Lexical Analyzer

Compilers Project 1

Eric Kuxhausen

# Introduction

This project consisted of building a lexical analyzer for a modified version of the Pascal grammar. The role of the lexical analyzer is to read source written in the Pascal language and translate that into a list of tokens which can be analyzed by the next stage of a compiler. To complete this project, I first designed finite state automata on paper to recognize the various lexical components of Pascal, then programmed these machines and surrounding logic using the Java language.  
  
Methodology

To translate Pascal source code into a representation convenient to work with throughout the Pascal compiler as a whole, the source code is run through a Lexar, such as implemented in this project. The Lexar consists primarily of design time knowledge of the lexical conventions of Pascal, but it also augmented with knowledge of Pascal’s reserved words at runtime through use of a reserved words file. The reserved word file provides a mapping between the characters used to express each reserved word and the logical representation of that reserved word, which is used throughout the rest of the compiler. This feature further insulates the bulk of the compiler from aesthetic changes in the Pascal language.

# Implementation

This process, tokenization, is accomplished by a series of state machines each capable of processing different parts of the language. Most of the logic is encoded in finite state automate, but there are a few machines that use internal variables for convenience. Machines are grouped into java methods that return a Token representing what the machine has matched for the input under consideration. There is an order of precedence in the calling of the machines such that more general machines like Id or Int do not preclude the matching of more specific things like reserved words or Reals.

# Discussion and Conclusions

This project furthered my knowledge of finite automata and enum types in Java. I initially wanted to be able to uniquely distinguish between all tokens with a single attribute, intending to extensively take advantage of java’s type checking of enums. However, that proved impractical for a variety of reasons. For the benefit of the parser, there needed to be a convenient way to match whole categories of tokens, such as reserved words or relops. Additionally, the function that loads in the reserved word file would benefit from all of the reserved word types being grouped together. Upon deciding to create additional enums specific to reserved words, relops, addops, and mullops, I tried constructing the illusion of a union type in java by creating a wrapper object around the enums. However after multiple refactors I abandoned that and moved to the more traditional type, attribute structure suggested by this assignment’s input and output.

# References

* Compiler & Pascal References
  + Compilers Principles, Techniques, and Tools
    - Aho, Sethi, and Ullman.
    - ISBN 0201100886
  + Wikipedia page for Pascal
    - <http://en.wikipedia.org/wiki/Pascal_%28programming_language%29>
  + Christian Mann’s Pascal fuzz tester
    - http://personal.utulsa.edu/~christian-mann/cgi-bin/compilers/fuzz.cgi
* Java Programming References
  + Java 1.7 Docs
    - <http://docs.oracle.com/javase/7/docs>
  + StackOverflow.com

# Appendix I: Sample Inputs and Outputs

## reservedwords.txt

program PROGRAM 0

var VAR 0

array ARRAY 0

of OF 0

integer INT\_NAME 0

real REAL\_NAME 0

procedure PROC 0

begin BEGIN 0

end END 0

if IF 0

then THEN 0

else ELSE 0

while WHILE 0

do DO 0

call CALL 0

not NOT 0

or ADDOP 2

div MULOP 2

mod MULOP 3

and MULOP 4

## allLexValid.pas

program stuff ( things );.,:

array [2..5] of integer real

call procedure begin end

:= if then else while do

= <> < <= >= >

+- or

\*/ div mod and

01 2 3.4 5.6E-7 8.9E1

## allLexValid.listing

1 program stuff ( things );.,:

