**CS 4013 Project 1:**

**Lexical Analyzer**

David Glover

October 1, 2014

**Table of Contents**

**Introduction** …………………………………………………………………………………… 1

**Methodology** …………………………………………………………………………………... 1

**Implementation** ……………………………………………………………………………….. 2

**Discussion and Conclusions** ………………………………………………………………. 3

**References** …………………………………………………………………………………….. 3

**Appendix Ia: Sample Inputs** …………………………………..…………………………..... 4

**Appendix Ib: Sample Outputs** ……………………………………………………………... 5

**Appendix II: Program Listings** ……………………………………………………………. 10

**Introduction**

The compilation process is made up of six main phases: lexical analysis, syntax analysis, semantic analysis, intermediate code generation, code optimization, and target code generation. The role of a lexical analyzer is to “read the input characters [from the source program] and produce as output a sequence of tokens that the parser uses for syntax analysis.” (Aho 84) The largest part of this project is interpreting the stream of characters and classifying the resulting tokens as different types for use by the syntax analyzer, then next step of the compilation process.

**Methodology**

The lexical analyzer is a fairly straightforward program that separates tokens from a source program file to a token file, where they are listed and assigned token numbers and attribute numbers.

The first step of lexical analysis is to go through the source program line by line and separate the tokens based on their type (ARITHOP, RELOP, ID, reserved word, etc.). This can be done by reading the line character by character and determining the type of machine the character should be sorted to in order to create a token of that type. If the character is invalid or does not meet the form that is specified by the lexical analyzer, it is reported in the output files as type LEXERR.

After the program is read in and the tokens are separated, the lexical analyzer then writes to a listing file each line of the source program, including any errors found for each line. It will also write the tokens it has kept track of to a token file. Finally, while scanning the source program, the lexical analyzer will keep track of identifiers that are not reserved words, and store them in a symbol table, without duplicates.

**Implementation**

To implement the lexical analyzer, I decided to use the Python programming language. My program begins with creating all of the input and output files specific to the source programs, whether they have or do not have lexical errors in them, and passing them to the lexical analyzer. After being called by the driver, all the components necessary for the lexical analyzer to run are initialized. It then begins to loop through the passed source program line by line. In each loop, the line is written to the listing file as a string and to the buffer as a list of characters. Then, the program looks at the first character in the buffer and determines what type of character it is (i.e. letter, number, symbol, etc.) and places it in the string corresponding to the token type. From there, it loops through the rest of the characters in the buffer and sends them to their corresponding token type. When the last character in the buffer has been reached, the errors that have accumulated during the reading of the line are written to the listing file and the error list is cleared for the next line. Then, the tokens are written to the token file, including their type and attribute information. Finally, the buffer is emptied and the lexical analyzer loops to the next line of the source program. When the lexical analyzer reaches the end of the source program, it simply exits the loop and the program is complete.

When the characters are passed from the main loop, they go to a function full of if/elif/else statements to determine what type the character is. They are then passed to a function that decides which function the character goes to based on its type to be added to its respective token. My program has functionality for white space, relops, mulops, arithops, identifiers, numbers, and others.

The white space function, which catches spaces and blank lines, is simply passed over and the program continues.

The relop function, which catches colons, equal signs, and less than/greater than signs, adds the passed character to its respective string. If the character passed to it is a different type from the next character, the string is evaluated and stored as a token, and is then emptied.

The mulop and arithop functions, which catch multiplication and division signs and plus and minus signs, respectively, work similarly to the relop function.

The identifier function catches all valid identifiers of form (letter)(letter or number)… shorter than 10 characters. If this size is exceeded, an error is added to the error list and the token is typed LEXERR. Otherwise, the program loops through characters and adds them to the respective string until the next character is of a different type. It then will check if the identifier is a reserved word. If not, it is added to the symbol table if it hasn’t already. Finally, the identifier’s storage string is emptied.

The number function checks the character, which could be a number, a decimal point, or ‘e.’ In short, this function keeps a number variable to store the number that is passed to the function one character at a time. It checks if the integer is ten digits or less, the characteristic and mantissa of the number are each five characters or less, and the exponent two characters or less, and adds errors to the error list and changes the type to LEXERR otherwise. If the number is valid, it will be separated to either ‘Integer,’ ‘Real,’ or ‘Long real.’

Finally, the other function takes care of any other symbols that are valid, including parentheses, brackets, semicolons, colons, commas, and dots not being used for numbers. Any other symbol passed to this function must be invalid since it did not fit with any other function.

**Discussion and Conclusions**

When this project was first assigned, I immediately thought that this would be one of the hardest software projects I had taken on here at TU. However, I broke it up into smaller, more manageable pieces and found it easier to work on it. I split the project up into input/output functionality, character typing and determining the machine they need to be sorted to, and the machines themselves. The input/output coding did not take long, even though this was the first “real” program I had written in Python. Determining the types for the characters took a little more time, but after studying samples of Pascal code, I was able to determine how the different character types related to the different machine types. Writing the machines proved to be the toughest, but most important, part of this project. I worked out some examples on pencil and paper to more easily visualize how the characters should be added to strings, and in turn to token entries or errors.

