Paul Kummer

Dr. Hanku Lee

Compilers CSIS 455-01

10/03/2021

Lab 04: Lexing & Parsing

Purpose of Lab

The purpose of this lab is to show how to implement a Lexer and Parser. The Lexer will read a file and convert the characters from the file into tokens that will have a specified meaning. In the process of Lexing, whitespace and newlines will be removed. The line numbers will be tracked, which will be used in future debugging processes. Each token that is created will have a value saved to that token and is used in Parsing. A token can be a word, number, or just a token.

After Lexing, the parser will create an abstract syntax tree (AST). This AST will determine what should be done when evaluating the tokens. The lab demonstrates this by using the Visitor Pattern to visit each node of the tree as it is being created.

Example Code

Main

Main.java

This runs the main line of logic of the compiler. First, the Lexer will be called to run on the input.txt file. Then the lexer object is passed into the Parser, which will begin the parsing.

C: > Users > pakum > Desktop > compilers > lab04 > code > Main.java > ...

```
import java.io.*;
import java.util.*;

public class Main
{
    Run | Debug
    public static void main(String[] args) throws IOException
    {
        Lexer lexer = new Lexer();
        Parser parser = new Parser(lexer);
    }
}

```

Lexer

Lexer.java

The Lexer is responsible for turning characters into meaningful objects. First it will open a file and begin reading each character of the file. If the character is a number, more characters are read until there isn't a number. Then a number token will be created with an integer value. If the character being read is a letter, more letters or numbers will be read and concatenated until a space or operator is encountered. Then a word token will be created with a lexeme value.

While the Lexer is creating new tokens, each token is saved into a hashtable. This table is useful for preventing duplicates of tokens. Also, there can be reserved words too, which will be indicated with a tag value of "true". Otherwise, the words tag value will be "false", indicating it is not a reserved word.

C: > Users > pakum > Desktop > compilers > lab04 > code > Lexer.java > Lexer > scan()

```java
import java.io.*;
import java.util.*;

public class Lexer
{
    public int line = 1;
    private char peek = ' ';

    private FileInputStream in;
    private BufferedInputStream bin;

    private Hashtable<String, Word> words = new Hashtable<String, Word>();

    public Lexer()
    {
        reserve(new Word("true", Tag.TRUE));
        reserve(new Word("false", Tag.FALSE));

        setupIOStream();
    }

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

    void setupIOStream()
    {
        try
        {
            in = new FileInputStream("input.txt");
            bin = new BufferedInputStream(in);
        }

        catch(IOException e)
        {
            System.out.println("IOException: ");
        }
    }

    void readch() throws IOException
    {
        peek = (char) bin.read();
    }

    public Token scan() throws IOException
    {
        for( ; ; readch())
        {
            if(peek == ' ' || peek == '\t')
            {
                continue;
            }

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

            else
            {
                break;
            }
        }

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

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

            return new Num(v);
        }

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

            do
            {
                b.append(peek);
                readch();
            } while(Character.isLetterOrDigit(peek));

            String s = b.toString();
            Word w = words.get(s);

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

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

            return w;
        }

        Token t = new Token(peek);
        peek = ' ';

        return t;
    }
}
```

Token.java

Tokens are used to be the smallest meaningful unit in the compilation process. Each token has a value assigned to it that is held within a variable called tag. This tag value is important for Parsing.

C: > Users > pakum > Desktop > compilers > lab04 > code > Token.java > Token

```java
public class Token
{
    public final int tag;

    public Token (int t)
    {
        tag = t;
    }

    public String toString()
    {
        return "" + (char)tag;
    }
}
```

Tag.java

The Tag class is used to hold special identifying values that are used in the Parsing phase.

C: > Users > pakum > Desktop > compilers > lab04 > code > Tag.java > Tag > NUM

```
// File Tag.java

public class Tag
{
    public final static int
        NUM = 256,
        ID = 257,
        TRUE = 258,
        FALSE = 259,
        EOF = 65535;
}
```

Num.java

Num is a subclass of Token, and it stores a tag value of indicating it is a number. Also, a Num has an additional value, which is an integer associated with the Num token.

