
    {

    }

    public StatementNode(StatementNode head)
    {
        this.head = head;
    }

    public StatementNode(Node node)
    {
        this.node = node;
    }

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

The StatementNode has identical functionality as the DeclaratioNode. However, it has a attribute, isConditional, that serves as a flag to help determine if a statement should derive to Boolean.

Execution

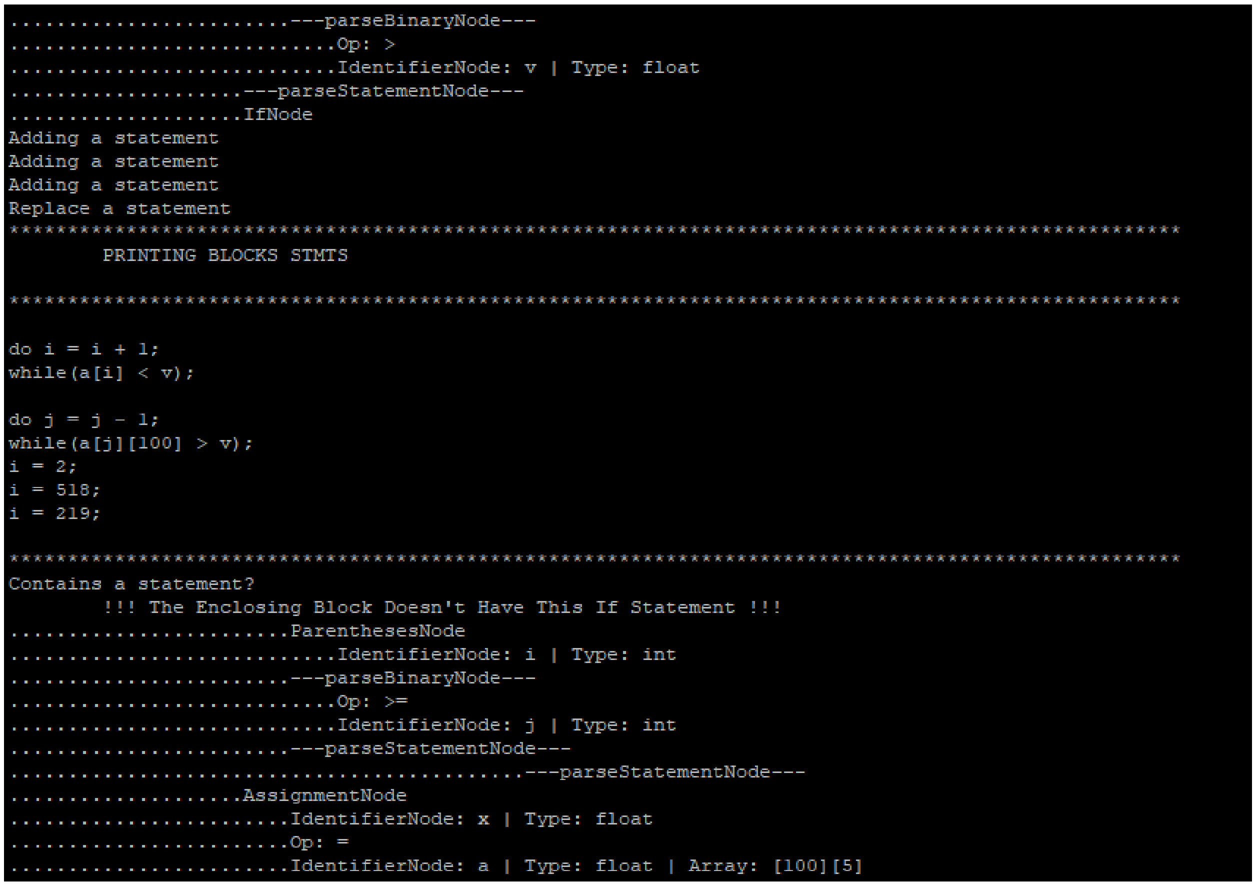
Lexing and Parsing



```
        [ LEXING ]
        --- Complete ---

        [ PARSING ]
CompilationUnit
....BlockStatementNode
........DeclarationNode
............TypeNode: int
............IdentifierNode: i | Type: int
........DeclarationNode
............TypeNode: int
............IdentifierNode: j | Type: int
........DeclarationNode
............TypeNode: float
............IdentifierNode: v | Type: float
........DeclarationNode
............TypeNode: float
............IdentifierNode: x | Type: float
........DeclarationNode
............TypeNode: float
................ArrayTypeNode
................Array Dimension: 100
....................ArrayTypeNode
....................Array Dimension: 5
............IdentifierNode: a | Type: float | Array: [100][5]
............---parseStatementNode---
............WhileNode
................ParenthesesNode
....................TrueNode: true
............---parseStatementNode---
............BlockStatementNode
....................---parseStatementNode---
....................DoNode
....................---parseStatementNode---
....................AssignmentNode
........................IdentifierNode: i | Type: int
........................Op: =
........................IdentifierNode: i | Type: int
........................---parseBinaryNode---
............................Op: +
............................NumNode: 1
........................ParenthesesNode
............................IdentifierNode: a | Type: float | Array: [100][5]
........................---parseArrayAccessNode---
```

