CS 4013: Compiler Construction
Project 1

Benjamin James

November 29, 2017

Introduction

This project is a lexical analyzer for the Pascal language. The lexer is a standalone program written in C that reads a source file and a reserved word file and produces tokens, detects and displays errors, produces a listing file, and prints out a symbol table. This is done by breaking up the input into lines and reading each line into a 72-character max buffer to be parsed. Each line is sent to a machine that breaks up lexemes and finds the appropriate representation.

Methodology

The purpose of the lexical analyzer is to break up a source file into little pieces that can be properly analyzed by the parser. These little pieces are called tokens and can be passed to a parser of a compiler to produce a parse tree. To do this, each token has to have a type and attribute to show type and value differences to the parser. After reading in reserved words, the lexer is able to determine each lexeme and its corresponding type-attribute pair. According to the specifications of the language provided[1] (e.g. 72 chars max line) errors are also detected and reported. Determining each token is done by finite state machines, specifically NFA with epsilons. This is done as shown in class by using a forward and backward pointer to the line. Specifically, this is advancing to each machine in order, setting the backward pointer if the machine succeeds, otherwise reseting the forward pointer to the backward pointer. Each type has a corresponding machine.

Implementation

The implementation is done in ANSI C. See Appendix II for details. The first step is reading in all the reserved words into a linked list. Then, all machines are added to the machine linked list.

Each machine is a struct consisting of a function pointer that has the logic for the machine, the forward pointer, and the backward pointer. The token to be returned is also part of the struct. The advantage to this architecture is that iterating through the linked list, each function can be called until one returns a token.

A token is (as discussed in class) a struct consisting a type, a flag specifying the attribute type, and a union that contains a pointer or an attribute value.

The numerical machines (integer, real, long real) were the most complex due to the different states and errors that could be produced. In all of the numerical machines, the regex $+$ occurs so frequently that a separate function, digit˙plus, has been made to compartmentalize the code. The function also returns the length of the regex match, so a too long integer, mantissa, exponent, etc. can be measured. The lexeme also can be checked for a leading or trailing zero when applicable.

The IDRES machine was also somewhat complex, due to the steps of checking the length, checking the reserved word list, checking the symbol table, then adding to the symbol table, stopping when necessary. A full linear search was done on both linked lists of the reserved words and identifiers, something that could be improved in a later version.

The symbol table is the linked list of IDs that are found in the program. At the moment, the printed table contains the pointer to the char* string, then the lexeme. The pointers are unique and cross-referenced in the token table, so they could be searched for by the parser. Errors are printed to the listing file when they occur, and the corresponding error can also be placed in the token file.

Discussion and Conclusions

A few test cases were made to demonstrate the validity of the lexer. Errors planted in the code were correctly found and dealt with accordingly. Each symbol found was placed in the symbol table.

The lexer was run with all tests using valgrind -v –leak-check=full, and no memory leaks were found. Additionally, the program was compiled with -Wall -Wextra to show all warnings, and the only error to show up was unused parameter.

For optimizations, a binary search tree may be beneficial to use in the future instead of a linked list for IDs and reserved words for faster search.

References

[1] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison-Wesley series in computer science and information processing, Addison-Wesley Publishing Company, 1986.

Appendix I: Sample Inputs and Outputs

bad_lex

Listing 1: bad_lex.pas

```
abcdefghij
abcdefghijk
@

12345678901
1234567890

12345.3
123456.3

1.12345
1.123456


123
0123
01.2
01.2E2

1230
1.2
1.20
1.20E-12

1.2E-10
1.2E-123
1.2E+5
1.2E+123


e#
3.4E+;
3.E4+;
34E+-;
E3.4+;

abcdefghijklmnopqrstuvwxyz0123456789abcdefghijklmnopqrstuvwxyz012345678
abcdefghijklmnopqrstuvwxyz0123456789abcdefghijklmnopqrstuvwxyz0123456789
```

Listing 2: bad_lex.list

```
1       abcdefghij
2       abcdefghijk
LEXERR: ID too long:            abcdefghijk
3       @
LEXERR: Unrecognized symbol:            @
4
5       12345678901
LEXERR: Int too long:           12345678901
6       1234567890
7
8       12345.3
9       123456.3
LEXERR: Mantissa too long:              123456.3
10
```

```
11      1.12345
12      1.123456
LEXERR: Fraction too long:              1.123456
13
14
15      123
16      0123
LEXERR: Leading zero:           0123
17      01.2
LEXERR: Leading zero:           01.2
18      01.2E2
LEXERR: Leading zero:           01.2E2
19
20      1230
21      1.2
22      1.20
LEXERR: Trailing zero:          1.20
23      1.20E-12
LEXERR: Trailing zero:          1.20E-12
24
25      1.2E-10
LEXERR: Trailing zero:          1.2E-10
26      1.2E-123
LEXERR: Exponent too long:              1.2E-123
27      1.2E+5
28      1.2E+123
LEXERR: Exponent too long:              1.2E+123
29
30
31      e#
LEXERR: Unrecognized symbol:            #
32      3.4E+;
LEXERR: No exponent:            3.4E+
33      3.E4+;
LEXERR: No fractional part:             3.E4
34      34E+-;
35      E3.4+;
36
37      abcdefghijklmnopqrstuvwxyz0123456789abcdefghijklmnopqrstuvwxyz012345678
LEXERR: ID too long:
   abcdefghijklmnopqrstuvwxyz0123456789abcdefghijklmnopqrstuvwxyz012345678
38
LEXERR: Line too long:
```

Listing 3: bad_lex.sym

```
0x55442b0       E3
0x5544010       E
0x5543a00       e
0x55410f0       abcdefghij
```

