Paul Kummer

Dr. Hanku Lee

Compilers CSIS 455-01

11/29/2021

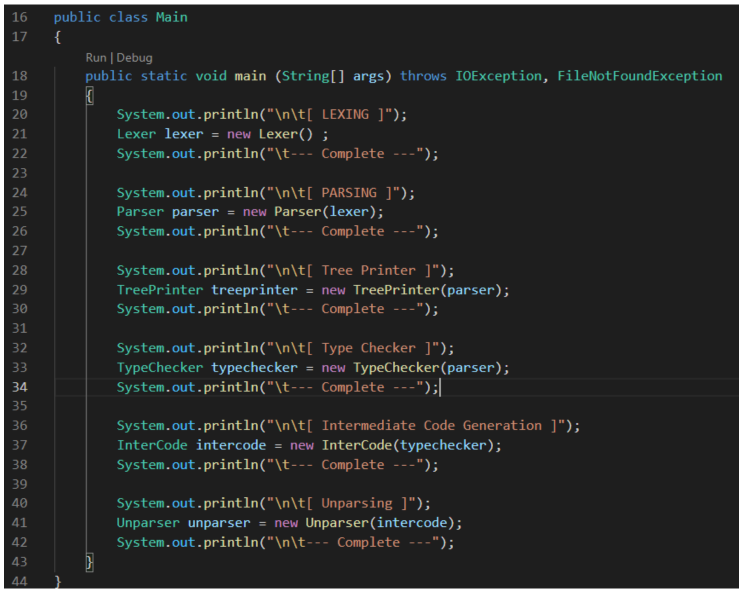
Lab 12: Intermediate Code Generation

Purpose of Lab

This lab’s purpose is to generate intermediate code from the intercode phase of the compiler. To accomplish the task of creating intermediate code, a new directory called “inter” is created to store the new files associated with the intermediate code generation. Three of the files are new nodes of GotoNode.java, LabelNode.java, and TempNode.java. Also, the directory contains the InterCode.java file, which is responsible for adding in new statements needed for an assembly-like language. After the inter phase is complete, the unparsing phase will generate a file with assembly-like code called output.txt.

Example Code

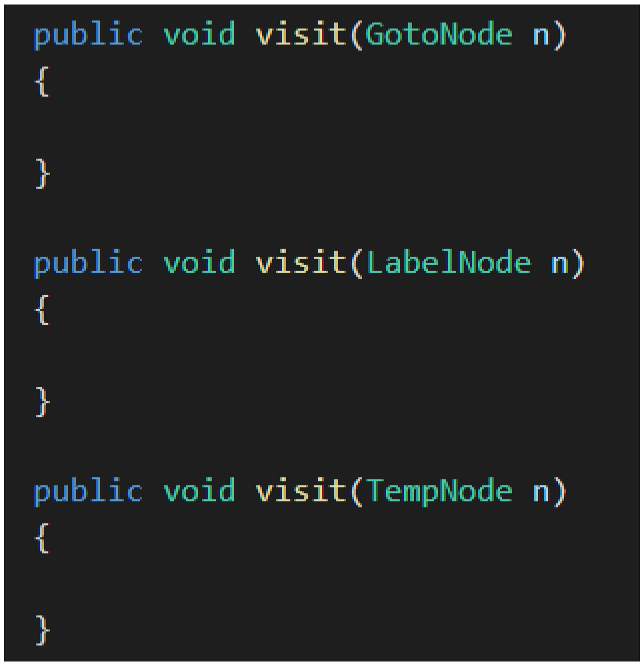
Main.java



```java
public class Main
{
    public static void main (String[] args) throws IOException, FileNotFoundException
    {
        System.out.println("\n\t[ LEXING ]");
        Lexer lexer = new Lexer() ;
        System.out.println("\t--- Complete ---");

        System.out.println("\n\t[ PARSING ]");
        Parser parser = new Parser(lexer);
        System.out.println("\t--- Complete ---");

        System.out.println("\n\t[ Tree Printer ]");
        TreePrinter treeprinter = new TreePrinter(parser);
        System.out.println("\t--- Complete ---");

        System.out.println("\n\t[ Type Checker ]");
        TypeChecker typechecker = new TypeChecker(parser);
        System.out.println("\t--- Complete ---");

        System.out.println("\n\t[ Intermediate Code Generation ]");
        InterCode intercode = new InterCode(typechecker);
        System.out.println("\t--- Complete ---");

        System.out.println("\n\t[ Unparsing ]");
        Unparser unparser = new Unparser(intercode);
        System.out.println("\n\t--- Complete ---");
    }
}
```

The intermediate code generation phase has been added right after the type checking is complete. The TypeChecker object is passed into the intercode phase and uses its objects to create three address code if necessary.

