Sam Beckmann
Compiler Construction
Project 1: Lexical Analyzer

Introduction

My project is titled *Paper*, and its source code can be found at github.com/samvbeckmann/paper. It is a compiler for a subset of the Pascal language, written in C.

Project 1, the lexical analyzer, reads source Pascal files as input and outputs both a “listing” file and a “token” file for each input. The listing file contains the source code, with line numbers added, as well as a listing of all errors in each line. For example:

SRC

```
      program test #@
```

LISTING FILE

```
      1         program test #@
      LEXERR:   Unrecognized Symbol:   #
      LEXERR:   Unrecognized Symbol:   @
```

The token file contains all the tokens in the source program, each listed on it’s own line.

SRC

```
      program test #@
```

TOKEN FILE

```
      1            program      10                 0
      1            test         50                 0x2840ab83d
      1            #            99                 1
      1            @            99                 1
```

Methodology

The lexical analyzer works as a NFAϵ, running the source file through 6 machines in series, until one returns a valid token. The last machine, catch_all, is guaranteed to return a token, so the compiler will be able to match all elements of the source file. The basic loop of the program is reading a line from the source program → Matching tokens in the line, advancing a pointer after each token matched → Getting a new line when the end of the line is reached by the pointer. This pointer is referred to as the “back” pointer, which points to the location in the file after which no token has been matched. Each machine uses a “forward” pointer, which begins equal to the back pointer and advances as the line is read. If the machine matches a token, the back pointer is set the forward pointer. If a machine fails to match a token, the forward pointer is reset to the back pointer for the next machine.

Implementation

The implementation of the analyzer is divided into three major parts, the analyzer framework, the machines, and the symbol table. The framework is responsible for reading the input files, calling each of the machines in order, and writing the returned tokens to the token and listing files. The machines file contains each of the seven machines, as well as the data structures associated with the tokens. The symbol table file manages the linked lists of ids and reserved words.

The seven machines are whitespace, long_real, real, int, id_res, relop, and catchall. The whitespace machine is called first, and is the only machine that does not return a token. The whitespace machine returns a new position for the forward pointer, advancing it past any whitespace. If there is no whitespace at the forward pointer, the return value matches the input. All other machines return an **optional_token**, which is either a token or null, with the exception of the catch_all machine, which is guaranteed to always return a token. If the catch_all machine does not recognize any valid token, it returns an “unrecognized symbol” lexerr token.

Reserved words are read in from a RESERVED_WORDS file in the same directory as the executable. Each line of the file contains a reserved word, a token type, and an attribute, space-separated. This file is read into a linked list that all IDs are compared against before they are added to the symbol table. the symbol table is another linked list, which contains all IDs found in the program.

Discussion and Conclusions

Although nothing is being done with the tokens yet, setting up the parsing of inputs files put a solid, easy to expand upon structure for *Paper*. The symbol table is easily accessible, tokens are stored in expandable and convenient data structures, which are largely created by factories. One of my goals when designing the data structures and layout of this project was to make it easy to add additional projects, and I believe the project is well set up for this.

The project is 1189 lines, is fully documented, and uses no external C libraries. For testing, *Paper* was complied using gcc on a machine running OS X El Capitan. The project is licensed under the MIT license. I am the sole contributor to this project.

Appendix 1: Sample Inputs and Outputs

Standard Test File (number_test.pas)

SRC

```
program numbertest(input, output);
var num1, num2: integer;

function fibonacci(first, second, num: integer): integer;
begin
    if num <= 0 then fibonacci := second
    else fibonacci := fibonacci(second, first + second, num - 1)
end;

function gcd(first, second: integer): integer;
begin
    if second = 0 then gcd := first
    else gcd := gcd(second, first mod second)
end;

function sumbelow(first, second, max: integer): integer;
var temp total: integer;
begin
    temp := 1;
    total := 0;
    while temp < max do
        begin
            if (temp mod first = 0) or (temp mod second = 0) then
                total := total + temp;
            temp := temp + 1
        end;
    sumbelow := total
end;

begin
    read(num1, num2);
    writeln(num1, fibonacci(0, 1, num1 - 1));
    writeln(gcd(num1, num2));
    write(num1, num2);
    writeln(sumbelow(num1, num2, 300));
end.
```

LISTING FILE

```
1         program numbertest(input, output);
2         var num1, num2: integer;
3
4         function fibonacci(first, second, num: integer): integer;
5         begin
6             if num <= 0 then fibonacci := second
7             else fibonacci := fibonacci(second, first + second, num - 1)
8         end;
9
10        function gcd(first, second: integer): integer;
11        begin
12            if second = 0 then gcd := first
13            else gcd := gcd(second, first mod second)
14        end;
15
16        function sumbelow(first, second, max: integer): integer;
17        var temp total: integer;
18        begin
```

```
19            temp := 1;
20            total := 0;
21            while temp < max do
22                begin
23                    if (temp mod first = 0) or (temp mod second = 0) then
24                        total := total + temp;
25                    temp := temp + 1
26                end;
27            sumbelow := total
28        end;
29
30        begin
31            read(num1, num2);
32            writeln(num1, fibonacci(0, 1, num1 − 1));
33            writeln(gcd(num1, num2));
34            write(num1, num2);
35            writeln(sumbelow(num1, num2, 300));
36        end.
```

