Paul Kummer

Dr. Hanku Lee

Compilers CSIS 455-01

10/07/2021

Lab 05: PrettyPrinter

Purpose of Lab

The purpose of this lab was to show students how the parser can read in a document containing characters that could make up a programming language and begin to process the data. Instead of trying to convert the data into a lower-level language, the characters are read in and determined what they mean in the sense of a programming language. For instance, this lab will create nodes for different parts of a programming language structure like blocks and line statements.

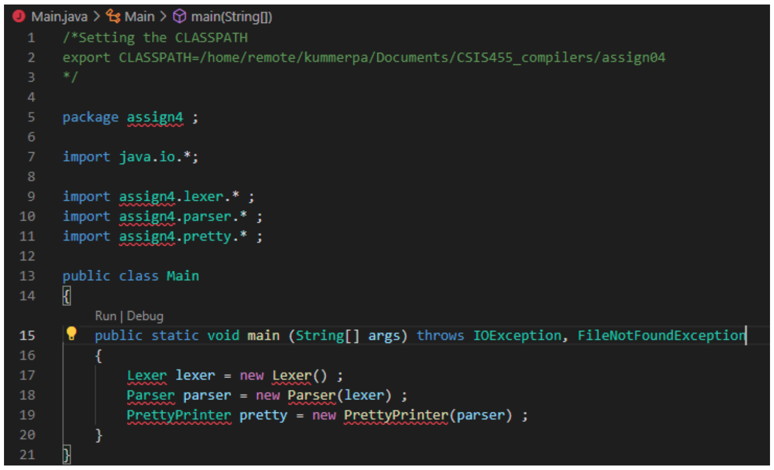
Additionally, the characters that are read are formatted using a newly created class called “PrettyPrinter”. This class will justify tokens and print them with good formatting practice for common programming languages. All code within blocks will be indented, and all line statements will be in one line and terminated with a semicolon. The “PrettyPrinter” class will overload the “ASTVisitor” class to do its own formatting. Using “PrettyPrinter” demonstrates how errors could be potentially detected or corrected with poorly formatted code or code that is missing a semicolon.

Example Code

Main

Main.java

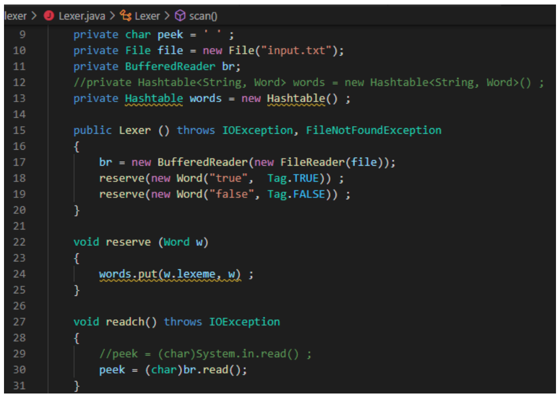
This executes the main line of logic for the program. It is currently separated into three different phases. The first phase is lexing, which is reading in the input.txt and turning it into tokens. The second phase is parsing, which is reading in the tokens and determining what there meaning is. Then the last phase is formatting, which will organize and appropriately space tokens based on common code formatting practices.



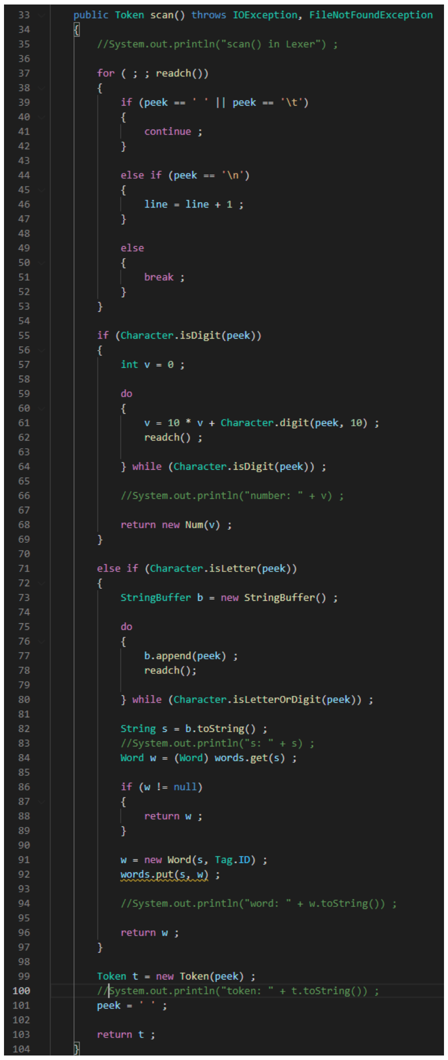
```java
/*Setting the CLASSPATH
export CLASSPATH=/home/remote/kummerpa/Documents/CSIS455_compilers/assign04
*/

package assign4 ;

import java.io.*;

import assign4.lexer.* ;
import assign4.parser.* ;
import assign4.pretty.* ;

public class Main
{
    public static void main (String[] args) throws IOException, FileNotFoundException
    {
        Lexer lexer = new Lexer() ;
        Parser parser = new Parser(lexer) ;
        PrettyPrinter pretty = new PrettyPrinter(parser) ;
    }
}
```