Overall, the project ended up taking longer than I had hoped due to other classwork, but after splitting the project into bite-sized chunks, it ended up being slightly easier than I had originally expected. I look forward to proceeding with the next project, the syntax analyzer, and learning how exactly it contributes to the compilation process.

**References**

Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*.

1st ed. Bell Laboratories, 1986. Print.

**Appendix Ia: Sample Inputs**

**src\_program\_no\_errors**

program test (input, output);

var a : integer;

var b : real;

var c : array [1 .. 5] of integer;

var d : real;

var t : boolean;

function fun1(x:integer; y:real;

z:array [1 .. 2] of integer;

q: real) : integer;

begin

x := z[1] \* 2;

y := (z[1] - z[2]) div 4;

z[1] := x mod 2;

q := y + 1.23e2;

end;

begin

if b = 0 then b := a \* 3.14

b := c[4] / c[5];

t := d >= a;

a := a or c[1];

a := c[1] and c[3];

end.

**src\_program\_with\_errors**

$@

var a : array [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14] of integer;

extralongtoken

12345678901

123456.789012e345

**src\_program\_syntax\_errors**

3.4E

3.4E+

3.E+

3.4E

3.E4

$@

**reserved\_words**

and 3 MULOP 3 AND

array 8 ARRAY 0 NULL

begin 9 BEGIN 0 NULL

boolean 10 BOOLEAN 0 NULL

div 3 MULOP 4 DIV

do 11 DO 0 NULL

else 12 ELSE 0 NULL

end 13 END 0 NULL

function 14 FUNCTION 0 NULL

if 15 IF 0 NULL

integer 16 INTEGER 0 NULL

mod 3 MULOP 5 MOD

of 17 OF 0 NULL

or 4 ARITHOP 3 OR

program 18 PROGRAM 0 NULL

real 19 REAL 0 NULL

then 20 THEN 0 NULL

var 21 VAR 0 NULL

while 22 WHILE 0 NULL

**Appendix Ib: Sample Outputs**

**listing\_file\_without\_errors**

1 program test (input, output);

2 var a : integer;

3 var b : real;

4 var c : array [1 .. 5] of integer;

5 var d : real;

6 var t : boolean;

7

8 function fun1(x:integer; y:real;

9 z:array [1 .. 2] of integer;

10 q: real) : integer;

11 begin

12 x := z[1] \* 2;

13 y := (z[1] - z[2]) div 4;

14 z[1] := x mod 2;

15 q := y + 1.23e2;

16 end;

17

18 begin

19 if b = 0 then b := a \* 3.14

20 b := c[4] / c[5];

21 t := d >= a;

22 a := a or c[1];

23 a := c[1] and c[3];

24 end.

**listing\_file\_with\_errors**

1 $@

LEXERR: Unrecognized symbol: $

LEXERR: Unrecognized symbol: @

2 var a : array [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14] of integer;

LEXERR: Line exceeds buffer size

3 extralongtoken

LEXERR: Token exceeded max length: extralongtoken

4 12345678901

LEXERR: Extra long integer: 12345678901

5 123456.789012e345

LEXERR: Extra long characteristic: 123456.789012e345

LEXERR: Extra long mantissa: 123456.789012e345

LEXERR: Extra long exponent: 123456.789012e345

**listing\_file\_syntax\_errors**

1 3.4E

2 3.4E+

3 3.E+

4 3.4E

5 3.E4

6 $@

LEXERR: Unrecognized symbol: $

LEXERR: Unrecognized symbol: @

**token\_file\_without\_errors**

Line No. Lexeme Token Type Attribute/Address

1 program 18 (PROGRAM) 0 (NULL)

1 test 1 (ID) e632973e (PTR TO SYM TAB)

1 ( 24 (LEFT PAREN) 0 (NULL)

1 input 1 (ID) 54cdb9f6 (PTR TO SYM TAB)

1 , 26 (COMMA) 0 (NULL)

1 output 1 (ID) 23542cfc (PTR TO SYM TAB)

1 ) 25 (RIGHT PAREN) 0 (NULL)

1 ; 23 (SEMICOLON) 0 (NULL)

2 var 21 (VAR) 0 (NULL)

2 a 1 (ID) 7fe6c96e (PTR TO SYM TAB)

2 : 2 (RELOP) 7 (COLON)

2 integer 16 (INTEGER) 0 (NULL)

2 ; 23 (SEMICOLON) 0 (NULL)

3 var 21 (VAR) 0 (NULL)

3 b 1 (ID) f92486ed (PTR TO SYM TAB)

3 : 2 (RELOP) 7 (COLON)

3 real 19 (REAL) 0 (NULL)

3 ; 23 (SEMICOLON) 0 (NULL)

4 var 21 (VAR) 0 (NULL)

4 c 1 (ID) 6e5b9acb (PTR TO SYM TAB)

4 : 2 (RELOP) 7 (COLON)

4 array 8 (ARRAY) 0 (NULL)