Listing 4: bad_lex.tok

```
1       abcdefghij        1     0x55410f0
2       abcdefghijk       99    1
3       @         99    2
5       12345678901       99    3
6       1234567890        6     1234567890
8       12345.3   5     12345
9       123456.3          99    4
11      1.12345   5     1
```

```
 9  12      1.123456          99    5
10  15      123       6       123
11  16      0123      99      6
12  17      01.2      99      6
13  18      01.2E2    99      6
14  20      1230      6       1230
15  21      1.2       5       1
16  22      1.20      99      7
17  23      1.20E-12          99    7
18  25      1.2E-10   99      7
19  26      1.2E-123          99    9
20  27      1.2E+5    5       120000
21  28      1.2E+123          99    9
22  31      e         1       0x5543a00
23  31      #         99      2
24  32      3.4E+     99      10
25  32      ;         33      59
26  33      3.E4      99      11
27  33      +         7       43
28  33      ;         33      59
29  34      34        6       34
30  34      E         1       0x5544010
31  34      +         7       43
32  34      -         7       45
33  34      ;         33      59
34  35      E3        1       0x55442b0
35  35      .         25      46
36  35      4         6       4
37  35      +         7       43
38  35      ;         33      59
39  37      abcdefghijklmnopqrstuvwxyz0123456789abcdefghijklmnopqrstuvwxyz012345678   99
        1
40  39      EOF       0       0
```

bad_syn

Listing 5: bad_syn.pas

```
1  program test(input, output);
2          var a : integer;
3          var b : array[1..a] of real;
4          function f1(a : integer, x : int) : real ;
5                  var c : integer;
6                  function f2(p : integer) : integer ;
7                  var d : real;
8                  begin
9                          f2 := 5 + p;
10                         if p > d then
11                                 if p = 5 then
12                                         f2 := f + 10
13                         else
14                                 f2 := 100
15                 end
16         begin
17                 f1 := f2(a) * x
18         end
19         function f2(a : real) : real ;
20         var b : integer;
21         begin
22                 f3 := 10 * f1(a, 2.3);
23         end
24 begin
25         f1(5, 3.2)
26 end.
27 begin
28
29 end
```

Listing 6: bad_syn.list

```
1        program test(input, output);
2                var a : integer;
3                var b : array[1..a] of real;
4                function f1(a : integer, x : int) : real ;
5                        var c : integer;
6                        function f2(p : integer) : integer ;
7                        var d : real;
8                        begin
9                                f2 := 5 + p;
10                               if p > d then
11                                       if p = 5 then
12                                               f2 := f + 10
13                               else
14                                       f2 := 100
15                       end
16               begin
17                       f1 := f2(a) * x
18               end
19               function f2(a : real) : real ;
20               var b : integer;
21               begin
22                       f3 := 10 * f1(a, 2.3);
23               end
24       begin
25               f1(5, 3.2)
```

```
26 26       end.
27 27       begin
28 28
29 29       end
```

Listing 7: bad_syn.sym

```
1  0x5548310        f3
2  0x5546590        f
3  0x55450d0        d
4  0x55449d0        p
5  0x5544820        f2
6  0x5544350        c
7  0x5543de0        int
8  0x5543b90        x
9  0x5543710        f1
10 0x5542e30        b
11 0x5542960        a
12 0x55425d0        output
13 0x55423d0        input
14 0x5542220        test
```

Listing 8: bad_syn.tok

```
1  1        program    20     0
2  1        test       1      0x5542220
3  1        (          27     40
4  1        input      1      0x55423d0
5  1        ,          34     44
6  1        output     1      0x55425d0
7  1        )          28     41
8  1        ;          33     59
9  2        var        23     0
10 2        a          1      0x5542960
11 2        :          32     58
12 2        integer    17     0
13 2        ;          33     59
14 3        var        23     0
15 3        b          1      0x5542e30
16 3        :          32     58
17 3        array      10     0
18 3        [          29     91
19 3        1          6      1
20 3        ..         26     26
21 3        a          1      0x5542960
22 3        ]          30     93
23 3        of         19     0
24 3        real       21     0
25 3        ;          33     59
26 4        function            15     0
27 4        f1         1      0x5543710
28 4        (          27     40
29 4        a          1      0x5542960
30 4        :          32     58
31 4        integer    17     0
32 4        ,          34     44
33 4        x          1      0x5543b90
34 4        :          32     58
35 4        int        1      0x5543de0
36 4        )          28     41
37 4        :          32     58
```

```
4       real        21      0
4       ;           33      59
5       var         23      0
5       c           1       0x5544350
5       :           32      58
5       integer     17      0
5       ;           33      59
6       function            15      0
6       f2          1       0x5544820
6       (           27      40
6       p           1       0x55449d0
6       :           32      58
6       integer     17      0
6       )           28      41
6       :           32      58
6       integer     17      0
6       ;           33      59
7       var         23      0
7       d           1       0x55450d0
7       :           32      58
7       real        21      0
7       ;           33      59
8       begin       11      0
9       f2          1       0x5544820
9       :=          31      0
9       5           6       5
9       +           7       43
9       p           1       0x55449d0
9       ;           33      59
10      if          16      0
10      p           1       0x55449d0
10      >           4       45
10      d           1       0x55450d0
10      then        22      0
11      if          16      0
11      p           1       0x55449d0
11      =           4       43
11      5           6       5
11      then        22      0
12      f2          1       0x5544820
12      :=          31      0
12      f           1       0x5546590
12      +           7       43
12      10          6       10
13      else        13      0
14      f2          1       0x5544820
14      :=          31      0
14      100         6       100
15      end         14      0
16      begin       11      0
17      f1          1       0x5543710
17      :=          31      0
17      f2          1       0x5544820
17      (           27      40
17      a           1       0x5542960
17      )           28      41
17      *           3       42
17      x           1       0x5543b90
18      end         14      0
19      function            15      0
```

```
19      f2          1     0x5544820
19      (           27    40
19      a           1     0x5542960
19      :           32    58
19      real        21    0
19      )           28    41
19      :           32    58
19      real        21    0
19      ;           33    59
20      var         23    0
20      b           1     0x5542e30
20      :           32    58
20      integer     17    0
20      ;           33    59
21      begin       11    0
22      f3          1     0x5548310
22      :=          31    0
22      10          6     10
22      *           3     42
22      f1          1     0x5543710
22      (           27    40
22      a           1     0x5542960
22      ,           34    44
22      2.3         5     2
22      )           28    41
22      ;           33    59
23      end         14    0
24      begin       11    0
25      f1          1     0x5543710
25      (           27    40
25      5           6     5
25      ,           34    44
25      3.2         5     3
25      )           28    41
26      end         14    0
26      .           25    46
27      begin       11    0
29      end         14    0
30      EOF         0     0
```