2 array [2..5] of integer real

3 call procedure begin end

4 := if then else while do

5 = <> < <= >= >

6 +- or

7 \*/ div mod and

8 01 2 3.4 5.6E-7 8.9E1

## allLexValid.token

Line No. Lexeme TOKEN-TYPE ATTRIBUTE

1 program RESWRD PROGRAM

1 stuff ID stuff

1 ( OPENPAREN NULL

1 things ID things

1 ) CLOSEPAREN NULL

1 ; SEMICOLON NULL

1 . EOF NULL

1 , COMMA NULL

1 : COLON NULL

2 array RESWRD ARRAY

2 [ OPENBRACKET NULL

2 2 NUM 2

2 .. DOTDOT NULL

2 5 NUM 5

2 ] CLOSEBRACKET NULL

2 of RESWRD OF

2 integer RESWRD INT\_NAME

2 real RESWRD REAL\_NAME

3 call RESWRD CALL

3 procedure RESWRD PROC

3 begin RESWRD BEGIN

3 end RESWRD END

4 := ASSIGNOP NULL

4 if RESWRD IF

4 then RESWRD THEN

4 else RESWRD ELSE

4 while RESWRD WHILE

4 do RESWRD DO

5 = RELOP EQ

5 <> RELOP NEQ

5 < RELOP LT

5 <= RELOP LTE

5 >= RELOP GTE

5 > RELOP GT

6 + ADDOP PLUS

6 - ADDOP MINUS

6 or ADDOP OR

7 \* MULOP TIMES

7 / MULOP SLASH

7 div MULOP DIV

7 mod MULOP MOD

7 and MULOP AND

8 01 NUM 01

8 2 NUM 2

8 3.4 NUM 3.4

8 5.6E-7 NUM 5.6E-7

8 8.9E1 NUM 8.9E1

## allLexErrors.pas

#@%! reallyLongWord 0001305 12345678901

12345678901.1 1.12345678901 00.1

12345678901.1E1 1.12345678901E1 1.1E123 00.1E1 0.1E00

## allLexErrors.listing

1 #@%! reallyLongWord 0001305 12345678901

LEXERR: Unrecog Symbol

LEXERR: Unrecog Symbol

LEXERR: Unrecog Symbol

LEXERR: Unrecog Symbol

LEXERR: Invalid ID: too long

LEXERR: Invalid INT: multiple leading zeros

LEXERR: Invalid INT: too long

2 12345678901.1 1.12345678901 00.1

LEXERR: Invalid REAL: xx too long

LEXERR: Invalid REAL: yy too long

LEXERR: Invalid REAL: multiple leading zeros in xx

3 12345678901.1E1 1.12345678901E1 1.1E123 00.1E1 0.1E00

LEXERR: Invalid REAL: xx too long

LEXERR: Invalid REAL: yy too long

LEXERR: Invalid REAL: zz too long

LEXERR: Invalid REAL: multiple leading zeros in xx

LEXERR: Invalid REAL: multiple leading zeros in zz

## allLexErrors.token

Line No. Lexeme TOKEN-TYPE ATTRIBUTE

1 # LEXERR Unrecog Symbol

1 @ LEXERR Unrecog Symbol

1 % LEXERR Unrecog Symbol

1 ! LEXERR Unrecog Symbol

1 reallyLongWord LEXERR Invalid ID: too long

1 0001305 LEXERR Invalid INT: multiple leading zeros

1 12345678901 LEXERR Invalid INT: too long

2 12345678901.1 LEXERR Invalid REAL: xx too long

2 1.12345678901 LEXERR Invalid REAL: yy too long

2 00.1 LEXERR Invalid REAL: multiple leading zeros in xx

3 12345678901.1E1 LEXERR Invalid REAL: xx too long

3 1.12345678901E1 LEXERR Invalid REAL: yy too long

3 1.1E123 LEXERR Invalid REAL: zz too long

3 00.1E1 LEXERR Invalid REAL: multiple leading zeros in xx

3 0.1E00 LEXERR Invalid REAL: multiple leading zeros in zz

# Appendix II: Program Listings

package kuxhausen;

import java.util.Scanner;

/\*\*

\* @author Eric Kuxhausen

\*/

public class Project1 {

public static void main(String[] args) {

for (String filename : args) {

Scanner file = Lexar.getFile("input/" + filename + ".pas");

if (file != null) {

Lexar l = new Lexar(file);

while (true) {

if (l.getNextToken() == null)

break;

}

Utils.writeListingFile("output/" + filename + ".listing", l.getTokenList(),

l.getSourceBuffer());

Utils.writeTokenFile("output/" + filename + ".token", l.getTokenList());

}

}

}

}

package kuxhausen;

import java.util.ArrayList;

import java.util.HashMap;

import java.util.Scanner;

import java.io.\*;

import static java.lang.System.out;

import static kuxhausen.Token.\*;

/\*\*

\* @author Eric Kuxhausen

\*/

public class Lexar {

private HashMap<String, Token> reservedWordTable = new HashMap<String, Token>();

private SourceBuffer source = new SourceBuffer();

private SourcePointer srcPos = new SourcePointer();

private SymbolTable symbols = new SymbolTable();

private ArrayList<Token> tokens = new ArrayList<Token>();

public Lexar(Scanner file) {

loadReservedWordTable();

while (file.hasNextLine()) {

// Read source into buffer

// Per project spec, only consider upto 71 characters per line including \n

String line = file.nextLine();

source.addLine(line.substring(0, Math.min(71, line.length())) + "\n");