TOKEN FILE

```
1   program                   10     0
1   numbertest                50     0x7fc22bc031e0
1   (                         32     0
1   input                     50     0x7fc22bc03560
1   ,                         31     0
1   output                    50     0x7fc22bc03640
1   )                         33     0
1   ;                         30     0
2   var                       23     0
2   num1                      50     0x7fc22bc03720
2   ,                         31     0
2   num2                      50     0x7fc22bc038c0
2   :                         36     0
2   integer                   90     1
2   ;                         30     0
4   function                  11     0
4   fibonacci                 50     0x7fc22bc039a0
4   (                         32     0
4   first                     50     0x7fc22bc03ba0
4   ,                         31     0
4   second                    50     0x7fc22bc03c80
4   ,                         31     0
4   num                       50     0x7fc22bc03d60
4   :                         36     0
4   integer                   90     1
4   )                         33     0
4   :                         36     0
4   integer                   90     1
4   ;                         30     0
5   begin                     13     0
6   if                        15     0
6   num                       50     0x7fc22bc03d60
6   <=                        80     3
6   0                         40     1
6   then                      16     0
6   fibonacci                 50     0x7fc22bc039a0
6   :=                        37     0
6   second                    50     0x7fc22bc03c80
7   else                      17     0
7   fibonacci                 50     0x7fc22bc039a0
7   :=                        37     0
7   fibonacci                 50     0x7fc22bc039a0
7   (                         32     0
7   second                    50     0x7fc22bc03c80
```

```
  7  ,                          31     0
  7  first                      50     0x7fc22bc03ba0
  7  +                          70     1
  7  second                     50     0x7fc22bc03c80
  7  ,                          31     0
  7  num                        50     0x7fc22bc03d60
  7  -                          70     2
  7  1                          40     1
  7  )                          33     0
  8  end                        14     0
  8  ;                          30     0
 10  function                   11     0
 10  gcd                        50     0x7fc22bc03e40
 10  (                          32     0
 10  first                      50     0x7fc22bc03ba0
 10  ,                          31     0
 10  second                     50     0x7fc22bc03c80
 10  :                          36     0
 10  integer                    90     1
 10  )                          33     0
 10  :                          36     0
 10  integer                    90     1
 10  ;                          30     0
 11  begin                      13     0
 12  if                         15     0
 12  second                     50     0x7fc22bc03c80
 12  =                          80     5
 12  0                          40     1
 12  then                       16     0
 12  gcd                        50     0x7fc22bc03e40
 12  :=                         37     0
 12  first                      50     0x7fc22bc03ba0
 13  else                       17     0
 13  gcd                        50     0x7fc22bc03e40
 13  :=                         37     0
 13  gcd                        50     0x7fc22bc03e40
 13  (                          32     0
 13  second                     50     0x7fc22bc03c80
 13  ,                          31     0
 13  first                      50     0x7fc22bc03ba0
 13  mod                        60     4
 13  second                     50     0x7fc22bc03c80
 13  )                          33     0
 14  end                        14     0
 14  ;                          30     0
 16  function                   11     0
 16  sumbelow                   50     0x7fc22bd00b00
 16  (                          32     0
 16  first                      50     0x7fc22bc03ba0
 16  ,                          31     0
 16  second                     50     0x7fc22bc03c80
 16  ,                          31     0
 16  max                        50     0x7fc22bd017e0
 16  :                          36     0
 16  integer                    90     1
 16  )                          33     0
 16  :                          36     0
 16  integer                    90     1
 16  ;                          30     0
 17  var                        23     0
 17  temp                       50     0x7fc22bd01a40
 17  total                      50     0x7fc22bd01d60
 17  :                          36     0
 17  integer                    90     1
```

```
17   ;                          30     0
18   begin                      13     0
19   temp                       50     0x7fc22bd01a40
19   :=                         37     0
19   1                          40     1
19   ;                          30     0
20   total                      50     0x7fc22bd01d60
20   :=                         37     0
20   0                          40     1
20   ;                          30     0
21   while                      18     0
21   temp                       50     0x7fc22bd01a40
21   <                          80     1
21   max                        50     0x7fc22bd017e0
21   do                         19     0
22   begin                      13     0
23   if                         15     0
23   (                          32     0
23   temp                       50     0x7fc22bd01a40
23   mod                        60     4
23   first                      50     0x7fc22bc03ba0
23   =                          80     5
23   0                          40     1
23   )                          33     0
23   or                         70     3
23   (                          32     0
23   temp                       50     0x7fc22bd01a40
23   mod                        60     4
23   second                     50     0x7fc22bc03c80
23   =                          80     5
23   0                          40     1
23   )                          33     0
23   then                       16     0
24   total                      50     0x7fc22bd01d60
24   :=                         37     0
24   total                      50     0x7fc22bd01d60
24   +                          70     1
24   temp                       50     0x7fc22bd01a40
24   ;                          30     0
25   temp                       50     0x7fc22bd01a40
25   :=                         37     0
25   temp                       50     0x7fc22bd01a40
25   +                          70     1
25   1                          40     1
26   end                        14     0
26   ;                          30     0
27   sumbelow                   50     0x7fc22bd00b00
27   :=                         37     0
27   total                      50     0x7fc22bd01d60
28   end                        14     0
28   ;                          30     0
30   begin                      13     0
31   read                       50     0x7fc22bd01de0
31   (                          32     0
31   num1                       50     0x7fc22bc03720
31   ,                          31     0
31   num2                       50     0x7fc22bc038c0
31   )                          33     0
31   ;                          30     0
32   writeln                    50     0x7fc22bd032a0
32   (                          32     0
32   num1                       50     0x7fc22bc03720
32   ,                          31     0
32   fibonacci                  50     0x7fc22bc039a0
32   (                              32     0
32   0                              40     1
32   ,                              31     0
32   1                              40     1
32   ,                              31     0
32   num1                           50     0x7fc22bc03720
32   -                              70     2
32   1                              40     1
32   )                              33     0
32   )                              33     0
32   ;                              30     0
33   writeln                        50     0x7fc22bd032a0
33   (                              32     0
33   gcd                            50     0x7fc22bc03e40
33   (                              32     0
33   num1                           50     0x7fc22bc03720
33   ,                              31     0
33   num2                           50     0x7fc22bc038c0
33   )                              33     0
33   )                              33     0
33   ;                              30     0
34   write                          50     0x7fc22bd03560
34   (                              32     0
34   num1                           50     0x7fc22bc03720
34   ,                              31     0
34   num2                           50     0x7fc22bc038c0
34   )                              33     0
34   ;                              30     0
35   writeln                        50     0x7fc22bd032a0
35   (                              32     0
35   sumbelow                       50     0x7fc22bd00b00
35   (                              32     0
35   num1                           50     0x7fc22bc03720
35   ,                              31     0
35   num2                           50     0x7fc22bc038c0
35   ,                              31     0
35   300                            40     1
35   )                              33     0
35   )                              33     0
35   ;                              30     0
36   end                            14     0
36   .                              38     0
```