C: > Users > pakum > Desktop > compilers > lab04 > code > Num.java > Num

```java
public class Num extends Token

    public final int value;

    public Num(int v)
    {
        super(Tag.NUM);
        value = v;
    }

    public String toString()
    {
        return "" + value;
    }
}
```

Word.java

Word is a subclass of Token, and it stores a tag value indicating it is a word. Also, a word has an additional value, which is a lexeme stored as a string associated with the Word token.

C: > Users > pakum > Desktop > compilers > lab04 > code > Word.java > Word

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

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

Parser

Parser.java

The parser is responsible for evaluating what the tokens should do based on their tags. This will result in an abstract syntax tree (AST) being created.

First, the parser will begin to read the tokens from the Lexer, using the Lexer's "scan" method, and put them into a queue. Whenever a newline is encountered, the tokens that have been read so far will be considered an expression and are sent to a new compilation unit, where they will be evaluated. This process will continue until the end of file (EOF) tag is encountered.

C: > Users > pakum > Desktop > compilers > lab04 > code > Parser.java > Parser > Parser(Lexer)

```java
import java.io.*;
import java.util.*;

class Parser
{
    public Parser(Lexer lex) throws IOException
    {
        Queue<Token> tokens = new LinkedList<Token>();
        Token tok = lex.scan();
        int lineNum = lex.line;

        
        while(tok.tag != Tag.EOF)
        {
            if(tok != null)
            {
                tokens.add(tok);
            }

            tok = lex.scan();

            //This will preserve original newlines
            while(lex.line > lineNum)
            {
                Node root = new CompilationUnit(tokens);
                root.accept(new ASTVisitor());
                System.out.print("\n\n\t[ New Expression ]\n");

                //lineNum++; //This will add as many lines as there were in the document
                lineNum = lex.line; //This will add only one line

                tokens.clear();
            }
        }

        Node root = new CompilationUnit(tokens);
        root.accept(new ASTVisitor());
    }
}
```

Nodes

The Nodes are used in the visitor pattern and allow each node to have its own stored values and private methods, like a print method. Some nodes have a unique identifier, which is somewhat redundant because the token already has the value stored. More work should be done to utilize the tokens into nodes.

Node.java

Node is the base class used to build and AST and is utilized by other classes as a parent class.

C: > Users > pakum > Desktop > compilers > lab04 > code > Node.java > ...

```
import java.util.*;

public class Node
{
    Token tok = null;

    public Node()
    {
    }

    public Node(Token t)
    {
        tok = t;
    }

    public void accept(ASTVisitor v)
    {
        System.out.println("Node: " + tok.toString());
    }
}
```

LiteralNode.java

A literal node cannot be broken down any further and has a string representation.

C: > Users > pakum > Desktop > compilers > lab04 > code > LiteralNode.java > LiteralNode

```java
//will not call accept because the visitor method is not used

public class LiteralNode extends Node
{
    public String literal;

    public LiteralNode()
    {

    }

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

    @Override
    public void accept(ASTVisitor v)
    {
        printNode();
    }

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

AssignmentNode.java

An assignment node is used to assign a value to a node on the left-hand-side of the equals operator. The right-hand-side can be any expression represented as a node.

C: > Users > pakum > Desktop > compilers > lab04 > code > AssignmentNode.java > AssignmentNode

```java
public class AssignmentNode extends Node

    String ident = "=";

    Node left;
    Node right;

    public AssignmentNode()
    {

    }

    public AssignmentNode (Node l, Node r)
    {
        this.left = l;
        this.right = r;
    }

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

    void printNode()
    {
        System.out.println("Operand: " + ident);
    }
}
```

CompilationUnit.java

The compilation unit is the starting point of building an expression. It stores a queue of tokens that will become nodes and be evaluated into an AST.

C: > Users > pakum > Desktop > compilers > lab04 > code > CompilationUnit.java > ...

