Paul Kummer

Dr. Hanku Lee

Compilers CSIS 455-01

11/04/2021

Lab 08: Adding Types and Arrays

Purpose of Lab

This lab's purpose is to expand on the last version of the compiler. In this version, declarations are added that will give an identifier a basic type of int, real, Boolean, or char. Also, support for an array is added. By adding support for types, many new nodes have to be added and subsequently, more code has to be modified to support new or removed nodes or attributes.

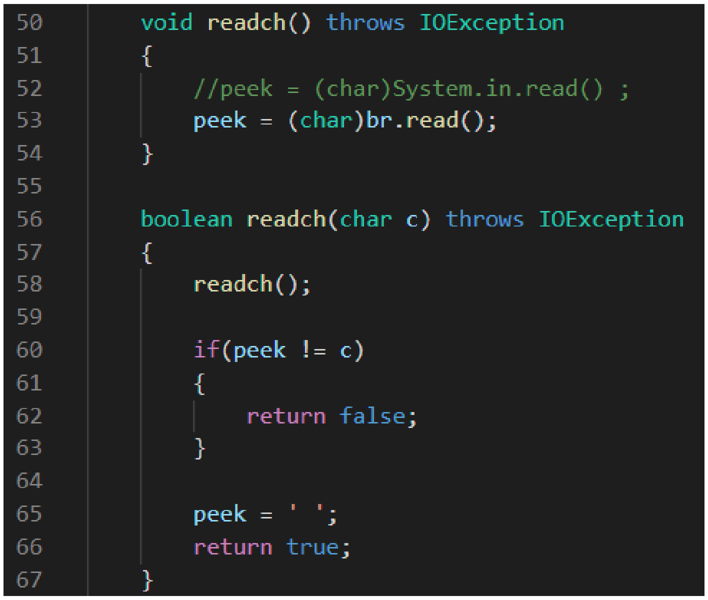
Example Code

The code in Main.java remains unchanged from previous assignments. Many other files have had some changes made to them. For starter, the parser had all the nodes moved to a new folder named "ast". Literals are removed and converted to either num or real in all files too. Additional support for types is added as well. The parser and unparser are modified to support the new nodes.

lexer

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.

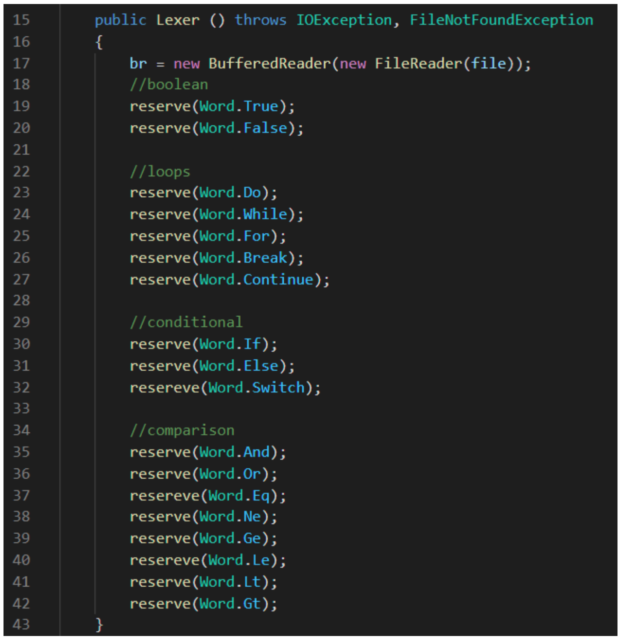
lexer.java



```
    void readch() throws IOException
    {
        //peek = (char)System.in.read() ;
        peek = (char)br.read();
    }

    boolean readch(char c) throws IOException
    {
        readch();

        if(peek != c)
        {
            return false;
        }

        peek = ' ';
        return true;
    }
```

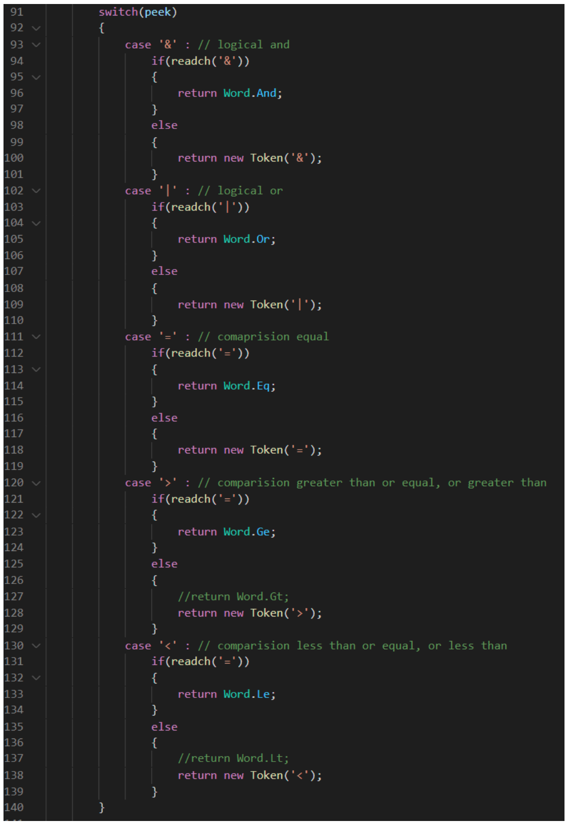
A new readch() method now will return a Boolean that depends on the lookahead.

lexer.java cont.



```
public Lexer () throws IOException, FileNotFoundException
{
    br = new BufferedReader(new FileReader(file));
    //boolean
    reserve(Word.True);
    reserve(Word.False);

    //loops
    reserve(Word.Do);
    reserve(Word.While);
    reserve(Word.For);
    reserve(Word.Break);
    reserve(Word.Continue);

    //conditional
    reserve(Word.If);
    reserve(Word.Else);
    resereve(Word.Switch);

    //comparison
    reserve(Word.And);
    reserve(Word.Or);
    resereve(Word.Eq);
    reserve(Word.Ne);
    reserve(Word.Ge);
    resereve(Word.Le);
    reserve(Word.Lt);
    reserve(Word.Gt);
}
```

