

CS 4013: Compiler Construction: Project 1

Nate Beckemeyer

September 2016

Introduction

For Project 1, I wrote a lexical analyzer in C for Pascal. The purpose of the lexical analyzer is to break down the Pascal source code into the parts needed to construct a parse tree. To achieve that goal, the lexical analyzer identifies each lexeme in the code, such as the parts of a type declaration, ‘:’ and ‘integer’ or ‘real’, the beginning of a program, ‘program’, or perhaps the multiplication operator, ‘*’. It converts this lexeme into a token that later parts of the compiler can readily use.

The user can specify a source document for the analyzer, and the analyzer will create a listing file and a token file. The listing file contains the source program, line-by-line, but points out any lexical errors that occur. The token file contains the line number, the lexeme corresponding to the token, the type of the token, and the attribute value of the token.

1 Methodology

The lexical analyzer is simply a series of machines that parse the file. The machines are Whitespace, ID/RES, NumMachine, Grouping, CatchAll, RELOP, ADDOP, and MULOP. These machines break down the file into its corresponding tokens, which are output to a file. A loop repeats this process until an end of file token is encountered.

If errors in the source program are encountered while parsing, the machines will send an error. The error will be displayed beneath the corresponding line in the listing file and passed as a token to the token file.

2 Implementation

The reserved words, special to the subset of the Pascal language that we’re studying, are loaded in from a file at the start of the program, along with the proper token category and type, should the word be encountered during parsing.

The Whitespace machine matches all whitespace, and returns the appropriate token. This token is usually ignored, except for newline characters, which are used to update listing and token file information. The ID/RES machine matches reserved words and IDs from the symbol table. If a word that could be used as an ID has not been encountered before, it will be added to the symbol table. The NumMachine matches numbers: ints, reals, and long reals. The Grouping machine matches certain “grouping” punctuation, such as opening and closing parentheses and brackets. The CatchAll machine catches various punctuation series. The RELOP machine matches relative (comparative) operations, the ADDOP machine matches adding operations (including ‘or’), and the MULOP machine matches multiplicative operations (including ‘and’).

The machines are evaluated in the order in which they were specified above. The machines are passed a token, a string, and a starting point in that string.

If the machine matches the token, it updates the token's type, attribute, and gives the location of the corresponding lexeme. Regardless, it returns the new location of the pointer in the string. That location may be the same, if no match were found; however, if a match were discovered, this update will signify that a valid token has been generated, and there is no need to throw an error.

If any errors are encountered while parsing, the error will be added to a queue. Once the generated token is returned (or an error signifying an unrecognized symbol is given), then all of the errors in the queue will be pulled and displayed in the listing file, and in the token file.

3 Discussion & Conclusions

The analyzer involved learning a lot about C. I'm glad that I took the opportunity to try to do this project in this language.

One aspect of this project that I'm particularly proud of is that it will identify all errors in a single lexeme, rather than just one. Ultimately, that difference will make life much easier for the programmer.

I wrote this compiler in C, with no external code of any kind. It was compiled with clang on macOS Sierra.

Appendix 1: Sample Inputs and Outputs

Reserved Words		
and	MULOP	2
array	ARRAY	0
begin	CONTROL	0
div	MULOP	3
do	CONTROL	1
else	CONTROL	2
end	CONTROL	3
function	CONTROL	4
if	CONTROL	5
integer	TYPE	1
mod	MULOP	4
not	INVERSE	0
of	ARRAY	2
or	ADDOP	2
procedure	CONTROL	6
program	CONTROL	7
real	TYPE	2
then	CONTROL	8
var	VAR	0
while	CONTROL	9

3.1 Error-Filled Source File

Error Source Code

```

program fib(input, output);
var excessivelyLongIntegerArrayName : array [1..12] of integer;

begin
  init;
  writeln(123456789012345);
  writeln(123456.123456E003);
  23.47E
  037E-2
  writeln(01.45620E02);

  4E+.
  3.4E+
  34E.
  E4.3
  $@
end.?
```

Error Listing File

```

1      program fib(input, output);
2      var excessivelyLongIntegerArrayName : array [1..12] of integer;
LEXERR:      ID length exceeded 10 characters:excessivelyLongIntegerArrayName
3
4      begin
5          init;
6          writeln(123456789012345);
LEXERR:      Int length exceeded 10 characters:      123456789012345
7          writeln(123456.123456E003);
LEXERR:      Leading 0 in exponent:      123456.123456E003
LEXERR:      Exponent part of long real exceeded 2 characters:      123456.123456E003
```