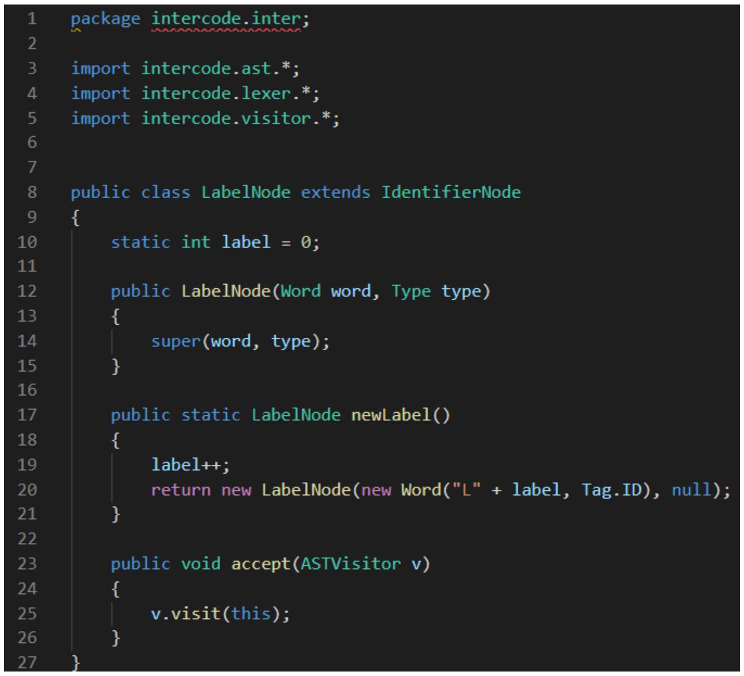
ASTVisitor.java



```
public void visit(GotoNode n)
{

}

public void visit(LabelNode n)
{

}

public void visit(TempNode n)
{

}
```

Three blank visit methods are added for a default method for the new intermediate code nodes.

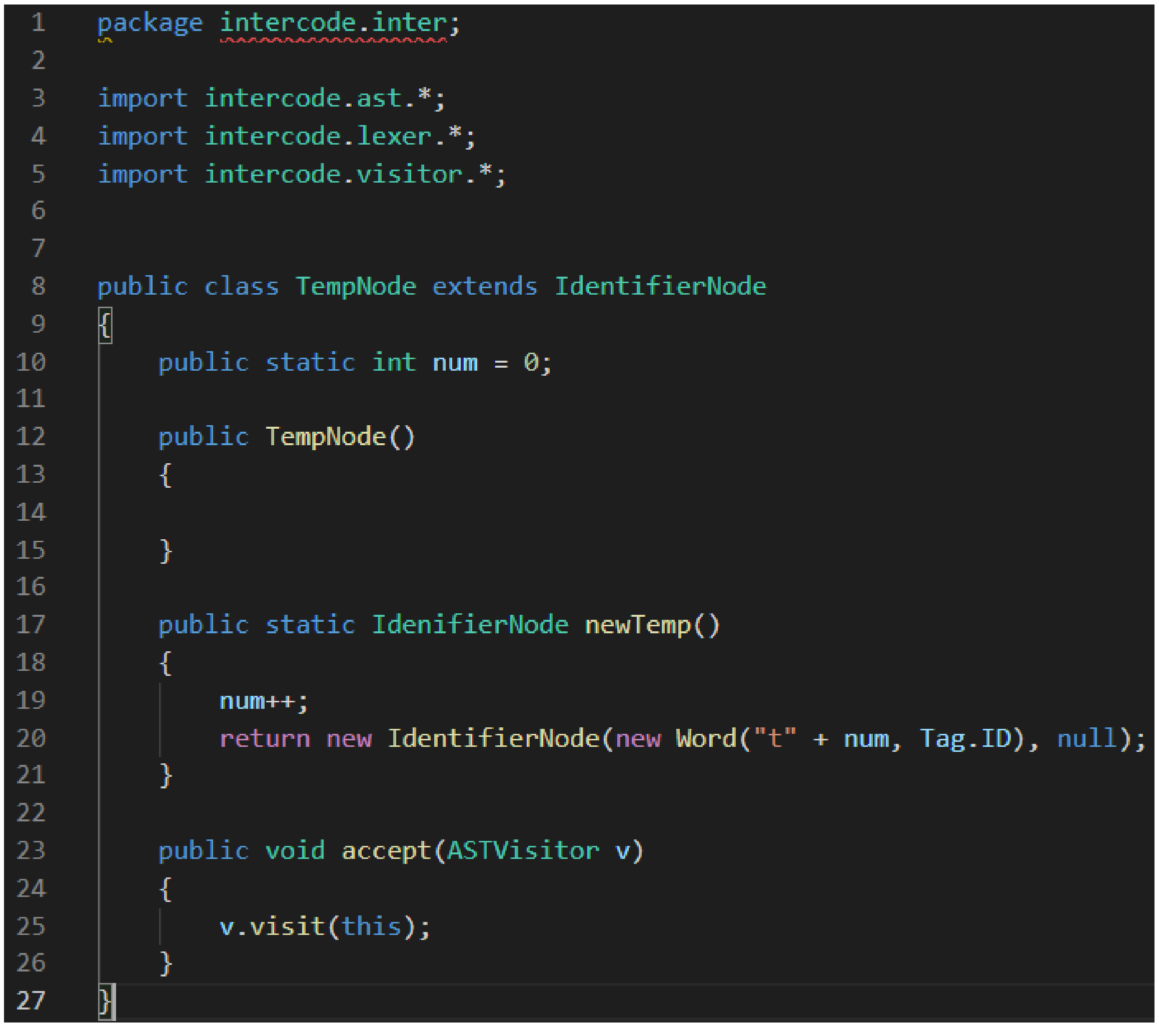
LabelNode.java



```
package intercode.inter;

import intercode.ast.*;
import intercode.lexer.*;
import intercode.visitor.*;


public class LabelNode extends IdentifierNode
{
    static int label = 0;

    public LabelNode(Word word, Type type)
    {