}

file.close();

}

private void loadReservedWordTable() {

try {

Scanner wordFile = new Scanner(new BufferedReader(new FileReader("input/reservedwords.txt")));

while (wordFile.hasNextLine() && wordFile.hasNext()) {

String lexeme = wordFile.next();

String resType = wordFile.next();

int attribute = wordFile.nextInt();

if (resType.equals(TokType.ADDOP.toString())) {

reservedWordTable.put(lexeme, new Token(TokType.ADDOP, attribute, lexeme, srcPos));

} else if (resType.equals(TokType.MULOP.toString())) {

reservedWordTable.put(lexeme, new Token(TokType.MULOP, attribute, lexeme, srcPos));

} else {

for (ResWordAttr tt : ResWordAttr.values()) {

if (resType.equals(tt.toString())) {

reservedWordTable.put(lexeme, new Token(TokType.RESWRD, tt.ordinal(), lexeme, srcPos));

}

}

}

}

wordFile.close();

out.println("successfully loaded " + reservedWordTable.size()

+ " reserved words from reservedwords.txt");

} catch (FileNotFoundException e) {

out.println("reservedwords.txt not found");

}

}

public Token getNextToken() {

Token result = null;

result = reservedWordsMachine();

if (result == null) {

whitespaceMachine();

if (!source.hasNext(srcPos)) // check there is more after removing whitespace

return result;

result = idMachine();

}

if (result == null) {

result = realMachine();

}

if (result == null) {

result = intMachine();

}

if (result == null) {

result = relopMachine();

}

if (result == null) {

result = catchAllMachine();

}

if (result != null)

tokens.add(result);

return result;

}

private boolean isWhiteSpace(char c) {

if (c == ' ' || c == '\t' || c == '\n')

return true;

return false;

}

private boolean isLetter(char c) {

if (c >= 'a' && c <= 'z')

return true;

if (c >= 'A' && c <= 'Z')

return true;

return false;

}

private boolean isDigit(char c) {

if (c >= '0' && c <= '9')

return true;

return false;

}

private boolean isEOF(char c) {

return (c == '.');

}

private Token reservedWordsMachine() {

SourcePointer backup = srcPos.clone();

// first consume whitespace expected before id / reserved words

boolean hasConsumedWhitespace = false;

if (this.srcPos.lineNum == 0 && srcPos.charInLineNum == 0) {

hasConsumedWhitespace = true; // whitespace not needed before first char in source

}

while (source.hasNext(srcPos) && isWhiteSpace(source.peek(srcPos))) {

source.advanceChar(srcPos);

hasConsumedWhitespace = true;

}

if (hasConsumedWhitespace) {

String candidate = "";

// next consume one letter

if (source.hasNext(srcPos) && isLetter(source.peek(srcPos))) {

candidate += source.advanceChar(srcPos);

// next consume any following letters or digits

while (source.hasNext(srcPos)

&& (isLetter(source.peek(srcPos)) || isDigit(source.peek(srcPos)))) {

candidate += source.advanceChar(srcPos);

}

// if candidate is followed by whitespace or EOF

if (source.hasNext(srcPos)

&& (isWhiteSpace(source.peek(srcPos)) || isEOF(source.peek(srcPos)))) {

// check reserved word table

if (reservedWordTable.containsKey(candidate)) {

Token result = reservedWordTable.get(candidate).clone();

result.position = srcPos.clone();

return result;

}

}

}

}

// if no token matched, revert source pointer and return null

srcPos = backup;

return null;

}

private Token idMachine() {

SourcePointer backup = srcPos.clone();

String candidate = "";

// consume one letter

if (source.hasNext(srcPos) && isLetter(source.peek(srcPos))) {

candidate += source.advanceChar(srcPos);

// next consume any following letters or digits

while (source.hasNext(srcPos)

&& (isLetter(source.peek(srcPos)) || isDigit(source.peek(srcPos)))) {

candidate += source.advanceChar(srcPos);

}

if (candidate.length() > 10)

return new Token(TokType.LEXERR, "Invalid ID: too long", candidate, srcPos);

// Check add id to symbol table

Token t = new Token(TokType.ID, candidate, candidate, srcPos);

if (!symbols.table.containsKey(candidate))

symbols.table.put(candidate, t);

return t;

}

// if no token matched, revert source pointer and return null

srcPos = backup;

return null;