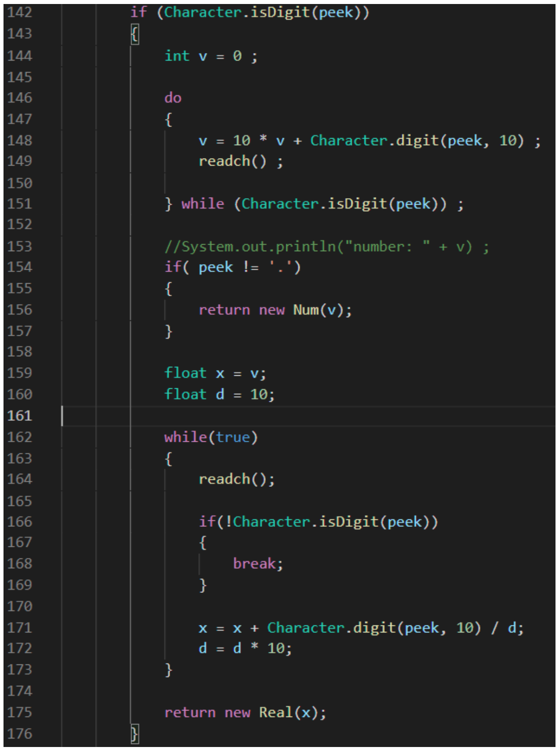
More tokens are reserved to make parsing more efficient for matching the lookahead and to prevent variables from using certain keywords.



```
switch(peek)
{
    case '&' : // logical and
        if(readch('&'))
        {
            return Word.And;
        }
        else
        {
            return new Token('&');
        }
    case '|' : // logical or
        if(readch('|'))
        {
            return Word.Or;
        }
        else
        {
            return new Token('|');
        }
    case '=' : // comaprision equal
        if(readch('='))
        {
            return Word.Eq;
        }
        else
        {
            return new Token('=');
        }
    case '>' : // comparision greater than or equal, or greater than
        if(readch('='))
        {
            return Word.Ge;
        }
        else
        {
            //return Word.Gt;
            return new Token('>');
        }
    case '<' : // comparision less than or equal, or less than
        if(readch('='))
        {
            return Word.Le;
        }
        else
        {
            //return Word.Lt;
            return new Token('<');
        }
}
```

A switch statement will now create tokens of operational significance to make parsing easier.

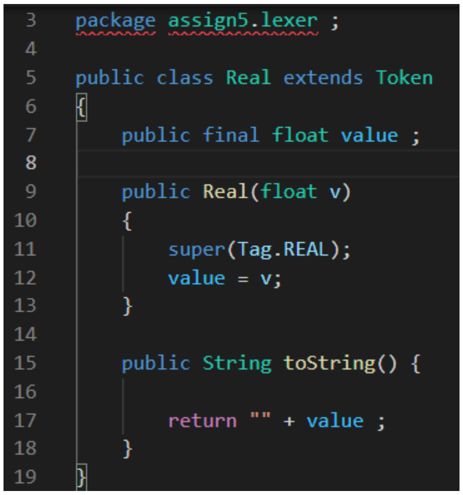
lexer.java cont.



```
        if (Character.isDigit(peek))
        {
            int v = 0 ;

            do
            {
                v = 10 * v + Character.digit(peek, 10) ;
                readch() ;

            } while (Character.isDigit(peek)) ;

            //System.out.println("number: " + v) ;
            if( peek != '.')
            {
                return new Num(v);
            }

            float x = v;
            float d = 10;

            while(true)
            {
                readch();

                if(!Character.isDigit(peek))
                {
                    break;
                }

                x = x + Character.digit(peek, 10) / d;
                d = d * 10;
            }

            return new Real(x);
        }
```

The scanner will now check for a decimal point after a series of digits to determine if the number is a float instead of an int. If there is a decimal point the token will be a Real.

Real.java



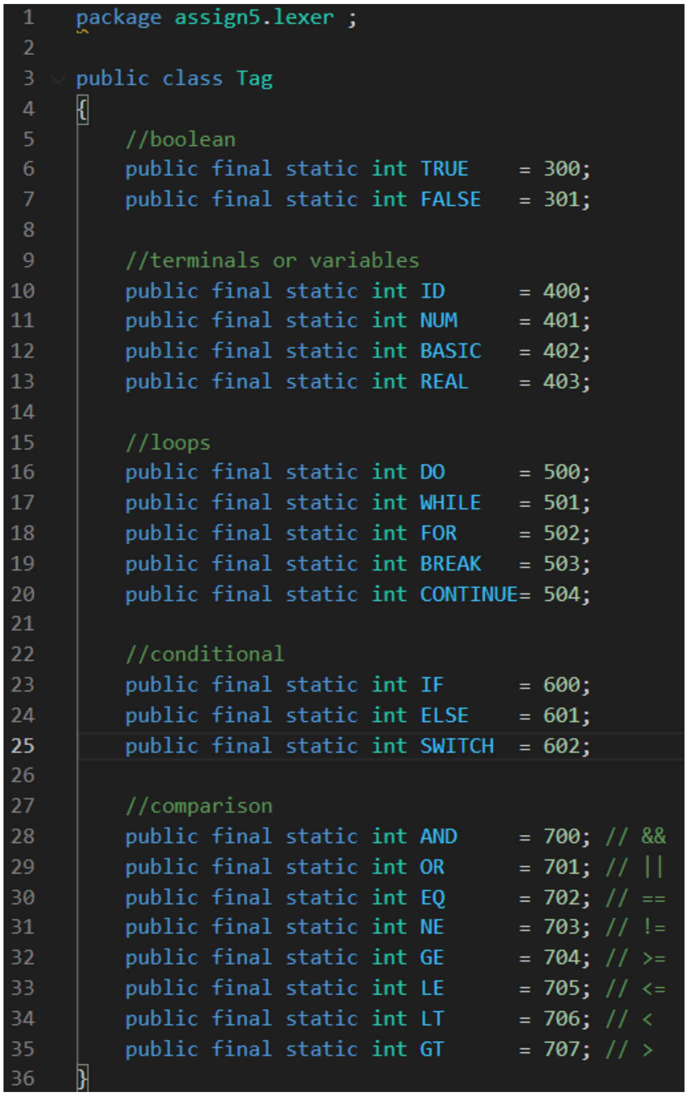
```
package assign5.lexer ;

public class Real extends Token
{
    public final float value ;

    public Real(float v)
    {
        super(Tag.REAL);
        value = v;
    }

    public String toString() {

        return "" + value ;
    }
}
```

The new class Real now will be assigned to any floating point decimal values.