Error Test File (error_test.pas)

SRC

```
0123 12345678901
1234.1234E+123 1234.1234E-0 1234.1234E+03
123456.123 123.4567890 00.73 24.000
^% #
abc123def456
```

LISTING FILE

```
1         0123 12345678901
LEXERR:   Leading Zeroes:     0123
LEXERR:   Extra Long Integer: 12345678901
2         1234.1234E+123 1234.1234E-0 1234.1234E+03
LEXERR:   Extra Long Real:    1234.1234E+123
LEXERR:   Leading Zeroes:     1234.1234E-0
LEXERR:   Leading Zeroes:     1234.1234E+03
3         123456.123 123.4567890 00.73 24.000
LEXERR:   Extra Long Real:    123456.123
```

```
LEXERR:   Extra Long Real:    123.4567890
LEXERR:   Leading Zeroes:     00.73
LEXERR:   Leading Zeroes:     24.000
4         ^% #
LEXERR:   Unrecognized Sym:   ^
LEXERR:   Unrecognized Sym:   %
LEXERR:   Unrecognized Sym:   #
5         abc123def456
LEXERR:   Extra Long ID:      abc123def456
```

TOKEN FILE

```
         1   0123                      99     5
         1   12345678901               99     3
         2   1234.1234E+123            99     4
         2   1234.1234E-0              99     5
         2   1234.1234E+03             99     5
         3   123456.123                99     4
         3   123.4567890               99     4
         3   00.73                     99     5
         3   24.000                    99     5
         4   ^                         99     1
         4   %                         99     1
         4   #                         99     1
         5   abc123def456              99     2
```

Appendix 2: Code Listing

ANALYZER.C

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "machines.h"
#include "analyzer.h"
#include "symbols.h"

int main(int argc, char *argv[])
{
        for(int i = 1; i < argc; i++) {
                compile_file(argv[i]);
        }
}

/*
 * Compiles the given Pascal file.
 * Creates two files in the directory of the given file:
 *      - .listing file which displays the source with line numbers and errors.
 *      - .tokens file which has a line for each token in the source.
 *
 * Arguments: src -> path to source file.
 */
static void compile_file(char src[])
{
        global_sym_table = malloc(sizeof(struct Symbol));
        global_sym_table -> ptr = NULL;

        FILE *sfp;
        FILE *lfp;
        FILE *tfp;
        FILE *rfp;

        char noext[40];
        strcpy(noext, src);
        *(strrchr(noext, '.') + 1) = '\0';

        char lfname[50];
        strcpy(lfname, noext);
        strcat(lfname, "listing");

        char tkname[50];
        strcpy(tkname, noext);
        strcat(tkname, "tokens");

        sfp = fopen(src, "r");
        lfp = fopen(lfname, "w");
        tfp = fopen(tkname, "w");
        rfp = fopen("RESERVED_WORDS", "r");

        if (sfp == NULL) {
                fprintf(stderr, "Source file \"%s\" does not exist.\n", src);
                return;
        } else if (rfp == NULL) {
                fprintf(stderr, "RESERVED_WORDS file not found.\n");
                return;
        }

        initialize_reserved_words(rfp);

        char buff[72];
        int line = 0;
        fgets(buff, 72, (FILE*) sfp);
        while(!feof(sfp)) {
                fprintf(lfp, "%-10d", ++line);
                fputs(buff, lfp);
                generate_tokens(line, buff, tfp, lfp);
                fgets(buff, 72, (FILE*) sfp);
        }

        fclose(sfp);
        fclose(lfp);
        fclose(tfp);
        fclose(rfp);
}

/*
 * Adds all tokens for the line into the token file.
 * Reports lexical errors to the listing file.
 *
 * Arguments: line -> line number that is currently being read.
 *            buff -> char array that contins a line of the source file.
 *            tfp -> Pointer to the token file that tokens are written to.
 *            lfp -> Pointer to the listing file, where errors are written.
 */
static void generate_tokens(int line, char buff[], FILE *tfp, FILE *lfp)
{
        char *forward = buff;
        char *back = buff;