        super(word, type);
    }

    public static LabelNode newLabel()
    {
        label++;
        return new LabelNode(new Word("L" + label, Tag.ID), null);
    }

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

The LabelNode will be used to generate labels used for goto statements in the intermediate language. Every new label will be created with the static method newLabel(). When the new label is generated, it will increment the label counter by one so that no labels can share the same value.

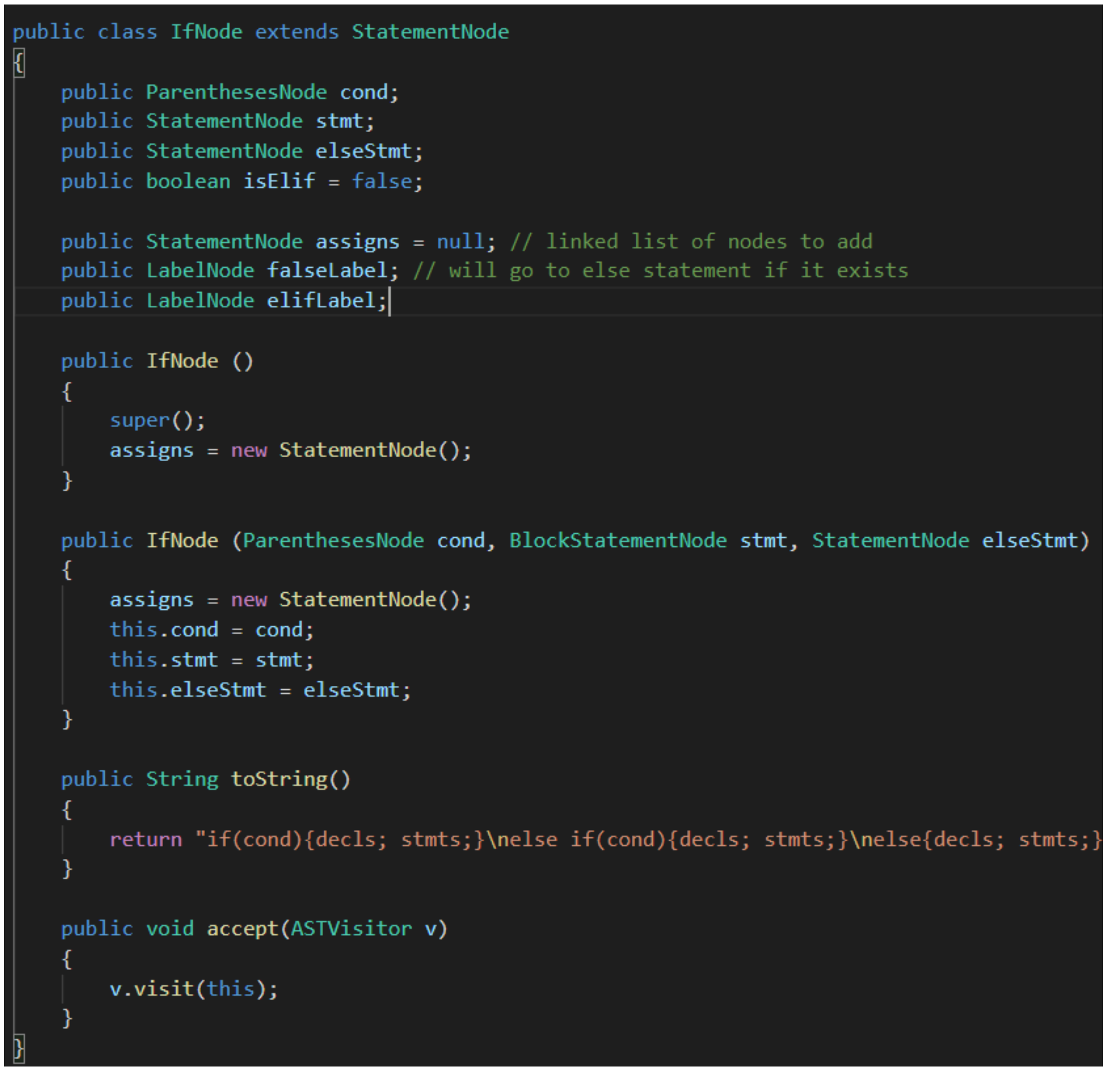
TempNode.java