4 [ 29 (OPEN BRACKET) 0 (NULL)

4 1 33 (INTEGER) 0 (NULL)

4 .. 27 (DOT DOT) 0 (NULL)

4 5 33 (INTEGER) 0 (NULL)

4 ] 30 (CLOSE BRACKET) 0 (NULL)

4 of 17 (OF) 0 (NULL)

4 integer 16 (INTEGER) 0 (NULL)

4 ; 23 (SEMICOLON) 0 (NULL)

5 var 21 (VAR) 0 (NULL)

5 d 1 (ID) 454976d5 (PTR TO SYM TAB)

5 : 2 (RELOP) 7 (COLON)

5 real 19 (REAL) 0 (NULL)

5 ; 23 (SEMICOLON) 0 (NULL)

6 var 21 (VAR) 0 (NULL)

6 t 1 (ID) 3ab79c87 (PTR TO SYM TAB)

6 : 2 (RELOP) 7 (COLON)

6 boolean 10 (BOOLEAN) 0 (NULL)

6 ; 23 (SEMICOLON) 0 (NULL)

8 function 14 (FUNCTION) 0 (NULL)

8 fun1 1 (ID) b7bfddaf (PTR TO SYM TAB)

8 ( 24 (LEFT PAREN) 0 (NULL)

8 x 1 (ID) 2622f4f6 (PTR TO SYM TAB)

8 : 2 (RELOP) 7 (COLON)

8 integer 16 (INTEGER) 0 (NULL)

8 ; 23 (SEMICOLON) 0 (NULL)

8 y 1 (ID) 069c3677 (PTR TO SYM TAB)

8 : 2 (RELOP) 7 (COLON)

8 real 19 (REAL) 0 (NULL)

8 ; 23 (SEMICOLON) 0 (NULL)

9 z 1 (ID) 73f5e944 (PTR TO SYM TAB)

9 : 2 (RELOP) 7 (COLON)

9 array 8 (ARRAY) 0 (NULL)

9 [ 29 (OPEN BRACKET) 0 (NULL)

9 1 33 (INTEGER) 0 (NULL)

9 .. 27 (DOT DOT) 0 (NULL)

9 2 33 (INTEGER) 0 (NULL)

9 ] 30 (CLOSE BRACKET) 0 (NULL)

9 of 17 (OF) 0 (NULL)

9 integer 16 (INTEGER) 0 (NULL)

9 ; 23 (SEMICOLON) 0 (NULL)

10 q 1 (ID) 7eef7ab8 (PTR TO SYM TAB)

10 : 2 (RELOP) 7 (COLON)

10 real 19 (REAL) 0 (NULL)

10 ) 25 (RIGHT PAREN) 0 (NULL)

10 : 2 (RELOP) 7 (COLON)

10 integer 16 (INTEGER) 0 (NULL)

10 ; 23 (SEMICOLON) 0 (NULL)

11 begin 9 (BEGIN) 0 (NULL)

12 x 1 (ID) 2622f4f6 (PTR TO SYM TAB)

12 := 2 (ASSIGNOP) 0 (NULL)

12 z 1 (ID) 73f5e944 (PTR TO SYM TAB)

12 [ 29 (OPEN BRACKET) 0 (NULL)

12 1 33 (INTEGER) 0 (NULL)

12 ] 30 (CLOSE BRACKET) 0 (NULL)

12 \* 3 (MULOP) 1 (MULT)

12 2 33 (INTEGER) 0 (NULL)

12 ; 23 (SEMICOLON) 0 (NULL)

13 y 1 (ID) 069c3677 (PTR TO SYM TAB)

13 := 2 (ASSIGNOP) 0 (NULL)

13 ( 24 (LEFT PAREN) 0 (NULL)

13 z 1 (ID) 73f5e944 (PTR TO SYM TAB)

13 [ 29 (OPEN BRACKET) 0 (NULL)

13 1 33 (INTEGER) 0 (NULL)

13 ] 30 (CLOSE BRACKET) 0 (NULL)

13 - 4 (ARITHOP) 2 (SUB)

13 z 1 (ID) 73f5e944 (PTR TO SYM TAB)

13 [ 29 (OPEN BRACKET) 0 (NULL)

13 2 33 (INTEGER) 0 (NULL)

13 ] 30 (CLOSE BRACKET) 0 (NULL)

13 ) 25 (RIGHT PAREN) 0 (NULL)

13 div 3 (MULOP) 4 (DIV)

13 4 33 (INTEGER) 0 (NULL)

13 ; 23 (SEMICOLON) 0 (NULL)

14 z 1 (ID) 73f5e944 (PTR TO SYM TAB)

14 [ 29 (OPEN BRACKET) 0 (NULL)

14 1 33 (INTEGER) 0 (NULL)

14 ] 30 (CLOSE BRACKET) 0 (NULL)

14 := 2 (ASSIGNOP) 0 (NULL)

14 x 1 (ID) 2622f4f6 (PTR TO SYM TAB)

14 mod 3 (MULOP) 5 (MOD)

