

CS 4013: Compiler Construction: Project 1

Nate Beckemeyer

September 2016

Introduction

For project 1, I wrote a lexical analyzer in C for Pascal. 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.

The purpose of the lexical analyzer is to

1 Methodology

The lexical analyzer is an

2 Implementation

The implementation.

3 Discussion & Conclusions

The discussions & conclusions.

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);
  writeln(23.47E);
  writeln(037);
  writeln(0.4562E23);
  writeln(01.45620E02);
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          writeln(23.47E);
LEXERR:      Missing exponent part of long real:      23.47E
9          writeln(037);
```

```

LEXERR:                Leading 0 in int:                037
10                    writeln(0.4562E23);
11                    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
12                    end.?.
LEXERR:                Unrecognized symbol:              ?

```

Line	Lexeme	Error Token File	Token Type	Token Attribute
1	program		CONTROL	7
1	fib		ID	0x7fdb2bd00b50
1	(GROUP	0
1	input		ID	0x7fdb2bd00c90
1	,		PUNC	0
1	output		ID	0x7fdb2bd00e10
1)		GROUP	1
1	;		PUNC	1
2	var		VAR	0
2	excessivelyLongIntegerArrayName		ID	0x7fdb2bd013b0
2	excessivelyLongIntegerArrayName		LEXERR	1
2	:		TYPE	0
2	array		ARRAY	0
2	[GROUP	2
2	1		INT	1
2	..		ARRAY	1
2	12		INT	12
2]		GROUP	3
2	of		ARRAY	2
2	integer		TYPE	1
2	;		PUNC	1
4	begin		CONTROL	0
5	init		ID	0x7fdb2bd01c00
5	;		PUNC	1
6	writeln		ID	0x7fdb2bd01e20
6	(GROUP	0
6	123456789012345		INT	-2045911175
6	123456789012345		LEXERR	2
6)		GROUP	1
6	;		PUNC	1
7	writeln		ID	0x7fdb2bd01e40
7	(GROUP	0
7	123456.123456E003		REAL	123456123.456000
7	123456.123456E003		LEXERR	10
7	123456.123456E003		LEXERR	5
7	123456.123456E003		LEXERR	4
7	123456.123456E003		LEXERR	3
7)		GROUP	1
7	;		PUNC	1
8	writeln		ID	0x7fdb2bd01e40
8	(GROUP	0
8	23.47E		REAL	23.470000
8	23.47E		LEXERR	6
8)		GROUP	1
8	;		PUNC	1
9	writeln		ID	0x7fdb2bd01e40

9	(GROUP	0
9	037	INT	37
9	037	LEXERR	7
9)	GROUP	1
9	;	PUNC	1
10	writeln	ID	0x7fdb2bd01e40
10	(GROUP	0
10	0.4562E23	REAL	45620000000000003145728.000000
10)	GROUP	1
10	;	PUNC	1
11	writeln	ID	0x7fdb2bd01e40
11	(GROUP	0
11	01.45620E02	REAL	145.620000
11	01.45620E02	LEXERR	9
11	01.45620E02	LEXERR	10
11	01.45620E02	LEXERR	8
11)	GROUP	1
11	;	PUNC	1
12	end	CONTROL	3
12	.	PUNC	2
12	?	LEXERR	0
13	EOF	FILEEND	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			
Line	Lexeme	Token Type	Token Attribute
1	program	CONTROL	7
1	fib	ID	0x7f8b64c033a0
1	(GROUP	0
1	input	ID	0x7f8b64c034e0
1	,	PUNC	0
1	output	ID	0x7f8b64c03660
1)	GROUP	1
1	;	PUNC	1
2	var	VAR	0
2	n	ID	0x7f8b64c03840
2	,	PUNC	0
2	p	ID	0x7f8b64c03920
2	:	TYPE	0
2	integer	TYPE	1
2	;	PUNC	1
3	var	VAR	0
3	q	ID	0x7f8b64c03c40
3	:	TYPE	0
3	real	TYPE	2
3	;	PUNC	1
4	var	VAR	0
4	numsArray	ID	0x7f8b64c04000

4	:	TYPE	0
4	array	ARRAY	0
4	[GROUP	2
4	1	INT	1
4	..	ARRAY	1
4	12	INT	12
4]	GROUP	3
4	of	ARRAY	2
4	integer	TYPE	1
4	;	PUNC	1
6	function	CONTROL	4
6	fib	ID	0x7f8b64c033c0
6	(GROUP	0
6	a	ID	0x7f8b64c04840
6	:	TYPE	0
6	integer	TYPE	1
6	;	PUNC	1
6	b	ID	0x7f8b64c04ac0
6	,	PUNC	0
6	c	ID	0x7f8b64c04ba0
6	:	TYPE	0
6	real	TYPE	2
6)	GROUP	1
6	:	TYPE	0
