Paul Kummer

Dr. Hanku Lee

Compilers CSIS 455-01

10/29/2021

Lab 07: Implementing Precedence with Binary Expression

Purpose of Lab

This lab's purpose is to implement a solution to create binary expressions with a proper abstract syntax tree (AST). With the previous examples of the binary expression method, operator precedence is not taken into consideration, so an addition operation could happen before a multiplication. With the new parseBinaryExpression method, a binary expression will be read, but if the operator precedence is higher than the previous, the method will be recursively called. By doing this, the AST will be shifted so that the lowest precedence operators will be the parents of the higher precedence operators, resulting in multiple binary branches what will be evaluated before the lower precedence operators. However, since the parseBinaryExpression method cannot know what operators are going to be read in ahead of time, it cannot accurately display a correct AST as it parses the tree. Therefore, another class called TreePrinter is created to display the hierarchy of the nodes in the AST.

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.

lexer

The lexer phase has been modified to include more keywords that will be used in future development.

lexer.java

```
    public Lexer () throws IOException, FileNotFoundException
    {
        br = new BufferedReader(new FileReader(file));
        reserve(Word.True);
        reserve(Word.False);
        reserve(Word.If);
        reserve(Word.Else);
        reserve(Word.Do);
        reserve(Word.While);
        reserve(Word.Break);
    }
```

Now there are more reserved words in the lexer, which will be used in future program development.

word.java

```
xer > Word.java > {} assign5.lexer
package assign5.lexer ;

public class Word extends Token {

    public String lexeme = "" ;

    public static final Word True   = new Word("true",  Tag.TRUE) ;
    public static final Word False  = new Word("false", Tag.FALSE) ;
    public static final Word If     = new Word("if", Tag.IF);
    public static final Word Else   = new Word("else", Tag.ELSE);
    public static final Word Do     = new Word("do", Tag.DO);
    public static final Word While  = new Word("while", Tag.WHILE);
    public static final Word Break  = new Word("break", Tag.BREAK);

    public Word (String s, int tag) {

        super(tag) ;
        lexeme = s ;
    }

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

New words were created, which are going to be keywords that are reserved for use in declarations.

tag.java

```
xer > Tag.java > {} assign5.lexer
package assign5.lexer ;

public class Tag
{
    public final static int FALSE   = 262;
    public final static int ID      = 264;
    public final static int NUM     = 270;
    public final static int TRUE    = 274;
    public final static int DO      = 280;
    public final static int WHILE   = 281;
    public final static int IF      = 282;
    public final static int ELSE    = 283;
    public final static int BREAK   = 284;
}
```

Tags are updated with more values so that parsing will be simpler when keywords are implemented over just using the assignment operation.

parsing

The parsing phase has now been modified to support operator precedence with every new expression. Also, as some additional features, the operations can be parenthesized and unary negate functionality is supported too. The parser cannot create an accurate AST.

parser.java

```java
Node parseBinaryNode (Node lhs, int precedence) //impossible to build a correct AST using this while it reads
{
    //timestamp - 52:44, 47:56
    // If the current op's precedence is higher than that of
    // the previous, then keep traversing down to create a new
    // BinaryNode for a binary expression with higher level precedence
    // Otherwise, create a new BinaryNode for the current lhs and rhs
    int levelDown = 0;
    Token opTok = null;
    Node rhs = null;
    int op = 0;
    lhs = lhs == null? new BinaryNode(): lhs;

    /*
    let the ')' signal an end to the recursion, but it must be cleared after
    the recursive calls complete. The lhs is returned if ')' is encountered
    */
    while(getPrecedence(look.tag) >= precedence && look.tag != ')')
    {
        opTok = look;
        op = getPrecedence(look.tag);

        level++;
        levelDown++;

        //must check before move for end of expression symbol ')'
        if(look.tag != ')')
        {
            move();//advance over the operator

            dots();

            rhs = new FactorNode(); //factor could be the end of the expression IE: a = 2 + ();
            rhs.accept(this);
        }

        if(look.tag != ';' && look.tag != ')')
        {
            if(getPrecedence(look.tag) > op)
            {
                level++;
                dots();
                System.out.println("operator: " + look);