14 2 33 (INTEGER) 0 (NULL)

14 ; 23 (SEMICOLON) 0 (NULL)

15 q 1 (ID) 7eef7ab8 (PTR TO SYM TAB)

15 := 2 (ASSIGNOP) 0 (NULL)

15 y 1 (ID) 069c3677 (PTR TO SYM TAB)

15 + 4 (ARITHOP) 1 (ADD)

15 1.23e2 32 (LONG REAL) 0 (NULL)

15 ; 23 (SEMICOLON) 0 (NULL)

16 end 13 (END) 0 (NULL)

16 ; 23 (SEMICOLON) 0 (NULL)

18 begin 9 (BEGIN) 0 (NULL)

19 if 15 (IF) 0 (NULL)

19 b 1 (ID) f92486ed (PTR TO SYM TAB)

19 = 2 (RELOP) 6 (EQ)

19 0 33 (INTEGER) 0 (NULL)

19 then 20 (THEN) 0 (NULL)

19 b 1 (ID) f92486ed (PTR TO SYM TAB)

19 := 2 (ASSIGNOP) 0 (NULL)

19 a 1 (ID) 7fe6c96e (PTR TO SYM TAB)

19 \* 3 (MULOP) 1 (MULT)

19 3.14 34 (REAL) 0 (NULL)

20 b 1 (ID) f92486ed (PTR TO SYM TAB)

20 := 2 (ASSIGNOP) 0 (NULL)

20 c 1 (ID) 6e5b9acb (PTR TO SYM TAB)

20 [ 29 (OPEN BRACKET) 0 (NULL)

20 4 33 (INTEGER) 0 (NULL)

20 ] 30 (CLOSE BRACKET) 0 (NULL)

20 / 3 (MULOP) 2 (DIV)

20 c 1 (ID) 6e5b9acb (PTR TO SYM TAB)

20 [ 29 (OPEN BRACKET) 0 (NULL)

20 5 33 (INTEGER) 0 (NULL)

20 ] 30 (CLOSE BRACKET) 0 (NULL)

20 ; 23 (SEMICOLON) 0 (NULL)

21 t 1 (ID) 3ab79c87 (PTR TO SYM TAB)

21 := 2 (ASSIGNOP) 0 (NULL)

21 d 1 (ID) 454976d5 (PTR TO SYM TAB)

21 >= 2 (RELOP) 3 (GE)

21 a 1 (ID) 7fe6c96e (PTR TO SYM TAB)

21 ; 23 (SEMICOLON) 0 (NULL)

22 a 1 (ID) 7fe6c96e (PTR TO SYM TAB)

22 := 2 (ASSIGNOP) 0 (NULL)

22 a 1 (ID) 7fe6c96e (PTR TO SYM TAB)

22 or 4 (ARITHOP) 3 (OR)

22 c 1 (ID) 6e5b9acb (PTR TO SYM TAB)

22 [ 29 (OPEN BRACKET) 0 (NULL)

22 1 33 (INTEGER) 0 (NULL)

22 ] 30 (CLOSE BRACKET) 0 (NULL)

22 ; 23 (SEMICOLON) 0 (NULL)

23 a 1 (ID) 7fe6c96e (PTR TO SYM TAB)

23 := 2 (ASSIGNOP) 0 (NULL)

23 c 1 (ID) 6e5b9acb (PTR TO SYM TAB)

23 [ 29 (OPEN BRACKET) 0 (NULL)

23 1 33 (INTEGER) 0 (NULL)

23 ] 30 (CLOSE BRACKET) 0 (NULL)

23 and 3 (MULOP) 3 (AND)

23 c 1 (ID) 6e5b9acb (PTR TO SYM TAB)

23 [ 29 (OPEN BRACKET) 0 (NULL)

23 3 33 (INTEGER) 0 (NULL)

23 ] 30 (CLOSE BRACKET) 0 (NULL)

23 ; 23 (SEMICOLON) 0 (NULL)

24 end 13 (END) 0 (NULL)

24 . 28 (DOT) 0 (NULL)

24 $ 7 (EOF) 0 (NULL)

**token\_file\_with\_errors**

Line No. Lexeme Token Type Attribute/Address

1 $ 99 (LEXERR) 6 (UNREC SYM)

1 @ 99 (LEXERR) 6 (UNREC SYM)

2 var 21 (VAR) 0 (NULL)

2 a 1 (ID) f5ea0e5c (PTR TO SYM TAB)

2 : 2 (RELOP) 7 (COLON)

2 array 8 (ARRAY) 0 (NULL)

2 [ 29 (OPEN BRACKET) 0 (NULL)

2 1 33 (INTEGER) 0 (NULL)

2 , 26 (COMMA) 0 (NULL)

2 2 33 (INTEGER) 0 (NULL)

2 , 26 (COMMA) 0 (NULL)

2 3 33 (INTEGER) 0 (NULL)

2 , 26 (COMMA) 0 (NULL)

2 4 33 (INTEGER) 0 (NULL)

2 , 26 (COMMA) 0 (NULL)