The lexing and parsing phase both perform as expected.

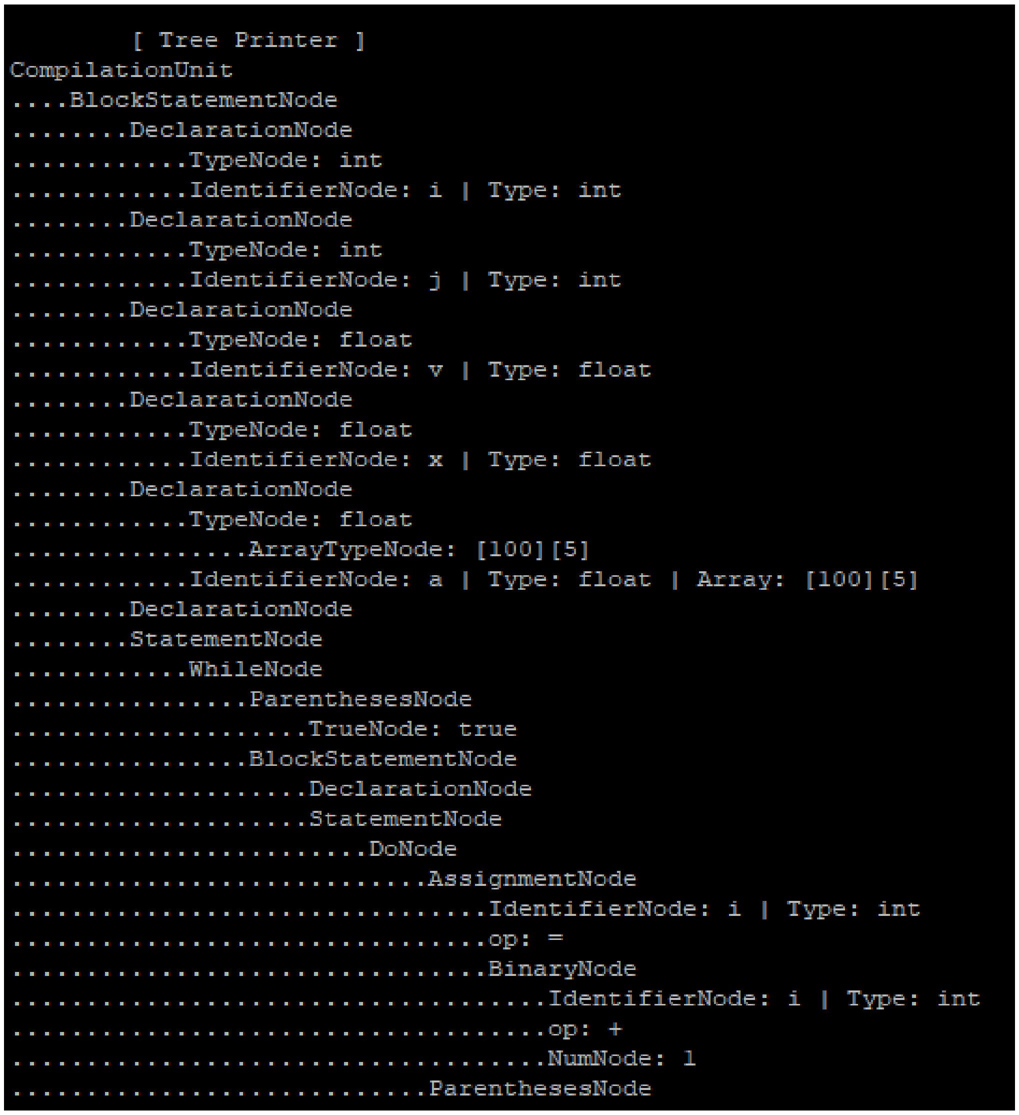
Parsing with statement injection



```
........................---parseBinaryNode---
............................Op: >
............................IdentifierNode: v | Type: float
....................---parseStatementNode---
....................IfNode
Adding a statement
Adding a statement
Adding a statement
Replace a statement
****************************************************************************************************
        PRINTING BLOCKS STMTS

****************************************************************************************************

do i = i + 1;
while(a[i] < v);

do j = j - 1;
while(a[j][100] > v);
i = 2;
i = 518;
i = 219;

****************************************************************************************************
Contains a statement?
        !!! The Enclosing Block Doesn't Have This If Statement !!!
........................ParenthesesNode
............................IdentifierNode: i | Type: int
........................---parseBinaryNode---
............................Op: >=
............................IdentifierNode: j | Type: int
........................---parseStatementNode---
............................................---parseStatementNode---
....................AssignmentNode
........................IdentifierNode: x | Type: float
........................Op: =
........................IdentifierNode: a | Type: float | Array: [100][5]
```