```java
import java.util.*;

public class CompilationUnit extends Node
{
    //public AssignmentNode assign;
    Queue<Token> toks;

    public CompilationUnit(Queue<Token> t)
    {
        this.toks = t;
    }

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

AdditionNode.java

The addition node holds two nodes, a left-hand-side, and a right-hand-side, that should be combined to form a new value.

```
C: > Users > pakum > Desktop > compilers > lab04 > code > AdditionNode.java > AdditionNode
public class AdditionNode extends Node
{
    String ident = "+";

    Node left;
    Node right;

    public AdditionNode()
    {

    }

    public AdditionNode (Node l, Node r)
    {
        this.left = l;
        this.right = r;
    }


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

    void printNode()
    {
        System.out.println("Operand: " + ident);
    }
}
```

SubtractioNode.java

The subtraction node holds two nodes, a left-hand-side, and a right-hand-side, that should be subtracted to form a new value.

C: > Users > pakum > Desktop > compilers > lab04 > code > SubtractionNode.java > SubtractionNode

```java
public class SubtractionNode extends Node
{
    String ident = "-";

    Node left;
    Node right;

    public SubtractionNode()
    {

    }

    public SubtractionNode (Node l, Node r)
    {
        this.left = l;
        this.right = r;
    }


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

    void printNode()
    {
        System.out.println("Operand: " + ident);
    }
}
```

MultiplicationNode.java

The multiplication node holds two nodes, a left-hand-side, and a right-hand-side, that should be multiplied to form a new value.

: > Users > pakum > Desktop > compilers > lab04 > code > MultiplicationNode.java > MultiplicationNode > MultiplicationNode(Node, Node)

```java
public class MultiplicationNode extends Node
{
    String ident = "*";

    Node left;
    Node right;

    public MultiplicationNode()
    {

    }

    public MultiplicationNode (Node l, Node r)
    {
        this.left = l;
        this.right = r;
    }


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

    void printNode()
    {
        System.out.println("Operand: " + ident);
    }
}
```

DivisionNode.java

The division node holds two nodes, a left-hand-side, and a right-hand-side, that should be divided to form a new value.

C: > Users > pakum > Desktop > compilers > lab04 > code > DivisionNode.java > DivisionNode > DivisionNode(Node, Node)

```java
public class DivisionNode extends Node
{
    String ident = "/";

    Node left;
    Node right;

    public DivisionNode()
    {

    }

    public DivisionNode (Node l, Node r)
    {
        this.left = l;
        this.right = r;
    }


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

    void printNode()
    {
        System.out.println("Operand: " + ident);
    }
}
```

ModuloNode.java

The modulo node holds two nodes, a left-hand-side, and a right-hand-side, that should be divided and have the remainder returned to form a new value.

C: > Users > pakum > Desktop > compilers > lab04 > code > ModuloNode.java > ModuloNode > ModuloNode(Node, Node)

```java
public class ModuloNode extends Node
{
    String ident = "%";

    Node left;
    Node right;

    public ModuloNode()
    {

    }

    public ModuloNode (Node l, Node r)
    {
        this.left = l;
        this.right = r;
    }


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

    void printNode()
    {
        System.out.println("Operand: " + ident);
    }
}
```

Visitors

ASTVisitor.java

The visitor is where the parsing really happens. When each node is visited, it will call accept on a node, which begins the processing of that node. To begin the parsing, the base node of compilation unit is visited first. This node will check if the next tokens in the queue are part of an operation or a literal. Based on what the first and second tokens are, switch statements will create a new node of whatever type is appropriate. For example, if the first node is a literal and the second node is '+', then an assignment node will be created and visited. However, if the first token is a literal and the second is null, then the literal node will be created and visited. If parentheses are encountered, a new compilation unit is created with a new queue of only the tokens within the parentheses. This compilation unit can be used as a node in any other operation.

