Paul Kummer

Dr. Hanku Lee

Compilers CSIS 455-01

10/23/2021

Lab 06: Assignment 4 Overview

Purpose of Lab

This lab's purpose is to show students how to implement a solution for assignment 4. Then all the students will have more similar versions of their compilers and there will be less fragmentation of design.

Example Code

The code in Main.java and the lexer files remain unchanged from previous assignments. For the parser, the identifier node and The input.txt has had more expressions added to test out reading in multiple statements.

Visitor

ASTVisitor.java

```
public class ASTVisitor {

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

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

    public void visit (StatementsNode n)
    {
        if(n.stmts != null)
        {
            n.assign.accept(this); //This will have to change to accept function calls
            n.stmts.accept(this);
        }
    }

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

    public void visit (ExpressionNode n)
    {
        //n.term.accept(this);
        //n.expr.accept(this);
        //n.bin.accept(this);
    }

    public void visit (AssignmentNode n)
    {
        //n.id.accept(this);
        //n.right.accept(this);
    }

    public void visit (BinaryNode n)
    {
        //n.term.accept(this) ;
        //n.expr.accept(this) ;
    }

    public void visit (UnaryNode n)
    {
        //n.id.accept(this);
    }

    public void visit (TermNode n)
    {
        //n.unary.accept(this);
        //n.id.accept(this);
    }

    /*
    public void visit (Factor n)
    {

    }
    */

    //this is a variable
    public void visit (LiteralNode n)
    {
        //n.printNode();
    }

    //this is a terminal symbol number or string
    public void visit (IdentifierNode n)
    {
        //n.printNode();
    }
}
```

The ASTVisitor is mostly unchanged. However, it now accepts multiple recursive StatementsNodes to be called. For this program the ASTVisitor class is unused.

Parser

CompilationUnit.java

```
5   public class CompilationUnit extends Node
6   {
7       public BlockStatementNode block;
8
9       public CompilationUnit ()
10      {
11
12      }
13
14      public CompilationUnit (BlockStatementNode block)
15      {
16          this.block = block;
17      }
18
19      public void accept(ASTVisitor v)
20      {
21
22          v.visit(this);
23      }
24  }
```

The CompiliationUnit is unchanged and still accepts a BlockStatementNode as an attribute.

BlockStatementNode.java

```
7   public class BlockStatementNode extends Node
8   {
9       public StatementsNode stmts;
10
11      //start of singley linked list, if used
12      public StatementNode head = null;
13
14      public BlockStatementNode()
15      {
16
17      }
18
19      //
20      public BlockStatementNode(StatementsNode stmts)
21      {
22          this.stmt = stmts;
23      }
24
25      //use this for singley linked list version
26      public BlockStatementNode(StatementNode stmt)
27      {
28
29          this.head = stmt;
30      }
31
32      public void accept(ASTVisitor v)
33      {
34          v.visit(this);
35      }
36  }
```

The BlockStatementNode now has StatementsNode as an attribute.

StatementsNode.java

```java
public class StatementsNode
{
    /*
    assign will only allow a statement node to assign values. If
    a statement calls a function or declares a variable, this will not work.
    */
    public StatementsNode stmts = null;
    public AssignmentNode assign;

    //This is used in a different version
    //public StatementNode stmt = null;

    public StatementsNode()
    {

    }

    //This is used with the singley linked list
    public StatementsNode(StatementsNode stmts, StatementNode stmt)
    {
        this.stmt = stmt;
        this.stmts = stmts;
    }

    public StatementsNode(StatementsNode stmts, AssignmentNode assign)
    {
        this.assign = assign;
        this.stmts = stmts;
    }

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

Instead of using StatementNode and using a singly linked list of StatementNodes, now StatementsNode will use recursion to call more StatementsNode and will every new StatementsNode an AssignmentNode will be created. This will have to be changed in the future of the compiler, since more than assignments can occur with every statement.

AssignmentNode.java

```java
public class AssignmentNode extends Node
{
    public Token op;
    public TermNode  id;
    public ExpressionNode right; //this will allow unary and binary expressions

    public AssignmentNode ()
    {

    }

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

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

The AssignmentNode now takes in a TermNode as the left-hand-side of the assignment operator. This TermNode should always derive to a IdentifierNode. The Token op will always be the ‘=’ character and then the right-hand-side will be an ExpressionNode.

