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

```
3   public class Tag
4   {
5       //null
6       public final static int NULL    = 0;
7
8       //boolean
9       public final static int TRUE    = 300;
10      public final static int FALSE   = 301;
11
12      //terminals, variables, types
13      public final static int ID      = 400;
14      public final static int NUM     = 401;
15      public final static int BASIC   = 402;
16      public final static int REAL    = 403;
17      public final static int INDEX   = 404;
18
19      //loops
20      public final static int DO      = 500;
21      public final static int WHILE   = 501;
22      public final static int FOR     = 502;
23      public final static int BREAK   = 503;
24      public final static int CONTINUE= 504;
25
26      //conditional
27      public final static int IF      = 600;
28      public final static int ELSE    = 601;
29      public final static int SWITCH  = 602;
30
31      //comparison
32      public final static int AND     = 700; // &&
33      public final static int OR      = 701; // ||
34      public final static int EQ      = 702; // ==
35      public final static int NE      = 703; // !=
36      public final static int GE      = 704; // >=
37      public final static int LE      = 705; // <=
38      public final static int LT      = 706; // <
39      public final static int GT      = 707; // >
40  }
```

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

ASTVisitor.java

```
87      public void visit (ParenthesesNode n)
88      {
89          n.expr.accept(this);
90      }
91
92      public void visit (IfNode n)
93      {
94          if(n.cond != null)
95          {
96              n.cond.accept(this);
97          }
98          if(n.stmt != null)
99          {
100             n.stmt.accept(this);
101         }
102         if(n.elseStmt != null)
103         {
104             n.elseStmt.accept(this);
105         }
106     }
107
108     public void visit (WhileNode n)
109     {
110         if(n.cond != null)
111         {
112             n.cond.accept(this);
113         }
114         if(n.stmt != null)
115         {
116             n.stmt.accept(this);
117         }
118     }
119
120     public void visit (DoNode n)
121     {
122         if(n.stmt != null)
123         {
124             n.stmt.accept(this);
125         }
126         if(n.cond != null)
127         {
128             n.cond.accept(this);
129         }
130     }
131
132     public void visit (ArrayAccessNode n)
133     {
134
135     }
136
137     public void visit (ArrayDimsNode n)
138     {
139         n.size.accept(this);
140
141         if(n.dim != null)
142         {
143             n.dim.accept(this);
144         }
145     }
146
147     public void visit (BreakStmtNode n)
148     {
149
150     }
151
152     public void visit (TrueNode n)
153     {
154
155     }
156
157     public void visit (FalseNode n)
158     {
159
160     }
```

The visitor now has the added nodes listed and functional.

ArrayAccessNode.java.

```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

```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

```
431     public void visit (StatementsNode n)
432     {
433         if (look.tag != '}')
434         {
435             level++;
436             n.stmt = new parseStatementNode(n.stmt);
437             n.stmt.accept(this);
438 
439             n.stmts = new StatementsNode();
440             n.stmts.accept(this);
441             level--;
442         }
443     }
444 
445     public StatementNode parseStatementNode (StatementNode stmt)
446     {
447         dots();
448         System.out.println("---parseStatementNode---");
449 
450         switch(look.tag)
451         {
452             //Tag.NUM and Tag.REAL are not options since it would
453             //to change a terminal literal's value
454             case Tag.ID :
455                 stmt = new AssignmentNode(); // Word = bool; | Wor
456                 break;
457             case Tag.IF :
458                 stmt = new IfNode(); // if ( bool ) blockstmt | if
459                 break;
460             case Tag.WHILE :
461                 stmt = new WhileNode(); // while ( bool ) blockstm
462                 break;
463             case Tag.DO :
464                 stmt = new DoNode(); // do blockstmt while ( bool
465                 break;
466             case Tag.BREAK : // break ;
467                 stmt = new BreakStmtNode();
468                 break;
469             case Tag.FOR :              // for ( AssignmentNode; B
470                 stmt = new ForNode();
471                 break;
472             case '{' :
473                 n.node = new BlockStatementNode(); // blockstmt
474                 break;
475             default :
476                 break;
477             //The break and continue tags should be handled in the
478         }
479 
480         stmt.accept(this); //May need a cast
481         return stmt;
482     }
```