Num.java

The num.java file is the former literal token. It will hold any integer values

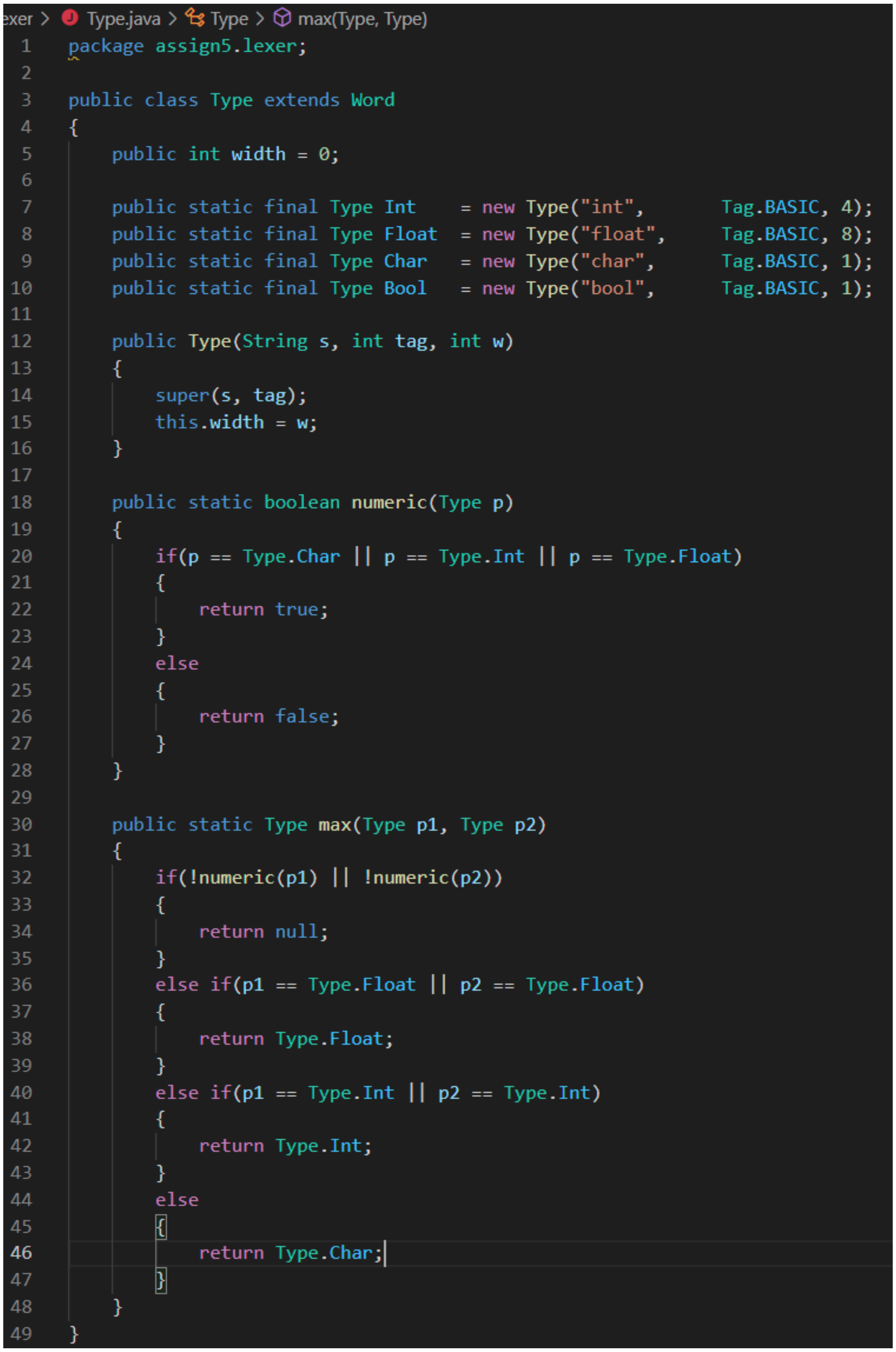
Tag.java



```java
package assign5.lexer ;

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

    //terminals or variables
    public final static int ID      = 400;
    public final static int NUM     = 401;
    public final static int BASIC   = 402;
    public final static int REAL    = 403;

    //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; // >
}
```

The tag values have been updated to support many more options within a program’s context.