LEXERR:	Fractional part of real exceeded 5 characters:	123456.123456E003
LEXERR:	Integer part of real exceeded 5 characters:	123456.123456E003
8	23.47E	
LEXERR:	Missing exponent part of long real:	23.47E
9	037E-2	
LEXERR:	Excessive leading 0 in real:	037E-2
10	writeln(01.45620E02);	
LEXERR:	Trailing 0 in real:	01.45620E02
LEXERR:	Leading 0 in exponent:	01.45620E02
LEXERR:	Excessive leading 0 in real:	01.45620E02
11		
12	4E+.	
LEXERR:	Missing exponent part of long real:	4E+.
LEXERR:	Attempt to use real exponent:	4E+.
13	3.4E+	
LEXERR:	Missing exponent part of long real:	3.4E+
14	34E.	
LEXERR:	Missing exponent part of long real:	34E.
LEXERR:	Attempt to use real exponent:	34E.
15	E4.3	
16	\$@	
LEXERR:	Unrecognized symbol:	\$
LEXERR:	Unrecognized symbol:	@
17	end.?	
LEXERR:	Unrecognized symbol:	?

Error Token File				
0	ASSIGNOP			
1	FILEEND			
2	RELOP			
3	ID			
4	CONTROL			
5	ADDOP			
6	MULOP			
7	WS			
8	ARRAY			
9	TYPE			
10	VAR			
11	INT			
12	REAL			
13	PUNC			
14	GROUP			
15	INVERSE			
16	NOOP			
17	LEXERR			
Line	Lexeme	Token Type	Token Attribute	
1	program	4	7	
1	fib	3	0x7fef24033c0	
1	(14	0	
1	input	3	0x7fef2403500	
1	,	13	0	
1	output	3	0x7fef2403680	
1)	14	1	
1	;	13	1	
2	var	10	0	
2	2excessivelyLongIntegerArrayName	17	1	
2	:	9	0	

2	array	8	0
2	[14	2
2	1	11	1
2	..	8	1
2	12	11	12
2]	14	3
2	of	8	2
2	integer	9	1
2	;	13	1
4	begin	4	0
5	init	3	0x7fef2404460
5	;	13	1
6	writeln	3	0x7fef2404680
6	(14	0
6	123456789012345	17	2
6)	14	1
6	;	13	1
7	writeln	3	0x7fef2404680
7	(14	0
7	123456.123456E003	17	10
7	123456.123456E003	17	5
7	123456.123456E003	17	4
7	123456.123456E003	17	3
7)	14	1
7	;	13	1
8	23.47E	17	6
9	037E-2	17	8
10	writeln	3	0x7fef2404680
10	(14	0
10	01.45620E02	17	9
10	01.45620E02	17	10
10	01.45620E02	17	8
10)	14	1
10	;	13	1
12	4E+.	17	6
12	4E+.	17	11
13	3.4E+	17	6
14	34E.	17	6
14	34E.	17	11
15	E4	3	0x7fef2405fb0
15	.3	12	0.300000
16	\$	17	0
16	@	17	0
17	end	4	3
17	.	13	2
17	?	17	0
18	EOF	1	0

3.2 Error-Free Source File

Correct Source Code

```
program fib(input, output);
var n, p: integer;
var q: real;
var numsArray : array [1..12] of integer;

function fib(a : integer; b, c : real) : real;
begin
    if a <= 1 then fib := c
    else fib := fib(a - 1, c, b + c)
end;

function fib2(a : integer) : integer;
var b, c, sum : integer;
begin
    a := a - 1;
    b := 0;
    sum := 1;
    c := b;
    while (not a < -1) do
        begin
            a := a - 1;
            b := sum;
            sum := c + sum;
            c := b
        end;
    fib2 := sum;
end;

procedure init();
begin
    n := 12;
    if (1 and 2) or 3 then p := 12
    else p := 14;
    numsArray[3] := 15.56;
    q := 12
end;

begin
    init;
    writeln(+fib(n, 0, 1)*q/p + 4);
    writeln(fib2(n));
    writeln(numsArray[3])
end.
```

Correct Listing File

1	program fib(input, output);
2	var n, p: integer;
3	var q: real;
4	var numsArray : array [1..12] of integer;
5	
6	function fib(a : integer; b, c : real) : real;
7	begin
8	if a <= 1 then fib := c
9	else fib := fib(a - 1, c, b + c)
10	end;

```

11
12      function fib2(a : integer) : integer;
13      var b, c, sum : integer;
14      begin
15          a := a - 1;
16          b := 0;
17          sum := 1;
18          c := b;
19          while (not a < -1) do
20              begin
21                  a := a - 1;
22                  b := sum;
23                  sum := c + sum;
24                  c := b
25              end;
26          fib2 := sum;
27      end;
28
29      procedure init();
30      begin
31          n := 12;
32          if (1 and 2) or 3 then p := 12
33          else p := 14;
34          numsArray[3] := 15.56;
35          q := 12
36      end;
37
38      begin
39          init;
40          writeln(+fib(n, 0, 1)*q/p + 4);
41          writeln(fib2(n));
42          writeln(numsArray[3])
43      end.