2 5 33 (INTEGER) 0 (NULL)

2 , 26 (COMMA) 0 (NULL)

2 6 33 (INTEGER) 0 (NULL)

2 , 26 (COMMA) 0 (NULL)

2 7 33 (INTEGER) 0 (NULL)

2 , 26 (COMMA) 0 (NULL)

2 8 33 (INTEGER) 0 (NULL)

2 , 26 (COMMA) 0 (NULL)

2 9 33 (INTEGER) 0 (NULL)

2 , 26 (COMMA) 0 (NULL)

2 10 33 (INTEGER) 0 (NULL)

2 , 26 (COMMA) 0 (NULL)

2 11 33 (INTEGER) 0 (NULL)

2 , 26 (COMMA) 0 (NULL)

2 12 33 (INTEGER) 0 (NULL)

2 , 26 (COMMA) 0 (NULL)

2 13 33 (INTEGER) 0 (NULL)

2 , 26 (COMMA) 0 (NULL)

2 14 33 (INTEGER) 0 (NULL)

2 ] 30 (CLOSE BRACKET) 0 (NULL)

2 of 17 (OF) 0 (NULL)

2 integer 16 (INTEGER) 0 (NULL)

2 ; 23 (SEMICOLON) 0 (NULL)

3 extralongtoken 99 (LEXERR) 1 (EXTRA LONG TKN)

4 12345678901 99 (LEXERR) 5 (EXTRA LONG INT)

5 123456.789012e345 99 (LEXERR) 2 (EXTRA LONG CHAR)

5 123456.789012e345 99 (LEXERR) 3 (EXTRA LONG MANT)

5 123456.789012e345 99 (LEXERR) 4 (EXTRA LONG EXP)

5 $ 7 (EOF) 0 (NULL)

**token\_file\_syntax\_errors**

Line No. Lexeme Token Type Attribute/Address

1 3.4 34 (REAL) 0 (NULL)

1 E 1 (ID) ad24af74 (PTR TO SYM TAB)

2 3.4 34 (REAL) 0 (NULL)

2 E 1 (ID) ad24af74 (PTR TO SYM TAB)

2 + 4 (ARITHOP) 1 (ADD)

3 3 33 (INTEGER) 0 (NULL)

3 . 28 (DOT) 0 (NULL)

3 E 1 (ID) ad24af74 (PTR TO SYM TAB)

3 + 4 (ARITHOP) 1 (ADD)

4 3.4 34 (REAL) 0 (NULL)

4 E 1 (ID) ad24af74 (PTR TO SYM TAB)

5 3 33 (INTEGER) 0 (NULL)

5 . 28 (DOT) 0 (NULL)

5 E4 1 (ID) b9f27bf9 (PTR TO SYM TAB)

6 $ 99 (LEXERR) 6 (UNREC SYM)

6 @ 99 (LEXERR) 6 (UNREC SYM)

6 $ 7 (EOF) 0 (NULL)

**symbols\_without\_errors**

test e632973e

input 54cdb9f6

output 23542cfc

a 7fe6c96e

b f92486ed

c 6e5b9acb

d 454976d5

t 3ab79c87

fun1 b7bfddaf

x 2622f4f6

y 069c3677

z 73f5e944

q 7eef7ab8

**symbols\_with\_errors**

a f5ea0e5c

**symbols\_syntax\_errors**

E ad24af74

E4 b9f27bf9

**Appendix II: Program Listings**

**driver\_proj1.py**

'''

@author David Glover

CS 4013 - Compiler Construction

Project 1 - Lexical Analyzer

September 26, 2014

'''

from lexical\_analyzer import \*

def main\_proj1():

# Input files

src\_without\_errors\_file = open('src\_program\_no\_errors', 'r')

src\_with\_errors\_file = open('src\_program\_with\_errors', 'r')

# Output files

listing\_with\_errors\_file = open('listing\_file\_with\_errors', 'w')

listing\_without\_errors\_file = open('listing\_file\_without\_errors', 'w')

token\_with\_errors\_file = open('token\_file\_with\_errors', 'w')

token\_without\_errors\_file = open('token\_file\_without\_errors', 'w')

symbols\_with\_errors\_file = open('symbols\_with\_errors', 'w')

symbols\_without\_errors\_file = open('symbols\_without\_errors', 'w')

# Run the lexical analyzer for both source programs

Lexical\_Analyzer(src\_without\_errors\_file, listing\_without\_errors\_file, token\_without\_errors\_file, symbols\_without\_errors\_file)

Lexical\_Analyzer(src\_with\_errors\_file, listing\_with\_errors\_file, token\_with\_errors\_file, symbols\_with\_errors\_file)

# Source programs with syntax errors - Cannot be caught yet

src\_file = open('src\_program\_syntax\_errors', 'r')

listing\_file = open('listing\_file\_syntax\_errors', 'w')

token\_file = open('token\_file\_syntax\_errors', 'w')

symbol\_file = open('symbols\_syntax\_errors', 'w')

Lexical\_Analyzer(src\_file, listing\_file, token\_file, symbol\_file)

main\_proj1()