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

parser.java: ParenthesesNode

```
484     public void visit (ParenthesesNode n)
485     {
486         dots();
487         System.out.println("ParenthesesNode");
488 
489         match('(');
490 
491         level++;
492         switch(look.tag)
493         {
494             case ')' :
495                 n.expr = new ParenthesesNode();
496                 n.expr.accept(this);
497                 break;
498             case Tag.ID :
499                 n.expr = new IdentifierNode();
500                 n.expr.accept(this);
501                 break;
502             case '[' :
503                 n.expr = parseArrayAccessNode((IdentifierNode)n.expr);
504                 break;
505             case Tag.NUM :
506                 n.expr = new NumNode();
507                 n.expr.accept(this);
508                 break;
509             case Tag.REAL :
510                 n.expr = new RealNode();
511                 n.expr.accept(this);
512                 break;
513             case Tag.TRUE :
514                 n.expr = new TrueNode();
515                 n.expr.accept(this);
516                 break;
517             case Tag.FALSE :
518                 n.expr = new FalseNode();
519                 n.expr.accept(this);
520                 break;
521             default :
522                 break;
523         }
524 
525         if(look.tag != ')')
526         {
527             n.expr = parseBinaryNode(n.expr, 0);
528         }
529         level--;
530     }
```

This node will execute whatever is in a parenthesized statement.

Parser.java: IfNode

```
767     public void visit(IfNode n)
768     {
769         dots();
770         System.out.println("IfNode");
771 
772         level++;
773 
774         if(look.tag == Tag.IF)
775         {
776             match(Tag.IF);
777 
778             n.cond = new ParenthesesNode();
779             n.cond.accept(this);
780 
781             if(look.tag == '{')
782             {
783                 n.stmt = new BlockStatementNode();
784                 n.stmt.accept(this);
785             }
786             else
787             {
788                 n.stmt = parseStatementNode(n.stmt);
789             }
790         }
791         else if(look.tag == Tag.ELSE)
792         {
793             match(Tag.ELSE);
794 
795             if(look.tag == '{')
796             {
797                 n.stmt = new BlockStatementNode();
798                 n.stmt.accept(this);
799             }
800             else
801             {
802                 n.stmt = parseStatementNode(n.stmt);
803             }
804         }
805         else
806         {
807             error("IfNode: unrecognized command");
808         }
809 
810         level--;
811     }
```

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

```
693     public void visit(WhileNode n)
694     {
695         dots();
696         System.out.println("WhileNode");
697         level++;
698 
699         match(Tag.WHILE);
700 
701         n.cond = new ParenthesesNode();
702         n.cond.accept(this);
703 
704         if(look.tag == '{')
705         {
706             n.stmt = new BlockStatementNode();
707             n.stmt.accept(this);
708         }
709         else
710         {
711             n.stmt = parseStatementNode(n.stmt);
712         }
713         level--;
714     }
```

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

Parser.java: DoNode

```
672     public void visit(DoNode n)
673     {
674         dots();
675         System.out.println("DoNode");
676 
677         match(Tag.DO);
678 
679         if(look.tag == '{')
680         {
681             level++;
682             n.stmt = new BlockStatementNode();
683             n.stmt.accept(this);
684             level--;
685         }
686         else
687         {
688             n.stmt = parseStatementNode(n.stmt);
689         }
690 
691         match(Tag.WHILE);
692 
693         level++;
694         n.cond = new ParenthesesNode();
695         n.cond.accept(this);
696         level--;
697 
698         match(';');
699     }
```