```

Correct Token File

```

0  ASSIGNOP
1  FILEEND
2  RELOP
3  ID
4  CONTROL
5  ADDOP
6  MULOP
7  WS
8  ARRAY
9  TYPE
10 VAR
11 INT
12 REAL
13 PUNC
14 GROUP
15 INVERSE
16 NOOP
17 LEXERR

```

Line	Lexeme	Token Type	Token Attribute
1	program	4	7
1	fib	3	0x7f98e04033c0
1	(14	0
1	input	3	0x7f98e0403500

1	,	13	0
1	output	3	0x7f98e0403680
1)	14	1
1	;	13	1
2	var	10	0
2	n	3	0x7f98e0403860
2	,	13	0
2	p	3	0x7f98e0403940
2	:	9	0
2	integer	9	1
2	;	13	1
3	var	10	0
3	q	3	0x7f98e0403c60
3	:	9	0
3	real	9	2
3	;	13	1
4	var	10	0
4	numsArray	3	0x7f98e0404020
4	:	9	0
4	array	8	0
4	[14	2
4	1	11	1
4	..	8	1
4	12	11	12
4]	14	3
4	of	8	2
4	integer	9	1
4	;	13	1
6	function	4	4
6	fib	3	0x7f98e04033c0
6	(14	0
6	a	3	0x7f98e0404860
6	:	9	0
6	integer	9	1
6	;	13	1
6	b	3	0x7f98e0404ae0
6	,	13	0
6	c	3	0x7f98e0404bc0
6	:	9	0
6	real	9	2
6)	14	1
6	:	9	0
6	real	9	2
6	;	13	1
7	begin	4	0
8	if	4	5
8	a	3	0x7f98e0404860
8	<=	2	1
8	1	11	1
8	then	4	8
8	fib	3	0x7f98e04033c0
8	:=	0	0
8	c	3	0x7f98e0404bc0
9	else	4	2
9	fib	3	0x7f98e04033c0
9	:=	0	0
9	fib	3	0x7f98e04033c0

9	(14	0
9	a	3	0x7f98e0404860
9	-	5	1
9	1	11	1
9	,	13	0
9	c	3	0x7f98e0404bc0
9	,	13	0
9	b	3	0x7f98e0404ae0
9	+	5	0
9	c	3	0x7f98e0404bc0
9)	14	1
10	end	4	3
10	;	13	1
12	function	4	4
12	fib2	3	0x7f98e0405fa0
12	(14	0
12	a	3	0x7f98e0404860
12	:	9	0
12	integer	9	1
12)	14	1
12	:	9	0
12	integer	9	1
12	;	13	1
13	var	10	0
13	b	3	0x7f98e0404ae0
13	,	13	0
13	c	3	0x7f98e0404bc0
13	,	13	0
13	sum	3	0x7f98e0406720
13	:	9	0
13	integer	9	1
13	;	13	1
14	begin	4	0
15	a	3	0x7f98e0404860
15	:=	0	0
15	a	3	0x7f98e0404860
15	-	5	1
15	1	11	1
15	;	13	1
16	b	3	0x7f98e0404ae0
16	:=	0	0
16	0	11	0
16	;	13	1
17	sum	3	0x7f98e0406720
17	:=	0	0
17	1	11	1
17	;	13	1
18	c	3	0x7f98e0404bc0
18	:=	0	0
18	b	3	0x7f98e0404ae0
18	;	13	1
19	while	4	9
19	(14	0
19	not	15	0
19	a	3	0x7f98e0404860
19	<	2	0
19	-1	11	-1

19)	14	1
19	do	4	1
20	begin	4	0
21	a	3	0x7f98e0404860
21	:=	0	0
21	a	3	0x7f98e0404860
21	-	5	1
21	1	11	1
21	;	13	1
22	b	3	0x7f98e0404ae0
22	:=	0	0
22	sum	3	0x7f98e0406720
22	;	13	1
23	sum	3	0x7f98e0406720
23	:=	0	0
23	c	3	0x7f98e0404bc0
23	+	5	0
23	sum	3	0x7f98e0406720
23	;	13	1
24	c	3	0x7f98e0404bc0
24	:=	0	0
24	b	3	0x7f98e0404ae0
25	end	4	3
25	;	13	1
26	fib2	3	0x7f98e0405fa0
26	:=	0	0
26	sum	3	0x7f98e0406720
26	;	13	1
27	end	4	3
27	;	13	1
29	procedure	4	6
29	init	3	0x7f98e0408c20
29	(14	0
29)	14	1
29	;	13	1
30	begin	4	0
31	n	3	0x7f98e0403860
31	:=	0	0
31	12	11	12
31	;	13	1
32	if	4	5
32	(14	0
32	1	11	1
32	and	6	2
32	2	11	2
32)	14	1
32	or	5	2
32	3	11	3
32	then	4	8
32	p	3	0x7f98e0403940
32	:=	0	0
32	12	11	12
33	else	4	2
33	p	3	0x7f98e0403940
33	:=	0	0
33	14	11	14
33	;	13	1