**reserved\_word.py**

class Reserved\_Word:

def \_\_init\_\_(self, lexeme, token\_type, token, attribute\_type, attribute):

self.lexeme = lexeme

self.token\_type = token\_type

self.token = token

self.attribute\_type = attribute\_type

self.attribute = attribute

**token.py**

class Token:

def \_\_init\_\_(self, lexeme, token\_type, token, attribute\_type, attribute):

self.lexeme = lexeme

self.token\_type = token\_type

self.token = token

self.attribute\_type = attribute\_type

self.attribute = attribute

**lexical\_analyzer.py**

from reserved\_word import \*

from token import \*

import random

# Initialize static variables

MAX\_BUFFER\_SIZE = 72

ID\_LENGTH = INT\_LENGTH = 10

X\_LENGTH = Y\_LENGTH = 5

Z\_LENGTH = 2

class Lexical\_Analyzer:

def \_\_init\_\_(self, source\_file, listing\_file, token\_file, symbol\_file):

# Variables shared with other functions

global error\_list, token\_list, function\_list, reserved\_words\_list, symbol\_list

global token, relop, mulop, arithop, number, other, is\_real, has\_exponent

global int\_part, real\_part, exponent, number\_is\_valid

# Initialize variables

int\_part = real\_part = exponent = ''

line\_number = 1

lex\_buffer = []

error\_list = []

reserved\_words\_list = []

token\_list = []

symbol\_list = []

token = relop = mulop = arithop = number = other = ''

number\_is\_valid = True

is\_real = has\_exponent = False

# Read in data from the reserved words file

reserved\_words\_file = open('reserved\_words', 'r')

for line in reserved\_words\_file:

words = line.split(' ')

reserved\_words\_list.append(Reserved\_Word(words[0], int(words[1]), words[2], int(words[3]), words[4]))

# Write the token file header

token\_file.write('Line No.\tLexeme\t\t\t\tToken Type\t\t\t\tAttribute/Address\n')

for line in source\_file:

line = line.strip()

listing\_file.write(str(line\_number) + '\t\t' + line + '\n')

if len(line) > MAX\_BUFFER\_SIZE:

error\_list.append('LEXERR: Line exceeds buffer size')

# Fill the buffer

for char in line:

lex\_buffer.append(char)

# Evaluate the first char of the buffer if it is not empty

next\_char\_type = curr\_char\_type = -1

if lex\_buffer:

curr\_char\_type = self.pick\_machine(lex\_buffer[0])

else:

curr\_char\_type = 0

if len(lex\_buffer) >= 2 and curr\_char\_type == self.pick\_machine(lex\_buffer[1], curr\_char\_type):

self.pick\_function(curr\_char\_type, lex\_buffer[0], False, symbol\_file)

# Evaluate the rest of the buffer

for i in range(len(lex\_buffer)):

next\_char\_type = self.pick\_machine(lex\_buffer[i], curr\_char\_type)

if next\_char\_type != curr\_char\_type or i > 1:

self.pick\_function(curr\_char\_type, lex\_buffer[i - 1], next\_char\_type != curr\_char\_type, symbol\_file)

curr\_char\_type = next\_char\_type

if lex\_buffer:

self.pick\_function(curr\_char\_type, lex\_buffer[len(lex\_buffer) - 1], True, symbol\_file)

# Write all the errors for the line to the listing file, if any

while len(error\_list) >= 1:

listing\_file.write(error\_list.pop(0) + '\n')

# Pretty print the tokens for the line to the token file, if any

while len(token\_list) >= 1:

tkn = token\_list.pop(0)

tabs = '\t\t\t\t\t'

token\_file.write(str(line\_number) + '\t\t\t' + str(tkn.lexeme) +

tabs[0:len(tabs) - len(str(tkn.lexeme)) / 4] + str(tkn.token\_type) +

'\t(' + str(tkn.token) + ')' + tabs[0:len(tabs) - ((len(str(tkn.token)) + 2) / 4)] + str(tkn.attribute\_type) + tabs[0:len(tabs) - len(str(tkn.attribute\_type)) / 4 - 2] +

'(' + tkn.attribute + ')' + tabs + '\n')

# Empty the buffer for the next line

while len(lex\_buffer) >= 1:

lex\_buffer.pop()

line\_number += 1

# Write EOF to the token file

token\_file.write(str(line\_number - 1) + '\t\t\t$\t\t\t\t\t7\t(EOF)\t\t\t\t0\t\t\t(NULL)')