IdentifierNode.java

```
6   public class IdentifierNode extends Node
7   {
8       public Token w;
9       public String id;
10
11      public IdentifierNode()
12      {
13
14      }
15
16      public IdentifierNode(Token w)
17      {
18          this.w = w;
19          this.id = "" + (char)w.tag;
20      }
21
22      public IdentifierNode(Word w)
23      {
24          this.w = w;
25          this.id = w.lexeme;
26      }
27
28      public void accept(ASTVisitor v)
29      {
30          v.visit(this);
31      }
32
33      public void printNode()
34      {
35          System.out.print("IdentNode: " + id);
36      }
37  }
```

The IdentifierNode has not changed at all, but it now is being used more correctly as opposed to previous assignments where it was being used in place of a LiteralNode.

LiteralNode.java

```java
//This is for number or string and a terminal
public class LiteralNode extends Node
{
    public Num v;
    public int literal;
    public String string;

    public LiteralNode ()
    {

    }

    public LiteralNode (Num v)
    {
        this.v = v ;
        this.literal = v.value;
        this.string = "" + v.value;
    }

    public LiteralNode (String string)
    {
        this.string = string ;
    }

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

    public void printNode ()
    {
        System.out.println("LiteralNode: " + literal) ;
    }
}
```

The LiteralNode is now being used and it stores a number or a string literal. This node is a terminal symbol.

BinaryNode.java

```
7   public class BinaryNode
8   {
9       public Node left = null;
10      //public IdentifierNode leftId = null;
11      //public LiteralNode leftLit = null;
12      //public ExpressionNode leftExpr = null;
13      public Token op = null;
14      public ExpressionNode right = null;
15
16      public BinaryNode()
17      {
18
19      }
20
21      public BinaryNode(Node left)
22      {
23          this.left = left;
24      }
25
26      public BinaryNode(Node left, ExpressionNode right)
27      {
28          this.left = left;
29          this.right = right;
30      }
31
32      public void accept(ASTVisitor v)
33      {
34          v.visit(this);
35      }
36  }
```

The BinaryNode has made one minor change. Instead of op being an IdentifierNode, it is now just a token. This change doesn't really alter the program much. It still has a Node on the left-hand-side and an expression on the right-hand-side.

Parser.java

```java
public void visit (BinaryNode n)
{
    //n.left: handled by ExpressionNode

    System.out.println("BinaryNode");
    n.op = look;

    level++;
    dots();

    //Does not handle operator presedence
    switch(look.tag)
    {
        case '+' :
            System.out.println("Op: +");
            break;
        case '-' :
            System.out.println("Op: -");
            break;
        case '*' :
            System.out.println("Op: *");
            break;
        case '/' :
            System.out.println("Op: /");
            break;
        case '%' :
            System.out.println("Op: %");
            break;
        default :
            //System.out.println("\tThrowing an op Error!");
            error("Binary Expression Could Not Match Operator");
            break;
    }

    //advance past operator
    move();

    level--;

    //n.op.accept(this); don't use this because the match advances the tokens

    level++;
    dots();

    //second operand
    n.right = new ExpressionNode();
    n.right.accept(this);

    level--;
}

//Unary: child of term, always negative otherwise it is an identifier node
public void visit (UnaryNode n)
{
    System.out.println("UnaryNode");

    if(look.tag == '-')
    {
        //unary minus operator
        match('-');

        level++;
        dots();

        //first operand
        n.term = new TermNode();
        n.term.accept(this);

        level--;
    }
    else
    {
        error("Unary Error");
    }
}

//Term: child of expression, binary
public void visit (TermNode n)
{
    System.out.println("TermNode");

    level++;
    dots();

    if(look.tag == '-')
    {
        n.unary = new UnaryNode();
        n.unary.accept(this);
    }
    else if(look.tag == Tag.ID)
    {
        n.id = new IdentifierNode((Word)look);
        n.id.accept(this);
    }
    else if(look.tag == Tag.NUM)
    {
        n.lit = new LiteralNode((Num)look);
        n.lit.accept(this);
    }
    else
    {
        System.out.println("Error parsing TermNode");
    }

    level--;
}

public void visit (LiteralNode n)
{
    n.printNode();
    match(Tag.NUM);
}
```