34	numsArray	3	0x7f98e0409b40
34	[14	2
34	3	11	3
34]	14	3
34	:=	0	0
34	15.56	12	15.560000
34	;	13	1
35	q	3	0x7f98e0403c60
35	:=	0	0
35	12	11	12
36	end	4	3
36	;	13	1
38	begin	4	0
39	init	3	0x7f98e0408c20
39	;	13	1
40	writeln	3	0x7f98e040a5c0
40	(14	0
40	+	5	0
40	fib	3	0x7f98e04033c0
40	(14	0
40	n	3	0x7f98e0403860
40	,	13	0
40	0	11	0
40	,	13	0
40	1	11	1
40)	14	1
40	*	6	0
40	q	3	0x7f98e0403c60
40	/	6	1
40	p	3	0x7f98e0403940
40	+	5	0
40	4	11	4
40)	14	1
40	;	13	1
41	writeln	3	0x7f98e040a5c0
41	(14	0
41	fib2	3	0x7f98e0405fa0
41	(14	0
41	n	3	0x7f98e0403860
41)	14	1
41)	14	1
41	;	13	1
42	writeln	3	0x7f98e040a5c0
42	(14	0
42	numsArray	3	0x7f98e040b380
42	[14	2
42	3	11	3
42]	14	3
42)	14	1
43	end	4	3
43	.	13	2
44	EOF	1	0

Appendix 2: Program Listings

```
LinkedList.c
#include<stdlib.h>
#include<stdio.h>

#include "linkedlist.h"

int add(LinkedList* list, void *data, size_t size)
{
    struct node* addition = malloc(sizeof(*addition));
    addition -> data = malloc(size);
    addition -> next = (list -> head);
    // Do a byte-by-byte copy of the data
    for (int i = 0; i < size; i++)
        *(char *) (addition -> data + i) = *(char *) (data + i);
    list -> size++;

    list -> head = addition;

    return list -> size;
}

void* pop(LinkedList* list)
{
    struct node* head = list -> head;
    struct node* next = head -> next;

    void* data = head -> data;
    list -> head = next;
    list -> size--;

    //free(head); // TODO this is necessary; should fix
    return data;
}

LinkedList.h
#ifndef LINKED_H_
#define LINKED_H_

// Behaves like a stack
struct node {
    void* data;
    struct node* next;
};

typedef struct LinkedNodes {
    struct node* head;
    int size;
} LinkedList;

// Add an item to the front of the linked list
int add(LinkedList* list, void* data, size_t size);

// Pop an item from the front of the linked list
void* pop(LinkedList* list);
```

```

#endif // LINKED_H_

```

```

Processor.h
#endif PROCESSOR_H_
#define PROCESSOR_H_

enum TokenType {ASSIGNOP, FILEEND, RELOP, ID, CONTROL,
                ADDOP, MULOP, WS, ARRAY, TYPE, VAR,
                INT, REAL, PUNC, GROUP, INVERSE, NOOP, LEXERR};

extern const char* catNames[];

// The token data type (essentially a tuple :: (TokenType, int/id))
typedef struct T_Type {
    enum TokenType category;
    int start;
    int length;
    union {
        int type;
        double val;
        char* id;
    };
} Token;

Token* getNextToken();
int passLine(char* newLine);
int initializeTokens(FILE* resFile);

#endif // PROCESSOR_H_

```

```

Processor.c
#include<stdlib.h>
#include<stdio.h>
#include<stdbool.h>
#include<ctype.h>
#include<string.h>

#include "processor.h"
#include "../dataStructures/linkedList/linkedList.h"
#include "../lexicalanalyzer.h"

const char* catNames[] = {"ASSIGNOP", "FILEEND", "RELOP", "ID", "CONTROL",
                          "ADDOP", "MULOP", "WS", "ARRAY", "TYPE", "VAR",
                          "INT", "REAL", "PUNC", "GROUP", "INVERSE", "NOOP", "LEXERR"};

static char* buffer;
// Begin machine listings
/*****
*                               ID/RES                               *
*****/
int getIndex(const char** array, size_t arr_size, char* item)
{
    while (arr_size > 0)
    {
        if (strcmp(array[arr_size - 1], item) == 0)