        while (*forward != '\n') {
                forward = ws_machine(forward);
                back = forward;

                struct Token token = match_token(forward);
                if (token.is_id) {
                        fprintf(tfp, "%4d\t%-20s\t%-2d\t%-p\n",
                                        line,
                                        token.lexeme,
                                        token.token_type,
                                        token.attribute.ptr);
                } else {
                        fprintf(tfp, "%4d\t%-20s\t%-2d\t%-d\n",
                                        line,
                                        token.lexeme,
                                        token.token_type,
                                        token.attribute.attribute);
                }

                if (token.token_type == 99) {
                        fprintf(lfp, "LEXERR:   %-20s%s\n",
                                error_codes[token.attribute.attribute- 1],
                                token.lexeme);
                }

                forward = token.forward;
                back = forward;
        }
}

/*
 * Runs a buffer through all of the machines to match a token.
 *
 * Arguments: forward -> Pointer to memory location to begin reading from.
 *
 * Returns: Token that was matched from one of the machines. Some token will
 *          always be matched by the catch-all machine, so this is garunteed.
 */
static struct Token match_token(char *forward)
{
        union Optional_Token result;

        result = longreal_machine(forward);
        if (result.nil != NULL)
                return result.token;

        result = real_machine(forward);
        if (result.nil != NULL)
                return result.token;

        result = int_machine(forward);
        if (result.nil != NULL)
                return result.token;

        result = id_res_machine(forward);
        if (result.nil != NULL)
                return result.token;

        result = relop_machine(forward);
        if (result.nil != NULL)
                return result.token;

        return catchall_machine(forward);
}
```

ANALYER.H

```
#ifndef ANALYZER_H
#define ANALYZER_H

#include "machines.h"

/*
 * Constant array of error code strings. Used for reporting error in a human
 * readable format in the listing file.
 */
const char * const error_codes[] = {
                "Unrecognized Sym:",
                "Extra Long ID:",
                "Extra Long Integer:",
                "Extra Long Real:",
                "Leading Zeroes:" };

/*
 * Compiles the given Pascal file.
 * Creates two files in the directory of the given file:
 *      - .listing file which displays the source with line numbers and errors.
 *      - .tokens file which has a line for each token in the source.
 *
 * Arguments: src -> path to source file.
 */
static void compile_file(char src[]);

/*
 * Adds all tokens for the line into the token file.
 * Reports lexical errors to the listing file.
 *
 * Arguments: line -> line number that is currently being read.
 *            buff -> char array that contins a line of the source file.
 *            tfp -> Pointer to the token file that tokens are written to.
 *            lfp -> Pointer to the listing file, where errors are written.
 */
static void generate_tokens(int line, char buff[], FILE *tfp, FILE *lfp);

/*
 * Runs a buffer through all of the machines to match a token.
 *
 * Arguments: forward -> Pointer to memory location to begin reading from.
 *
 * Returns: Token that was matched from one of the machines. Some token will
 *          always be matched by the catch-all machine, so this is garunteed.
 */
static struct Token match_token(char *forward);

#endif
```

MACHINES.C

```
#include "machines.h"
#include "word_defs.h"
#include "symbols.h"
#include <string.h>
#include <ctype.h>
#include <stdbool.h>
#include <stdlib.h>

/*
 * Factory for Optional_Tokens.
 * Takes in needed parameters for a token, and makes an Optional_Token with
 * those parameters. Abstracts the creation of Optional_Token structs.
 *
 * Arguments: lexeme -> Literal of matched lexeme.
 *            type -> Integer representation of token's type.
 *            attr -> Integer representation of token's attribute.
 *            forward -> Pointer to the char after this lexeme ended in buffer.
 *
 * Returns: An Optional_Token with the given parameters. Not a null optional.
 */
union Optional_Token make_optional(
                        char lexeme[],
                        int type,
                        int attr,
                        char *forward) {
        return wrap_token(make_token(lexeme, type, attr, forward));
}

/*
 * Factory for Tokens.
 * Takes in needed parameters for a token, and makes an Optional_Token with
 * those paratmers. Abstracts the creation of Token structs.
 *
 * Arguments: lexeme -> Literal of matched lexeme.
 *            type -> Integer representation of token's type.
 *            attr -> Integer representation of token's attribute.
 *            forward -> Pointer to the char after this lexeme ended in buffer.
 *
 * Returns: A Token with the given parameters. This does not create an id
token.
 */
struct Token make_token(char lexeme[], int type, int attr, char *forward) {
        struct Token token;
        strcpy(token.lexeme, lexeme);
        token.token_type = type;
        token.is_id = 0;
        token.attribute.attribute = attr;
        token.forward = forward;
        return token;
}

/*
 * Creates an Optional_Token which is nil.
 * Used as standard factory of nil Optional_Token structs.
 *
 * Returns: Optional_Token with "nil" as the token.
 */
union Optional_Token null_optional() {
        union Optional_Token op_token;
        op_token.nil = NULL;
        return op_token;
}

/*
 * Wraps a token as an Optional_Token, so that it can be returned as such.
 *
 * Arguments: token -> Token that is to be wrapped.
 *
 * Returns: Optional_Token that contains the paramter "token"
 */
union Optional_Token wrap_token(struct Token token)
{
        union Optional_Token op_token;
        op_token.token = token;
        return op_token;
}

/*
 * Reads a series of digits until a non-digit character is read, returning a
 * buffer of read digits.
 *
 * Arguments: forward -> Pointer to where begin reading.
 *
 * Returns: char pointer to buffer or read digits.
 */
static char * read_digits(char *forward) {
        char * buff = malloc(30);
        int i = 0;
        char value = *forward++;
        while (isdigit(value)) {
                buff[i] = value;
                value = *forward++;
                i++;
        }
        buff[i] = '\0';
        return buff;
}

/*
 * Machine that matches whitespace.
 *
 * Arguments: forward -> Pointer to memory location to begin reading from.