gcd

Listing 9: gcd.pas

```
program example(input, output);
var x: integer; var y: integer;
function gcd(a:integer; b: integer): integer;
begin
        if b = 0 then gcd := a
        else gcd := gcd(b, a mod b)
end;

begin
        out := read(x, y);
        out := write(gcd(x, y))
end.
```

Listing 10: gcd.list

```
1       program example(input, output);
2       var x: integer; var y: integer;
3       function gcd(a:integer; b: integer): integer;
4       begin
5               if b = 0 then gcd := a
6               else gcd := gcd(b, a mod b)
7       end;
8
9       begin
10              out := read(x, y);
11              out := write(gcd(x, y))
12      end.
13
```

Listing 11: gcd.sym

```
0x55455f0       write
0x5544f90       read
0x5544d40       out
0x55435b0       b
0x55432c0       a
0x5543110       gcd
0x5542ce0       y
0x55428b0       x
0x5542570       output
0x5542370       input
0x55421c0       example
```

Listing 12: gcd.tok

```
1       program   20    0
1       example   1     0x55421c0
1       (         27    40
1       input     1     0x5542370
1       ,         34    44
1       output    1     0x5542570
1       )         28    41
1       ;         33    59
2       var       23    0
2       x         1     0x55428b0
2       :         32    58
2       integer   17    0
```

```
2       ;           33    59
2       var         23    0
2       y           1     0x5542ce0
2       :           32    58
2       integer     17    0
2       ;           33    59
3       function            15    0
3       gcd         1     0x5543110
3       (           27    40
3       a           1     0x55432c0
3       :           32    58
3       integer     17    0
3       ;           33    59
3       b           1     0x55435b0
3       :           32    58
3       integer     17    0
3       )           28    41
3       :           32    58
3       integer     17    0
3       ;           33    59
4       begin       11    0
5       if          16    0
5       b           1     0x55435b0
5       =           4     43
5       0           6     0
5       then        22    0
5       gcd         1     0x5543110
5       :=          31    0
5       a           1     0x55432c0
6       else        13    0
6       gcd         1     0x5543110
6       :=          31    0
6       gcd         1     0x5543110
6       (           27    40
6       b           1     0x55435b0
6       ,           34    44
6       a           1     0x55432c0
6       mod         3     37
6       b           1     0x55435b0
6       )           28    41
7       end         14    0
7       ;           33    59
9       begin       11    0
10      out         1     0x5544d40
10      :=          31    0
10      read        1     0x5544f90
10      (           27    40
10      x           1     0x55428b0
10      ,           34    44
10      y           1     0x5542ce0
10      )           28    41
10      ;           33    59
11      out         1     0x5544d40
11      :=          31    0
11      write       1     0x55455f0
11      (           27    40
11      gcd         1     0x5543110
11      (           27    40
11      x           1     0x55428b0
11      ,           34    44
```

```
11      y           1       0x5542ce0
11      )           28      41
11      )           28      41
12      end         14      0
12      .           25      46
14      EOF         0       0
```

test

Listing 13: test.pas

```
program test(input, output);
        var a : integer;
        var b : array[1..5] of real;
        function f1(a : integer; x : real) : real ;
                var c : integer;
                function f2(p : integer) : integer ;
                var d : real;
                begin
                        f2 := 5 + p
                end;
        begin
                f1 := f2(a) * x
        end;
        function f3(a : real) : real ;
        var b : integer;
        begin
                f3 := 10 * f1(a, 2.3)
        end;
begin
        out := f3(5.3)
end.
```

Listing 14: test.list

```
1       program test(input, output);
2               var a : integer;
3               var b : array[1..5] of real;
4               function f1(a : integer; x : real) : real ;
5                       var c : integer;
6                       function f2(p : integer) : integer ;
7                       var d : real;
8                       begin
9                               f2 := 5 + p
10                      end;
11              begin
12                      f1 := f2(a) * x
13              end;
14              function f3(a : real) : real ;
15              var b : integer;
16              begin
17                      f3 := 10 * f1(a, 2.3)
18              end;
19      begin
20              out := f3(5.3)
21      end.
22
```

Listing 15: test.sym

```
0x55479d0       out
0x5546420       f3
0x5544fb0       d
0x55448b0       p
0x5544700       f2
0x5544230       c
0x5543b30       x
0x55436b0       f1
```

```
0x5542dd0       b
0x5542900       a
0x5542570       output
0x5542370       input
0x55421c0       test
```