```

```

        return arr_size - 1;
        arr_size--;
    }
    return -1;
}

// The tables & arrays and stuff
char** reservedWords;
int numReserved;
static enum TokenType* categories;
static int* attributes;

LinkedList* symbolTable;

static LinkedList* errorList;
static struct node* errorHead;

void throwError(enum TokenType category, int type, int start, int length)
{
    Token* errToken = malloc(sizeof(*errToken));
    errToken -> category = category;
    errToken -> type = type;
    errToken -> start = start;
    errToken -> length = length;

    add(errorList, errToken, sizeof(*errToken));
}

// Initialization stuff
int initResWords(FILE* resFile)
{
    static const int length = 11;
    LinkedList* resWords = malloc(sizeof(*resWords));
    LinkedList* cats = malloc(sizeof(*cats));
    LinkedList* attrs = malloc(sizeof(*attrs));

    char word[length] = {0};
    char category[length] = {0};
    int attr = 0;
    //while (fgets(word, length, resFile))
    while (true)
    {
        fscanf(resFile, "%s", word);
        if (feof(resFile))
            break;
        fscanf(resFile, "%s", category); // The actual name.
        fscanf(resFile, "%d", &attr);
        numReserved = add(resWords, &word, length*sizeof(char));
        add(cats, &category, length*sizeof(char));
        add(attrs, &attr, sizeof(int));
    }

    // Initialize the lexeme table
    reservedWords = malloc(numReserved*sizeof(char*));
    struct node* node = resWords -> head;

```

```

    for (size_t i = 0; i < numReserved; i++) {
        reservedWords[i] = (char *) node -> data;
        node = node -> next;
    }

    // Initialize the category table
    categories = malloc(numReserved*sizeof(enum TokenType));
    node = cats -> head;

    for (size_t i = 0; i < numReserved; i++) {
        categories[i] = (enum TokenType) getIndex(catNames,
                                                    sizeof(catNames)/sizeof(char*),
                                                    (char *) node -> data);
        node = node -> next;
    }

    // Initialize the attribute table
    attributes = malloc(numReserved*sizeof(int));
    node = attrs -> head;

    for (size_t i = 0; i < numReserved; i++) {
        attributes[i] = *(int *) node -> data;
        node = node -> next;
    }

    return 0;
}

int initSymbolTable()
{
    symbolTable = malloc(sizeof(*symbolTable));
    symbolTable -> head = 0;
    return 0;
}

int isReserved(char* word)
{
    // Check the reserved words table for a match first
    for (size_t i = 0; i < numReserved; i++) {
        if (!reservedWords[i] || strcmp(reservedWords[i], word) == 0) // Match
            return i;
    }

    return -1;
}

char* knownID(char* word)
{
    // Then check the symbol table
    struct node* node = symbolTable -> head;
    while (node)
    {
        if (strcmp(node -> data, word) == 0) // Match
            return (char *) (node -> data);
        node = node -> next;
    }
}

```



```

    return NULL;
}

int idres(Token* storage, char* str, int start)
{
    int initial = start;
    LinkedList* id = malloc(sizeof(*id));
    storage->category = ID;
    storage->type = 0;
    char next = str[start];
    if (isalpha(next)) // Can actually be an id/reserved
    {
        size_t wordSize = 0;
        do
        {
            wordSize = add(id, &next, sizeof(char));
            start++;
            next = str[start];
        } while(isalpha(next) || isdigit(next)); // Match ID

        // The string of the id name
        char* name = malloc((wordSize + 1)*sizeof(char));
        name[wordSize] = '\0';
        struct node* node = id->head;
        for (size_t i = 0; i < wordSize; i++) {
            name[wordSize - i - 1] = *(char*)(node->data);
            node = node->next;
        }

        int index = -1;
        char* address = 0;
        if ((index = isReserved(name)) >= 0)
        { // It's a reserved word!
            storage->category = categories[index];
            storage->type = attributes[index];
        }
        else if ((address = knownID(name)))
        {
            storage->id = address;
        } else
        {
            add(symbolTable, name, sizeof(char*)); // (wordSize + 1)*sizeof(char));
            storage->id = (char*)(symbolTable->head->data);
        }
    }

    if (start - initial > 10) // ID Too long err
    {
        storage->category = NOOP;
        throwError(LEXERR, 1, initial, start - initial);
    }
    return start;
}

/*****
*                               END ID/RES                               *
*****/