When the IfNode is visited, some statements are injected before the If statement. The result in parsing shows the statements being added and replaced. Then after the statements are added. The current statements in the enclosing block are printed to show what has been created so far. This proves that the statements are in fact created before the IfNode is finished. To verify this, another statement will display if the current statement is in the list of statements in the enclosing block.

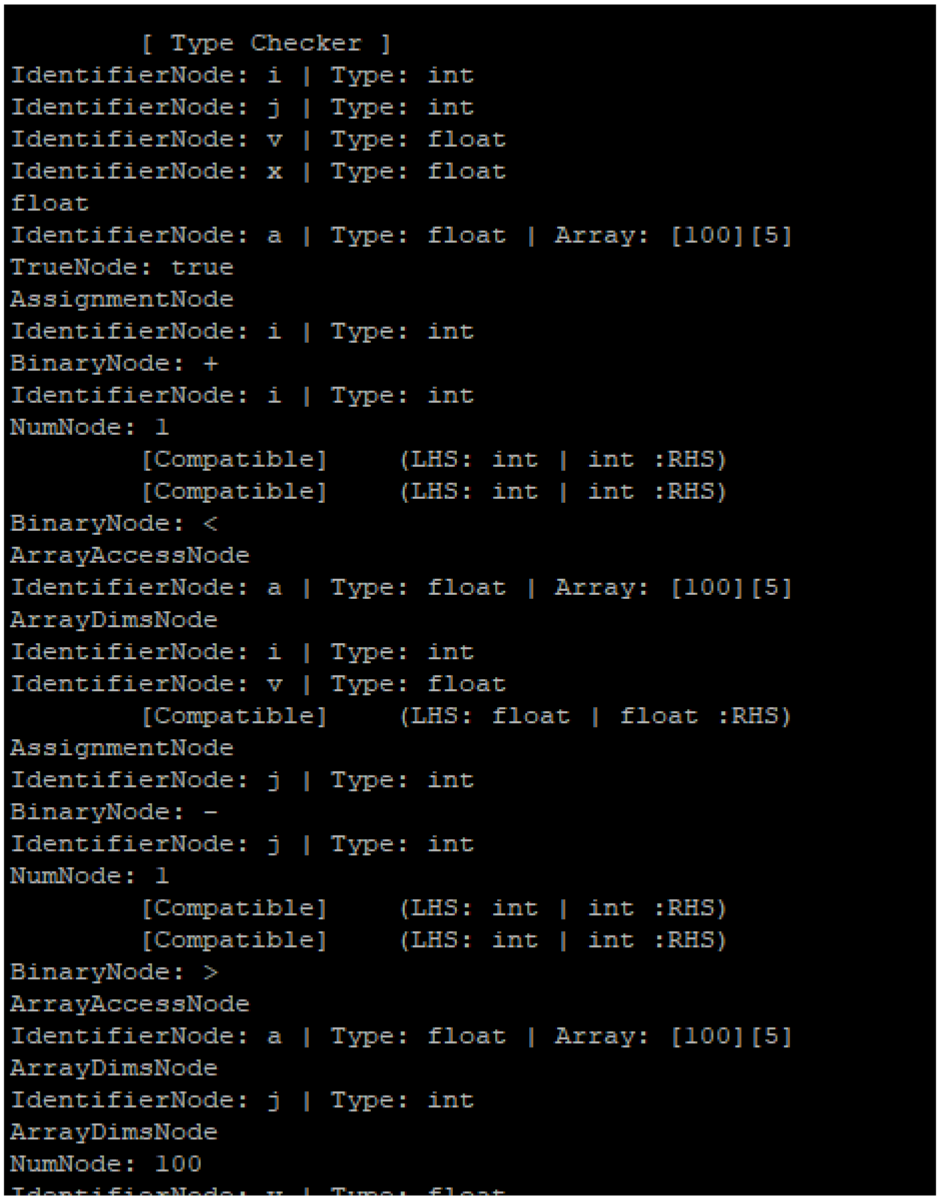
Tree printing



```
        [ Tree Printer ]
CompilationUnit
....BlockStatementNode
........DeclarationNode
............TypeNode: int
............IdentifierNode: i | Type: int
........DeclarationNode
............TypeNode: int
............IdentifierNode: j | Type: int
........DeclarationNode
............TypeNode: float
............IdentifierNode: v | Type: float
........DeclarationNode
............TypeNode: float
............IdentifierNode: x | Type: float
........DeclarationNode
............TypeNode: float
................ArrayTypeNode: [100][5]
............IdentifierNode: a | Type: float | Array: [100][5]
........DeclarationNode
........StatementNode
............WhileNode
................ParenthesesNode
....................TrueNode: true
................BlockStatementNode
....................DeclarationNode
....................StatementNode
........................DoNode
............................AssignmentNode
................................IdentifierNode: i | Type: int
................................op: =
................................BinaryNode
....................................IdentifierNode: i | Type: int
....................................op: +
....................................NumNode: 1
............................ParenthesesNode
```