Lexer.java

This code will read in characters from input.txt and turn them into tokens. The tokens can be a number, word, or a generic token. For this lab the lexer had to be modified to support reading a file and putting that file into a buffer that is read by the method “readch()”. Also, the “scan()” method was modified to use an if else statement instead of just if statements.



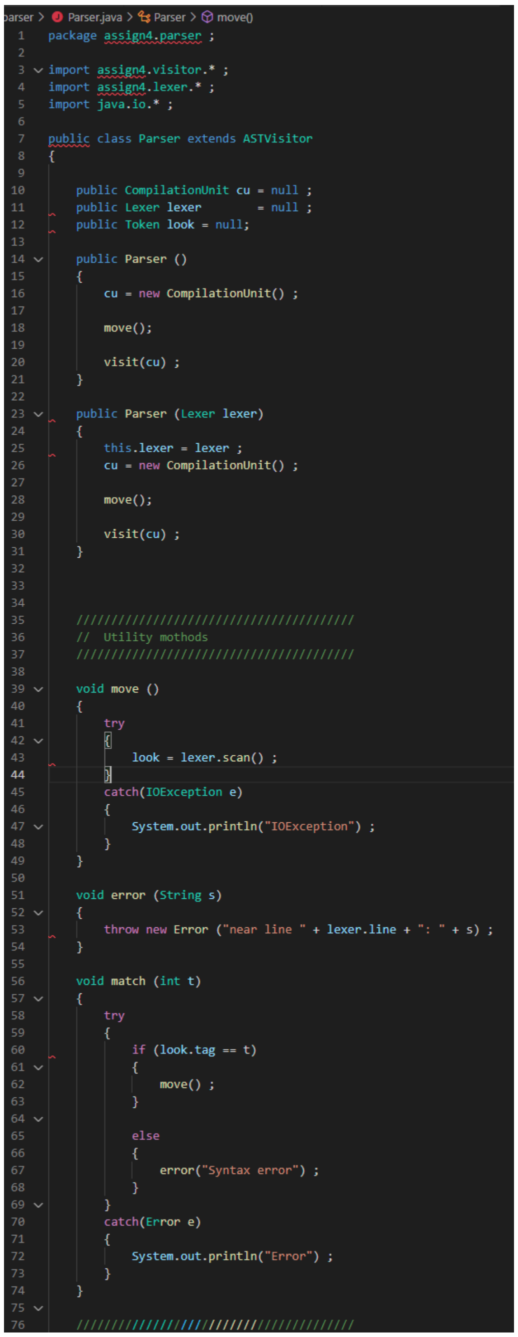
```java
    private char peek = ' ' ;
    private File file = new File("input.txt");
    private BufferedReader br;
    //private Hashtable<String, Word> words = new Hashtable<String, Word>() ;
    private Hashtable words = new Hashtable() ;

    public Lexer () throws IOException, FileNotFoundException
    {
        br = new BufferedReader(new FileReader(file));
        reserve(new Word("true",  Tag.TRUE)) ;
        reserve(new Word("false", Tag.FALSE)) ;
    }

    void reserve (Word w)
    {
        words.put(w.lexeme, w) ;
    }

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



```java
    public Token scan() throws IOException, FileNotFoundException
    {
        //System.out.println("scan() in Lexer") ;

        for ( ; ; readch())
        {
            if (peek == ' ' || peek == '\t')
            {
                continue ;
            }

            else if (peek == '\n')
            {
                line = line + 1 ;
            }

            else
            {
                break ;
            }
        }

        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) ;

            return new Num(v) ;
        }

        else if (Character.isLetter(peek))
        {
            StringBuffer b = new StringBuffer() ;

            do
            {
                b.append(peek) ;
                readch();

            } while (Character.isLetterOrDigit(peek)) ;

            String s = b.toString() ;
            //System.out.println("s: " + s) ;
            Word w = (Word) words.get(s) ;

            if (w != null)
            {
                return w ;
            }

            w = new Word(s, Tag.ID) ;
            words.put(s, w) ;