6	real	TYPE	2
6	;	PUNC	1
7	begin	CONTROL	0
8	if	CONTROL	5
8	a	ID	0x7f8b64c04860
8	<=	RELOP	1
8	1	INT	1
8	then	CONTROL	8
8	fib	ID	0x7f8b64c033c0
8	:=	ASSIGNOP	0
8	c	ID	0x7f8b64c04bc0
9	else	CONTROL	2
9	fib	ID	0x7f8b64c033c0
9	:=	ASSIGNOP	0
9	fib	ID	0x7f8b64c033c0
9	(GROUP	0
9	a	ID	0x7f8b64c04860
9	-	ADDOP	1
9	1	INT	1
9	,	PUNC	0
9	c	ID	0x7f8b64c04bc0
9	,	PUNC	0
9	b	ID	0x7f8b64c04ae0
9	+	ADDOP	0
9	c	ID	0x7f8b64c04bc0
9)	GROUP	1
10	end	CONTROL	3
10	;	PUNC	1
12	function	CONTROL	4
12	fib2	ID	0x7f8b64c05f80
12	(GROUP	0
12	a	ID	0x7f8b64c04860
12	:	TYPE	0

12	integer	TYPE	1
12)	GROUP	1
12	:	TYPE	0
12	integer	TYPE	1
12	;	PUNC	1
13	var	VAR	0
13	b	ID	0x7f8b64c04ae0
13	,	PUNC	0
13	c	ID	0x7f8b64c04bc0
13	,	PUNC	0
13	sum	ID	0x7f8b64c06700
13	:	TYPE	0
13	integer	TYPE	1
13	;	PUNC	1
14	begin	CONTROL	0
15	a	ID	0x7f8b64c04860
15	:=	ASSIGNOP	0
15	a	ID	0x7f8b64c04860
15	-	ADDOP	1
15	1	INT	1
15	;	PUNC	1
16	b	ID	0x7f8b64c04ae0
16	:=	ASSIGNOP	0
16	0	INT	0
16	;	PUNC	1
17	sum	ID	0x7f8b64c06720
17	:=	ASSIGNOP	0
17	1	INT	1
17	;	PUNC	1
18	c	ID	0x7f8b64c04bc0
18	:=	ASSIGNOP	0
18	b	ID	0x7f8b64c04ae0
18	;	PUNC	1
19	while	CONTROL	9
19	(GROUP	0
19	not	INVERSE	0
19	a	ID	0x7f8b64c04860
19	<	RELOP	0
19	-1	INT	1
19)	GROUP	1
19	do	CONTROL	1
20	begin	CONTROL	0
21	a	ID	0x7f8b64c04860
21	:=	ASSIGNOP	0
21	a	ID	0x7f8b64c04860
21	-	ADDOP	1
21	1	INT	1
21	;	PUNC	1
22	b	ID	0x7f8b64c04ae0
22	:=	ASSIGNOP	0
22	sum	ID	0x7f8b64c06720
22	;	PUNC	1
23	sum	ID	0x7f8b64c06720
23	:=	ASSIGNOP	0
23	c	ID	0x7f8b64c04bc0
23	+	ADDOP	0
23	sum	ID	0x7f8b64c06720

23	;	PUNC	1
24	c	ID	0x7f8b64c04bc0
24	:=	ASSIGNOP	0
24	b	ID	0x7f8b64c04ae0
25	end	CONTROL	3
25	;	PUNC	1
26	fib2	ID	0x7f8b64c05fa0
26	:=	ASSIGNOP	0
26	sum	ID	0x7f8b64c06720
26	;	PUNC	1
27	end	CONTROL	3
27	;	PUNC	1
29	procedure	CONTROL	6
29	init	ID	0x7f8b64c08be0
29	(GROUP	0
29)	GROUP	1
29	;	PUNC	1
30	begin	CONTROL	0
31	n	ID	0x7f8b64c03860
31	:=	ASSIGNOP	0
31	12	INT	12
31	;	PUNC	1
32	if	CONTROL	5
32	(GROUP	0
32	1	INT	1
32	and	MULOP	2
32	2	INT	2
32)	GROUP	1
32	or	ADDOP	2
32	3	INT	3
32	then	CONTROL	8
32	p	ID	0x7f8b64c03940
32	:=	ASSIGNOP	0
32	12	INT	12
33	else	CONTROL	2
33	p	ID	0x7f8b64c03940
33	:=	ASSIGNOP	0
33	14	INT	14
33	;	PUNC	1
34	numsArray	ID	0x7f8b64c04020
34	[GROUP	2
34	3	INT	3
34]	GROUP	3
34	:=	ASSIGNOP	0
34	15.56	REAL	15.560000
34	;	PUNC	1
35	q	ID	0x7f8b64c03c60
35	:=	ASSIGNOP	0
35	12	INT	12
36	end	CONTROL	3
36	;	PUNC	1
38	begin	CONTROL	0
39	init	ID	0x7f8b64c08c00
39	;	PUNC	1
40	writeln	ID	0x7f8b64c0a560
40	(GROUP	0
40	+	ADDOP	0

40	fib	ID	0x7f8b64c033c0
40	(GROUP	0
40	n	ID	0x7f8b64c03860
40	,	PUNC	0
40	0	INT	0
40	,	PUNC	0
40	1	INT	1
40)	GROUP	1
40	*	MULOP	0
40	q	ID	0x7f8b64c03c60
40	/	MULOP	1
40	p	ID	0x7f8b64c03940
40	+	ADDOP	0
40	4	INT	4
40)	GROUP	1
40	;	PUNC	1
41	writeln	ID	0x7f8b64c0a580
41	(GROUP	0
41	fib2	ID	0x7f8b64c05fa0
41	(GROUP	0
41	n	ID	0x7f8b64c03860
41)	GROUP	1
41)	GROUP	1
41	;	PUNC	1
42	writeln	ID	0x7f8b64c0a580
42	(GROUP	0
42	numsArray	ID	0x7f8b64c04020
42	[GROUP	2
42	3	INT	3
42]	GROUP	3
42)	GROUP	1
43	end	CONTROL	3
43	.	PUNC	2
44	EOF	FILEEND	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, 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", "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;
    }
}