When nodes other than the compilation unit are visited, the node's private values, which are nodes, will be accepted. Also, as this occurs, the level of the AST will be shown with dots and the values of the node will be displayed. That way a visualization of the AST can be seen.

ASTVisitor.java cont.

: > Users > pakum > Desktop > compilers > lab04 > code > ASTVisitor.java > ASTVisitor > visit(CompilationUnit)

```java
import java.util.*;

public class ASTVisitor
{
    int level = 0;

    public void visit(CompilationUnit n)
    {
        System.out.print("Compilation Unit:\n");

        level++;
        Node left;
        Node right;
        Node operation;

        Queue<Token> newToks = new LinkedList<Token>();
        Token first = n.toks.poll();
        Token op = n.toks.peek();

        if(op != null)
        {
            right = new CompilationUnit(n.toks);

            switch(first.tag)
            {
                case Tag.NUM: //the token is a number
                    left = new LiteralNode(first.toString());
                    break;
                case Tag.ID: //the token is a word
                    left = new LiteralNode(first.toString());
                    break;
                case (int) '(':
                    while((char)n.toks.peek().tag != ')' &&  n.toks.peek() != null && (char)first.tag != ')')
                    {
                        newToks.add(n.toks.poll());
                    }
                    n.toks.poll(); //remove the ')' token
                    left = new CompilationUnit(newToks);
                    op = n.toks.peek();
                    break;
                default:
                    left = new LiteralNode(first.toString());
                    break;
            }

            switch((char)op.tag)
            {
                case '=':
                    operation = new AssignmentNode(left, right);
                    break;
                case '+':
                    operation = new AdditionNode(left, right);
                    break;
                case '-':
                    operation = new SubtractionNode(left, right);
                    break;
                case '*':
                    operation = new MultiplicationNode(left, right);
                    break;
                case '/':
                    operation = new DivisionNode(left, right);
                    break;
                case '%':
                    operation = new ModuloNode(left, right);
                    break;
                default:
                    operation = left;
                    break;
            }
            n.toks.poll(); //remove the operation symbol
            operation.accept(this);
        }

        else
        {
            switch(first.tag)
            {
                case Tag.NUM: //the token is a number
                    left = new LiteralNode(first.toString());
                    dots();
                    left.accept(this);
                    break;
                case Tag.ID: //the token is a word
                    dots();
                    left = new LiteralNode(first.toString());
                    left.accept(this);
                    break;
                default:
                    if(first.tag != Tag.EOF)
                    {
                        dots();
                        left = new LiteralNode(first.toString());
                        left.accept(this);
                    }
                    break;
            }
        }
        level--;
    }
```

ASTVisitor.java cont.

C: > Users > pakum > Desktop > compilers > lab04 > code > ASTVisitor.java > ASTVisitor > visit(CompilationUnit)

```java

    public void visit(AssignmentNode n)
    {
        dots();
        System.out.print("Assignment:\n");


        level++;
        dots();
        n.left.accept(this);
        level--;

        level++;
        dots();
        n.printNode();
        level--;

        level++;
        dots();
        n.right.accept(this);
        level--;
    }

    public void visit (AdditionNode n)
    {
        dots();
        System.out.print("Addition:\n");

        level++;
        dots();
        n.left.accept(this);
        level--;

        level++;
        dots();
        n.printNode();
        level--;

        level++;
        dots();
        n.right.accept(this);
        level--;
    }

    public void visit (SubtractionNode n)
    {
        dots();
        System.out.print("Subtraction:\n");

        level++;
        dots();
        n.left.accept(this);
        level--;

        level++;
        dots();
        n.printNode();
        level--;

        level++;
        dots();
        n.right.accept(this);
        level--;
    }

    public void visit (MultiplicationNode n)
    {
        dots();
        System.out.print("Multiplication:\n");

        level++;
        dots();
        n.left.accept(this);
        level--;

        level++;
        dots();
        n.printNode();
        level--;

        level++;
        dots();
        n.right.accept(this);
        level--;
    }

    public void visit (DivisionNode n)
    {
        dots();
        System.out.print("Division:\n");

        level++;
        dots();
        n.left.accept(this);
        level--;

        level++;
        dots();
        n.printNode();
        level--;

        level++;
        dots();
        n.right.accept(this);
        level--;
    }
```