            //System.out.println("word: " + w.toString()) ;

            return w ;
        }

        Token t = new Token(peek) ;
        //System.out.println("token: " + t.toString()) ;
        peek = ' ' ;

        return t ;
    }
```

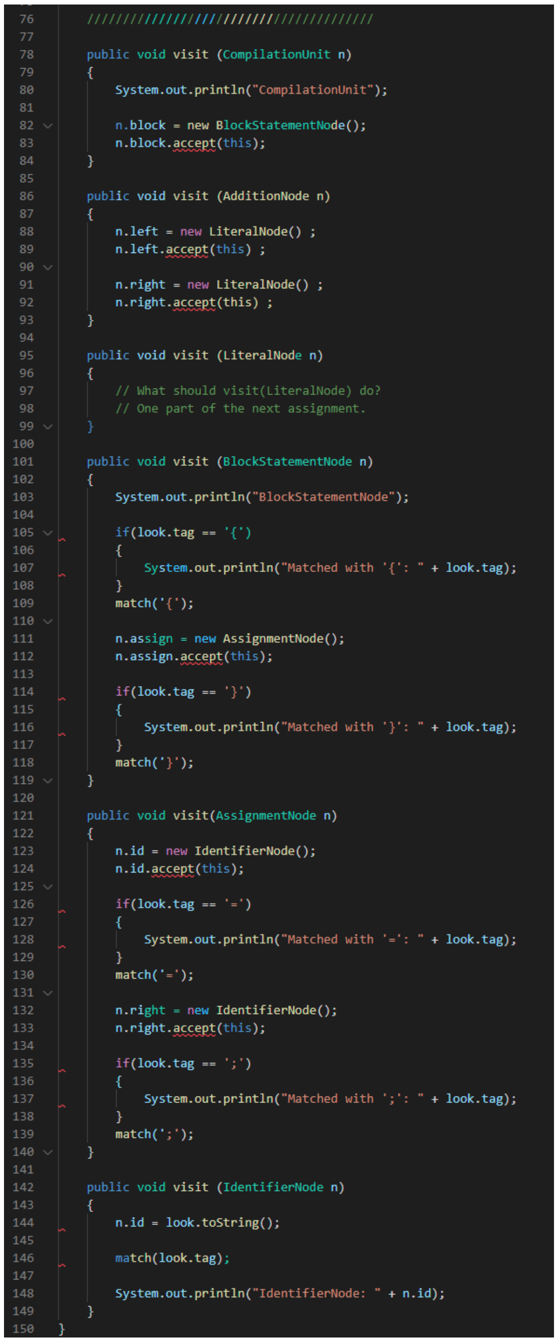
Parser.java

The parser determines what kind of node each token will represent. For this lab, and assignment node and a block statement node were added to give more structure to the tokens that are read in. The block statement node represents a container that holds many lines of codes between two curly braces. The assignment node represents any expression where a left-hand side of an expression is given the value of the right hand side. In this lab, the block statement only contains one assignment statement, but it will be modified in the future to hold more.



parser > Parser.java > Parser > move()

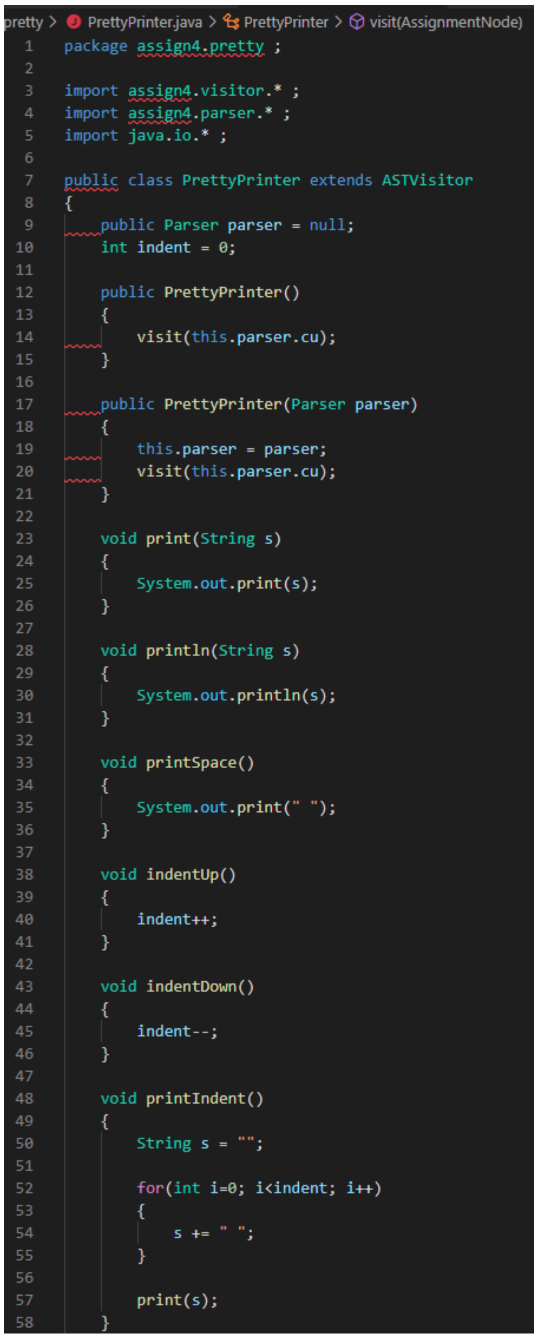
```java
package assign4.parser ;