```

```

int relop(Token* storage, char* str, int start)
{
    storage->category = RELOP;
    char next = str[start];
    switch (next) {
        case '<':
            start++;
            if (str[start] == '=')
            {
                storage->type = 1;
                start++;
            } else if (str[start] == '>')
            {
                storage->type = 5;
                start++;
            } else {
                storage->type = 0;
            }
            break;

        case '=':
            start++;
            storage->type = 2;
            break;

        case '>':
            start++;
            if (str[start] == '=')
            {
                storage->type = 4;
                start++;
            } else {
                storage->type = 3;
            }
            break;

        default: break; // Do not increment; continue on to the next machine.
    }

    return start;
}

int whitespace(Token* storage, char* str, int start)
{
    storage->category = WS;
    if (isspace(str[start]))
    {
        storage->type = 0;
        if (str[start] == '\n')
            storage->type = 1;
        start++;
    }
    return start;
}

```

```

int addop(Token* storage, char* str, int start)
{
    storage->category = ADDOP;
    switch (str[start])
    {
        case '+':
            storage->type = 0;
            start++;
            return start;

        case '-':
            storage->type = 1;
            start++;
            return start;

        default: break;
    }

    return start;
}

int mulop(Token* storage, char* str, int start)
{
    storage->category = MULOP;
    if (str[start] == '*')
    {
        storage->type = 0;
        start++;
    } else if (str[start] == '/')
    {
        storage->type = 1;
        start++;
    }

    return start;
}

int catchall(Token* storage, char* str, int start)
{
    if (strncmp(&str[start], ":", 2) == 0)
    {
        storage->category = ASSIGNOP;
        storage->type = 0;
        start += 2;
    } else if (strncmp(&str[start], "..", 2) == 0)
    {
        storage->category = ARRAY;
        storage->type = 1;
        start += 2;
    } else if (str[start] == ':'){
        storage->category = TYPE;
        storage->type = 0;
        start++;
    } else if (str[start] == ',')
    {
        storage->category = PUNC;
    }
}

```

```

        storage -> type = 0;
        start++;
    } else if (str[start] == ',')
    {
        storage -> category = PUNC;
        storage -> type = 1;
        start++;
    } else if (str[start] == '.')
    {
        storage -> category = PUNC;
        storage -> type = 2;
        start++;
    }

    return start;
}

// Assumes that "str" is valid as an integer.
char* parseNum(LinkedList* chars, bool real)
{
    char* num = malloc((chars -> size + 1) * sizeof(char));
    size_t count = chars -> size;
    num[count--] = 0;
    struct node* node = chars -> head;
    while (node)
    {
        num[count--] = *(char *)node -> data;
        node = node -> next;
    }

    return num;
}

int grouping(Token* storage, char* str, int start)
{
    storage -> category = GROUP;
    switch (str[start])
    {
        case '(':
            storage -> type = 0;
            start++;
            break;

        case ')':
            storage -> type = 1;
            start++;
            break;

        case '[':
            storage -> type = 2;
            start++;
            break;

        case ']':
            storage -> type = 3;
            start++;
            break;
    }
}

```

```

        default:
            break;
    }

    return start;
}

double parseReal(LinkedList* digits)
{
    char* array = parseNum(digits, true);
    double val = strtod(array, NULL);
    free(array);
    return val;
}

int parseInt(LinkedList* digits)
{
    char* array = parseNum(digits, false);
    int val = (int) strtol(array, NULL, 10);
    free(array);
    return val;
}

int numMachine(Token* storage, char* str, int start)
{
    int initial = start; // For keeping track
    bool real = false;
    bool hasE = false;
    bool started = false;
    bool leadZero = false;
    bool expLeadZero = false;
    bool trailZero = false;

    int sign = 1;
    int intLen = 0;
    int fractionLen = 0;
    int expLen = 0;
    bool errored = false;

    LinkedList* digits = malloc(sizeof(*digits));
    if (str[start] == '-' && isdigit(str[start + 1]))
    {
        add(digits, &str[start], sizeof(char*));
        start++;
    }
    else if (str[start] == '+' && isdigit(str[start + 1]))
        start++;

    while (isdigit(str[start])) // Match the beginning integer
    {
        if (str[start] == '0' && !started)
            leadZero = true;
        add(digits, &str[start], sizeof(char*));
        started = true;
        start++;
        intLen++;
    }