Listing 16: test.tok

```
1       program   20    0
1       test      1     0x55421c0
1       (         27    40
1       input     1     0x5542370
1       ,         34    44
1       output    1     0x5542570
1       )         28    41
1       ;         33    59
2       var       23    0
2       a         1     0x5542900
2       :         32    58
2       integer   17    0
2       ;         33    59
3       var       23    0
3       b         1     0x5542dd0
3       :         32    58
3       array     10    0
3       [         29    91
3       1         6     1
3       ..        26    26
3       5         6     5
3       ]         30    93
3       of        19    0
3       real      21    0
3       ;         33    59
4       function          15    0
4       f1        1     0x55436b0
4       (         27    40
4       a         1     0x5542900
4       :         32    58
4       integer   17    0
4       ;         33    59
4       x         1     0x5543b30
4       :         32    58
4       real      21    0
4       )         28    41
4       :         32    58
4       real      21    0
4       ;         33    59
5       var       23    0
5       c         1     0x5544230
5       :         32    58
5       integer   17    0
5       ;         33    59
6       function          15    0
6       f2        1     0x5544700
6       (         27    40
6       p         1     0x55448b0
6       :         32    58
6       integer   17    0
6       )         28    41
6       :         32    58
6       integer   17    0
```

```
6       ;           33      59
7       var         23      0
7       d           1       0x5544fb0
7       :           32      58
7       real        21      0
7       ;           33      59
8       begin       11      0
9       f2          1       0x5544700
9       :=          31      0
9       5           6       5
9       +           7       43
9       p           1       0x55448b0
10      end         14      0
10      ;           33      59
11      begin       11      0
12      f1          1       0x55436b0
12      :=          31      0
12      f2          1       0x5544700
12      (           27      40
12      a           1       0x5542900
12      )           28      41
12      *           3       42
12      x           1       0x5543b30
13      end         14      0
13      ;           33      59
14      function            15    0
14      f3          1       0x5546420
14      (           27      40
14      a           1       0x5542900
14      :           32      58
14      real        21      0
14      )           28      41
14      :           32      58
14      real        21      0
14      ;           33      59
15      var         23      0
15      b           1       0x5542dd0
15      :           32      58
15      integer     17      0
15      ;           33      59
16      begin       11      0
17      f3          1       0x5546420
17      :=          31      0
17      10          6       10
17      *           3       42
17      f1          1       0x55436b0
17      (           27      40
17      a           1       0x5542900
17      ,           34      44
17      2.3         5       2
17      )           28      41
18      end         14      0
18      ;           33      59
19      begin       11      0
20      out         1       0x55479d0
20      :=          31      0
20      f3          1       0x5546420
20      (           27      40
20      5.3         5       5
20      )           28      41
```

```
21      end         14      0
21      .           25      46
23      EOF         0       0
```

Appendix II: Program Listings

Listing 17: common/io.c

```
/* -*- C -*-
 *
 * io.c
 *
 * Author: Benjamin T James
 */

#include <stdlib.h>
#include <stddef.h>
#include "defs.h"
#include "io.h"
#include "util.h"

int read_line(struct line *buf, FILE *f)
{
    int c, ret = 0;
    unsigned offset = 0;
        buf->len = 0;
    while ((c = getc(f)) != EOF) {
        if (offset == buf->alloc) {
            buf->len += offset;
            offset = 0;
            buf->err = LEXERR_LINE_TOO_LONG;
        }
        buf->buf[offset++] = c;
        if (c == '\n') {
            buf->buf[offset] = '\0';
            break;
        }
    }
    buf->len += offset;
    if (c == EOF) {
        ret = -1;
    }
    return ret;
}

int open_file(const char* src_file, const char*
    ext, FILE** out)
{
    char *out_name;
    FILE *f;
    if (get_out_file(src_file, ext, &out_name) <
        0) {
        return -1;
    }
        f = fopen(out_name, "w");
    *out = f;
    free(out_name);
    if (f == NULL) {
        fprintf(stderr, "Could not open file \"%s
            \"\n", out_name);
        return -1;
    }

    return f == NULL ? -1 : 0;
}

int init_buf(struct line* l, size_t alloc)
{
    l->buf = malloc(alloc + 1);
    if (l->buf == NULL) {
        fprintf(stderr, "Could not allocate
            resources\n");
        return -1;
    }
    l->buf[alloc] = 0;
    l->alloc = alloc;
    l->err = 0;
    l->len = 0;
    return 0;
}

int free_buf(struct line *l)
{
    if (l->buf != NULL) {
        free(l->buf);
    }
    return 0;
}
```

Listing 18: common/io.h

```
/* -*- C -*-
 *
 * io.h
 *
 * Author: Benjamin T James
 */

#ifndef IO_H
#define IO_H
#include <stdio.h>
#include <stddef.h>

struct line {
    char *buf;
    int len;
    size_t alloc;
    int err;
};

int open_file(const char* src_file, const char*
    ext, FILE** out);
int init_buf(struct line* l, size_t alloc);
int free_buf(struct line* l);

int read_line(struct line* l, FILE *f);

#endif
```