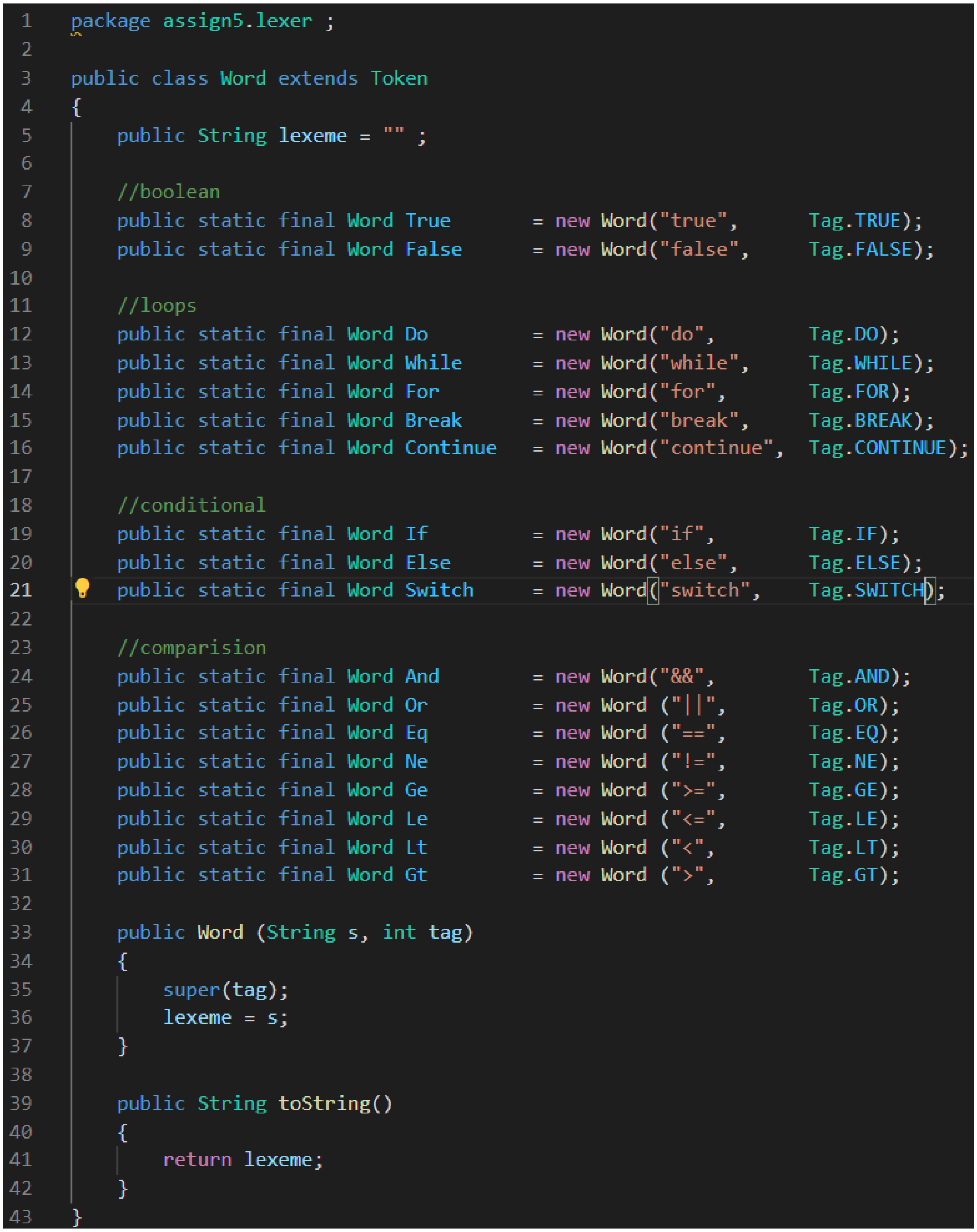
Type.java



```java
package assign5.lexer;

public class Type extends Word
{
    public int width = 0;

    public static final Type Int    = new Type("int",      Tag.BASIC, 4);
    public static final Type Float  = new Type("float",    Tag.BASIC, 8);
    public static final Type Char   = new Type("char",     Tag.BASIC, 1);
    public static final Type Bool   = new Type("bool",     Tag.BASIC, 1);

    public Type(String s, int tag, int w)
    {
        super(s, tag);
        this.width = w;
    }

    public static boolean numeric(Type p)
    {
        if(p == Type.Char || p == Type.Int || p == Type.Float)
        {
            return true;
        }
        else
        {
            return false;
        }
    }

    public static Type max(Type p1, Type p2)
    {
        if(!numeric(p1) || !numeric(p2))
        {
            return null;
        }
        else if(p1 == Type.Float || p2 == Type.Float)
        {
            return Type.Float;
        }
        else if(p1 == Type.Int || p2 == Type.Int)
        {
            return Type.Int;
        }
        else
        {
            return Type.Char;
        }
    }
}
```

This file adds support for a token word of a variable type. The types currently supported are int, float, char, and Boolean. Additionally, there are some methods added to determine if a type is numerical or find the maximum value.

Word.java



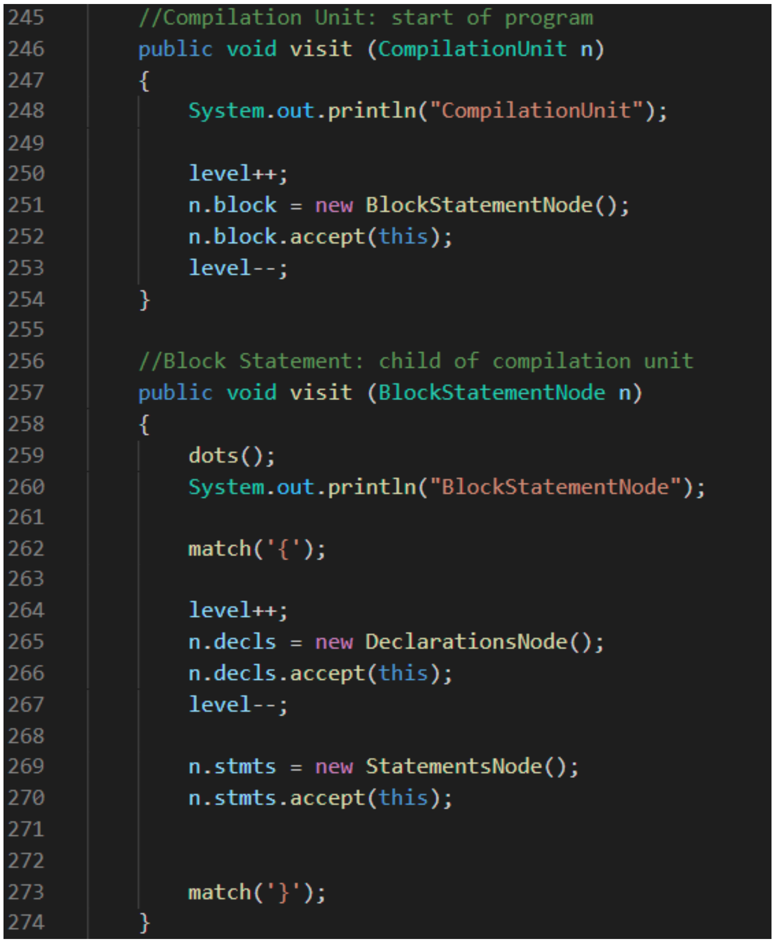
```java
package assign5.lexer ;

public class Word extends Token
{
    public String lexeme = "" ;

    //boolean
    public static final Word True        = new Word("true",      Tag.TRUE);
    public static final Word False       = new Word("false",     Tag.FALSE);

    //loops
    public static final Word Do          = new Word("do",        Tag.DO);
    public static final Word While       = new Word("while",     Tag.WHILE);
    public static final Word For         = new Word("for",       Tag.FOR);
    public static final Word Break       = new Word("break",     Tag.BREAK);
    public static final Word Continue    = new Word("continue",  Tag.CONTINUE);

    //conditional
    public static final Word If          = new Word("if",        Tag.IF);
    public static final Word Else        = new Word("else",      Tag.ELSE);
    public static final Word Switch      = new Word("switch",    Tag.SWITCH);

    //comparision
    public static final Word And         = new Word("&&",        Tag.AND);
    public static final Word Or          = new Word ("||",       Tag.OR);
    public static final Word Eq          = new Word ("==",       Tag.EQ);
    public static final Word Ne          = new Word ("!=",       Tag.NE);
    public static final Word Ge          = new Word (">=",       Tag.GE);
    public static final Word Le          = new Word ("<=",       Tag.LE);
    public static final Word Lt          = new Word ("<",        Tag.LT);
    public static final Word Gt          = new Word (">",        Tag.GT);

    public Word (String s, int tag)
    {
        super(tag);
        lexeme = s;
    }

    public String toString()
    {
        return lexeme;
    }
}
```

Many new words have been added to support new syntax for the program. It will now have supported words for Boolean, loops, conditional operators, and comparison operators.

parsing

The parsing phase now has additional support for declarations and the nodes used within the declaration.