}

/\*\*

\* consumes whitespace

\*/

private void whitespaceMachine() {

while (source.hasNext(srcPos) && isWhiteSpace(source.peek(srcPos))) {

source.advanceChar(srcPos);

}

}

private Token relopMachine() {

SourcePointer backup = srcPos.clone();

if (source.hasNext(srcPos)) {

String lex = "" + source.advanceChar(srcPos);

switch (lex) {

case "=":

return new Token(TokType.RELOP, RelopAttr.EQ.ordinal(), lex, srcPos);

case "<":

if (source.hasNext(srcPos)) {

if (source.hasNext(srcPos) && source.peek(srcPos) == '>') {

lex += source.advanceChar(srcPos);

return new Token(TokType.RELOP, RelopAttr.NEQ.ordinal(), lex, srcPos);

} else if (source.hasNext(srcPos) && source.peek(srcPos) == '=') {

lex += source.advanceChar(srcPos);

return new Token(TokType.RELOP, RelopAttr.LTE.ordinal(), lex, srcPos);

} else {

return new Token(TokType.RELOP, RelopAttr.LT.ordinal(), lex, srcPos);

}

}

break;

case ">":

if (source.hasNext(srcPos)) {

if (source.hasNext(srcPos) && source.peek(srcPos) == '=') {

lex += source.advanceChar(srcPos);

return new Token(TokType.RELOP, RelopAttr.GTE.ordinal(), lex, srcPos);

} else {

return new Token(TokType.RELOP, RelopAttr.GT.ordinal(), lex, srcPos);

}

}

break;

}

}

// if no token matched, revert source pointer and return null

srcPos = backup;

return null;

}

private Token intMachine() {

SourcePointer backup = srcPos.clone();

if (source.hasNext(srcPos) && isDigit(source.peek(srcPos))) {

String lex = "" + source.advanceChar(srcPos);

while (source.hasNext(srcPos) && isDigit(source.peek(srcPos))) {

lex += source.advanceChar(srcPos);

}

if (lex.startsWith("00"))

return new Token(TokType.LEXERR, "Invalid INT: multiple leading zeros", lex, srcPos);

if (lex.length() > 10)

return new Token(TokType.LEXERR, "Invalid INT: too long", lex, srcPos);

return new Token(TokType.NUM, lex, lex, srcPos);

}

// if no token matched, revert source pointer and return null

srcPos = backup;

return null;

}

private Token realMachine() {

SourcePointer backup = srcPos.clone();

String lex = "";

int xCount = 0;

boolean hasDot = false;

int yCount = 0;

boolean hasExp = false;

int zCount = 0;

while (source.hasNext(srcPos) && isDigit(source.peek(srcPos))) {

xCount++;

lex += source.advanceChar(srcPos);

}

if (source.hasNext(srcPos) && source.peek(srcPos) == '.') {

hasDot = true;

lex += source.advanceChar(srcPos);

while (source.hasNext(srcPos) && isDigit(source.peek(srcPos))) {

yCount++;

lex += source.advanceChar(srcPos);

}

}

SourcePointer notLongBackup = srcPos.clone();

if (source.hasNext(srcPos) && (source.peek(srcPos) == 'E' || source.peek(srcPos) == 'e')) {

hasExp = true;

lex += source.advanceChar(srcPos);

if (source.hasNext(srcPos) && (source.peek(srcPos) == '+' || source.peek(srcPos) == '-')) {

lex += source.advanceChar(srcPos);

}

while (source.hasNext(srcPos) && isDigit(source.peek(srcPos))) {

zCount++;

lex += source.advanceChar(srcPos);

}

}

if (xCount > 0 && hasDot && yCount > 0) {

if (lex.startsWith("00"))

return new Token(TokType.LEXERR, "Invalid REAL: multiple leading zeros in xx", lex, srcPos);

if (xCount > 5)

return new Token(TokType.LEXERR, "Invalid REAL: xx too long", lex, srcPos);

if (yCount > 5)

return new Token(TokType.LEXERR, "Invalid REAL: yy too long", lex, srcPos);

if (hasExp && zCount > 0) {

if (zCount > 2)

return new Token(TokType.LEXERR, "Invalid REAL: zz too long", lex, srcPos);

else if (lex.substring(lex.length() - zCount).startsWith("00"))

return new Token(TokType.LEXERR, "Invalid REAL: multiple leading zeros in zz", lex, srcPos);

else

return new Token(TokType.NUM, lex, lex, srcPos);