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

    level++;
    dots();

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

    level--;
}

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

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

    //Read in all statements until the end of the block

    level++;
    dots();

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

    level--;

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

//Statement: child of block
public void visit (StatementsNode n)
{
    System.out.println("StatementsNode");

    if(look.tag != '}')
    {
        level++;
        dots();

        //n.stmt = new StatementNode();
        //n.stmt.accept(this);
        n.assign = new AssignmentNode();
        n.assign.accept(this);
        match(';');

        level--;

        dots();

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

//UNUSED: Statement: child of block or statementsndoe
public void visit (StatementNode n)
{
    System.out.println("StatementNode");
    //In a fully functional version, this should also accept a function call

    level++;
    dots();

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

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

    level--;
}
```

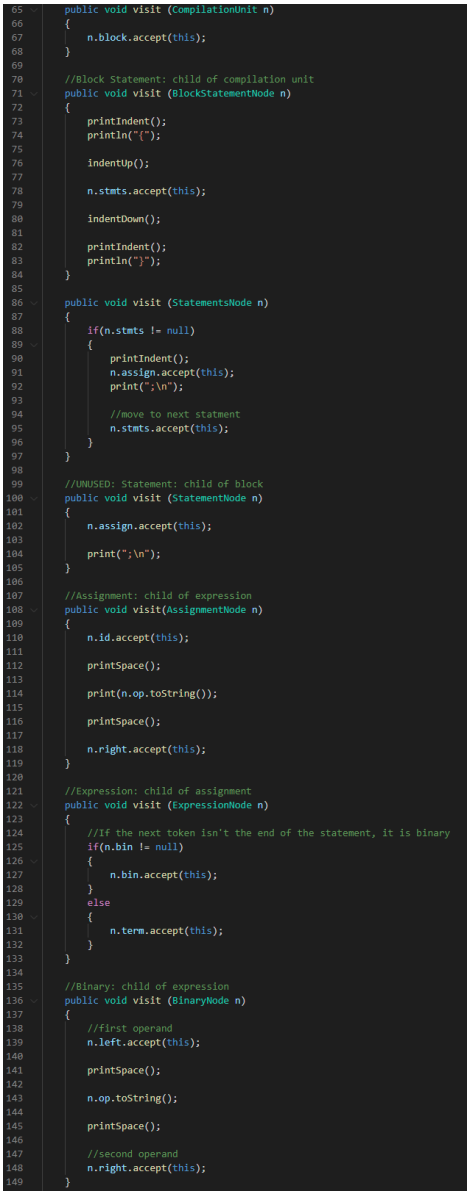
Parser.java Continued

```
196     //Assignment: child of expression
197     public void visit(AssignmentNode n)
198     {
199         System.out.println("AssignmentNode");
200 
201         level++;
202         dots();
203 
204         n.id = new TermNode();
205         n.id.accept(this);
206 
207         level--;
208 
209         level++;
210         dots();
211 
212         if(look.tag == '=')
213         {
214             n.op = look;
215             System.out.println("Op: =");
216         }
217         match('=');