Listing 19: common/defs.h

```
/* -*- C -*-
 *
 * defs.h
```

```
 *
 * Author: Benjamin T James
 */

#ifndef DEFS_H
#define DEFS_H

#define NOPRINT 1024

#define TOKEN_EOF 0
#define TOKEN_ID 1
#define TOKEN_ADDOP 2
#define TOKEN_MULOP 3
#define TOKEN_RELOP 4
#define TOKEN_NUM_REAL 5
#define TOKEN_NUM_INTEGER 6
#define TOKEN_SIGN 7

#define TOKEN_ARRAY 10
#define TOKEN_BEGIN 11
#define TOKEN_DO 12
#define TOKEN_ELSE 13
#define TOKEN_END 14
#define TOKEN_FUNCTION 15
#define TOKEN_IF 16
#define TOKEN_INTEGER 17
#define TOKEN_NOT 18
#define TOKEN_OF 19
#define TOKEN_PROGRAM 20
#define TOKEN_REAL 21
#define TOKEN_THEN 22
#define TOKEN_VAR 23
#define TOKEN_WHILE 24


#define TOKEN_PERIOD 25
#define TOKEN_ELLIPSIS 26
#define TOKEN_LPAREN 27
#define TOKEN_RPAREN 28
#define TOKEN_LBRACKET 29
#define TOKEN_RBRACKET 30
#define TOKEN_ASSIGN 31
#define TOKEN_COLON 32
#define TOKEN_SEMICOLON 33
#define TOKEN_COMMA 34



#define TOKEN_LT 41
#define TOKEN_LEQ 42
#define TOKEN_EQ 43
#define TOKEN_NEQ 44
#define TOKEN_GT 45
#define TOKEN_GEQ 46


#define LEXERR 99

#define TOKEN_WHITESPACE 1024
#define TOKEN_NEWLINE 1025


#define LEXERR_ID_TOO_LONG 1
#define LEXERR_UNREC_SYM 2
#define LEXERR_INT_TOO_LONG 3
#define LEXERR_MANTIS_TOO_LONG 4
#define LEXERR_FRAC_TOO_LONG 5
#define LEXERR_LEADING_ZERO 6
#define LEXERR_TRAILING_ZERO 7
#define LEXERR_LINE_TOO_LONG 8
#define LEXERR_EXP_TOO_LONG 9
#define LEXERR_NO_EXP 10
#define LEXERR_NO_FRAC 11



#define ID_STRLEN 10

#ifndef LINELEN
#define LINELEN 72
#endif

/* forward declarations */
typedef struct machine *machine_t;
typedef struct lex_state *lex_state_t;

#endif
```

Listing 20: common/util.c

```
/* -*- C -*-
 *
 * util.c
 *
 * Author: Benjamin T James
 */

#include "util.h"
#include <stdlib.h>
#include <string.h>

int get_file_without_ext(const char* f, char **
    to_write)
{
    char *loc, *buf;
        if (sdup(f, &buf) < 0) {
        return -1;
    }
    *to_write = buf;
    loc = strrchr(buf, '.');
    if (loc) {
        *loc = 0;
    } else {
        return -1;
    }
    return 0;
}

int get_out_file(const char* in_file, const char*
    extension, char **out_file)
{
    char *str, *buf;
```

```
    int i, ext_len, total_len;
    if (get_file_without_ext(in_file, &str) < 0) {
        fprintf(stderr, "File \"%s\" must have an
            extension\n", in_file);
        return -1;
    }
        i = strlen(str);
    ext_len = strlen(extension) + 1; /* for
        decimal */
        total_len = i + ext_len;
        buf = malloc(total_len + 1); /* for null
            terminator */
    if (buf == NULL) {
        fprintf(stderr, "Could not allocate
            resources\n");
        return -1;
    }
    buf[total_len] = 0;
    sprintf(buf, "%s.%s", str, extension);
    free(str);
    *out_file = buf;
    return 0;
}

int get_str(char *f, char *b, char **ret)
{
    int len = (f - b) + 1;
    char *buf = malloc(len + 1);
    if (buf == NULL) {
        fprintf(stderr, "Could not allocate
            resources\n");
        return -1;
    }
    memcpy(buf, b, len);
    buf[len] = 0;
    *ret = buf;
    return 0;
}

int sdup(const char* s, char **ret)
{
    int len = strlen(s);
    char *buf = malloc(len + 1);
    if (buf == NULL) {
        fprintf(stderr, "Could not allocate
            resources\n");
        return -1;
    }
    buf[len] = 0;
    memcpy(buf, s, len);
    *ret = buf;
    return 0;
}
```

Listing 21: common/util.h

```
/* -*- C -*-
 *
 * util.h
 *
 * Author: Benjamin T James
 */

#ifndef UTIL_H
#define UTIL_H
#include <stdio.h>


int get_out_file(const char* in_file, const char*
    extension, char **out_file);
int get_file_without_ext(const char* f, char **
    to_write);
int get_str(char *f, char *b, char **ret);

int sdup(const char* s, char **ret);

#endif
```

Listing 22: common/token.c

```
/* -*- C -*-
 *
 * token.c
 *
 * Author: Benjamin T James
 */

#include "token.h"
#include "defs.h"

int token_id(struct token *t, char *ptr)
{
    t->is_id = 1;
    t->type = TOKEN_ID;
    t->val.ptr = ptr;
    return 0;
}

int token_add(struct token *t, int type, int attr)
{
    t->is_id = 0;
    t->type = type;
    t->val.attr = attr;
    return 0;
}

int token_println(FILE* f, int line, const char *
    lexeme, struct token t)
{
    if (t.type & NOPRINT) {
        return 0;
    } else if (t.is_id) {
        return fprintf(f, "%d\t%s\t %d\t%p\n",
                line, lexeme, t.type, t.val.ptr);
    } else {
        return fprintf(f, "%d\t%s\t %d\t%d\n",
                line, lexeme, t.type, t.val.attr)
                    ;
    }
}
```

```
char *token2str(int token)
{
    switch (token) {
    case TOKEN_ADDOP: return "ADDOP";
    case TOKEN_ARRAY: return "array";
    case TOKEN_ASSIGN: return ":=";
    case TOKEN_BEGIN: return "begin";
    case TOKEN_COLON: return ":";
    case TOKEN_COMMA: return ",";
    case TOKEN_DO: return "do";
    case TOKEN_ELLIPSIS: return "..";
    case TOKEN_ELSE: return "else";
    case TOKEN_END: return "end";
    case TOKEN_EOF: return "EOF";
    case TOKEN_FUNCTION: return "function";
    case TOKEN_ID: return "identifier";
    case TOKEN_IF: return "if";
    case TOKEN_INTEGER: return "integer";
    case TOKEN_LBRACKET: return "[";
    case TOKEN_LPAREN: return "(";
    case TOKEN_MULOP: return "MULOP";
    case TOKEN_NOT: return "not";
    case TOKEN_NUM_INTEGER: return "NUM_INTEGER";
    case TOKEN_NUM_REAL: return "NUM_REAL";
    case TOKEN_OF: return "of";
    case TOKEN_PERIOD: return ".";
    case TOKEN_PROGRAM: return "program";
    case TOKEN_RBRACKET: return "]";
    case TOKEN_REAL: return "real";
    case TOKEN_RELOP: return "RELOP";
    case TOKEN_RPAREN: return ")";
    case TOKEN_SEMICOLON: return ";";
    case TOKEN_SIGN: return "+ or -";
    case TOKEN_THEN: return "then";
    case TOKEN_VAR: return "var";
    case TOKEN_WHILE: return "while";
    case LEXERR: return "LEXERR";
    }
    return "UNKNOWN";
}
```