 * Returns: Pointer to first non-whitespace character matched.
 */
char * ws_machine(char *forward)
{
        char value;
        do {
                value = *forward++;
        } while (value == ' ' || value == '\t');
        forward--;

        return forward;
}

/*
 * Machine that reads real numbers containing an exponent, or "Long Reals".
 *
 * A long real consists of 1-5 digits, a decimal point, 1-5 digits, "E",
 * an optional sign (+|-), and 1-2 digits.
 *
 * Arguments: forward -> Pointer to memory location to begin reading from.
 *
 * Returns: an Optional_Token representing the matched long real, or a nil
 *          Optional_Token if no long real is matched.
 */
union Optional_Token longreal_machine(char *forward)
{
        char real_lit[30];
        bool extra_long = false;
        bool lead_zeros = false;

        char * first_part = read_digits(forward);
        int len = strlen(first_part);
        forward += len;
        strcpy(real_lit, first_part);

        if (len == 0)
                return null_optional();
        else if (len > 5)
                extra_long = true;
        else if (first_part[0] == '0' && len != 1)
                lead_zeros = true;

        char value = *forward++;
        if (value != '.')
                return null_optional();
        strncat(real_lit, &value, 1);

        char *second_part = read_digits(forward);
        len = strlen(second_part);
        forward += len;
        strcat(real_lit, second_part);

        if (len == 0)
                return null_optional();
        else if (len > 5)
                extra_long = true;
        else if (second_part[0] == '0' && len != 1)
                lead_zeros = true;

        value = *forward++;
        if (value != 'E')
                return null_optional();
        strncat(real_lit, &value, 1);

        value = *forward++;
        if (value == '-' || value == '+')
                strncat(real_lit, &value, 1);
        else
                forward--;

        char *exponent = read_digits(forward);
        len = strlen(exponent);
        forward += len;
        strcat(real_lit, exponent);

        if (len == 0)
                return null_optional();
        else if (len > 2)
                extra_long = true;
        else if (exponent[0] == '0')
                lead_zeros = true;

        if (extra_long)
                return make_optional(real_lit, LEXERR, EXTRA_LONG_REAL,
forward);
        else if (lead_zeros)
                return make_optional(real_lit, LEXERR, LEADING_ZEROES,
forward);
        else
                return make_optional(real_lit, NUM, LONG_REAL, forward);

        return null_optional();
}

/*
 * Machine that reads real numbers.
 *
 * A real number consists of 1-5 digits, a decimal point, and 1-5 digits.
 *
 * Arguments: forward -> Pointer to memory location to begin reading from.
 *
 * Returns: An Optional_Token representing the matched real, or a nil
 *          Optional_Token if no real number is matched.
 */
union Optional_Token real_machine(char *forward)
{
        char real_lit[30];
        bool extra_long = false;
        bool lead_zeros = false;

        char * first_part = read_digits(forward);
        int len = strlen(first_part);
        forward += len;
        strcpy(real_lit, first_part);

        if (len == 0)
                return null_optional();
        else if (len > 5)
                extra_long = true;
        else if (first_part[0] == '0' && len != 1)
                lead_zeros = true;

        char value = *forward++;
        if (value != '.')
                return null_optional();
        strncat(real_lit, &value, 1);

        char *second_part = read_digits(forward);
        len = strlen(second_part);
        forward += len;
        strcat(real_lit, second_part);

        if (len == 0)
                return null_optional();
        else if (len > 5)
                extra_long = true;
        else if (second_part[0] == '0' && len != 1)
                lead_zeros = true;

        if (extra_long)
                return make_optional(real_lit, LEXERR, EXTRA_LONG_REAL,
forward);
        else if (lead_zeros)
                return make_optional(real_lit, LEXERR, LEADING_ZEROES,
forward);
        else
                return make_optional(real_lit, NUM, REAL, forward);
}

/*
 * Machine that reads integers.
 *
 * An integer consists of 1-10 digits with no leading zeroes.
 *
 * Arguments: forward -> Pointer to memory location to begin reading from.
 *
 * Returns: An Optional_Token representing the matched integer, or a nil
 *          Optional_Token if no integer is matched.
 */
union Optional_Token int_machine(char *forward)
{
        char *digits = read_digits(forward);
        int len = strlen(digits);
        forward += len;

        if (len == 0)
                return null_optional();
        else if (digits[0] == '0' && len != 1)
                return make_optional(digits, LEXERR, LEADING_ZEROES, forward);
        else if (len > 10)
                return make_optional(digits, LEXERR, EXTRA_LONG_INT, forward);
        else
                return make_optional(digits, NUM, INTEGER, forward);
}