import assign4.visitor.* ;
import assign4.lexer.* ;
import java.io.* ;

public class Parser extends ASTVisitor
{

    public CompilationUnit cu = null ;
    public Lexer lexer        = null ;
    public Token look = null;

    public Parser ()
    {
        cu = new CompilationUnit() ;

        move();

        visit(cu) ;
    }

    public Parser (Lexer lexer)
    {
        this.lexer = lexer ;
        cu = new CompilationUnit() ;

        move();

        visit(cu) ;
    }



    ////////////////////////////////////////
    //  Utility mothods
    ////////////////////////////////////////

    void move ()
    {
        try
        {
            look = lexer.scan() ;
        }
        catch(IOException e)
        {
            System.out.println("IOException") ;
        }
    }

    void error (String s)
    {
        throw new Error ("near line " + lexer.line + ": " + s) ;
    }

    void match (int t)
    {
        try
        {
            if (look.tag == t)
            {
                move() ;
            }

            else
            {
                error("Syntax error") ;
            }
        }
        catch(Error e)
        {
            System.out.println("Error") ;
        }
    }

    ////////////////////////////////////////
```



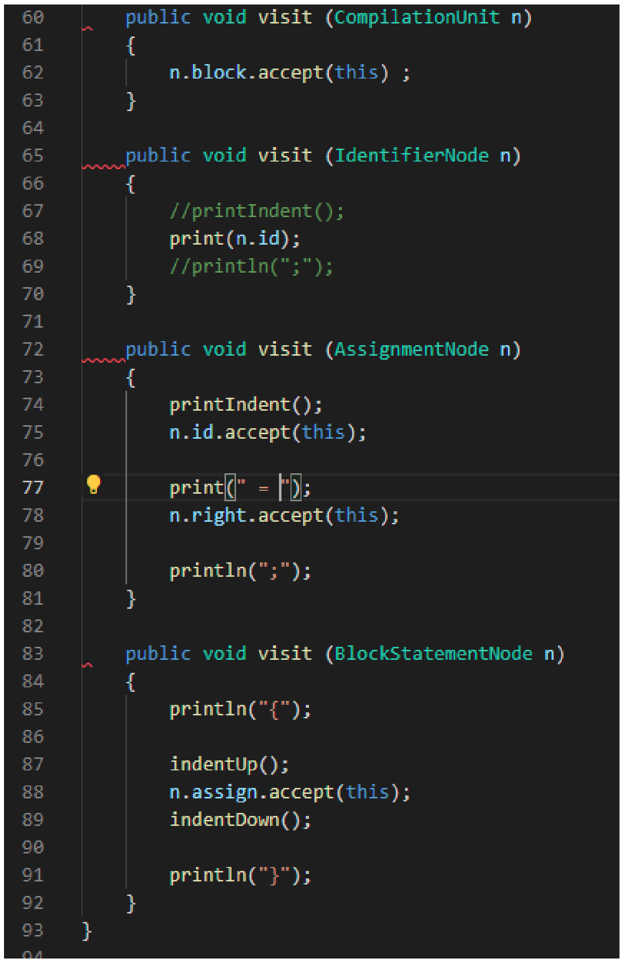
```java
    ////////////////////////////////////////

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

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

    public void visit (AdditionNode n)
    {
        n.left = new LiteralNode() ;
        n.left.accept(this) ;

        n.right = new LiteralNode() ;
        n.right.accept(this) ;
    }

    public void visit (LiteralNode n)
    {
        // What should visit(LiteralNode) do?
        // One part of the next assignment.
    }

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

        if(look.tag == '{')
        {
            System.out.println("Matched with '{': " + look.tag);
        }
        match('{');

        n.assign = new AssignmentNode();
        n.assign.accept(this);

        if(look.tag == '}')
        {
            System.out.println("Matched with '}': " + look.tag);
        }
        match('}');
    }

    public void visit(AssignmentNode n)
    {
        n.id = new IdentifierNode();
        n.id.accept(this);

        if(look.tag == '=')
        {
            System.out.println("Matched with '=': " + look.tag);
        }
        match('=');

        n.right = new IdentifierNode();
        n.right.accept(this);

        if(look.tag == ';')
        {
            System.out.println("Matched with ';': " + look.tag);
        }
        match(';');
    }

    public void visit (IdentifierNode n)
    {
        n.id = look.toString();

        match(look.tag);