} else {

srcPos = notLongBackup;

return new Token(TokType.NUM, lex, lex, srcPos);

}

}

// if no token matched, revert source pointer and return null

srcPos = backup;

return null;

}

private Token catchAllMachine() {

SourcePointer backup = srcPos.clone();

String lex = "" + source.advanceChar(srcPos);

switch (lex) {

case "(":

return new Token(TokType.OPENPAREN, null, lex, srcPos);

case ")":

return new Token(TokType.CLOSEPAREN, null, lex, srcPos);

case ";":

return new Token(TokType.SEMICOLON, null, lex, srcPos);

case ",":

return new Token(TokType.COMMA, null, lex, srcPos);

case "[":

return new Token(TokType.OPENBRACKET, null, lex, srcPos);

case "]":

return new Token(TokType.CLOSEBRACKET, null, lex, srcPos);

case "+":

return new Token(TokType.ADDOP, AddopAttr.PLUS.ordinal(), lex, srcPos);

case "-":

return new Token(TokType.ADDOP, AddopAttr.MINUS.ordinal(), lex, srcPos);

case "\*":

return new Token(TokType.MULOP, MulopAttr.TIMES.ordinal(), lex, srcPos);

case "/":

return new Token(TokType.MULOP, MulopAttr.SLASH.ordinal(), lex, srcPos);

}

if (lex.equals(":")) {

if (source.hasNext(srcPos) && source.peek(srcPos) == '=') {

lex += source.advanceChar(srcPos);

return new Token(TokType.ASSIGNOP, null, lex, srcPos);

} else

return new Token(TokType.COLON, null, lex, srcPos);

} else if (lex.equals(".")) {

if (source.hasNext(srcPos) && source.peek(srcPos) == '.') {

lex += source.advanceChar(srcPos);

return new Token(TokType.DOTDOT, null, lex, srcPos);

} else {

return new Token(TokType.EOF, null, lex, srcPos);

}

}

Token err = new Token(TokType.LEXERR, "Unrecog Symbol", lex, srcPos);

return err;

}

public void computeProjectZero() {

for (int i = 0; i < source.getNumLines(); i++) {

out.print(i + ". " + source.getLine(i));

}

}

public static Scanner getFile(String filepath) {

try {

return new Scanner(new BufferedReader(new FileReader(filepath)));

} catch (FileNotFoundException e) {

out.println("Source not found at " + filepath);

return null;

}

}

public ArrayList<Token> getTokenList() {

return tokens;

}

public SourceBuffer getSourceBuffer() {

return source;

}

}

package kuxhausen;

import java.util.ArrayList;

/\*\*

\* @author Eric Kuxhausen Stores source code with the requirements of: preserving line numbers,

\* providing access by line number, and facilitating per-character linear traversal with

\* SourcePointers.

\*/

public class SourceBuffer {

private ArrayList<String> sourceBuffer = new ArrayList<String>();

public SourceBuffer() {}

public void addLine(String line) {

sourceBuffer.add(line);

}

public int getNumLines() {

return sourceBuffer.size();

}

public String getLine(int number) {

return sourceBuffer.get(number);

}

public boolean hasNext(SourcePointer position) {

if (position.lineNum < sourceBuffer.size()

&& position.charInLineNum < sourceBuffer.get(position.lineNum).length()) {

return true;

} else

return false;

}

/\*\*

\* guard with hasNextCharacter() to prevent out of bounds issues

\*/

public char peek(SourcePointer position) {

return sourceBuffer.get(position.lineNum).charAt(position.charInLineNum);

}

public char advanceChar(SourcePointer position) {

if (hasNext(position)) {

char result = peek(position);

if (position.charInLineNum < sourceBuffer.get(position.lineNum).length() - 1) {

position.charInLineNum++;

} else {

position.lineNum++;

position.charInLineNum = 0;

}

return result;

}

return 0;

}

}

package kuxhausen;

/\*\*

\* @author Eric Kuxhausen

\*/

public class SourcePointer implements Cloneable {

public int lineNum;

public int charInLineNum;

@Override

public SourcePointer clone() {

SourcePointer copy = new SourcePointer();

copy.lineNum = lineNum;

copy.charInLineNum = charInLineNum;

return copy;

}

}

package kuxhausen;

/\*\*

\* @author Eric Kuxhausen

\*/

import java.util.HashMap;

public class SymbolTable {

public HashMap<String, Token> table;

public SymbolTable() {

table = new HashMap<String, Token>();

}

}

package kuxhausen;