```java
package intercode.inter;

import intercode.ast.*;
import intercode.lexer.*;
import intercode.visitor.*;


public class TempNode extends IdentifierNode
{
    public static int num = 0;

    public TempNode()
    {

    }

    public static IdenifierNode newTemp()
    {
        num++;
        return new IdentifierNode(new Word("t" + num, Tag.ID), null);
    }

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

The TempNode works very similar to the label node. However, it is responsible for storing a temporary value. By doing this, a binary expression can have the result of the expression stored into the temp variable, which can be used again with another binary expression. When this is done, very complex expressions can be turned into three address code.

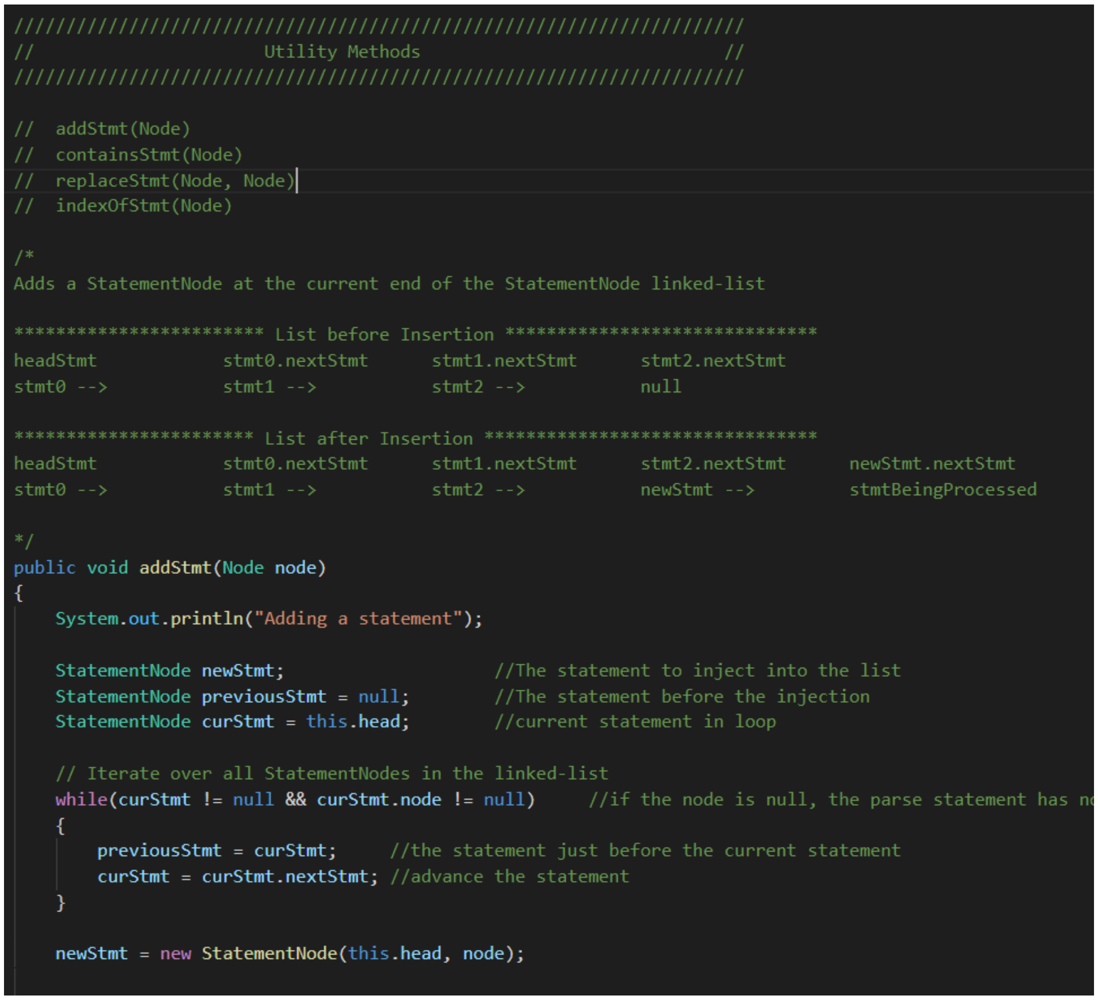
IfNode.java



```java
public class IfNode extends StatementNode
{
    public ParenthesesNode cond;
    public StatementNode stmt;
    public StatementNode elseStmt;
    public boolean isElif = false;

    public StatementNode assigns = null; // linked list of nodes to add
    public LabelNode falseLabel; // will go to else statement if it exists
    public LabelNode elifLabel;

    public IfNode ()
    {
        super();
        assigns = new StatementNode();
    }

    public IfNode (ParenthesesNode cond, BlockStatementNode stmt, StatementNode elseStmt)
    {
        assigns = new StatementNode();
        this.cond = cond;
        this.stmt = stmt;
        this.elseStmt = elseStmt;
    }

    public String toString()
    {
        return "if(cond){decls; stmts;}\nelse if(cond){decls; stmts;}\nelse{decls; stmts;}
    }

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

The IfNode has now been modified to store two LabelNodes. This will allow the InterCode phase to created links to “if”, “else if”, and “else” statements correctly. The labels assigned to the statements of the IfNode will be used strategically in the unparsing phase to direct the program counter to the correct label when conditions are met in a conditional expression or at the end of statement following a conditional expression. Additionally, the “assigns” variable is newly added to store a linked-list of the statements that will compose the three-address-code of the conditional expression.

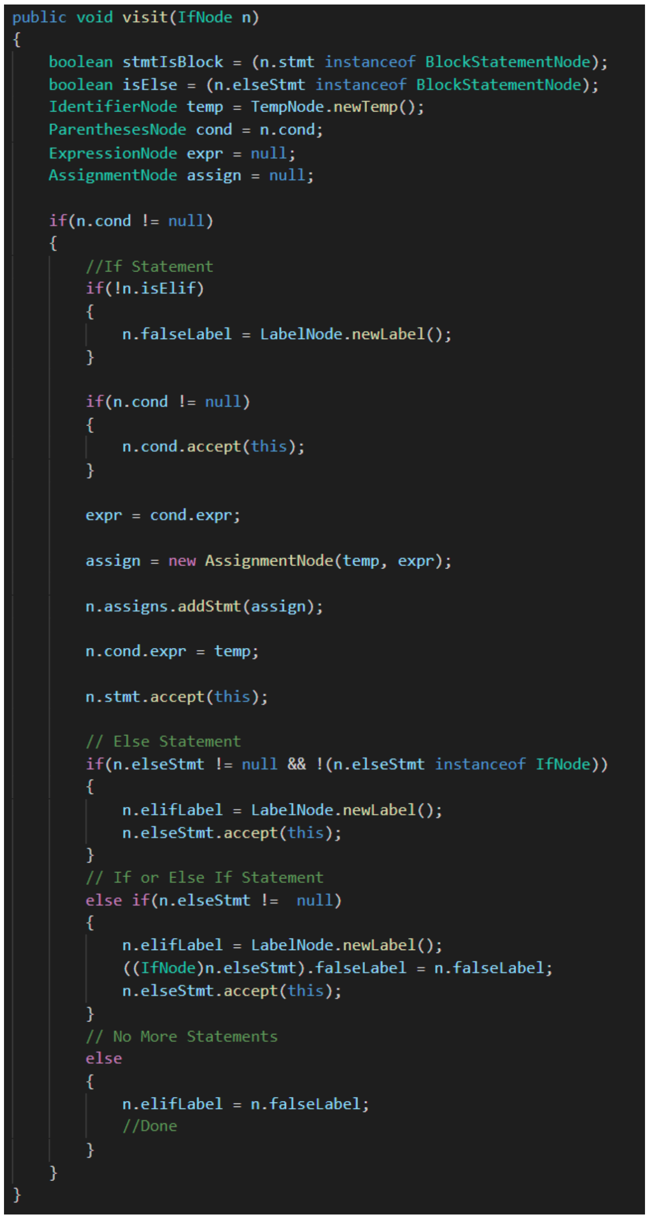
StatementNode.java