/*
 * Machine that matches ids and reserved words.
 *
 * An ID consists of a letter, followed by 0-9 digits or letters.
 * If the matched string is equivalent to a reserved word, returns the token
 * that represents the reserved word.
 * Otherwise, adds the ID to the symbol table if it is not already there,
 * and returns an Optional_Token containg the matched ID and a reference
 * to it in the symbol table.
 *
 * Arguments: forward -> Pointer to memory location to begin reading from.
 *
 * Returns: A LEXERR Optional_Token if an error is encountered, or a a nil
 *          Optional_Token if no id or reserved word is matched.
 */
union Optional_Token id_res_machine(char *forward)
{
        char word[30];
        int i = 0;
        char value = *forward++;
        while (isalnum(value)) {
                word[i] = value;
                value = *forward++;
                i++;
        }
        forward--;
        word[i] = '\0';

        if (i == 0)
                return null_optional();
        else if (i > 10)
                return make_optional(word, LEXERR, EXTRA_LONG_ID, forward);

        union Optional_Token res = check_reserved_words(word);
        if (res.nil != NULL) {
                res.token.forward = forward;
                return res;
        } else {
                struct Symbol *sym_ptr = add_symbol(word);
                struct Token token;
                strcpy(token.lexeme, word);
                token.token_type = ID;
                token.is_id = 1;
                token.attribute.ptr = sym_ptr;
                token.forward = forward;
                return wrap_token(token);
        }
}

/*
 * Machine that matches relational operators, or "Relops".
 *
 * Valid relops: '<', '>', '==', '<=', '>=', '<>'.
 *
 * Arguments: forward -> Pointer to memory location to begin reading from.
 *
 * Returns: An Optional_Token representing the matched relop, or a nil
 *          Optional_Token if no relop is matched.
 */
union Optional_Token relop_machine(char *forward)
{
        char value = *forward++;
        switch (value) {
        case '<':
                value = *forward++;
                switch (value) {
                case '>':
                        return make_optional("<>", RELOP, NEQ, forward);
                case '=':
                        return make_optional("<=", RELOP, LT_EQ, forward);
                default:
                        forward--;
                        return make_optional("<", RELOP, LT, forward);
                }
        case '>':
                value = *forward++;
                if (value == '=') {
                        return make_optional(">=", RELOP, GT_EQ, forward);
                } else {
                        forward--;
                        return make_optional(">", RELOP, GT, forward);
                }
        case '=':
                return make_optional("=", RELOP, EQ, forward);
        default:
                return null_optional();
        }
}
```

```
/*
 * Machine that caches all other tokens not matched by a previous machine.
 *
 * If no valid token is matched by this machine, it returns a LEXERR for an
 * unrecognized symbol. This garuntees this machine will always return a token.
 *
 * Arguments: forward -> Pointer to memory location to begin reading from.
 *
 * Returns: Token either containing a valid token, attribute pair, or a LEXERR
 *          token if no valid token is matched.
 */
struct Token catchall_machine(char *forward)
{
        char value = *forward++;
        char lexeme[2];

        switch (value) {
        case '+':
                return make_token("+", ADDOP, ADD, forward);
        case '-':
                return make_token("-", ADDOP, SUB, forward);
        case '*':
                return make_token("*", MULOP, MULT, forward);
        case '/':
                return make_token("/", MULOP, DIVIDE, forward);
        case ';':
                return make_token(";", SEMI, 0, forward);
        case ',':
                return make_token(",", COMMA, 0, forward);
        case '(':
                return make_token("(", PAREN_OPEN, 0, forward);
        case ')':
                return make_token(")", PAREN_CLOSE, 0, forward);
        case '[':
                return make_token("[", BR_OPEN, 0, forward);
        case ']':
                return make_token("]", BR_CLOSE, 0, forward);
        case ':':
                value = *forward++;
                if (value == '=') {
                        return make_token(":=", ASSIGN, 0, forward);
                } else {
                        forward--;
                        return make_token(":", COLON, 0, forward);
                }
        case '.':
                value = *forward++;
                if (value == '.') {
                        return make_token("..", TWO_DOT, 0, forward);
                } else {
                        forward--;
                        return make_token(".", DOT, 0, forward);
                }
        default:
                lexeme[0] = value;
                lexeme[1] = '\0';
                return make_token(lexeme, LEXERR, UNRECOG_SYM, forward);
        }
}
```

MACHINES.H

```
#ifndef MACHINES_H
#define MACHINES_H
```

```
/*
 * A token is the basic unit the Pascal interpretation.
 *
 * Fields: lexeme -> The literal from source that is this token.
 *         is_id -> 1 if this token represents an id, otherwise 0.
 *         token_type -> integer that represents this token's type.
 *         Attribute.attribute -> Integer that represents the type's attribute.
 *         Attribute.ptr -> Pointer to a symbol in the symbol table.
 *                          Used if this token is an id.
 *         forward -> Pointer to next position in buffer after lexeme.
 *                    Used to update the forward pointer, then discarded.
 */
struct Token {
        char lexeme[20];
        int is_id;
        int token_type;
        union Attribute {
                int attribute;
                struct Symbol *ptr;
        } attribute;
        char *forward;
};

/*
 * An Optional_Token is either a token or null.
 * Used as a return type for machines that may not match a token.
 *
 * Fields: nil -> Void pointer if the Optional_Token is nil.
 *         token -> Token if Optional_Token is not null.
 */
union Optional_Token {
        void *nil;
        struct Token token;
};

/*
 * Factory for Optional_Tokens.
 * Takes in needed parameters for a token, and makes an Optional_Token with
 * those parameters. Abstracts the creation of Optional_Token structs.
 *
 * Arguments: lexeme -> Literal of matched lexeme.
 *            type -> Integer representation of token's type.
 *            attr -> Integer representation of token's attribute.
 *            forward -> Pointer to the char after this lexeme ended in buffer.
 *
 * Returns: An Optional_Token with the given parameters. Not a null optional.
 */
union Optional_Token make_optional(char lexeme[], int type, int attr, char
*forward);