        System.out.println("IdentifierNode: " + n.id);
    }
}
```

PrettyPrinter.java

Pretty printer formats the token stream into what a programming language should look like. It does this by overloading the “ASTVisitor” class’s visit methods. For every node, there will be a visit method that will create an abstract syntax tree in the format of a programming language. Also, indentation and spacing will be systematically created to display the code in a easy to read format.


pretty > PrettyPrinter.java > PrettyPrinter > visit(AssignmentNode)

```java
package assign4.pretty ;

import assign4.visitor.* ;
import assign4.parser.* ;
import java.io.* ;

public class PrettyPrinter extends ASTVisitor
{
    public Parser parser = null;
    int indent = 0;

    public PrettyPrinter()
    {
        visit(this.parser.cu);
    }

    public PrettyPrinter(Parser parser)
    {
        this.parser = parser;
        visit(this.parser.cu);
    }

    void print(String s)
    {
        System.out.print(s);
    }

    void println(String s)
    {
        System.out.println(s);
    }

    void printSpace()
    {
        System.out.print(" ");
    }

    void indentUp()
    {
        indent++;
    }

    void indentDown()
    {
        indent--;
    }

    void printIndent()
    {
        String s = "";

        for(int i=0; i<indent; i++)
        {
            s += " ";
        }

        print(s);
    }
```



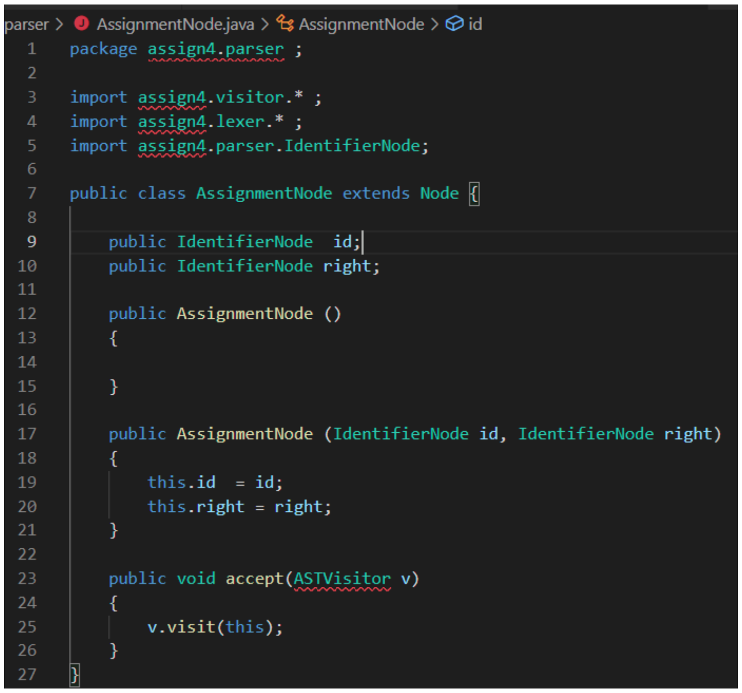
```java
    public void visit (CompilationUnit n)
    {
        n.block.accept(this) ;
    }

    public void visit (IdentifierNode n)
    {
        //printIndent();
        print(n.id);
        //println(";");
    }

    public void visit (AssignmentNode n)
    {
        printIndent();
        n.id.accept(this);

        print(" = ");
        n.right.accept(this);

        println(";");
    }

    public void visit (BlockStatementNode n)
    {
        println("{");

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

        println("}");
    }
}
```

Nodes

For every unique type of expression that can be represented there is a node to represent that expression. In that node, the tag identifier will be stored along with values specific to that node. Some nodes are generic and can be broken down into more specific nodes until an atomic unit is reached. For example, a binary expression has three parts. An identifier, operator, and another expression, which could be an identifier.

AssignmentNode.java

The assignment node stores two identifiers and of id and right. The right will in the future probably be the result of an expression.

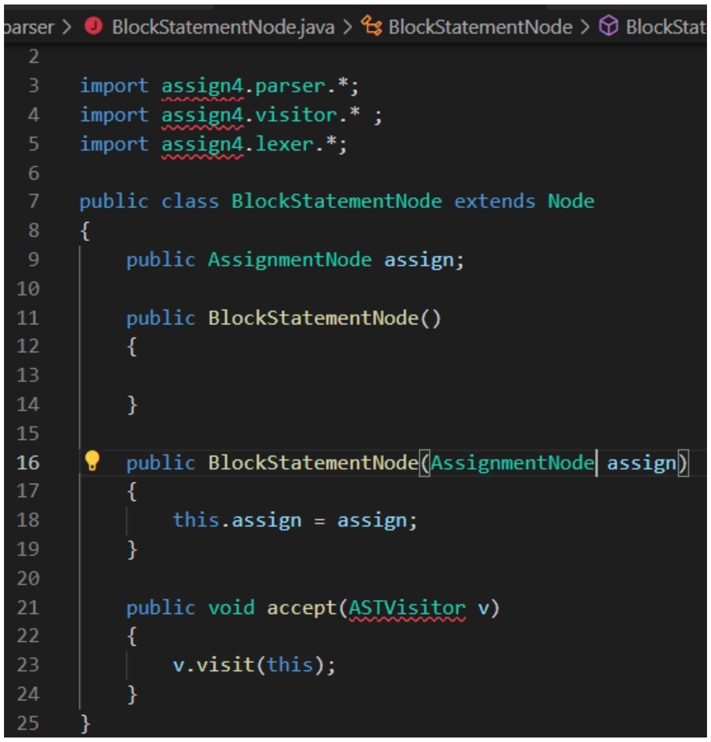


parser > AssignmentNode.java > AssignmentNode > id