# Close all files

reserved\_words\_file.close()

source\_file.close()

listing\_file.close()

token\_file.close()

symbol\_file.close()

def pick\_machine(self, char, char\_type = None):

if not char or char == ' ':

return 0

elif char == '=' or char == ':' or char == '<' or char == '>':

return 1

elif char == '\*' or char == '/':

return 2

elif char == '+' or char == '-':

return 3

if char\_type is None:

try:

if char.isalpha():

return 4

elif 9 >= int(char) >= 0:

return 5

except:

return 6

else:

try:

if char\_type == 4 and 9 >= int(char) >= 0:

return 4

elif 9 >= int(char) >= 0:

return 5

except:

if char\_type == 5 and (char == '.' or char == 'e' or char == 'E'):

return 5

elif char.isalpha():

return 4

else:

return 6

def pick\_function(self, char\_type, char, is\_last\_char, sym\_file):

if char\_type == 1:

self.relop(char, is\_last\_char)

elif char\_type == 2:

self.mulop(char, is\_last\_char)

elif char\_type == 3:

self.arithop(char, is\_last\_char)

elif char\_type == 4:

self.identifier(char, is\_last\_char, sym\_file)

elif char\_type == 5:

self.number(char, is\_last\_char, sym\_file)

elif char\_type == 6:

self.other(char, is\_last\_char)

def relop(self, char, is\_last\_char):

global relop

relop += char

if is\_last\_char:

if relop == ':=':

token\_list.append(Token(relop, 2, 'ASSIGNOP', 0, 'NULL'))

elif relop == '>':

token\_list.append(Token(relop, 2, 'RELOP', 1, 'GT'))

elif relop == '<':

token\_list.append(Token(relop, 2, 'RELOP', 2, 'LT'))

elif relop == '>=':

token\_list.append(Token(relop, 2, 'RELOP', 3, 'GE'))

elif relop == '<=':

token\_list.append(Token(relop, 2, 'RELOP', 4, 'LE'))

elif relop == '<>':

token\_list.append(Token(relop, 2, 'RELOP', 5, 'NE'))

elif relop == '=':

token\_list.append(Token(relop, 2, 'RELOP', 6, 'EQ'))

elif relop == ':':

token\_list.append(Token(relop, 2, 'RELOP', 7, 'COLON'))

relop = ''

def mulop (self, char, is\_last\_char):

global mulop

mulop += char

if is\_last\_char:

if mulop == '\*':

token\_list.append(Token(mulop, 3, 'MULOP', 1, 'MULT'))

else:

token\_list.append(Token(mulop, 3, 'MULOP', 2, 'DIV'))

mulop = ''

def arithop(self, char, is\_last\_char):

global arithop

arithop += char

if is\_last\_char:

if arithop == '+':

token\_list.append(Token(arithop, 4, 'ARITHOP', 1, 'ADD'))

else:

token\_list.append(Token(arithop, 4, 'ARITHOP', 2, 'SUB'))

arithop = ''

def identifier(self, char, is\_last\_char, sym\_file):

global token

token += char

if is\_last\_char:

if len(token) > ID\_LENGTH:

error\_list.append('LEXERR: Token exceeded max length: ' + token)

token\_list.append(Token(token, 99, 'LEXERR', 1, 'EXTRA LONG TKN'))

other = ''

token = ''

return

# Check if the token is a reserved word

for word in reserved\_words\_list:

if word.lexeme == token:

token\_list.append(Token(word.lexeme, word.token\_type, word.token, word.attribute\_type, word.attribute.strip('\n')))

token = ''

return

self.add\_id(token)

self.check\_symbols(sym\_file)

def number(self, char, is\_last\_char, sym\_file):

global number, is\_real, has\_exponent, int\_part, real\_part, exponent, number\_is\_valid

number += char

if char == '.':

if number[len(number) - 2: len(number)] == '..':

token\_list.append(Token(number.rstrip('..'), 33, 'INTEGER', 0, 'NULL'))

token\_list.append(Token('..', 27, 'DOT DOT', 0, 'NULL'))

has\_exponent = is\_real = False

number\_is\_valid = True

number = int\_part = real\_part = exponent = ''

return

is\_real = True

if not is\_last\_char:

return

elif char == 'e' or char == 'E':

has\_exponent = True

if not is\_last\_char:

return

char = ''

if has\_exponent:

exponent += char

elif is\_real:

real\_part += char

else:

int\_part += char

if is\_last\_char:

if len(int\_part) > X\_LENGTH and is\_real:

error\_list.append('LEXERR: Extra long characteristic: ' + number)

token\_list.append(Token(number, 99, 'LEXERR', 2, 'EXTRA LONG CHAR'))

number\_is\_valid = False

if len(real\_part) > Y\_LENGTH and is\_real:

error\_list.append('LEXERR: Extra long mantissa: ' + number)

token\_list.append(Token(number, 99, 'LEXERR', 3, 'EXTRA LONG MANT'))

number\_is\_valid = False

if len(exponent) > Z\_LENGTH and has\_exponent:

error\_list.append('LEXERR: Extra long exponent: ' + number)

token\_list.append(Token(number, 99, 'LEXERR', 4, 'EXTRA LONG EXP'))

number\_is\_valid = False

if len(int\_part) > INT\_LENGTH and not is\_real:

error\_list.append('LEXERR: Extra long integer: ' + int\_part)

token\_list.append(Token(int\_part, 99, 'LEXERR', 5, 'EXTRA LONG INT'))

number\_is\_valid = False

if number\_is\_valid:

if has\_exponent:

# |longreal|

if exponent and is\_real and real\_part and int\_part:

token\_list.append(Token(number, 32, 'LONG REAL', 0, 'NULL'))

# |dot|int|id|

elif is\_real and real\_part and not int\_part:

token\_list.append(Token('.', 28, 'DOT', 0, 'NULL'))

token\_list.append(Token(real\_part, 33, 'INTEGER', 0, 'NULL'))

self.add\_id('E' + exponent)

self.check\_symbols(sym\_file, 'E' + exponent)

# |int|dot|id|

elif is\_real and not real\_part and int\_part:

token\_list.append(Token(int\_part, 33, 'INTEGER', 0, 'NULL'))

token\_list.append(Token('.', 28, 'DOT', 0, 'NULL'))

self.add\_id('E' + exponent)

self.check\_symbols(sym\_file, 'E' + exponent)

# |dot|id|

elif is\_real and not real\_part and not int\_part:

token\_list.append(Token('.', 28, 'DOT', 0, 'NULL'))

self.add\_id('E' + exponent)

self.check\_symbols(sym\_file, 'E' + exponent)

# |int|id|

elif not is\_real and not real\_part and int\_part:

token\_list.append(Token(int\_part, 33, 'INTEGER', 0, 'NULL'))

self.add\_id('E' + exponent)

self.check\_symbols(sym\_file, 'E' + exponent)

# |id|

elif not is\_real and not real\_part and not int\_part:

self.add\_id('E' + exponent)

self.check\_symbols(sym\_file, 'E' + exponent)

# |real|id|

elif not exponent and is\_real and real\_part and int\_part:

token\_list.append(Token(int\_part + '.' + real\_part, 34, 'REAL', 0, 'NULL'))

self.add\_id('E')

self.check\_symbols(sym\_file, 'E')

elif is\_real and not has\_exponent:

if real\_part:

if int\_part:

# |real|

token\_list.append(Token(number, 34, 'REAL', 0, 'NULL'))

else:

# |dot|int|

token\_list.append(Token('.', 28, 'DOT', 0, 'NULL'))

token\_list.append(Token(real\_part, 33, 'INTEGER', 0, 'NULL'))

else:

if int\_part:

# |int|dot|

token\_list.append(Token(int\_part, 33, 'INTEGER', 0, 'NULL'))

token\_list.append(Token('.', 28, 'DOT', 0, 'NULL'))

elif not is\_real and not has\_exponent and int\_part:

# |int|

token\_list.append(Token(number, 33, 'INTEGER', 0, 'NULL'))

has\_exponent = is\_real = False

number\_is\_valid = True

number = int\_part = real\_part = exponent = ''

def other(self, char, is\_last\_char):

global other

other += char

if is\_last\_char or char != '.':

if other == ';':

token\_list.append(Token(other, 23, 'SEMICOLON', 0, 'NULL'))

elif other == '(':

token\_list.append(Token(other, 24, 'LEFT PAREN', 0, 'NULL'))

elif other == ')':

token\_list.append(Token(other, 25, 'RIGHT PAREN', 0, 'NULL'))

elif other == ',':

token\_list.append(Token(other, 26, 'COMMA', 0, 'NULL'))

elif char == '.':

if other == '..':

token\_list.append(Token(other, 27, 'DOT DOT', 0, 'NULL'))

else:

token\_list.append(Token(other, 28, 'DOT', 0, 'NULL'))

elif other == '[':

token\_list.append(Token(other, 29, 'OPEN BRACKET', 0, 'NULL'))

elif other == ']':

token\_list.append(Token(other, 30, 'CLOSE BRACKET', 0, 'NULL'))

elif other == ':':

token\_list.append(Token(other, 31, 'COLON', 0, 'NULL'))

else:

error\_list.append('LEXERR: Unrecognized symbol: ' + other)

token\_list.append(Token(other, 99, 'LEXERR', 6, 'UNREC SYM'))

other = ''

def add\_id(self, lex):

global symbol\_list, token\_list

hex\_rand = '%00000008x' % random.randrange(16 \*\* 8)

for word in symbol\_list:

if lex == word[0]:

hex\_rand = word[1]

break

token\_list.append(Token(lex, 1, 'ID', hex\_rand, 'PTR TO SYM TAB'))

def check\_symbols(self, sym\_file, tok = None):

# Check if the token is in the symbol list. Add if it is not.

global symbol\_list, token, token\_list

tabs = '\t\t\t'

if tok is None:

for word in symbol\_list:

if token == word[0]:

token = ''

return

symbol\_list.append([token, token\_list[len(token\_list) - 1].attribute\_type])

sym\_file.write(token + tabs[0:len(tabs) - len(str(token)) / 4] + symbol\_list[len(symbol\_list) - 1][1] + '\n')

token = ''

else:

for word in symbol\_list:

if tok == word[0]:

return

symbol\_list.append([tok, token\_list[len(token\_list) - 1].attribute\_type])

sym\_file.write(tok + tabs[0:len(tabs) - len(str(tok)) / 4] + symbol\_list[len(symbol\_list) - 1][1] + '\n')