218 
219         level--;
220 
221         level++;
222         dots();
223 
224         n.right = new ExpressionNode();
225         n.right.accept(this);
226 
227         level--;
228     }
229 
230     //Expression: child of assignment
231     public void visit (ExpressionNode n)
232     {
233         System.out.println("ExpressionNode");
234 
235         level++;
236         dots();
237 
238         //first operand or ID
239         n.term = new TermNode();
240         //This accept has to happen here or the token wont advance, causing the
241         //wrong look.tag for checking if it is binary
242         n.term.accept(this);
243 
244         level--;
245 
246 
247         //If the next token isn't the end of the statement, it is binary
248         if(look.tag != ';')
249         {
250             level++;
251             dots();
252 
253             n.bin = new BinaryNode(n.term);
254             n.bin.accept(this);
255 
256             level--;
257         }
258         else
259         {
260             //accept the id since it isn't binary
261         }
262     }
```

The Parser file undergone the most changes. The biggest change was the way StatementsNode recursively calls itself to derive an AssignmentNode and more StatementsNode. The next biggest changes were with BinaryNode and TermNode. BinaryNode now has its left-hand-side as a TermNode. This allows deriving to an IdentifierNode more easily. Also, the op attribute is no longer an IdentiferNode and is just a token instead. The BinaryNode still does not do operator precedence yet. Finally, the TermNode now can derive into a UnaryNode, IdentifierNode, or LiteralNode. This allows correct usage of IdentifierNode and LiteralNode, which was not done in the past.

PrettyPrinter

PrettyPrinter.java

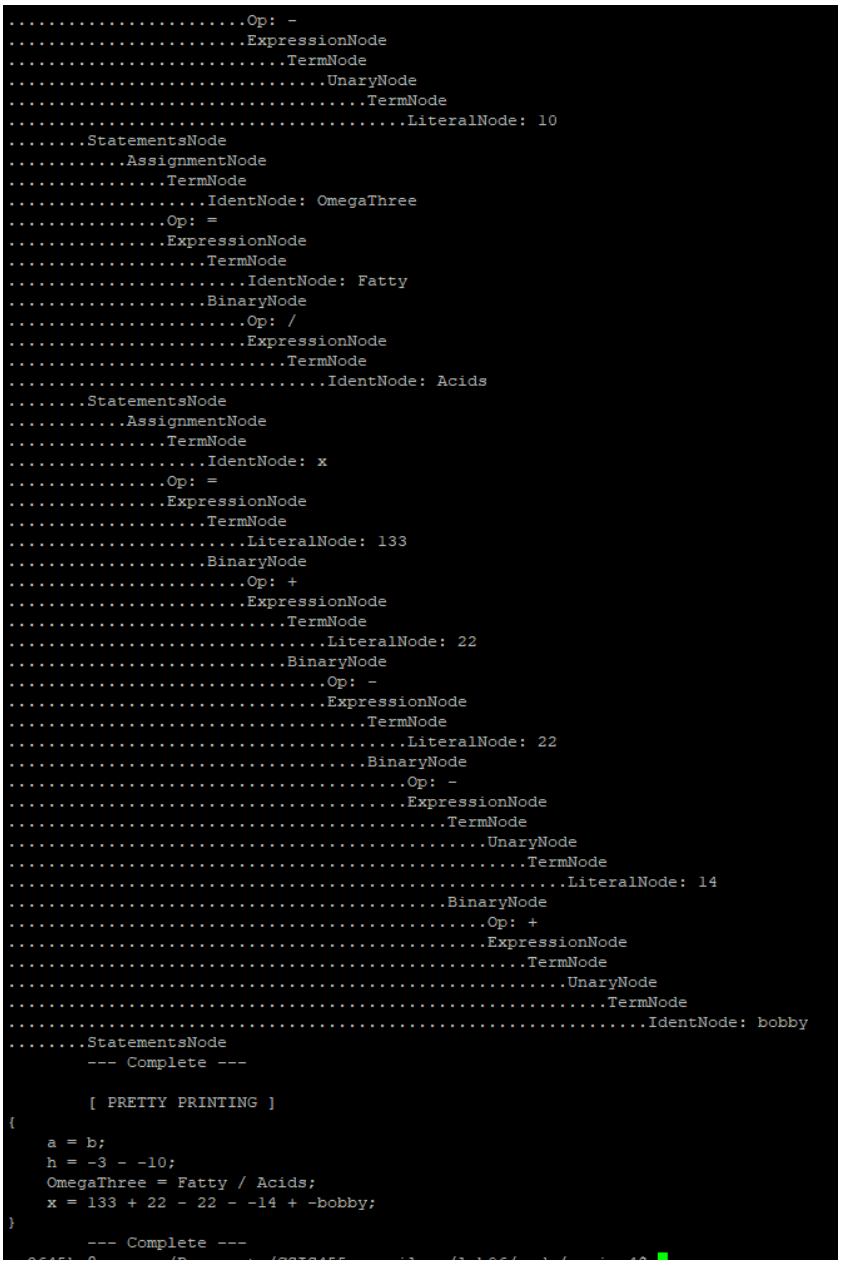