```
/////////////////////////////////////////////////////////////////////
//                      Utility Methods                            //
/////////////////////////////////////////////////////////////////////

//  addStmt(Node)
//  containsStmt(Node)
//  replaceStmt(Node, Node)
//  indexOfStmt(Node)

/*
Adds a StatementNode at the current end of the StatementNode linked-list

*********************** List before Insertion *****************************
headStmt            stmt0.nextStmt      stmt1.nextStmt      stmt2.nextStmt
stmt0 -->           stmt1 -->           stmt2 -->           null

********************** List after Insertion *******************************
headStmt            stmt0.nextStmt      stmt1.nextStmt      stmt2.nextStmt      newStmt.nextStmt
stmt0 -->           stmt1 -->           stmt2 -->           newStmt -->         stmtBeingProcessed

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

    StatementNode newStmt;                  //The statement to inject into the list
    StatementNode previousStmt = null;      //The statement before the injection
    StatementNode curStmt = this.head;      //current statement in loop

    // Iterate over all StatementNodes in the linked-list
    while(curStmt != null && curStmt.node != null)     //if the node is null, the parse statement has n
    {
        previousStmt = curStmt;     //the statement just before the current statement
        curStmt = curStmt.nextStmt; //advance the statement
    }

    newStmt = new StatementNode(this.head, node);
```

The StatementNodes now have the utility methods that were formerly in the parsing phase moved into the StatementNode itself. This allows the StatementNodes to be used as a singly linked list in any phase of the compilation.

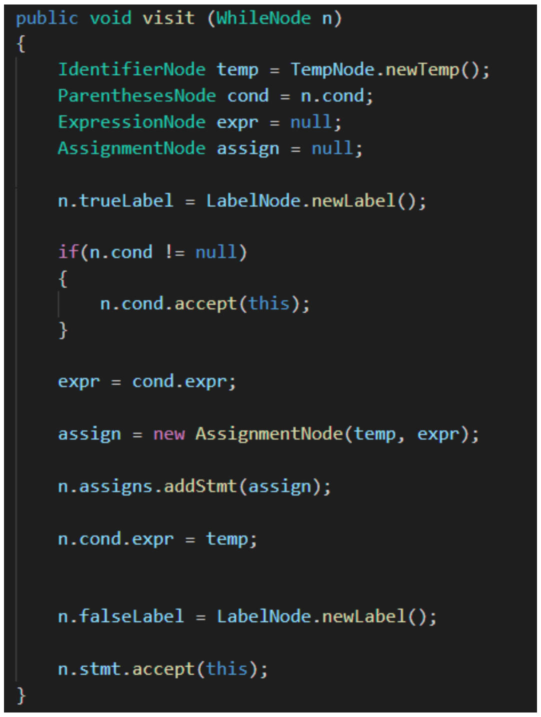
InterCode.java: IfNode



```java
public void visit(IfNode n)
{
    boolean stmtIsBlock = (n.stmt instanceof BlockStatementNode);
    boolean isElse = (n.elseStmt instanceof BlockStatementNode);
    IdentifierNode temp = TempNode.newTemp();
    ParenthesesNode cond = n.cond;
    ExpressionNode expr = null;
    AssignmentNode assign = null;

    if(n.cond != null)
    {
        //If Statement
        if(!n.isElif)
        {
            n.falseLabel = LabelNode.newLabel();
        }

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

        expr = cond.expr;

        assign = new AssignmentNode(temp, expr);

        n.assigns.addStmt(assign);

        n.cond.expr = temp;

        n.stmt.accept(this);