```java
package assign4.parser ;

import assign4.visitor.* ;
import assign4.lexer.* ;
import assign4.parser.IdentifierNode;

public class AssignmentNode extends Node {

    public IdentifierNode  id;
    public IdentifierNode right;

    public AssignmentNode ()
    {

    }

    public AssignmentNode (IdentifierNode id, IdentifierNode right)
    {
        this.id  = id;
        this.right = right;
    }

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

BlockStatementNode.java

The block statement node is used to represent a container that holds many related statements. Currently, the block statement only holds the value of one assignment node, but will be built upon more in the future.

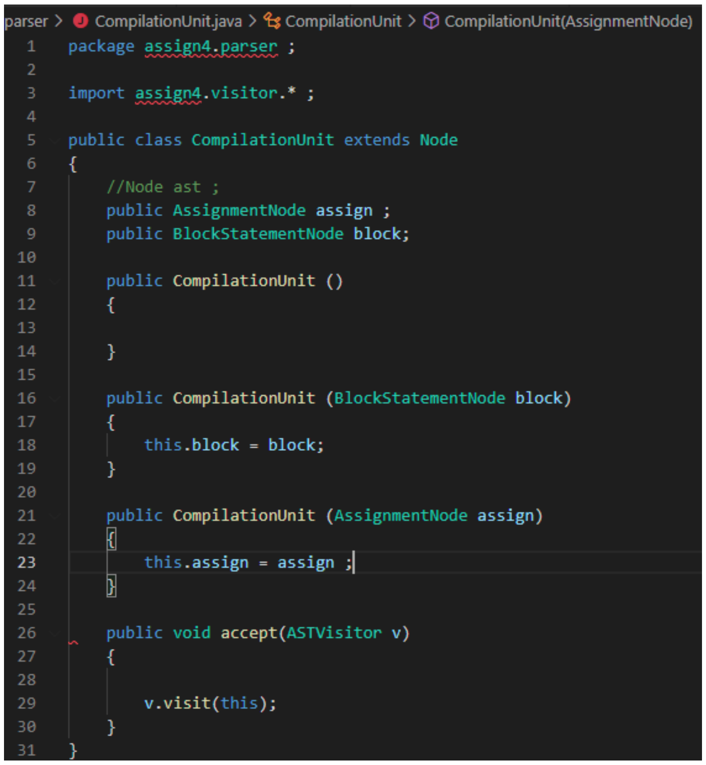


parser > BlockStatementNode.java > BlockStatementNode > BlockStat

```java

import assign4.parser.*;
import assign4.visitor.* ;
import assign4.lexer.*;

public class BlockStatementNode extends Node
{
    public AssignmentNode assign;

    public BlockStatementNode()
    {

    }

    public BlockStatementNode(AssignmentNode assign)
    {
        this.assign = assign;
    }

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

CompilationUnit.java

The compilation unit is the root of all nodes. It is used to start building an AST. There are two ways a AST can start, and it currently will accept either an assignment node or a block statement node.

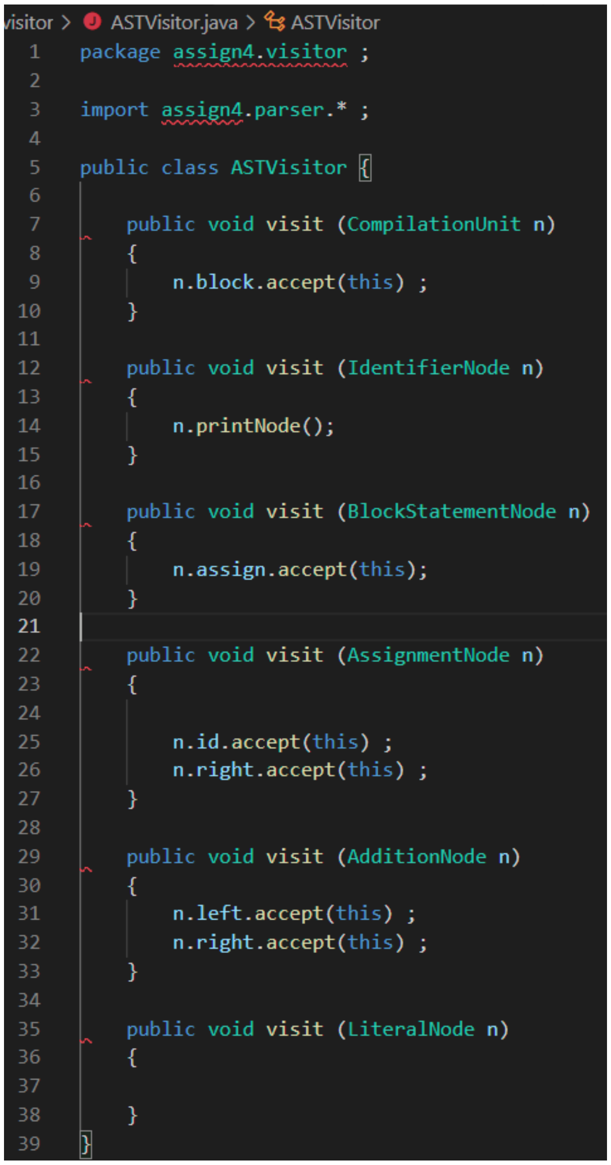

parser > CompilationUnit.java > CompilationUnit > CompilationUnit(AssignmentNode)