```

```

}
if (str[start] == '.' && isdigit(str[start + 1])) // Match the real
{
    add(digits, &str[start], sizeof(char*));
    real = true;
    start++;
    while (isdigit(str[start])) // The fraction part of the decimal
    {
        add(digits, &str[start], sizeof(char*));
        if (str[start] == '0')
            trailZero = true;
        else
            trailZero = false;
        start++;
        fractionLen++;
    }
}
if (str[start] == 'E') // Match the long real
{
    hasE = true;
    add(digits, &str[start], sizeof(char*));
    real = true;
    bool initialRun = true;
    start++;
    if (str[start] == '+' || str[start] == '-') // Sign in the exponent
        add(digits, &str[start++], sizeof(char*));
    if (str[start] == '.')
    {
        start++;
        errored = true;
    }
    while (isdigit(str[start])) // The exponent part (if applicable)
    {
        if (str[start] == '0' && initialRun)
            explLeadZero = true;
        initialRun = false;
        add(digits, &str[start], sizeof(char*));
        start++;
        explLen++;
    }
}
if (errored) // TODO Fix up, if necessary.
    throwError(LEXERR, 11, initial, start - initial);
if (real)
{
    if (intLen > 5) // Too long.
    {
        throwError(LEXERR, 3, initial, start - initial);
        errored = true;
    }
    if (fractionLen > 5) // Nope. Too long.
    {
        throwError(LEXERR, 4, initial, start - initial);
        errored = true;
    }
    if (explLen > 2) // Too long again.
    {

```

```

        throwError(LEXERR, 5, initial, start - initial);
        errored = true;
    }
    if (hasE && expLen == 0) // 3.4E what???
    {
        throwError(LEXERR, 6, initial, start - initial);
        errored = true;
    }

    storage -> val = parseReal(digits);

    if (leadZero && intLen > 1) // Leading zero error!
    {
        throwError(LEXERR, 8, initial, start - initial);
        errored = true;
    }
    if (expLeadZero)
    {
        throwError(LEXERR, 10, initial, start - initial);
        errored = true;
    }
    if (trailZero) // Trailing zero error!
    {
        throwError(LEXERR, 9, initial, start - initial);
        errored = true;
    }
    storage -> category = REAL;
} else
{
    if (intLen > 10)
    {
        throwError(LEXERR, 2, initial, start - initial);
        errored = true;
    }
    storage -> type = parseInt(digits);
    if (leadZero && !(storage -> type == 0))
    {
        throwError(LEXERR, 7, initial, start - initial);
        errored = true;
    }
    storage -> category = INT;
}
if (errored)
{
    storage -> category = NOOP;
}

return start;
}

// The processing
typedef int (*machine)(Token*, char*, int);
const static machine machines[] = {whitespace, idres, numMachine, grouping,
                                   catchall, relop, addop, mulop};

bool initialized = false;
int start;

```

```

int passLine(char* newLine)
{
    strcpy(buffer, newLine);
    start = 0;
    initialized = true;
    return 0;
}

Token* getNextToken()
{
    if (initialized) {
        while (errorList -> size > 0)
            passError((Token *) pop(errorList), buffer);

        Token* current = malloc(sizeof(*current));
        int end;
        current -> start = start;
        for (int i = 0; i < sizeof(machines)/sizeof(machine); i++)
        {
            current -> type = 0;
            end = (*machines[i])(current, buffer, start);
            if (end > start) {
                current -> length = end - start;
                start = end;
                return current;
            }
        }

        // Unrecognized symbol error. This error is manual because it takes
        // the place of a lexeme, rather than being processed during one.
        current -> category = LEXERR;
        current -> type = 0;
        current -> start = start;
        current -> length = 1;
        start++;
        return current;
    } else {
        fprintf(stderr, "%s\n", "Processor not initialized. Aborting.");
        return NULL;
    }
}

int initializeTokens(FILE* resFile)
{
    if (resFile) {
        buffer = malloc(sizeof(char)*73);
        initResWords(resFile);
        initSymbolTable();
        errorList = malloc(sizeof(*errorList));
    } else {
        fprintf(stderr, "%s\n", "Reserved words file for analyzer null!");
    }
    return 1;
}

```

LexicalAnalyzer.h

```

#ifndef LEXICAL_ANALYZER_H
#define LEXICAL_ANALYZER_H

int passError(Token* description, char* line);

#endif // LEXICAL_ANALYZER_H

```

```

LexicalAnalyzer.c
#include<stdio.h>
#include<stdlib.h>
#include "dataStructures/linkedList/linkedlist.h"
#include "machines/processor.h"

// Global file constants
static const char* lexErrs[] = {"Unrecognized symbol:",
                                "ID length exceeded 10 characters:",
                                "Int length exceeded 10 characters:",
                                "Integer part of real exceeded 5 characters:",
                                "Fractional part of real exceeded 5 characters:",
                                "Exponent part of long real exceeded 2 characters:",
                                "Missing exponent part of long real:",
                                "Leading 0 in int:",
                                "Excessive leading 0 in real:",
                                "Trailing 0 in real:",
                                "Leading 0 in exponent:",
                                "Attempt to use real exponent:"};

static const char TOKEN_PATH[] = "out/tokens.dat";
static const char LISTING_PATH[] = "out/listing.txt";
static const char RESWORD_PATH[] = "compiler/reswords.dat";
static const char* TEST_PATH;

static const int TokenLineSpace = 10;
static const int TokenTypeSpace = 15;
static const int TokenAttrSpace = 20;
static const int TokenLexSpace = 20;

static const int ListingLineSpace = 10;
static const int ListingErrSpace = 50;
static const int ListingLexSpace = 20;

static int LINE = 1;
static FILE* sourceFile;
static FILE* listingFile;
static FILE* tokenFile;

// Returns 1 on failure, 0 on success.
int init() {
    sourceFile = fopen(TEST_PATH, "r");
    listingFile = fopen(LISTING_PATH, "w+");
    tokenFile = fopen(TOKEN_PATH, "w+");
    FILE* resFile = fopen(RESWORD_PATH, "r");
    initializeTokens(resFile);
    fclose(resFile);

    if (sourceFile == NULL)
    {