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

```
484     public void visit (ParenthesesNode n)
485     {
486         dots();
487         System.out.println("ParenthesesNode");
488 
489         match('(');
490 
491         level++;
492         switch(look.tag)
493         {
494             case ')' :
495                 n.expr = new ParenthesesNode();
496                 n.expr.accept(this);
497                 break;
498             case Tag.ID :
499                 n.expr = new IdentifierNode();
500                 n.expr.accept(this);
501 
502                 if(look.tag == '[')
503                 {
504                     level--;
505                     n.expr = parseArrayAccessNode((IdentifierNode)n.expr);
506                     level++;
507                 }
508                 break;
509             case Tag.NUM :
510                 n.expr = new NumNode();
511                 n.expr.accept(this);
512                 break;
513             case Tag.REAL :
514                 n.expr = new RealNode();
515                 n.expr.accept(this);
516                 break;
517             case Tag.TRUE :
518                 n.expr = new TrueNode();
519                 n.expr.accept(this);
520                 break;
521             case Tag.FALSE :
522                 n.expr = new FalseNode();
523                 n.expr.accept(this);
524                 break;
525             default :
526                 break;
527         }
528         level--;
529 
530         if(look.tag != ')')
531         {
532             n.expr = parseBinaryNode(n.expr, 0);
533         }
534 
535         match(')');
536     }
```

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

Parser.java: ArrayElements

```
430     //Uses parseArrayAccessNode instead
431     public void visit (ArrayAccessNode n)
432     {
433         dots();
434         System.out.println("ArrayAccessNode");
435     }
436
437     public void visit (ArrayDimsNode n)
438     {
439         dots();
440         System.out.println("ArrayDimsNode");
441
442         match('[');
443
444         ExpressionNode index = null;
445
446         level++;
447         switch (look.tag)
448         {
449             case '(':
450                 index = new ParenthesesNode();
451                 break;
452             case Tag.ID:
453                 index = new IdentifierNode();
454                 break;
455             case Tag.NUM:
456                 index = new NumNode();
457                 break;
458             default:
459                 break;
460         }
461         index.accept(this);
462         level--;
463
464         if (look.tag != ']')
465         {
466             level++;
467             index = new parseBinaryNode(index, 0);
468             level--;
469         }
470
471         n.size = index;
472
473         if (look.tag == '[')
474         {
475             level++;
476             n.dim = new ArrayDimsNode();
477             n.dim.accept(this);
478             level--;
479         }
480     }
481
482     public ExpressionNode parseArrayAccessNode (IdentifierNode id)
483     {
484         dots();
485         System.out.println("---parseArrayAccessNode---");
486
487         level++;
488         ExpressioNode index = new ArrayDimsNode();
489         index.accept(this);
490         level--;
491
492         return new ArrayAccessNode(id, index);
493     }
```

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

```
677     public void visit(AssignmentNode n)
678     {
679         dots();
680         System.out.println("AssignmentNode");
681 
682         if(n.left == null) // id
683         {
684             level++;
685             n.left = new IdentifierNode((Word)look);
686             n.left.accept(this);
687             level--;
688         }
689 
690         //The array, n.left.array needs to be indexed
691         if (look.tag == '[') // loc [ bool ] | loc [ bool ] = bool
692         {
693             n.left = parseArrayAccessNode((IdentifierNode)n.left);
694         }
695 
696         if(look.tag == '=') // loc = bool
697         {
698             level++;
699             dots();
700             n.op = look;
701             System.out.println("Op: " + n.op);
702             match('=');
703 
704 
705             ExpressionNode rhsAssign = null;
706 
707             if(look.tag == '(')
708             {
709                 rhsAssign = new ParenthesesNode();
710                 rhsAssign.accept(this);
711             }
712             else if (look.tag == Tag.ID)
713             {
714                 rhsAssign = new IdentifierNode();
715                 rhsAssign.accept(this);
716 
717                 if (look.tag == '[')
718                 {
719                     level--;
720                     rhsAssign = parseArrayAccessNode((IdentifierNode)rhsAssign);
721                     level++;
722                 }
723             }
724             else if (look.tage == Tag.NUM)
725             {
726                 rhsAssign = new NumNode();
727                 rhsAssign.accept(this);
728             }
729             else if (look.tage == Tag.REAL)
730             {
731                 rhsAssign = new RealNode();
732                 rhsAssign.accept(this);
733             }
734             level--;
735 
736 
737             if(look.tag == ';')
738             {
739                 n.right = rhsAssign;
740             }
741             else
742             {
743                 dots();
744                 System.out.println("Op: " + n.op);
745 
746                 level++;
747                 n.right = (BinaryNode)parseBinExprNode(rhsAssign, 0);
748                 level--;
749             }
750             
751             match(';');
752 
753             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