```
65      public void visit (CompilationUnit n)
66      {
67          n.block.accept(this);
68      }
69
70      //Block Statement: child of compilation unit
71      public void visit (BlockStatementNode n)
72      {
73          printIndent();
74          println("{");
75
76          indentUp();
77
78          n.stmts.accept(this);
79
80          indentDown();
81
82          printIndent();
83          println("}");
84      }
85
86      public void visit (StatementsNode n)
87      {
88          if(n.stmts != null)
89          {
90              printIndent();
91              n.assign.accept(this);
92              print(";\n");
93
94              //move to next statment
95              n.stmts.accept(this);
96          }
97      }
98
99      //UNUSED: Statement: child of block
100     public void visit (StatementNode n)
101     {
102         n.assign.accept(this);
103
104         print(";\n");
105     }
106
107     //Assignment: child of expression
108     public void visit(AssignmentNode n)
109     {
110         n.id.accept(this);
111
112         printSpace();
113
114         print(n.op.toString());
115
116         printSpace();
117
118         n.right.accept(this);
119     }
120
121     //Expression: child of assignment
122     public void visit (ExpressionNode n)
123     {
124         //If the next token isn't the end of the statement, it is binary
125         if(n.bin != null)
126         {
127             n.bin.accept(this);
128         }
129         else
130         {
131             n.term.accept(this);
132         }
133     }
134
135     //Binary: child of expression
136     public void visit (BinaryNode n)
137     {
138         //first operand
139         n.left.accept(this);
140
141         printSpace();
142
143         n.op.toString();
144
145         printSpace();
146
147         //second operand
148         n.right.accept(this);
149     }
```

```
151     //Unary: child of term, always negative otherwise it is an identifier node
152     public void visit (UnaryNode n)
153     {
154         n.term.accept(this);
155     }
156
157     //Term: child of expression, binary
158     public void visit (TermNode n)
159     {
160         if(n.unary != null)
161         {
162             print("-");
163             n.unary.accept(this);
164         }
165         else if(n.id != null)
166         {
167             n.id.accept(this);
168         }
169         else if(n.lit != null)
170         {
171             n.lit.accept(this);
172         }
173         else
174         {
175             println("An Error Occured");
176         }
177     }
178
179     //this will assign an identifier a string
180     public void visit (LiteralNode n)
181     {
182         print(n.literal);
183     }
184
185     public void visit (IdentifierNode n)
186     {
187         print(n.id);
188     }
```

The PrettyPrinter has been changed to accept the now recursive calls of StatementsNode. StatementsNode will stop the recursion when a null stmts attribute is encountered. With every statement that is read, a newline is started, and the previous line is ended with a semicolon. The other prominent change that was made, was printing out the tokens for operators instead of calling accept on the IdentifierNode that held the operators value.

Execution



When the input.txt file is read in, the program will turn all the characters, ignoring whitespace, into tokens during the lexing phase. Then the program will turn the token stream into nodes of organized information that has more meaning during the parsing phase. As the tokens are organized into nodes, an abstract syntax tree will be created and it will be visually output to the user. Finally, the pretty printing phase will format the input text into properly spaced and organized code segments. It will first start with a block statement. Each block statement will have multiple statements that are terminated with a semicolon. Within each statement, there can be many different types of nodes, but each statement will start off with an assignment node.

Conclusion

This assignment revealed some flaws in my groups design for assignment 4. However, most of assignment 4 was done in a very similar way to how the lab was done. I though it was interesting using recursion to read in statements. However, I like the singly linked list design more. I realized how I was using the Identifier and Literal nodes incorrectly, but now it makes more sense as to why the two different nodes exist.