        // Else Statement
        if(n.elseStmt != null && !(n.elseStmt instanceof IfNode))
        {
            n.elifLabel = LabelNode.newLabel();
            n.elseStmt.accept(this);
        }
        // If or Else If Statement
        else if(n.elseStmt !=  null)
        {
            n.elifLabel = LabelNode.newLabel();
            ((IfNode)n.elseStmt).falseLabel = n.falseLabel;
            n.elseStmt.accept(this);
        }
        // No More Statements
        else
        {
            n.elifLabel = n.falseLabel;
            //Done
        }
    }
}
```

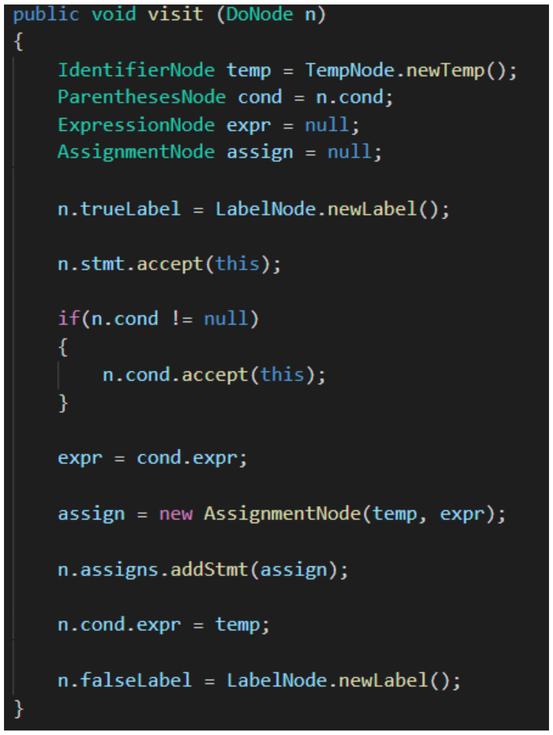
The newly created Intercode.java file contains methods like the unparser’s methods. However, the methods have been modified to not output anything, and insert new statements into the “IfNode”, “WhileNode’, and “DoNode”. These inserted statements will create three address code for determining the conditional statement. Additionally, labels will be added to the IfNode for the end of the statement and for each “else if” or “else” statement.

InterCode.java: WhileNode



```
public void visit (WhileNode n)
{
    IdentifierNode temp = TempNode.newTemp();
    ParenthesesNode cond = n.cond;
    ExpressionNode expr = null;
    AssignmentNode assign = null;

    n.trueLabel = LabelNode.newLabel();

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

    expr = cond.expr;

    assign = new AssignmentNode(temp, expr);

    n.assigns.addStmt(assign);

    n.cond.expr = temp;


    n.falseLabel = LabelNode.newLabel();

    n.stmt.accept(this);
}
```

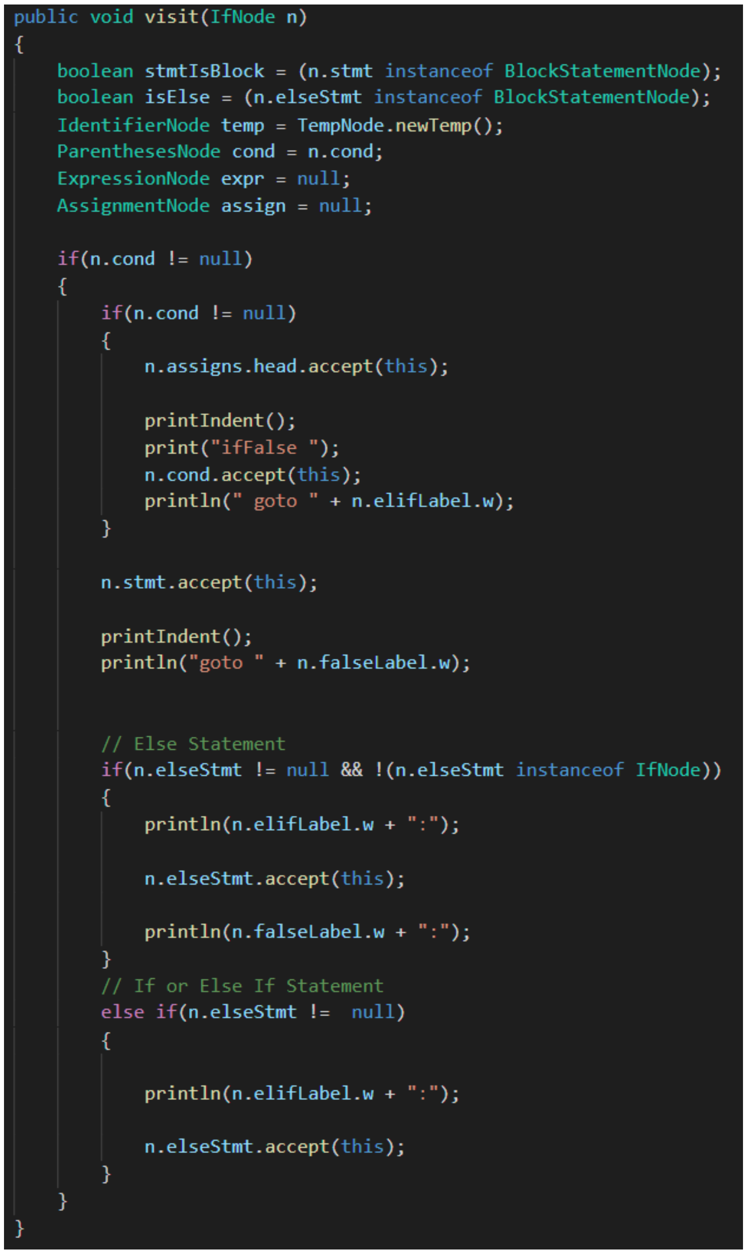
InterCode.java: DoNode



```
public void visit (DoNode n)
{
    IdentifierNode temp = TempNode.newTemp();
    ParenthesesNode cond = n.cond;
    ExpressionNode expr = null;
    AssignmentNode assign = null;

    n.trueLabel = LabelNode.newLabel();

    n.stmt.accept(this);