```



```

{
    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;

    if (str[start] == '-' && isdigit(str[start + 1]))
    {
        start++;
        sign = -1;
    }
    else if (str[start] == '+' && isdigit(str[start + 1]))
        start++;

    LinkedList* digits = malloc(sizeof(*digits));
    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++;
    while (isdigit(str[start])) // The exponent part (if applicable)
    {
        if (str[start] == '0' && initialRun)
            expLeadZero = true;
        initialRun = false;
        add(digits, &str[start], sizeof(char*));
        start++;
        expLen++;
    }
}
if (real)
{
    if (intLen > 5) // Too long.
        throwError(LEXERR, 3, initial, start - initial);
    if (fractionLen > 5) // Nope. Too long.
        throwError(LEXERR, 4, initial, start - initial);
    if (expLen > 2) // Too long again.
        throwError(LEXERR, 5, initial, start - initial);
    if (hasE && expLen == 0) // 3.4E what???
        throwError(LEXERR, 6, initial, start - initial);

    storage -> val = parseReal(digits);

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

```

```

        throwError(LEXERR, 7, initial, start - initial);
        storage -> category = INT;
    }

    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:"};

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;
    }
    fprintf(tokenFile, "%s%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%.s%.d\n", TokenLineSpace, LINE,
        TokenLexSpace, description -> length, &line[description -> start],
        TokenTypeSpace, catNames[description -> category], TokenAttrSpace,
        description -> type);
    fprintf(listingFile, "%s:%.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%.s%.d\n", TokenLineSpace, LINE, TokenLexSpace,
        3, "EOF", TokenTypeSpace, catNames[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) // 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%*.s", TokenLineSpace, LINE, TokenLexSpace,
            token -> length, &line[token -> start], TokenTypeSpace,
            catNames[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;
}

```