Listing 23: common/token.h

```
/* -*- C -*-
 *
 * token.h
 *
 * Author: Benjamin T James
 */

#ifndef TOKEN_H
#define TOKEN_H

#include <stdio.h>

union tok_val {
    int attr;
    void *ptr;
};

struct token {
    int type;
/* char *lex; */
    unsigned is_id : 1;
    union tok_val val;
};

int token_id(struct token *t, char *ptr);
int token_add(struct token *t, int type, int attr)
    ;
int token_println(FILE *f, int line, const char *
    lexeme, struct token t);
char* token2str(int token);
#endif
```

Listing 24: common/idres.c

```
/* -*- C -*-
 *
 * idres.c
 *
 * Author: Benjamin T James
 */

#include "idres.h"
#include "defs.h"
#include "util.h"
#include <math.h>
#include <stdlib.h>
#include <string.h>

int idres_print(FILE* f, struct idres **list)
{
    struct idres *node = *list;
    while (node != NULL) {
        fprintf(f, "%p\t%s\n", node->token.val.ptr
            , node->lexeme);
        node = node->next;
    }
    return 0;
}

int idres_insert(struct idres **list, char* lexeme
    , struct token token)
{
    int ret = 0;
    struct idres *root = malloc(sizeof(*root));
    root->lexeme = lexeme;
    root->type = 0;
    root->token = token;
    root->next = *list;
    *list = root;
    return ret;
}
int idres_add_rw(struct idres **list, char*
    c_lexeme, int type, int attr)
{
    char *lexeme;
    struct token tok;
    if (sdup(c_lexeme, &lexeme) < 0) {
        return -1;
    }
```

```
    token_add(&tok, type, attr);
    return idres_insert(list, lexeme, tok);
}

int idres_add_id(struct idres **list, char* c_lexeme)
{
    char *lexeme;
    struct token tok;
    if (sdup(c_lexeme, &lexeme) < 0) {
        return -1;
    }
    token_id(&tok, lexeme);
    return idres_insert(list, lexeme, tok);
}

int idres_add_id_attr(struct idres **list, char* c_lexeme, char* attr)
{
    char *lexeme;
    struct token tok;
    if (sdup(c_lexeme, &lexeme) < 0) {
        return -1;
    }
    token_id(&tok, lexeme);
    tok.val.ptr = attr;
    return idres_insert(list, lexeme, tok);
}

int idres_lookup(struct idres **list, void* ptr, struct idres **ret)
{
    struct idres *node = *list;
    while (node != NULL) {
        if (ptr == node->token.val.ptr) {
            *ret = node;
            return 0;
        }
        node = node->next;
    }
    return -1;
}
int idres_find(struct idres *node, char *lexeme, struct idres **ret)
{
    while (node != NULL) {
        if (!strcmp(lexeme, node->lexeme)) {
            *ret = node;
            return 0;
        }
        node = node->next;
    }
    return -1;
}

int idres_search(struct idres **list, char* lexeme, struct idres **ret)
{
    return idres_find(*list, lexeme, ret);
}

int idres_clean(struct idres **list)
{
    while (*list != NULL) {
        struct idres *prev = *list;
        *list = prev->next;
        free(prev->lexeme);
        free(prev);
    }
    return 0;
}

int idres_read(const char *filename, struct idres **list)
{
    FILE* f = fopen(filename, "r");
    void* addr = NULL;
    long count;
    char *lexeme = malloc(ID_STRLEN + 1);
    /* strlen(lexeme) guaranteed to be ID_STRLEN */
    for (count = 0; fscanf(f, "0x%p\t%s\n", &addr, lexeme) == 2; count++) {
        idres_add_id_attr(list, lexeme, addr);
    }
    free(lexeme);
    fclose(f);
    /*return idres_balance(list, count);*/
    return 0;
}
```

Listing 25: common/idres.h

```
/* -*- C -*-
 *
 * idres.h
 *
 * Author: Benjamin T James
 */

#ifndef IDRES_H
#define IDRES_H

#include <stdlib.h>
#include "token.h"
#include "io.h"

struct idres {
    char *lexeme;
    int type;
    struct token token;
    struct idres *next;
};

int idres_add_rw(struct idres **list, char* lexeme, int token, int attr);
int idres_add_id(struct idres **list, char* lexeme);

int idres_search(struct idres **list, char* lexeme, struct idres **ret);
```

```
int idres_lookup(struct idres **list, void* ptr,
    struct idres **ret);
int idres_clean(struct idres **list);
int idres_print(FILE* f, struct idres **list);

int idres_read(const char *filename, struct idres
    **list);

int idres_add_id_attr(struct idres **list, char*
    lexeme, char* attr);
#endif
```