```
1234    public void visit (BreakStmtNode n)
1235    {
1236        dots();
1237        System.out.println("BreakStmtNode");
1238
1239        match(Tag.BREAK);
1240        match(';');
1241    }
```

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

Parser.java: BinaryNode

```
1138    public void visit (BinaryNode n)
1139    {
1140        dots();
1141        System.out.println("BinaryNode");
1142
1143        level++;
1144        switch (look.tag)
1145        {
1146            case '(':
1147                n.left = new ParenthesesNode();
1148                break;
1149            case Tag.ID:
1150                n.left = new IdentifierNode();
1151
1152                if (look.tag == '[')
1153                {
1154                    level--;
1155                    n.left = parseArrayAccessNode((IdentifierNode)n.left);
1156                    level++;
1157                }
1158                break;
1159            case Tag.NUM:
1160                n.left = new NumNode();
1161                break;
1162            case Tag.REAL:
1163                n.left = new RealNode();
1164            default:
1165                break;
1166        }
1167
1168        BinaryNode binary = parseBinaryNode(n.left, 0);
1169        n.op = binary.op;
1170        n.right = binary.right;
1171        level--;
1172    }
```

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

```
112     int getPrecedence (int op)
113     {
114         switch(op)
115         {
116             case ')' :
117             case '(' :
118                 return 15;
119             case Tag.POSTINC : // x++
120             case Tag.POSTDEC : // x--
121                 return 14;
122             case '*' :
123             case '/' :
124             case '%' :
125                 return 13;
126             case '+' :
127             case '-' :
128                 return 12;
129             case Tag.RL : // <<
130             case Tag.RR : // >>
131             case Tag.LRR : // >>>
132                 return 11;
133             case '<' :
134             case '>' :
135             case Tag.GE : // >=
136             case Tag.LE : // <=
137                 return 9;
138             case Tag.EQ : // ==
139             case Tag.NE : // !=
140                 return 8;
141             case Tag.BITAND : // and
142             case '&' :
143                 return 7;
144             case '^' :
145             case Tag.XOR : // or
146                 return 6;
147             case Tag.IOR : // |
148                 return 5;
149             case Tag.AND : // &&
150                 return 4;
151             case Tag.OR : // ||
152                 return 3;
153             case Tag.TERNARY :  // comparison? x: y
154                 return 2;
155             case '=' :
156             case Tag.ADDEQ : // +=
157             case Tag.MINEQ : // -=
158             case Tag.MULEQ : // *=
159             case Tag.DIVEQ : // /=
160             case Tag.MODEQ : // %=
161             case Tag.ANDEQ : // &=
162             case Tag.XOREQ : // ^=
163             case Tag.RLEQ : // <<=
164             case Tag.RREQ : // >>=
165             case Tag.LLREQ : // >>>=
166                 return 1;
167             default :
168                 return -1;
169         }
170     }
```

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