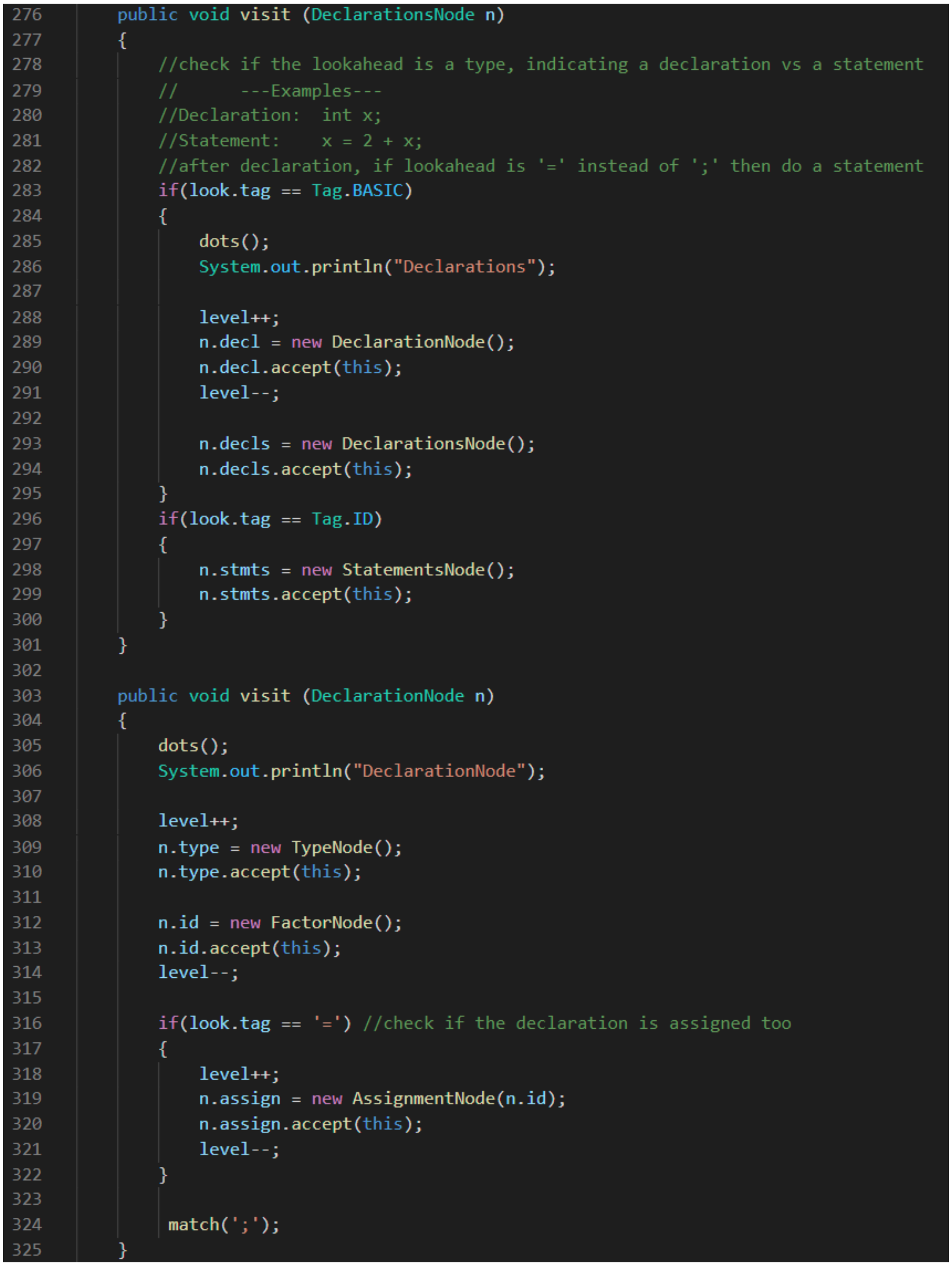
parser.java



```
    //Compilation Unit: start of program
    public void visit (CompilationUnit n)
    {
        System.out.println("CompilationUnit");

        level++;
        n.block = new BlockStatementNode();
        n.block.accept(this);
        level--;
    }

    //Block Statement: child of compilation unit
    public void visit (BlockStatementNode n)
    {
        dots();
        System.out.println("BlockStatementNode");

        match('{');

        level++;
        n.decls = new DeclarationsNode();
        n.decls.accept(this);
        level--;

        n.stmts = new StatementsNode();
        n.stmts.accept(this);


        match('}');
    }
```

The block statement now will support declarations followed by statements. In the future, there will be support for multiple block statements and statements will be allowed to come before declarations.