/\*\*

\* @author Eric Kuxhausen

\*/

public class Token implements Cloneable {

public TokType type;

public Object attribute;

public String lexeme;

public SourcePointer position;

public Token(TokType t, int attr, String lex, SourcePointer pos) {

this(t, (Object) attr, lex, pos);

}

public Token(TokType t, String attr, String lex, SourcePointer pos) {

this(t, (Object) attr, lex, pos);

}

private Token(TokType t, Object attr, String lex, SourcePointer pos) {

type = t;

attribute = attr;

lexeme = lex;

position = (pos != null) ? pos.clone() : null;

}

public Token clone() {

return new Token(type, attribute, lexeme, position.clone());

}

public PasType getNumType() {

if (type == TokType.NUM) {

if (lexeme.contains("."))

return PasType.REAL;

else

return PasType.INT;

}

return PasType.ERR;

}

public String getAttribute() {

if (attribute != null) {

if (attribute instanceof Integer && (int) attribute != -1) {

switch (type) {

case RESWRD:

return ResWordAttr.values()[(int) attribute].toString();

case RELOP:

return RelopAttr.values()[(int) attribute].toString();

case ADDOP:

return AddopAttr.values()[(int) attribute].toString();

case MULOP:

return MulopAttr.values()[(int) attribute].toString();

}

} else if (!(attribute instanceof Integer)) {

return attribute.toString();

}

}

return "NULL";

}

public RelopAttr getRelop() {

return RelopAttr.values()[(int) attribute];

}

public MulopAttr getMulop() {

return MulopAttr.values()[(int) attribute];

}

public AddopAttr getAddop() {

return AddopAttr.values()[(int) attribute];

}

public boolean fullTypeMatch(Token other) {

if (type == other.type) {

// if one of these types, have to compare attributes as well

if (type == TokType.RESWRD || type == TokType.RELOP || type == TokType.ADDOP

|| type == TokType.MULOP) {

// unless the attribute wasn't specified, in which case it's a wildcard

if ((int) attribute == -1 || (int) other.attribute == -1) {

return true;

}

if (((int) attribute) == ((int) other.attribute)) {

return true;

}

} else {

return true;

}

}

return false;

}

public static enum TokType {

RESWRD, ID, EOF, NUM, RELOP, ADDOP, MULOP, LEXERR, SYNTAXERR, SEMANTICERR, OPENPAREN, CLOSEPAREN, SEMICOLON, COMMA, COLON, OPENBRACKET, DOTDOT, CLOSEBRACKET, ASSIGNOP, $

}

public static enum ResWordAttr {

PROGRAM, VAR, ARRAY, OF, INT\_NAME, REAL\_NAME, PROC, BEGIN, END, IF, THEN, ELSE, WHILE, DO, CALL, NOT

}

public static enum RelopAttr {

EQ, NEQ, LT, LTE, GTE, GT

}

public static enum AddopAttr {

PLUS, MINUS, OR

}

public static enum MulopAttr {

TIMES, SLASH, DIV, MOD, AND

}

}

package kuxhausen;

import java.io.FileNotFoundException;

import java.io.PrintWriter;

import java.util.ArrayList;

import static kuxhausen.Token.\*;

public class Utils {

public static void writeListingFile(String filename, ArrayList<Token> tokens, SourceBuffer source) {

PrintWriter output = null;

try {

output = new PrintWriter(filename);

} catch (FileNotFoundException e) {

}

int lineNo = -1;

for (Token t : tokens) {

while (t.position.lineNum > lineNo) {

lineNo++;

output.print(String.format("%-8s", "" + (lineNo + 1)) + source.getLine(lineNo));

}

if (t.type == TokType.LEXERR)

output.println("LEXERR: " + t.attribute);

if (t.type == TokType.SYNTAXERR)

output.println("SYNTAXERR: " + t.attribute);

}

output.close();

}

public static void writeTokenFile(String filename, ArrayList<Token> tokens) {

PrintWriter output = null;

try {

output = new PrintWriter(filename);

} catch (FileNotFoundException e) {

}

String formatting = "%-10s%-20s%-20s%-10s";

output.println(String.format(formatting, "Line No.", "Lexeme", "TOKEN-TYPE", "ATTRIBUTE"));

for (Token t : tokens) {

if (t.type != TokType.$) {

output.println(String.format(formatting, (t.position.lineNum + 1), t.lexeme,

t.type.toString(), t.getAttribute()));

}

}

output.close();

}

}