```
83      public void visit (BlockState  visit (Par  Aa  ? of 1
84      {
85          dots();
86          println("BlockStatementNode");
87
88          indentUp();
89          if(n.decls != null)
90          {
91              n.decls.accept(this);
92          }
93          if(n.stmts != null)
94          {
95              n.stmts.accept(this);
96          }
97          indentDown();
98      }
99
100     public void visit (DeclarationsNode n)
101     {
102         if(n.decls != null)
103         {
104             dots();
105             println("Declarations");
106
107             indentUp();
108             n.decl.accept(this);
109             indentDown();
110
111             n.decls.accept(this);
112         }
113         if(n.stmts != null)
114         {
115             n.stmts.accept(this);
116         }
117     }
118
119     public void visit (DeclarationNode n)
120     {
121         dots();
122         println("DeclarationNode");
123
124         indentUp();
125         if(n.type != null)
126         {
127             n.type.accept(this);
128             n.id.accept(this);
129
130             if(n.assign != null) //assignment after declaration
131             {
132                 indentUp();
133                 n.assign.accept(this);
134                 indentDown();
135             }
136         }
137         indentDown();
138     }
139
140     public void visit (StatementsNode n)
141     {
142         if(n.stmts != null)
143         {
144             dots();
145             println("Statements");
146
147             indentUp();
148             n.stmt.accept(this);
149             indentDown();
150
151             n.stmts.accept(this);
152         }
153         if(n.decls != null)
154         {
155             indentUp();
156             n.decls.accept(this);
157             indentDown();
158         }
```

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

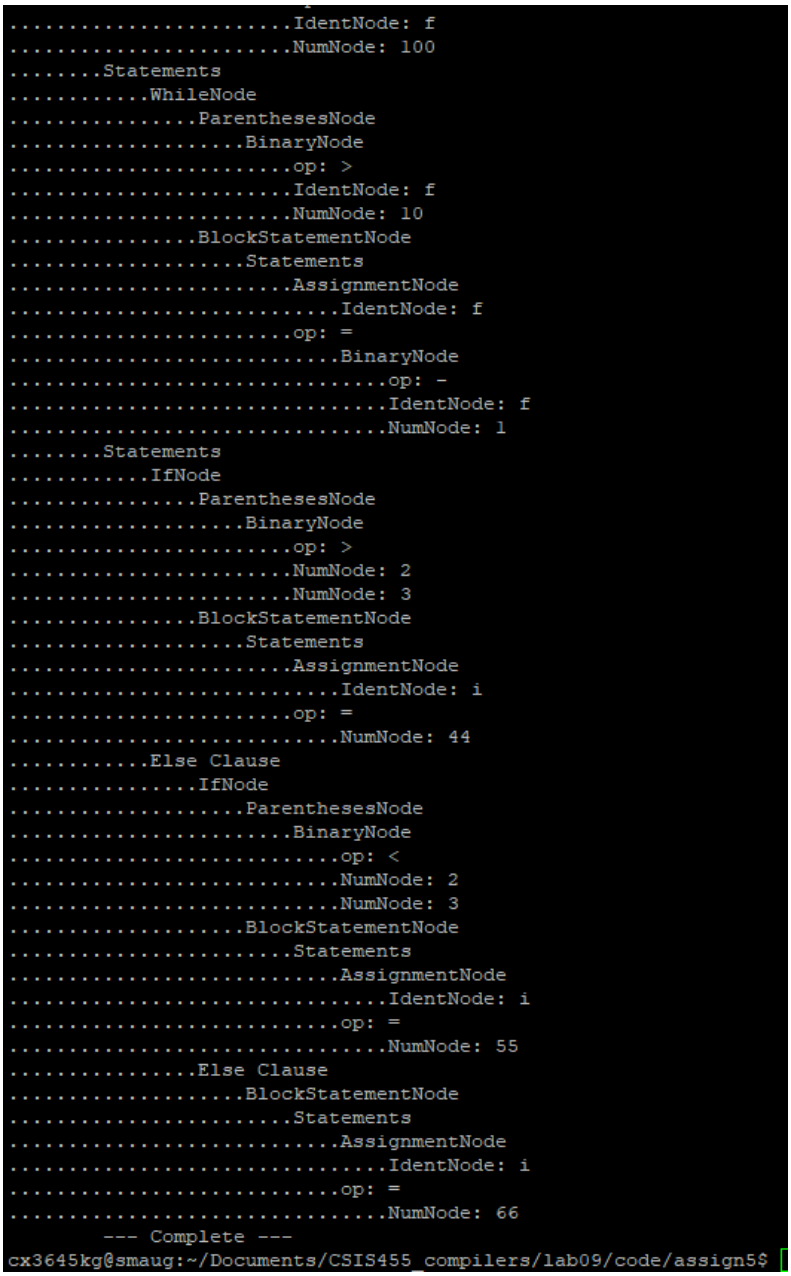
Unparser.java

```
    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.



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

Output.txt

output.txt - Notepad

File Edit Format View Help

```
{
    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.