/*
 * Factory for Tokens.
 * Takes in needed parameters for a token, and makes an Optional_Token with
 * those paratmers. Abstracts the creation of Token structs.
 *
 * Arguments: lexeme -> Literal of matched lexeme.
 *            type -> Integer representation of token's type.
 *            attr -> Integer representation of token's attribute.
 *            forward -> Pointer to the char after this lexeme ended in buffer.
 *
 * Returns: A Token with the given parameters. This does not create an id
token.
 */
struct Token make_token(char lexeme[], int type, int attr, char *forward);
```

```c
/*
 * Creates an Optional_Token which is nil.
 * Used as standard factory of nil Optional_Token structs.
 *
 * Returns: Optional_Token with "nil" as the token.
 */
union Optional_Token null_optional();

/*
 * Wraps a token as an Optional_Token, so that it can be returned as such.
 *
 * Arguments: token -> Token that is to be wrapped.
 *
 * Returns: Optional_Token that contains the paramter "token"
 */
union Optional_Token wrap_token(struct Token token);

/*
 * Machine that matches whitespace.
 *
 * Arguments: forward -> Pointer to memory location to begin reading from.

 * Returns: Pointer to first non-whitespace character matched.
 */
char * ws_machine(char *forward);

/*
 * Machine that reads real numbers containing an exponent, or "Long Reals".
 *
 * A long real consists of 1-5 digits, a decimal point, 1-5 digits, "E",
 * an optional sign (+|-), and 1-2 digits.
 *
 * Arguments: forward -> Pointer to memory location to begin reading from.
 *
 * Returns: an Optional_Token representing the matched long real, or a nil
 *          Optional_Token if no long real is matched.
 */
union Optional_Token longreal_machine(char *forward);

/*
 * Machine that reads real numbers.
 *
 * A real number consists of 1-5 digits, a decimal point, and 1-5 digits.
 *
 * Arguments: forward -> Pointer to memory location to begin reading from.
 *
 * Returns: An Optional_Token representing the matched real, or a nil
 *          Optional_Token if no real number is matched.
 */
union Optional_Token real_machine(char *forward);

/*
 * Machine that reads integers.
 *
 * An integer consists of 1-10 digits with no leading zeroes.
 *
 * Arguments: forward -> Pointer to memory location to begin reading from.
 *
 * Returns: An Optional_Token representing the matched integer, or a nil
 *          Optional_Token if no integer is matched.
 */
union Optional_Token int_machine(char *forward);
```

```
/*
 * Machine that matches ids and reserved words.
 *
 * An ID consists of a letter, followed by 0-9 digits or letters.
 * If the matched string is equivalent to a reserved word, returns the token
 * that represents the reserved word.
 * Otherwise, adds the ID to the symbol table if it is not already there,
 * and returns an Optional_Token containg the matched ID and a reference
 * to it in the symbol table.
 *
 * Arguments: forward -> Pointer to memory location to begin reading from.
 *
 * Returns: A LEXERR Optional_Token if an error is encountered, or a a nil
 *          Optional_Token if no id or reserved word is matched.
 */
union Optional_Token id_res_machine(char *forward);

/*
 * Machine that matches relational operators, or "Relops".
 *
 * Valid relops: '<', '>', '==', '<=', '>=', '<>'.
 *
 * Arguments: forward -> Pointer to memory location to begin reading from.
 *
 * Returns: An Optional_Token representing the matched relop, or a nil
 *          Optional_Token if no relop is matched.
 */
union Optional_Token relop_machine(char *forward);

/*
 * Machine that caches all other tokens not matched by a previous machine.
 *
 * If no valid token is matched by this machine, it returns a LEXERR for an
 * unrecognized symbol. This garuntees this machine will always return a token.
 *
 * Arguments: forward -> Pointer to memory location to begin reading from.
 *
 * Returns: Token either containing a valid token, attribute pair, or a LEXERR
 *          token if no valid token is matched.
 */
struct Token catchall_machine(char *forward);

#endif
```

SYMBOLS.C

```
#include "symbols.h"
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

struct Symbol *global_sym_table;
struct Reserved_Word *reserved_word_table;

/*
 * Adds a symbol to the symbol table if it is not already present. If the
symbol
 * is already present, returns a pointer to that Symbol.
 *
 * Arguments: word -> literal symbol to be added to the table.
 *
 * Returns: A pointer to the symbol in the table.
 */
struct Symbol * add_symbol(char word[])
```

```
{
        struct Symbol *current = global_sym_table;

        while (current -> ptr != NULL) {
                if (strcmp(current -> word, word) == 0)
                        return current;
                current = current -> ptr;
        }

        current -> ptr = malloc(sizeof(struct Symbol));
        strcpy(current -> word, word);
        current -> ptr -> ptr = NULL;

        return current;
}

/*
 * Adds a reserved word to the reserved word table.
 *
 * Arguments: word -> Literal of the word to be added.
 *            type -> Token type associated with the reserved word.
 *            attr -> Token attribute associated with the reserved word.
 *
 * Returns: A pointer to the reserved word added to the table.
 */
struct Reserved_Word * add_reserved_word(char word[], int type, int attr)
{
        struct Reserved_Word *current = reserved_word_table;

        while (current -> next != NULL) {
                current = current -> next;
        }

        current -> next = malloc(sizeof(struct Reserved_Word));
        strcpy(current -> word, word);
        current -> token_type = type;
        current -> attribute = attr;
        current -> next -> next = NULL;

        return current -> next;
}

/*
 * Checks if a given word is in the reserved word table.
 *
 * Arguments: word -> Literal of the word to be checked.
 *
 * Returns: The token associated with the reserved word. If no reserved word is
 *          found, returns a null Optional_Token.
 */
union Optional_Token check_reserved_words(char word[])
{
        struct Reserved_Word *current = reserved_word_table;

        do {
                if (strcmp(current -> word, word) == 0) {
                        return make_optional(word,
                                        current -> token_type,
                                        current -> attribute,
                                        NULL);
                }
                current = current -> next;
        } while (current -> next != NULL);
        return null_optional();
}