Listing 26: lexer/main.c

```
/* -*- C -*-
 *
 * main.c
 *
 * Author: Benjamin T James
 */

#include "state.h"
#include "defs.h"
#include "lexerr.h"

int handle_line(struct lex_state *s, int line_no)
{
    char *lexeme = NULL;
    struct machine state;
    state.f = s->buf.buf;
    state.b = state.f;
    state.tok.type = 0;
    if (s->buf.err == LEXERR_LINE_TOO_LONG) {
        print_error(s->list, s->buf.err, s->buf.
            buf);
    }
    while (state.tok.type != TOKEN_NEWLINE) {
        int ret = machine_iter(s, &state, &lexeme)
            ;
        if (ret < 0) {
            fprintf(stderr, "Machine not found\n");
            return -1;
        }
        if (state.tok.type == LEXERR) {
            print_error(s->list, state.tok.val.attr
                , lexeme);
        }
        token_println(s->token, line_no, lexeme,
            state.tok);
        free(lexeme);
        lexeme = NULL;
    }
    return 0;
}

int main(int argc, char **argv)
{
    struct lex_state s;
    struct token tok_eof;
    int line = 1;
    if (argc != 3) {
        fprintf(stderr, "Usage: %s source
            reservedWordFile\n", *argv);
        return -1;
    }

    if (state_init(argv[1], argv[2], LINELEN, &s)
        < 0) {
        return -1;
    }

    while (read_line(&s.buf, s.source) == 0) {
        fprintf(s.list, "%d\t%s", line, s.buf.buf)
            ;

        handle_line(&s, line);

        line++;
    }
    token_add(&tok_eof, TOKEN_EOF, 0);
    token_println(s.token, line, "EOF", tok_eof);
    idres_print(s.sym, &s.ids);
    state_cleanup(&s);
    return 0;
}
```

Listing 27: lexer/state.c

```
/* -*- C -*-
 *
 * state.c
 *
 * Author: Benjamin T James
 */

#include "state.h"
#include "util.h"

int resword_init(struct lex_state *st)
{
    int tok, attr;
    char *lexeme = malloc(st->buf.alloc);
    if (lexeme == NULL) {
        fprintf(stderr, "Could not allocate memory
            \n");
        return -1;
    }
    while (fscanf(st->res_word, "%s\t%d\t%d\n",
        lexeme, &tok, &attr) != EOF) {
        idres_add_rw(&st->rwords, lexeme, tok,
            attr);
    }
    free(lexeme);
    return 0;
}
int state_init(const char *source, const char *
    res_word,
           int line_len, struct lex_state *st)
{
    if (init_buf(&st->buf, line_len) < 0) {
        return -1;
    }
```

```
    if (open_file(source, "list", &st->list) < 0) {
        return -1;
    }
    if (open_file(source, "tok", &st->token) < 0) {
        return -1;
    }
    if (open_file(source, "sym", &st->sym) < 0) {
        return -1;
    }
    st->res_word = fopen(res_word, "r");
    if (st->res_word == NULL) {
        fprintf(stderr, "Could not open file \"%s\"\n", res_word);
        return -1;
    }
    st->source = fopen(source, "r");
    if (st->source == NULL) {
        fprintf(stderr, "Could not open file \"%s\"\n", source);
        return -1;
    }
    st->rwords = NULL;
    st->ids = NULL;
    if (resword_init(st) < 0) {
        return -1;
    }
    st->machines = NULL;
    if (machine_init(&st->machines) < 0) {
        return -1;
    }
    return 0;
}

int state_cleanup(struct lex_state *s)
{
    free_buf(&s->buf);
    fclose(s->source);
    fclose(s->res_word);
    fclose(s->sym);
    fclose(s->list);
    fclose(s->token);
    idres_clean(&s->rwords);
    idres_clean(&s->ids);
    machine_clean(&s->machines);
    return 0;
}
```

Listing 28: lexer/state.h

```
/* -*- C -*-
 *
 * state.h
 *
 * Author: Benjamin T James
 */

#ifndef STATE_H
#define STATE_H

#include <stdio.h>
#include "defs.h"
#include "io.h"
#include "idres.h"
#include "machine.h"

struct lex_state {
    /* inputs */
    FILE* source;
    FILE* res_word;

    /* outputs */
    FILE* sym;
    FILE* list;
    FILE* token;

    /* lexer state */
    struct line buf;
    struct idres *rwords;
    struct idres *ids;
        machine_t machines;
};

int state_init(const char *source, const char *
    res_word,
          int line_len, struct lex_state *st);
int resword_init(struct lex_state *s);
int state_cleanup(struct lex_state *s);
#endif
```