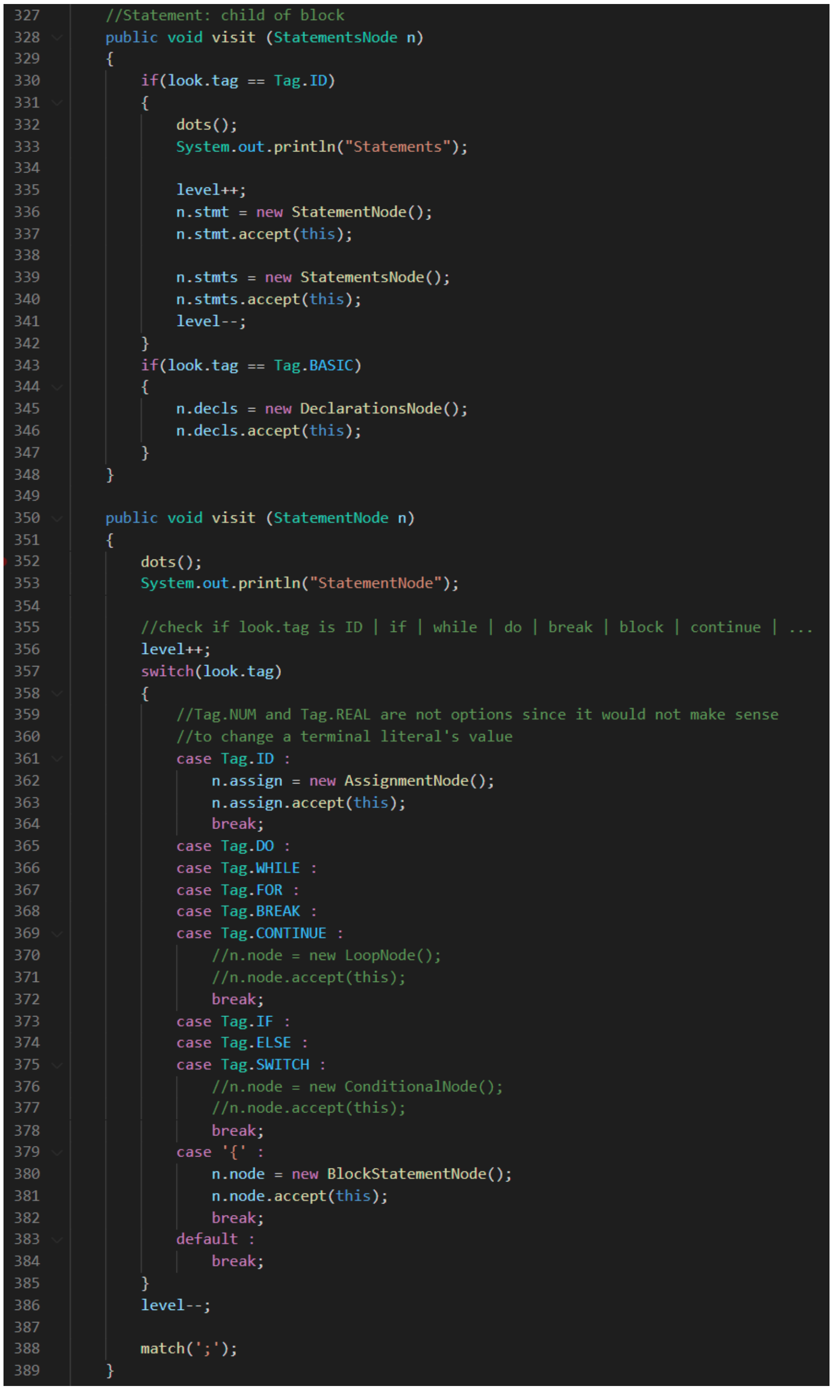
parser.java cont.



```java
    public void visit (DeclarationsNode n)
    {
        //check if the lookahead is a type, indicating a declaration vs a statement
        //      ---Examples---
        //Declaration:  int x;
        //Statement:    x = 2 + x;
        //after declaration, if lookahead is '=' instead of ';' then do a statement
        if(look.tag == Tag.BASIC)
        {
            dots();
            System.out.println("Declarations");

            level++;
            n.decl = new DeclarationNode();
            n.decl.accept(this);
            level--;

            n.decls = new DeclarationsNode();
            n.decls.accept(this);
        }
        if(look.tag == Tag.ID)
        {
            n.stmts = new StatementsNode();
            n.stmts.accept(this);
        }
    }

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

        level++;
        n.type = new TypeNode();
        n.type.accept(this);

        n.id = new FactorNode();
        n.id.accept(this);
        level--;

        if(look.tag == '=') //check if the declaration is assigned too
        {
            level++;
            n.assign = new AssignmentNode(n.id);
            n.assign.accept(this);
            level--;
        }

         match(';');
    }
```

The declarations node will now continuously read in declarations until a statement is encountered, which is denoted by the look having a tag of ID at the start of a new production. For each declaration a typeNode will determine the variables type of the factornode that is read in next. After the declaration, there is the option for assignment, which will be denoted by the look.tag being equal to the ‘=’ symbol.

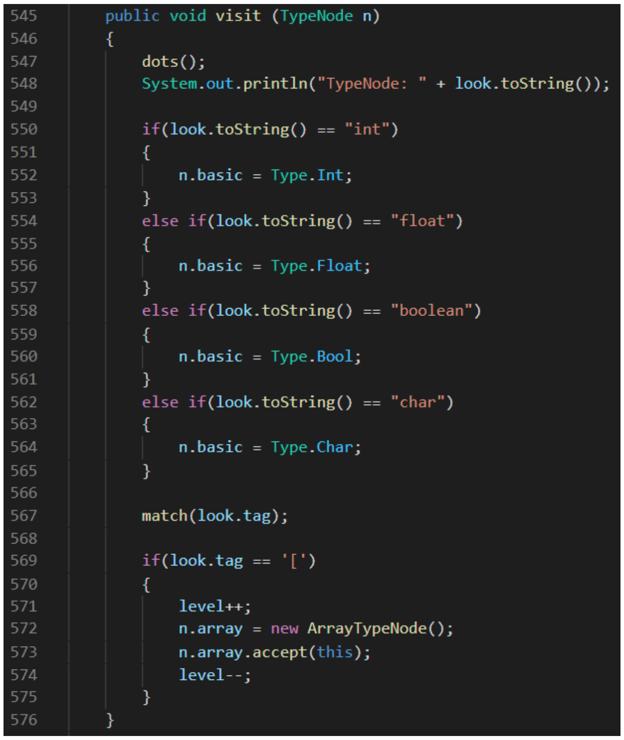
parser.java cont.



```java
    //Statement: child of block
    public void visit (StatementsNode n)
    {
        if(look.tag == Tag.ID)
        {
            dots();
            System.out.println("Statements");

            level++;
            n.stmt = new StatementNode();
            n.stmt.accept(this);

            n.stmts = new StatementsNode();
            n.stmts.accept(this);
            level--;
        }
        if(look.tag == Tag.BASIC)
        {
            n.decls = new DeclarationsNode();
            n.decls.accept(this);
        }
    }

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

        //check if look.tag is ID | if | while | do | break | block | continue | ...
        level++;
        switch(look.tag)
        {
            //Tag.NUM and Tag.REAL are not options since it would not make sense
            //to change a terminal literal's value
            case Tag.ID :
                n.assign = new AssignmentNode();
                n.assign.accept(this);
                break;
            case Tag.DO :
            case Tag.WHILE :
            case Tag.FOR :
            case Tag.BREAK :
            case Tag.CONTINUE :
                //n.node = new LoopNode();
                //n.node.accept(this);
                break;
            case Tag.IF :
            case Tag.ELSE :
            case Tag.SWITCH :
                //n.node = new ConditionalNode();
                //n.node.accept(this);
                break;
            case '{' :
                n.node = new BlockStatementNode();
                n.node.accept(this);
                break;
            default :
                break;
        }
        level--;

        match(';');
    }
```