ASTVisitor.java cont.

```java
    public void visit (ModuloNode n)
    {
        dots();
        System.out.print("Modulo:\n");

        level++;
        dots();
        n.left.accept(this);
        level--;

        level++;
        dots();
        n.printNode();
        level--;

        level++;
        dots();
        n.right.accept(this);
        level--;
    }

    public void visit(LiteralNode n)
    {
        level++;
        dots();;
        n.accept(this);
        level--;
    }

    public void visit(Node n)
    {
        level++;
        dots();
        n.accept(this);
        level--;
    }

    private void dots()
    {
        System.out.print(new String(new char[level*4]).replace('\0', '.'));
    }
}
```

Code Execution

As the code executes, the Lexer begins to read in “input.txt”, which contains some expression written out on each line. Then the lexer is passed into the parser, which starts creating nodes and making an AST. This AST is displayed as output. Below is the expressions in the file and what the result is of the program’s evaluation.

input.txt

c=b+1+input

c=l+mmmm

input = b + result

x =   b * (c -2 * x) + 33

program execution

```
cx3645kg@smaug:~/Documents/CSIS455_compilers/lab04/code$ rm *.class
cx3645kg@smaug:~/Documents/CSIS455_compilers/lab04/code$ javac *.java
cx3645kg@smaug:~/Documents/CSIS455_compilers/lab04/code$ java Main
Compilation Unit:
....Assignment:
........Literal: c
........Operand: =
........Compilation Unit:
............Addition:
................Literal: b
................Operand: +
................Compilation Unit:
....................Addition:
........................Literal: 1
........................Operand: +
........................Compilation Unit:
............................Literal: input


        [ New Expression ]
Compilation Unit:
....Assignment:
........Literal: c
........Operand: =
........Compilation Unit:
............Addition:
................Literal: l
................Operand: +
................Compilation Unit:
....................Literal: mmmm


        [ New Expression ]
Compilation Unit:
....Assignment:
........Literal: input
........Operand: =
........Compilation Unit:
............Addition:
................Literal: b
................Operand: +
................Compilation Unit:
....................Literal: result


        [ New Expression ]
Compilation Unit:
....Assignment:
........Literal: x
........Operand: =
........Compilation Unit:
............Multiplication:
................Literal: b
................Operand: *
................Compilation Unit:
....................Addition:
........................Compilation Unit:
............................Subtraction:
................................Literal: c
................................Operand: -
................................Compilation Unit:
....................................Multiplication:
........................................Literal: 2
........................................Operand: *
........................................Compilation Unit:
............................................Literal: x
........................Operand: +
........................Compilation Unit:
............................Literal: 33
cx3645kg@smaug:~/Documents/CSIS455_compilers/lab04/code$ 
```

Conclusion

After this lab, I have a much better understanding of how the visitor pattern works. Also, I learned how to create a parser in a different way than I have done before. I have made a parser in C++ that uses recursive descent, which this parser utilizes a similar concept by using a compilation unit as a node that is visited within a different compilation unit. Initially, I found the code hard to follow because it is spread out across many different files, but many of the files are used in similar ways, so the processing only occurs in one place.

I found this lab to be difficult to do and hard to figure out how to make the previous assignments work with this lab. I think the connection between this lab and the other assignments was weak. I was able to see some similarities, but I wasn’t sure how to implement it with the parser. I was able to make it work, but I don’t know if I did it the best way. Overall, this lab challenged me, and I enjoyed figuring out how to make it work. I had to go back and watch other lecture videos, consult the Compilers textbook, and review the visitor pattern.