/*
 * Initializes the reserved word table from the RESERVED_WORDS file.
 *
 * Arguments: rfp -> Pointer to the reserved word file.
 *
 * Returns: A pointer to the reserved word table.
 */
struct Reserved_Word * initialize_reserved_words(FILE *rfp)
{
        reserved_word_table = malloc(sizeof(struct Reserved_Word));
        reserved_word_table -> next = NULL;
        char buff[80];
        fgets(buff, 80, rfp);

        while(!feof(rfp)) {
                if (buff[0] != '\n') {
                        char word[11];
                        int type;
                        int attr;

                        sscanf(buff, "%s %d %d", word, &type, &attr);
                        add_reserved_word(word, type, attr);
                }
                fgets(buff, 80, rfp);
        }
        return reserved_word_table;
}
```

SYMBOLS.H

```
#ifndef SYMBOLS_H
#define SYMBOLS_H

#include "machines.h"
#include <stdio.h>

/*
 * A Symbol for an ID in the symbol table.
 *
 * Fields: word -> Literal of the lexeme symbol.
 *         ptr -> Pointer to the next symbol in the table.
 */
struct Symbol {
        char word[11];
        struct Symbol *ptr;
};

/*
 * Contains a reserved word from the reserved word table.
 *
 * Fields: word -> Literal of the reserved word.
 *         token_type -> Integer of token type assoicated with the word.
 *         attribute -> Integer of attribute associated with the word.
 *         next -> Pointer to the next reserved word in the table.
 */
struct Reserved_Word {
        char word[11];
        int token_type;
        int attribute;
        struct Reserved_Word *next;
};
```

```
/*
 * Global symbol table. Pointer to first item in the linked list.
 */
extern struct Symbol *global_sym_table;

/*
 * Rerved word table. Pointer fo first item in the linked list.
 */
extern struct Reserved_Word *reserved_word_table;

/*
 * Adds a symbol to the symbol table if it is not already present. If the
symbol
 * is already present, returns a pointer to that Symbol.
 *
 * Arguments: word -> literal symbol to be added to the table.
 *
 * Returns: A pointer to the symbol in the table.
 */
struct Symbol * add_symbol(char word[]);

/*
 * Checks if a given word is in the reserved word table.
 *
 * Arguments: word -> Literal of the word to be checked.
 *
 * Returns: The token associated with the reserved word. If no reserved word is
 *          found, returns a null Optional_Token.
 */
union Optional_Token check_reserved_words(char word[]);

/*
 * Initializes the reserved word table from the RESERVED_WORDS file.
 *
 * Arguments: rfp -> Pointer to the reserved word file.
 *
 * Returns: A pointer to the reserved word table.
 */
struct Reserved_Word * initialize_reserved_words(FILE *rfp);

#endif
```

WORD_DEFS.H

```
#ifndef WORD_DEFS_H
#define WORD_DEFS_H

// token types
#define PROGRAM 10
#define FUNCTION 11
#define PROCEDURE 12
#define BEGIN 13
#define END 14
#define IF 15
#define THEN 16
#define ELSE 17
#define WHILE 18
#define DO 19
#define NOT 20
#define ARRAY 21
#define OF 22
#define VAR 23
```

```
#define SEMI 30
#define COMMA 31
#define PAREN_OPEN 32
#define PAREN_CLOSE 33
#define BR_OPEN 34
#define BR_CLOSE 35
#define COLON 36
#define ASSIGN 37
#define DOT 38
#define TWO_DOT 39

#define NUM 40
#define ID 50
#define MULOP 60
#define ADDOP 70
#define RELOP 80
#define STANDARD_TYPE 90
#define LEXERR 99

// Addops
#define ADD 1
#define SUB 2
#define OR 3

// Mulops
#define MULT 1
#define DIVIDE 2
#define DIV 3
#define MOD 4
#define AND 5

// Relops
#define LT 1
#define GT 2
#define LT_EQ 3
#define GT_EQ 4
#define EQ 5
#define NEQ 6

// Standard types
#define INTEGER 1
#define REAL 2
#define LONG_REAL 3

// Error Codes
#define UNRECOG_SYM 1
#define EXTRA_LONG_ID 2
#define EXTRA_LONG_INT 3
#define EXTRA_LONG_REAL 4
#define LEADING_ZEROES 5

#endif
```

RESERVED_WORDS

```
and 60 5
array 21 0
begin 13 0
div 60 3
do 19 0
else 17 0
end 14 0
function 11 0
if 15 0
integer 90 1
```

```
mod 60 4
not 20 0
of 22 0
or 70 3
procedure 12 0
program 10 0
real 90 2
then 16 0
var 23 0
while 18 0
```