The tree printer creates an AST as expected

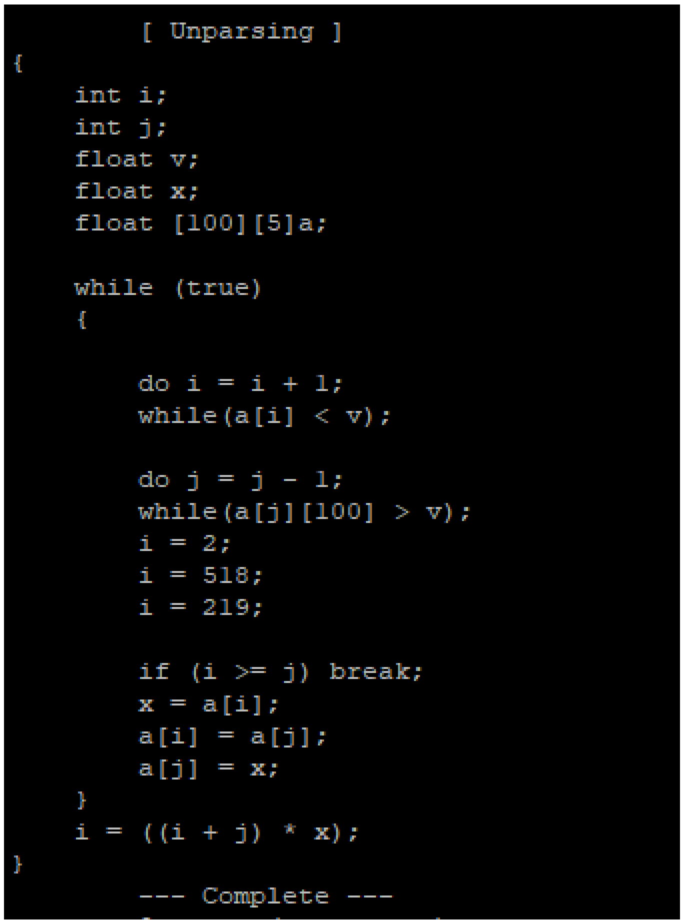
Type checking



```
        [ Type Checker ]
IdentifierNode: i | Type: int
IdentifierNode: j | Type: int
IdentifierNode: v | Type: float
IdentifierNode: x | Type: float
float
IdentifierNode: a | Type: float | Array: [100][5]
TrueNode: true
AssignmentNode
IdentifierNode: i | Type: int
BinaryNode: +
IdentifierNode: i | Type: int
NumNode: 1
        [Compatible]    (LHS: int | int :RHS)
        [Compatible]    (LHS: int | int :RHS)
BinaryNode: <
ArrayAccessNode
IdentifierNode: a | Type: float | Array: [100][5]
ArrayDimsNode
IdentifierNode: i | Type: int
IdentifierNode: v | Type: float
        [Compatible]    (LHS: float | float :RHS)
AssignmentNode
IdentifierNode: j | Type: int
BinaryNode: -
IdentifierNode: j | Type: int
NumNode: 1
        [Compatible]    (LHS: int | int :RHS)
        [Compatible]    (LHS: int | int :RHS)
BinaryNode: >
ArrayAccessNode
IdentifierNode: a | Type: float | Array: [100][5]
ArrayDimsNode
IdentifierNode: j | Type: int
ArrayDimsNode
NumNode: 100
IdentifierNode: v | Type: float
```

The type checker compares all the types from the binary expressions and assignments to determine that all the values are compatible, which is expected with the current input file.

Unparsing



```
        [ Unparsing ]
{
    int i;
    int j;
    float v;
    float x;
    float [100][5]a;

    while (true)
    {

        do i = i + 1;
        while(a[i] < v);

        do j = j - 1;
        while(a[j][100] > v);
        i = 2;
        i = 518;
        i = 219;

        if (i >= j) break;
        x = a[i];
        a[i] = a[j];
        a[j] = x;
    }
    i = ((i + j) * x);
}
        --- Complete ---
```

The unparser correctly formats and creates and output file of the code from the input file.

Conclusion

This lab showed me how to implement a data structure of a linked list into an existing program to eliminate recursion. More importantly, it gave me the tools needed to implement 3AC for creating intermediate code. The most complex part of the compiler so far has been getting the arrays to work correctly and ensuring that all the types are checked. There are many different spots in the program that types need to be compared to verify compatibility. The same holds true with grammar checking too.