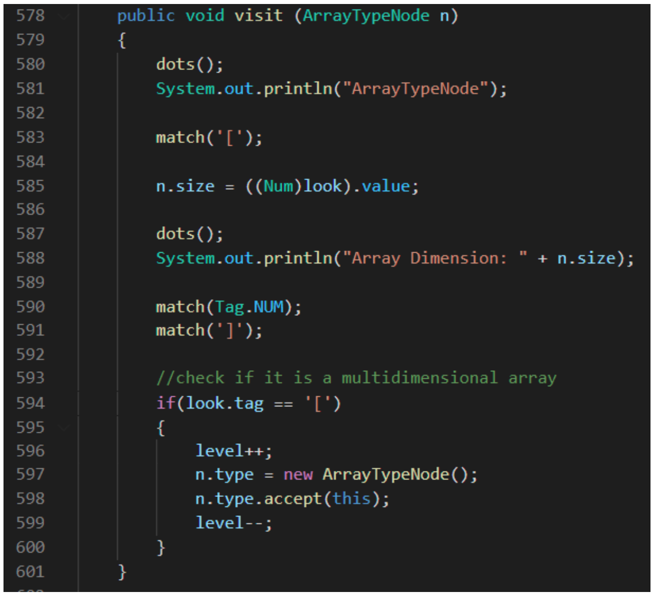
Statements and statement are a lot like the declarations and declaration. However, the statements will be some sort of programmatic operation like assigning a value, looping, or checking a condition.

parser.java cont.



```java
public void visit (TypeNode n)
{
    dots();
    System.out.println("TypeNode: " + look.toString());

    if(look.toString() == "int")
    {
        n.basic = Type.Int;
    }
    else if(look.toString() == "float")
    {
        n.basic = Type.Float;
    }
    else if(look.toString() == "boolean")
    {
        n.basic = Type.Bool;
    }
    else if(look.toString() == "char")
    {
        n.basic = Type.Char;
    }

    match(look.tag);

    if(look.tag == '[')
    {
        level++;
        n.array = new ArrayTypeNode();
        n.array.accept(this);
        level--;
    }
}
```

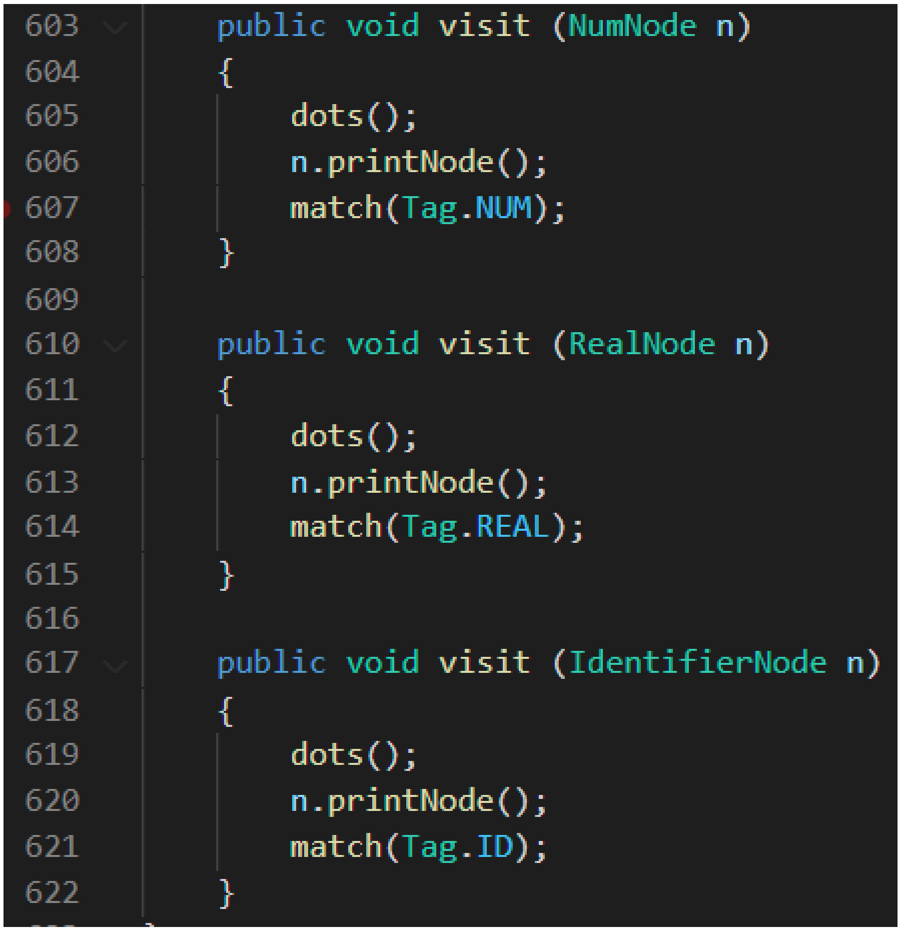
The TypeNode will determine the type of a variables by checking a token for being a basic type. If, after reading in the type, the token is ‘[‘, then an array node will be formed of a fixed size.



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

    match('[');

    n.size = ((Num)look).value;

    dots();
    System.out.println("Array Dimension: " + n.size);

    match(Tag.NUM);
    match(']');

    //check if it is a multidimensional array
    if(look.tag == '[')
    {
        level++;
        n.type = new ArrayTypeNode();
        n.type.accept(this);
        level--;
    }
}
```

When the square brackets are detected in the token stream, the parser will begin parsing together an array variable. If there is another left square bracket after the first array declaration, then the array is multidimensional and another array is parsed.

parser.java cont.