```java
package assign4.parser ;

import assign4.visitor.* ;

public class CompilationUnit extends Node
{
    //Node ast ;
    public AssignmentNode assign ;
    public BlockStatementNode block;

    public CompilationUnit ()
    {

    }

    public CompilationUnit (BlockStatementNode block)
    {
        this.block = block;
    }

    public CompilationUnit (AssignmentNode assign)
    {
        this.assign = assign ;
    }

    public void accept(ASTVisitor v)
    {

        v.visit(this);
    }
}
```

Visitor

ASTVisitor.java

Technically, there are two visitors; the pretty printer and the AST visitor. The “ASTVisitor” class is a general visitor and is used as a base class of pretty printer. It ensures that the nodes within nodes are visited by calling the “accept()” method of all the nodes of a class when it is visited.


isitor > ASTVisitor.java > ASTVisitor

```java
package assign4.visitor ;

import assign4.parser.* ;

public class ASTVisitor {

    public void visit (CompilationUnit n)
    {
        n.block.accept(this) ;
    }

    public void visit (IdentifierNode n)
    {
        n.printNode();
    }

    public void visit (BlockStatementNode n)
    {
        n.assign.accept(this);
    }

    public void visit (AssignmentNode n)
    {

        n.id.accept(this) ;
        n.right.accept(this) ;
    }

    public void visit (AdditionNode n)
    {
        n.left.accept(this) ;
        n.right.accept(this) ;
    }

    public void visit (LiteralNode n)
    {

    }
}
```

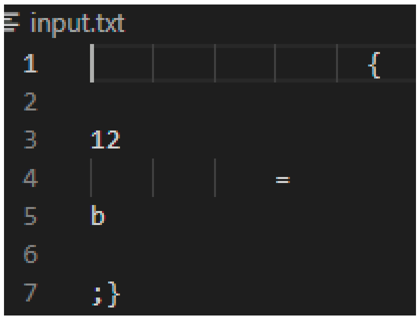
Execution

Execution of Main.java

Main is called and breaks the program into three phases. the first is the lexer. Then the parser will be called. Finally, the formatting of pretty printer is called.

input.txt

The sample file that contains the characters that will be converted into tokens and nodes, which will be displayed to the user.