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

    expr = cond.expr;

    assign = new AssignmentNode(temp, expr);

    n.assigns.addStmt(assign);

    n.cond.expr = temp;

    n.falseLabel = LabelNode.newLabel();
}
```

The WhileNode and DoNode work like the IfNode, but simpiler. It turns the conditional expression into a temporary variable for making three address code. Then labels are created for the beginning and end of the Node if necessary.

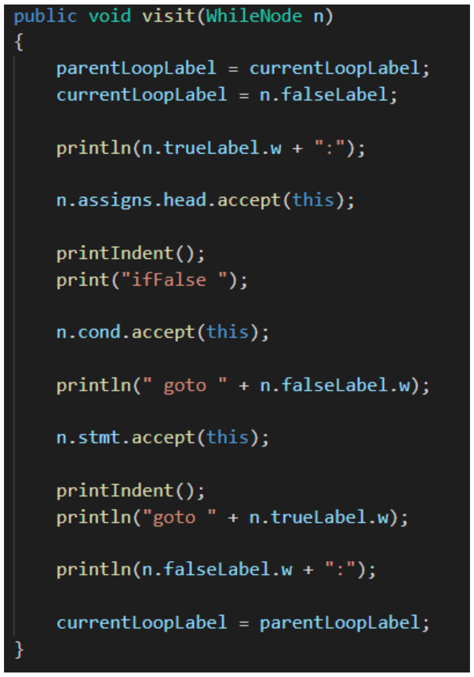
Unparser.java: IfNode



```java
public void visit(IfNode n)
{
    boolean stmtIsBlock = (n.stmt instanceof BlockStatementNode);
    boolean isElse = (n.elseStmt instanceof BlockStatementNode);
    IdentifierNode temp = TempNode.newTemp();
    ParenthesesNode cond = n.cond;
    ExpressionNode expr = null;
    AssignmentNode assign = null;

    if(n.cond != null)
    {
        if(n.cond != null)
        {
            n.assigns.head.accept(this);

            printIndent();
            print("ifFalse ");
            n.cond.accept(this);
            println(" goto " + n.elifLabel.w);
        }

        n.stmt.accept(this);

        printIndent();
        println("goto " + n.falseLabel.w);


        // Else Statement
        if(n.elseStmt != null && !(n.elseStmt instanceof IfNode))
        {
            println(n.elifLabel.w + ":");

            n.elseStmt.accept(this);

            println(n.falseLabel.w + ":");
        }
        // If or Else If Statement
        else if(n.elseStmt !=  null)
        {

            println(n.elifLabel.w + ":");

            n.elseStmt.accept(this);
        }
    }
}
```

The visit method for the IfNode in the Unparser is more complicated than the others. This is because the method must differentiate between an “if”, “else if”, and “else” statement. The “if” and “else if” statement require a goto command after the conditional statement executes that moves the program counter to the end of all the statements. The else statement does not require a goto at the end of its statement because the program counter will automatically advance to the next statement. When a conditional expression is false, the program counter will advance to the next conditional expression for an “else if” statement.

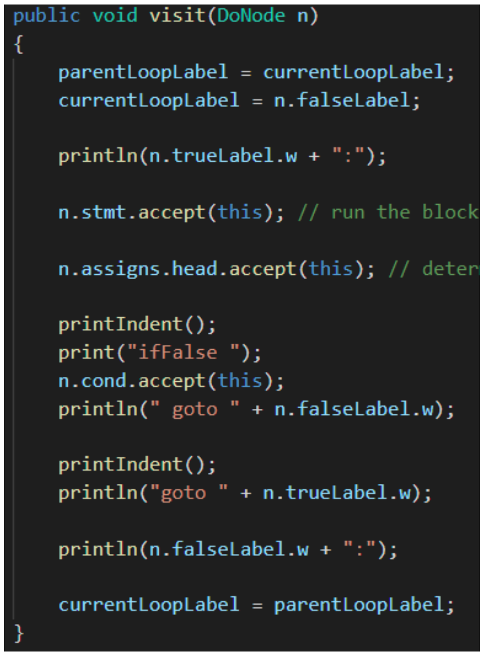
Unparser.java: WhileNode



```
public void visit(WhileNode n)
{
    parentLoopLabel = currentLoopLabel;
    currentLoopLabel = n.falseLabel;

    println(n.trueLabel.w + ":");

    n.assigns.head.accept(this);

    printIndent();
    print("ifFalse ");

    n.cond.accept(this);

    println(" goto " + n.falseLabel.w);

    n.stmt.accept(this);

    printIndent();
    println("goto " + n.trueLabel.w);

    println(n.falseLabel.w + ":");

    currentLoopLabel = parentLoopLabel;
}
```

The WhileNode has a start and end label so that the program can either continue to loop or break out of the loop by skipping past the last goto statement that moves the program counter to the beginning of the statement.

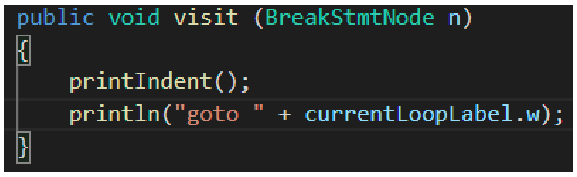
Unparser.java: DoNode



```
public void visit(DoNode n)
{
    parentLoopLabel = currentLoopLabel;
    currentLoopLabel = n.falseLabel;

    println(n.trueLabel.w + ":");

    n.stmt.accept(this); // run the block

    n.assigns.head.accept(this); // deter

    printIndent();
    print("ifFalse ");
    n.cond.accept(this);
    println(" goto " + n.falseLabel.w);

    printIndent();
    println("goto " + n.trueLabel.w);

    println(n.falseLabel.w + ":");