```java
    public void visit (NumNode n)
    {
        dots();
        n.printNode();
        match(Tag.NUM);
    }

    public void visit (RealNode n)
    {
        dots();
        n.printNode();
        match(Tag.REAL);
    }

    public void visit (IdentifierNode n)
    {
        dots();
        n.printNode();
        match(Tag.ID);
    }
```

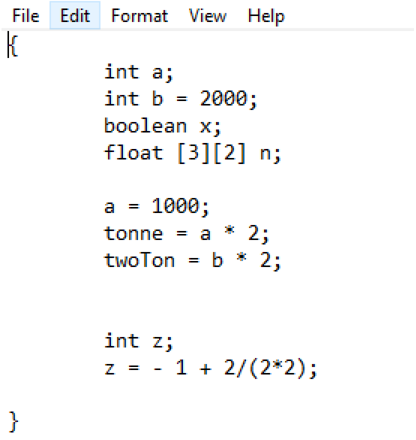
The LiteralNode has been replaced to support floats as well as integers. Now integers will be stored in a NumNode and floats will be in a RealNode. The IdentifierNode remains the same.

unparser

The unparser is the same from the last lab with the exceptions of new nodes being supported and visited appropriately. The AST still cannot be build correctly during parsing so the TreePrinter will be used to make an AST. The TreePrinter is mostly the same too, with the exception with support for the new and removed nodes.

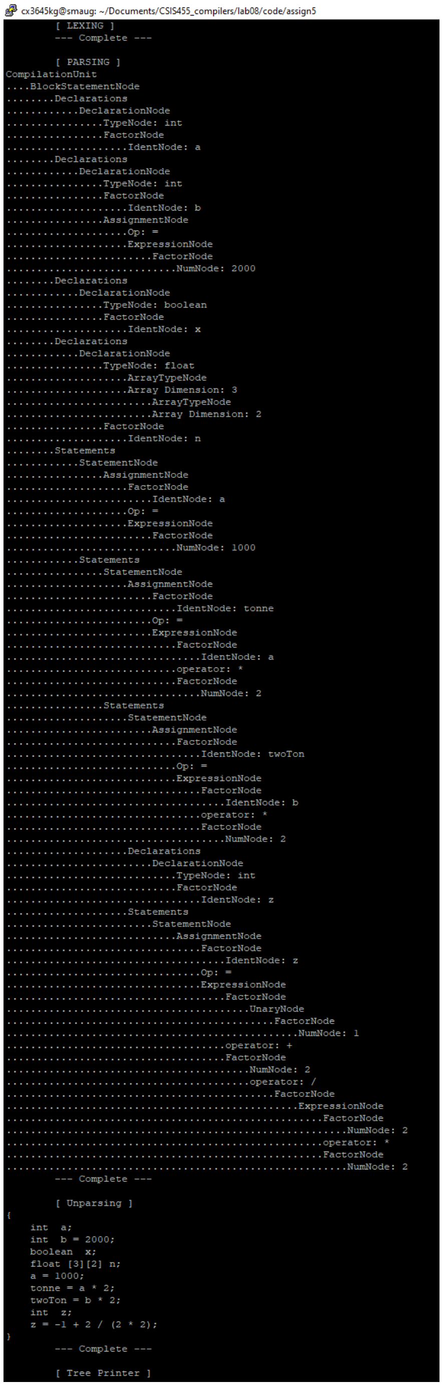
Execution

input.txt



```
{

        int a;
        int b = 2000;
        boolean x;
        float [3][2] n;

        a = 1000;
        tonne = a * 2;
        twoTon = b * 2;


        int z;
        z = - 1 + 2/(2*2);

}
```

terminal


cx3645kg@smaug: ~/Documents/CSIS455_compilers/lab08/code/assign5

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

        [ PARSING ]
CompilationUnit
....BlockStatementNode
........Declarations
............DeclarationNode
................TypeNode: int
................FactorNode
....................IdentNode: a
........Declarations
............DeclarationNode
................TypeNode: int
................FactorNode
....................IdentNode: b
................AssignmentNode
....................Op: =
....................ExpressionNode
........................FactorNode
............................NumNode: 2000
........Declarations
............DeclarationNode
................TypeNode: boolean
................FactorNode
....................IdentNode: x
........Declarations
............DeclarationNode
................TypeNode: float
....................ArrayTypeNode
....................Array Dimension: 3
........................ArrayTypeNode
........................Array Dimension: 2
................FactorNode
....................IdentNode: n
........Statements
............StatementNode
................AssignmentNode
....................FactorNode
........................IdentNode: a
....................Op: =
....................ExpressionNode
........................FactorNode
............................NumNode: 1000
............Statements
................StatementNode
....................AssignmentNode
........................FactorNode
............................IdentNode: tonne
........................Op: =
........................ExpressionNode
............................FactorNode
................................IdentNode: a
............................operator: *
............................FactorNode
................................NumNode: 2
................Statements
....................StatementNode
........................AssignmentNode
............................FactorNode
................................IdentNode: twoTon
............................Op: =
............................ExpressionNode
................................FactorNode
....................................IdentNode: b
................................operator: *
................................FactorNode
....................................NumNode: 2
....................Declarations
........................DeclarationNode
............................TypeNode: int
............................FactorNode
................................IdentNode: z
....................Statements
........................StatementNode
............................AssignmentNode
................................FactorNode
....................................IdentNode: z
................................Op: =
................................ExpressionNode
....................................FactorNode
........................................UnaryNode
............................................FactorNode
................................................NumNode: 1
....................................operator: +
....................................FactorNode
........................................NumNode: 2
........................................operator: /
............................................FactorNode
................................................ExpressionNode
....................................................FactorNode
........................................................NumNode: 2
....................................................operator: *
....................................................FactorNode
........................................................NumNode: 2
        --- Complete ---

        [ Unparsing ]
{
    int  a;
    int  b = 2000;
    boolean  x;
    float [3][2] n;
    a = 1000;
    tonne = a * 2;
    twoTon = b * 2;
    int  z;
    z = -1 + 2 / (2 * 2);
}
        --- Complete ---

        [ Tree Printer ]
```

The execution parsed the input as expected and displayed what it was doing. Also, the tree printer displayed a correct AST as expected with a complex operation.

Conclusion

This assignment required a lot of additional code to support the new nodes and features in the lexer and parser. Overall, the changes were not difficult, but some of the errors during compilation were difficult to track down, and I should have compiled more often when I was coding. Some of the choices I've made in previous assignments diverged my program from where it was supposed to be, and I had to make changes to bring it closer to what it is supposed to be. I am now beginning to see how the program will be able to read in the tokens, parse them, and then create machine code. Also, I'm seeing the importance of keeping the code organized and adding features like tag id's to reduce the amount of code later on.