```

```

        fprintf(stderr, "%s\n", "Source was null?");
        fclose(listingFile);
        return 1;
    }
    if (tokenFile == NULL)
    {
        fprintf(stderr, "%s\n", "Token file could not be created.");
        fclose(listingFile);
        return 1;
    }
    for (size_t i = ASSIGNOP; i <= LEXERR; i++) {
        fprintf(tokenFile, "%-5zu%s\n", i, catNames[i]);
    }
    fprintf(tokenFile, "%*s%*s%*s%*s\n", TokenLineSpace, "Line",
                                                TokenLexSpace, "Lexeme",
                                                TokenTypeSpace, "Token Type",
                                                TokenAttrSpace, "Token Attribute");

    return 0;
}

int passError(Token* description, char* line)
{
    fprintf(tokenFile, "%*d%*.*s%*d%*d\n", TokenLineSpace, LINE,
        TokenLexSpace, description -> length, &line[description -> start],
        TokenTypeSpace, description -> category, TokenAttrSpace,
        description -> type);
    fprintf(listingFile, "%*s:%*s%*.*s\n", ListingLineSpace - 1,
        catNames[description -> category], ListingErrSpace,
        lexErrs[description -> type], ListingLexSpace, description -> length,
        &line[description -> start]);
    return 0;
}

void writeEOFToken()
{
    fprintf(tokenFile, "%*d%*.*s%*d%*d\n", TokenLineSpace, LINE, TokenLexSpace,
        3, "EOF", TokenTypeSpace, FILEEND, TokenAttrSpace, 0);
}

void updateLine(char* line)
{
    passLine(line);
    fprintf(listingFile, "%*d\t\t%s", ListingLineSpace, LINE, line);
}

void writeToken(Token* token, char* line)
{
    if (token -> category == WS || token -> category == NOOP) // Don't bother including in the output file.
        return;
    if (token -> category == LEXERR) // For catching the unrecognized symbol error
    {
        passError(token, line);
        return;
    }
}

```

```

    fprintf(tokenFile, "%d%*.s%d", TokenLineSpace, LINE, TokenLexSpace,
            token -> length, &line[token -> start], TokenTypeSpace,
            token -> category);
    switch (token -> category) {
        case REAL:
            fprintf(tokenFile, "%*f", TokenAttrSpace, token -> val);
            break;

        case ID:
            fprintf(tokenFile, "%*p", TokenAttrSpace, token -> id);
            break;

        default:
            fprintf(tokenFile, "%*d", TokenAttrSpace, token -> type);
            break;
    }
    fprintf(tokenFile, "\n");
}

// void printWords(LinkedList* list)
// {
//     struct node* node = list->head;
//     while (node)
//     {
//         printf("Printing symbol: %s\n", (char *) node->data);
//         node = node -> next;
//     }
// }

int run()
{
    char line[72];
    if (fgets(line, sizeof(line), sourceFile) != NULL)
        updateLine(line);
    Token* next = malloc(sizeof(*next));
    while ((next = getNextToken()))
    {
        writeToken(next, line);
        if (next -> category == WS && next -> type == 1)
        {
            LINE++;
            if (fgets(line, sizeof(line), sourceFile) != NULL)
            {
                updateLine(line);
            } else { // Error or end of file (assume the latter)
                writeEOFToken();
                return 0;
            }
        }
    }
    return 1;
}

int main(int argc, char *argv[]) {
    if (argc != 2) {

```

```
        fprintf(stderr, "%s\n", "Expected exactly one file to compile!");
    } else {
        TEST_PATH = argv[1];
    }
    if (init() == 0) {
        if (run() != 0)
            fprintf(stderr, "%s\n", "Run failed. Could not terminate properly.");
    } else {
        fprintf(stderr, "%s\n", "Initialization process failed in lexical analyzer.");
    }
    fclose(listingFile);
    return 0;
}
```