                /*
                while the precedence of the current operator is greater than the previous
                operator, move the greater precedence down the AST by moving it from the
                right hand side to the left hand side and creating a new right hand side.
                */
                while(getPrecedence(look.tag) > op)
                {
                    //move the left hand side to right hand side
                    rhs = parseBinaryNode(rhs, getPrecedence(look.tag));

                    if(look.tag == ')') //end of expression
                    {
                        level--;
                        break;
                    }
                }

                level--;
            }
            else
            {
                dots();
                System.out.println("operator: " + look);

                //level -= levelDown;
                //return lhs;
            }
        }

        lhs = new BinaryNode(lhs, opTok, rhs);
    }

    level -= levelDown;
    return lhs;
}
```

This function will process all binary expressions and rotate the tree root to the operator with the lowest precedence first. Any operation that is parenthesized will be performed on its own and added as its own individual expression node to the AST. When the ‘)’ symbol is encountered all recursion stops and makes the calls return to the caller, which can be another expression. The visit to factorNode will handle all unary operations, calling a new expression, making a literal, or making an identifier.

parser.java cont.

```java
    //Statement: child of block
    public void visit (StatementsNode n)
    {
        //timestamp - 38:27

        if(look.tag != '}')
        {
            dots();
            System.out.println("StatementsNode");

            level++;
            dots();

            //This may need to change from assignment node
            n.assign = new AssignmentNode();
            n.assign.accept(this);
            match(';');

            level--;

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

    //Assignment: child of stmts
    public void visit(AssignmentNode n)
    {
        //timestamp - 38:49
        System.out.println("AssignmentNode");

        level++;
        dots();

        //left-hand-side
        n.left = new FactorNode();
        n.left.accept(this);

        level--;

        level++;
        dots();

        if(look.tag == '=')
        {
            n.op = look;
            System.out.println("Op: =");
        }
        else
        {
            error("AssignmentNode missing '=' operator");
        }
        match('=');

        level--;

        level++;
        dots();

        //Does this really need to test for identifier or literal
        //That happens in the termnode. The expressionNode also
        //checks for binary or unary
        n.right = new ExpressionNode();
        n.right.accept(this);

        level--;
    }
```

The visit to StatementsNode will read in all statements and call a new AssignmentNode with every new statement. The AssignmentNode will create a new FactorNode, which will be assigned a value from the ExpressionNode that is created next.

parser.java cont.

```
    //Expression: child of assignment | factor
    public void visit (ExpressionNode n)
    {
        //timestamp - 41:59 lecture video uses binary expression only
        System.out.println("ExpressionNode");

        level++;
        dots();

        FactorNode rhs_assign = new FactorNode();
        rhs_assign.accept(this);

        level--;

        if(look.tag == ';') //the expression is unary
        {
            //n.fact is rhs of assign now
            n.fact = rhs_assign;
        }
        else //the expression is binary
        {
            level++;
            dots();

            System.out.println("operator: " + look);

            level--;

            //n.bin is rhs of assign now
            n.bin = (BinaryNode)parseBinaryNode(rhs_assign, 0); //0 is the default level for operator precedence
            //System.out.println("\t---- Root Node Operator: " + n.bin.op + " ----");

            //This character must remain in parseBinaryNode to force the recursion to stop
            if(look.tag == ')')
            {
                move(); //clear the end of expression character
            }
        }
    }

    //Binary: child of expression | parseBinNode
    public void visit (BinaryNode n)
    {
        /*
        ALL BINARY NODES ARE HANDLED WITH parseBinaryNode