```
                    {

12
                =
b

;}
```

terminal output

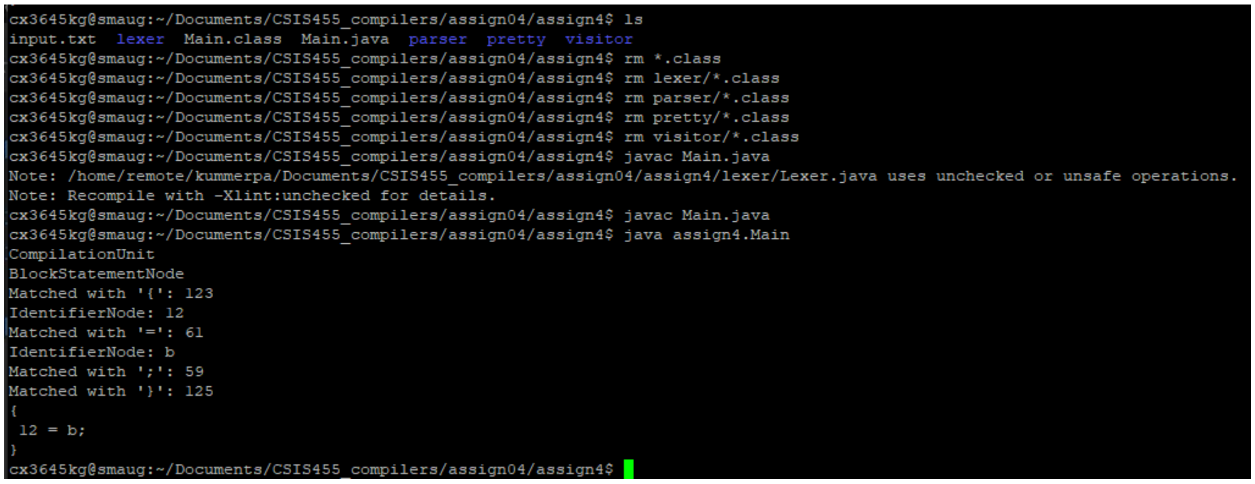
Once the java program is compiled correctly, the lexer opens and reads the input.txt file into a buffer and begins reading in each character from the file. Whatever token the character or characters becomes depends on what the first character is of any meaningful grouping of characters. Then the parser will start to read the tokens and make nodes based on the matching of token tag identifiers. Finally, the nodes will be visited and output to the terminal in an orderly manner that is easy to read.

First, the program reads in the whitespace and removes it. Then it encounters a right curly brace, which is turned into a token with the id of the ascii value of 123. Next whitespace is removed again, and a number is encountered, which leads to consecutive numbers being read. This results in a number token being made with the tag id of 12. As usual, whitespace is removed, and another character is read and converted to a token because it is not a letter or number. The id of this token is 61, which is the ascii value of '=' symbol. Whitespace is removed and the character b is read and turned into a word token with the ascii value of 'b'. Next whitespace is removed and the final two tokens of ';' and '}' are read and assigned their own tag ids.

Next, a computation unit is initialized and leads to the creating of a block statement unit. This then expects a '{' token and matches it. Next it expects an assignment statement and creates and assignment node. The assignment node creates to identifier nodes and matches the tag ids of the '12', '=', 'b', and ';'. The identifier nodes are actually matched within the identifier node and not assignment node. Finally, the last curly brace is matched within the original block statement node.

Finally, all the nodes are visited starting with the compilation unit. The compilation unit accepts its block statement, visiting the block statement. This will cause the block statement to accept and visit its assignment statement that will accept and visit its two identifiers. When these to identifiers are done with their visit method, control will start traveling back to the callees.

With every visit method, formatting methods from pretty printer are added between accept calls. When this is done, the proper code format is displayed.



```
cx3645kg@smaug:~/Documents/CSIS455_compilers/assign04/assign4$ ls
input.txt  lexer  Main.class  Main.java  parser  pretty  visitor
cx3645kg@smaug:~/Documents/CSIS455_compilers/assign04/assign4$ rm *.class
cx3645kg@smaug:~/Documents/CSIS455_compilers/assign04/assign4$ rm lexer/*.class
cx3645kg@smaug:~/Documents/CSIS455_compilers/assign04/assign4$ rm parser/*.class
cx3645kg@smaug:~/Documents/CSIS455_compilers/assign04/assign4$ rm pretty/*.class
cx3645kg@smaug:~/Documents/CSIS455_compilers/assign04/assign4$ rm visitor/*.class
cx3645kg@smaug:~/Documents/CSIS455_compilers/assign04/assign4$ javac Main.java
Note: /home/remote/kummerpa/Documents/CSIS455_compilers/assign04/assign4/lexer/Lexer.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
cx3645kg@smaug:~/Documents/CSIS455_compilers/assign04/assign4$ javac Main.java
cx3645kg@smaug:~/Documents/CSIS455_compilers/assign04/assign4$ java assign4.Main
CompilationUnit
BlockStatementNode
Matched with '{': 123
IdentifierNode: 12
Matched with '=': 61
IdentifierNode: b
Matched with ';': 59
Matched with '}': 125
{
 12 = b;
}
cx3645kg@smaug:~/Documents/CSIS455_compilers/assign04/assign4$
```

Conclusion

This lab was very helpful in clarifying some confusion I was having about how to tie together the lexer and parser using the visitor pattern. I managed to do it in a previous assignment, but this is much clearer and easier to follow with the method shown in this lab. The way the code is laid out makes it easy to see how it can be used to start taking in stored high-level code and turn it into intermediate code or machine code. I also, began to see how syntax errors could be detected upon compile time. For example, when the pretty printer doesn't match an expected character, the compiler could throw an error with a line number telling the programmer where there is some code that is wrong. I found this lab to be very helpful in understanding how what we have been doing so far is supposed to work in a larger picture.

Additionally, I learned more about how to work with CLASSPATH to aid in compiling larger java programs.