    currentLoopLabel = parentLoopLabel;
}
```

The DoNode behaves exactly like the WhileNode, except that the condition isn’t evaluated until all statements in the block have been read. Then whether the program counter moves to the top label or bottom label is determined.

Unparser.java: BreakStmtNode

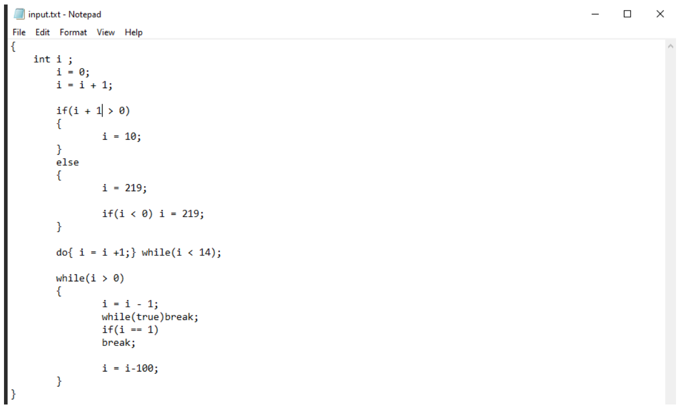


```java
public void visit (BreakStmtNode n)
{
    printIndent();
    println("goto " + currentLoopLabel.w);
}
```

The BreakStmtNode is very simple, and simply directs the program counter to the label outside of the enclosing loop, which is stored like how the enclosing block is stored in the parser. Whenever a loop is visited, the false label for the loop is stored to the enclosing loop and the parent loop is stored as well. Therefore, nested loops should be able to be escaped.

Execution

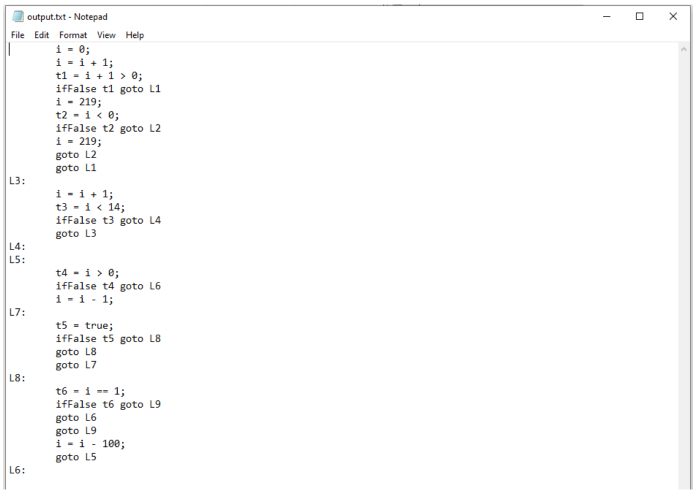
Input.txt



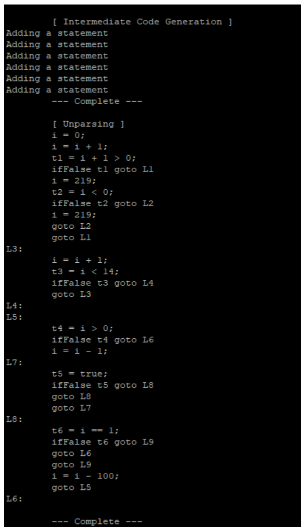
```
{
    int i ;
        i = 0;
        i = i + 1;

        if(i + 1 > 0)
        {
                i = 10;
        }
        else
        {
                i = 219;

                if(i < 0) i = 219;
        }

        do{ i = i +1;} while(i < 14);

        while(i > 0)
        {
                i = i - 1;
                while(true)break;
                if(i == 1)
                break;

                i = i-100;
        }
}
```

The Input file test the compiler's ability to handle "do", "while", and "if" statements. Also, the break statement is tested to ensure it points to the correct label.

Output.txt



```
        i = 0;
        i = i + 1;
        t1 = i + 1 > 0;
        ifFalse t1 goto L1
        i = 219;
        t2 = i < 0;
        ifFalse t2 goto L2
        i = 219;
        goto L2
        goto L1
L3:
        i = i + 1;
        t3 = i < 14;
        ifFalse t3 goto L4
        goto L3
L4:
L5:
        t4 = i > 0;
        ifFalse t4 goto L6
        i = i - 1;
L7:
        t5 = true;
        ifFalse t5 goto L8
        goto L8
        goto L7
L8:
        t6 = i == 1;
        ifFalse t6 goto L9
        goto L6
        goto L9
        i = i - 100;
        goto L5
L6:
```



```
        [ Intermediate Code Generation ]
Adding a statement
Adding a statement
Adding a statement
Adding a statement
Adding a statement
Adding a statement
        --- Complete ---

        [ Unparsing ]
        i = 0;
        i = i + 1;
        t1 = i + 1 > 0;
        ifFalse t1 goto L1
        i = 219;
        t2 = i < 0;
        ifFalse t2 goto L2
        i = 219;
        goto L2
        goto L1
L3:
        i = i + 1;
        t3 = i < 14;
        ifFalse t3 goto L4
        goto L3
L4:
L5:
        t4 = i > 0;
        ifFalse t4 goto L6
        i = i - 1;
L7:
        t5 = true;
        ifFalse t5 goto L8
        goto L8
        goto L7
L8:
        t6 = i == 1;
        ifFalse t6 goto L9
        goto L6
        goto L9
        i = i - 100;
        goto L5
L6:

        --- Complete ---
```

The labels and goto statements all are positioned correctly and allow the program to flow correctly.

Conclusion

This lab showed me important techniques used for converting the code that has gone through the lexing, parsing, and type checking phases into an intermediate like code in the intercode and unparsing phase. The most challenging part was getting the labels to work and display correctly within the IfNode. The break statement also required some thought before determining a fairly simple way to make it work. My arrays still need some work and are not fully functional, so I was unable to test them well with the intermediate code.