        */
    }
```

The visit to ExpressionNode will create a new FactorNode and visit it then determine if the assignment is a unary or binary assignment. If it is unary, the assignment node will only be assigned the value of the one factor node on the right-hand-side. If it is binary, then the parseBinaryNode() method is called to continue reading the rest of the operation and create a proper AST. It cannot display a correct AST as it reads tokens, only correct the AST as it reads tokens. Since the BinaryNode is handled completely by the parseBinNode() method, the BinaryNode visit method isn't needed anymore, but kept for historic purposes.

parser.java cont.

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

        level++;

        if(look.tag == '-')
        {
            dots();
            n.unary = new UnaryNode();
            n.unary.accept(this);
        }
        else if(look.tag == Tag.ID)
        {
            dots();
            n.id = new IdentifierNode((Word)look);
            n.id.accept(this);
        }
        else if(look.tag == Tag.NUM)
        {
            dots();
            n.lit = new LiteralNode((Num)look);
            n.lit.accept(this);
        }
        else if(look.tag == '(')
        {
            dots();
            match('(');
            n.expr = new ExpressionNode();
            n.expr.parenthesis = true;
            n.expr.accept(this);
        }
        else if(look.tag == ')')
        {
            level--;
            return;
        }
        else
        {
            error("FactorNode could not recognize the token");
        }

        level--;
    }

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

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

            level++;
            dots();

            //first operand
            n.fact = new FactorNode();
            n.fact.accept(this);

            level--;
        }
        else
        {
            error("UnaryNode there was no '-' before the term");
        }
    }
```

These two visit calls to Unary and Factor will handle creation of literals and identifiers. If the current token indicates some form of operation, the appropriate operation will occur like making a new expression or just return to the caller without giving any attributes a value.

unparser

The unparser has similar functionality as PrettyPrinter from the last lab. However, because the AST is updated as tokens are read in, what the unparser is displaying isn't an accurate reflection of the AST. It is however an accurate representation of what the parser is doing as tokens are read. The expression is displayed in infix format. To solve the way the AST is displayed, TreePrinter is created to show the correct AST in prefix format.

TreePrinter.java

unparser > TreePrinter.java > TreePrinter > visit(BinaryNode)

```java
package assign5.unparser ;

import assign5.visitor.* ;
import assign5.parser.* ;

public class TreePrinter extends ASTVisitor
{
    public Parser parser = null;

    int level = 0;
    String indent = "....";
    int indentLevel = 0;

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

    ////////////////////////////////////////////////
    //              Utility Methods               //
    ////////////////////////////////////////////////

    // print the dots for displaying the abstract syntax tree (AST)
    private void dots()
    {
        System.out.print(new String(new char[indentLevel*4]).replace('\0', '.'));
    }

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

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

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

    void indentUp()
    {
        indentLevel++;
    }

    void indentDown()
    {
        indentLevel--;
    }

    void printIndent()
    {
        String s = "";
        for(int indent=0; indent<indentLevel; indent++)
        {
            s += "    ";
        }

        print(s);
    }
```

```java
    ////////////////////////////////////////////////
    //               Visit Methods                //
    ////////////////////////////////////////////////

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

        indentUp();
        n.block.accept(this) ;
        indentDown();
    }

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

        indentUp();
        n.stmts.accept(this);
        indentDown();
    }

    public void visit (StatementsNode n)
    {
        if(n.stmts != null)
        {
            dots();
            println("StatementsNode");

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

    public void visit (AssignmentNode n)
    {
        dots();
        println("AssignmentNode");

        indentUp();
        dots();
        n.left.accept(this);
        indentDown();

        dots();
        println("op: =");

        indentUp();
        dots();
        n.right.accept(this);
        indentDown();
    }
```

TreePrinter.java cont.

```java
    public void visit (ExpressionNode n)
    {
        println("ExpressionNode");

        indentUp();
        if(n.fact != null)
        {
            dots();
            n.fact.accept(this);
        }

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

        if(n.bin != null)
        {
            dots();
            n.bin.accept(this);
        }
        indentDown();
    }

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

        indentUp();

        dots();
        println("op: " + n.op);

        dots();
        n.left.accept(this) ;

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

        indentDown();
    }

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

        indentUp();
        dots();
        n.fact.accept(this);
        indentDown();
    }
```

```java
    public void visit (FactorNode n)
    {
        if(n.unary != null || n.id != null || n.lit != null || n.expr != null)
        {
            println("FactorNode");

            indentUp();
            if(n.unary != null)
            {
                dots();
                n.unary.accept(this);
            }
            if(n.id != null)
            {
                dots();
                n.id.accept(this);
            }
            if(n.lit != null)
            {
                dots();
                n.lit.accept(this);
            }
            if(n.expr != null)
            {
                dots();
                n.expr.accept(this);
            }
            indentDown();
        }
    }

    //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();
    }

    public void visit(Node n)
    {
        //Do Nothing
    }
}
```

Visiting TreePrinter will result in all the nodes created during the parsing to be visited and display what level they are in the AST along with what their parent is and the operation to be performed. With every accept call the level in incremented and decremented afterwards. If a node is a leaf, then the value of that leaf is printed along with its level.

Execution

input.txt

```
input.txt - Notepad
File Edit Format View Help
                    {

sum =
-(
                    (-30* -31 + 32)
        +
                            -20
                    *
                            (50+-51)
);
a=c;
}
```

terminal

```
        [ PARSING ]
CompilationUnit
....BlockStatementNode
........StatementsNode
............AssignmentNode
................FactorNode
....................IdentNode: sum
................Op: =
................ExpressionNode
....................FactorNode
........................UnaryNode
............................FactorNode
................................ExpressionNode
....................................FactorNode
........................................ExpressionNode
............................................FactorNode
................................................UnaryNode
....................................................FactorNode
........................................................LiteralNode: 30
............................................operator: *
............................................FactorNode
................................................UnaryNode
....................................................FactorNode
........................................................LiteralNode: 31
............................................operator: +
................................................FactorNode
....................................................LiteralNode: 32
....................................operator: +
....................................FactorNode
........................................UnaryNode
............................................FactorNode
................................................LiteralNode: 20
........................................operator: *
............................................FactorNode
................................................ExpressionNode
....................................................FactorNode
........................................................LiteralNode: 50
....................................................operator: +
....................................................FactorNode
........................................................UnaryNode
............................................................FactorNode
................................................................LiteralNode: 51
....StatementsNode
........AssignmentNode
............FactorNode
................IdentNode: a
............Op: =
............ExpressionNode
................FactorNode
....................IdentNode: c
        --- Complete ---

        [ Unparsing ]
{
    sum = -((-30 * -31 + 32) + -20 * (50 + -51));
    a = c;
}
        --- Complete ---

        [ Tree Printer ]
CompilationUnit
....BlockStatementNode
........StatementsNode
............AssignmentNode
................FactorNode
....................IdentNode: sum
............op: =
................ExpressionNode
....................FactorNode
........................UnaryNode
............................FactorNode
................................ExpressionNode
....................................BinaryNode
........................................op: +
........................................FactorNode
............................................ExpressionNode
................................................BinaryNode
....................................................op: +
....................................................BinaryNode
........................................................op: *
........................................................FactorNode
............................................................UnaryNode
................................................................FactorNode
....................................................................LiteralNode: 30
........................................................FactorNode
............................................................UnaryNode
................................................................FactorNode
....................................................................LiteralNode: 31
....................................................FactorNode
........................................................LiteralNode: 32
........................................BinaryNode
............................................op: *
............................................FactorNode
................................................UnaryNode
....................................................FactorNode
........................................................LiteralNode: 20
............................................FactorNode
................................................ExpressionNode
....................................................BinaryNode
........................................................op: +
........................................................FactorNode
............................................................LiteralNode: 50
........................................................FactorNode
............................................................UnaryNode
................................................................FactorNode
....................................................................LiteralNode: 51
........StatementsNode
............AssignmentNode
................FactorNode
....................IdentNode: a
............op: =
................ExpressionNode
....................FactorNode
........................IdentNode: c
        --- Complete ---
```

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 demonstrated an alternate way to parse binary expressions in an effective way. However, because of the method used, a tree printer must be created as opposed to the method I was using previously, but it may be better in the long run because other operators with different precedence can be added easily. I think the method in this lab was a little harder to follow for parsing than my previous version because recursion can be a difficult concept to visualize what is happening. However, I did learn another way to parse a binary expression, and I was able to implement some of the functionality of my previous code to work with the parseBinaryNode method. I do think the TreePrinter class is a nice way to show the real AST and it helps with understanding what the parseBinaryNode was doing. Overall, this lab will be very useful for implementing new features later.