Listing 29: lexer/fsm.c

```
/* -*- C -*-
 *
 * fsm.c
 *
 * Author: Benjamin T James
 */

#include <ctype.h>
#include <string.h>
#include "fsm.h"
#include "defs.h"
#include "util.h"

int digit_plus(struct machine *m)
{
    int len = 1;
    if (!isdigit(*m->f)) {
        return 0;
    }
    m->f++;
    while (isdigit(*m->f)) {
        m->f++;
        len++;
    }
    return len;
}

int fsm_relop(struct machine *m, struct lex_state
    *ls)
```

```
{
    if (*m->f == '>') {
        m->f++;
        if (*m->f == '=') {
            token_add(&m->tok, TOKEN_RELOP,
                TOKEN_GEQ);
        } else {
            m->f--;
            token_add(&m->tok, TOKEN_RELOP,
                TOKEN_GT);
        }
    } else if (*m->f == '=') {
        token_add(&m->tok, TOKEN_RELOP, TOKEN_EQ);
    } else if (*m->f == '<') {
        m->f++;
        if (*m->f == '=') {
            token_add(&m->tok, TOKEN_RELOP,
                TOKEN_LEQ);
        } else if (*m->f == '>') {
            token_add(&m->tok, TOKEN_RELOP,
                TOKEN_NEQ);
        } else {
            m->f--;
            token_add(&m->tok, TOKEN_RELOP,
                TOKEN_LT);
        }
    } else {
        return 0;
    }
    return 1;
}
int fsm_integer(struct machine *m, struct
    lex_state *ls)
{
    char *lexeme;
    int result, len;
    if (!digit_plus(m)) {
        return 0;
    }
    m->f--;
    if (get_str(m->f, m->b, &lexeme) < 0) {
        return -1;
    }
    len = strlen(lexeme);
    if (len > 10) {
        token_add(&m->tok, LEXERR,
            LEXERR_INT_TOO_LONG);
    } else if ((lexeme[0] == '0' && len > 1)
          || (lexeme[0] == '-' && lexeme[1] == '0'
              && len > 2)) {
        token_add(&m->tok, LEXERR,
            LEXERR_LEADING_ZERO);
    } else {
        result = strtol(lexeme, NULL, 10);
        token_add(&m->tok, TOKEN_NUM_INTEGER,
            result);
    }
    free(lexeme);
    return 1;
}

int fsm_real(struct machine *m, struct lex_state *
    ls)
{
    char *lexeme;
    double result;
    int len, mantis_len = 0, frac_len = 0;
    mantis_len = digit_plus(m);
    if (mantis_len == 0 || *m->f != '.') {
        return 0;
    }
    m->f++;
    frac_len = digit_plus(m);
    if (frac_len == 0) {
        return 0;
    }
    m->f--;
    if (get_str(m->f, m->b, &lexeme) < 0) {
        return -1;
    }

    len = strlen(lexeme);
    if (mantis_len > 5) {
        token_add(&m->tok, LEXERR,
            LEXERR_MANTIS_TOO_LONG);
    } else if (frac_len > 5) {
        token_add(&m->tok, LEXERR,
            LEXERR_FRAC_TOO_LONG);
    } else if (lexeme[0] == '0' || (lexeme[0] == '
        -' && lexeme[1] == '0')) {
        token_add(&m->tok, LEXERR,
            LEXERR_LEADING_ZERO);
    } else if (lexeme[len-1] == '0') {
        token_add(&m->tok, LEXERR,
            LEXERR_TRAILING_ZERO);
    } else {
        result = strtod(lexeme, NULL);
        token_add(&m->tok, TOKEN_NUM_REAL, (int)
            result);
    }
    free(lexeme);
    return 1;
}
int fsm_long_real(struct machine *m, struct
    lex_state *ls)
{
    char *lexeme;
    double result;
    int len, tz = 0, mantis_len = 0, frac_len = 0,
         exp_len = 0;
    mantis_len = digit_plus(m);
    if (mantis_len == 0 || *m->f != '.') {
        return 0;
    }
    m->f++;
    frac_len = digit_plus(m);
    if (/* frac_len == 0 || */*m->f != 'E') {
        return 0;
    }
    if (get_str(m->f - 1, m->b, &lexeme) < 0) {
```

```
        return -1;
    }
    if (lexeme[strlen(lexeme) - 1] == '0') {
        tz = 1; /* set trailing zero flag to 1 */
    }

    free(lexeme);
    m->f++;
    if (*m->f == '-' || *m->f == '+') {
        m->f++;
    }
    exp_len = digit_plus(m);
    /* if (exp_len == 0) { */
    /*  /\* err *\/ */
    /*  return 0; */
    /* } */
    m->f--;
    if (get_str(m->f, m->b, &lexeme) < 0) {
        return -1;
    }
    len = strlen(lexeme);
    if (frac_len == 0) {
        token_add(&m->tok, LEXERR, LEXERR_NO_FRAC);
    } else if (exp_len == 0) {
        token_add(&m->tok, LEXERR, LEXERR_NO_EXP);
    } else if (mantis_len > 5) {
        token_add(&m->tok, LEXERR,
            LEXERR_MANTIS_TOO_LONG);
    } else if (frac_len > 5) {
        token_add(&m->tok, LEXERR,
            LEXERR_FRAC_TOO_LONG);
    } else if (exp_len > 2) {
        token_add(&m->tok, LEXERR,
            LEXERR_EXP_TOO_LONG);
    } else if (lexeme[0] == '0' || (lexeme[0] == '
        -' && lexeme[1] == '0')) {
        token_add(&m->tok, LEXERR,
            LEXERR_LEADING_ZERO);
    } else if (tz || lexeme[len-1] == '0') {
        token_add(&m->tok, LEXERR,
            LEXERR_TRAILING_ZERO);
    } else {
        result = strtod(lexeme, NULL);
        token_add(&m->tok, TOKEN_NUM_REAL, (int)
            result);
    }
    free(lexeme);
    return 1;
}

int fsm_addop(struct machine *m, struct lex_state
    *ls)
{
    if (*m->f == '+') {
        token_add(&m->tok, TOKEN_SIGN, '+');
        return 1;
    } else if (*m->f == '-') {
        token_add(&m->tok, TOKEN_SIGN, '-');
        return 1;
    } else if (*m->f == 'o') {
        m->f++;
        if (*m->f == 'r') {
            token_add(&m->tok, TOKEN_ADDOP, '|');
            return 1;
        }
    }
    return 0;
}
int fsm_mulop(struct machine *m, struct lex_state
    *ls)
{
    if (*m->f == '*') {
        token_add(&m->tok, TOKEN_MULOP, '*');
        return 1;
    } else if (*m->f == '/') {
        token_add(&m->tok, TOKEN_MULOP, '/');
        return 1;
    } else if (*m->f == 'd') {
        m->f++;
        if (*m->f == 'i' && m->f++ && *m->f == 'v'
            ) {
            token_add(&m->tok, TOKEN_MULOP, '/');
            return 1;
        }
    } else if (*m->f == 'm') {
        m->f++;
        if (*m->f == 'o' && m->f++ && *m->f == 'd'
            ) {
            token_add(&m->tok, TOKEN_MULOP, '%');
            return 1;
        }
    } else if (*m->f == 'a') {
        m->f++;
        if (*m->f == 'n' && m->f++ && *m->f == 'd'
            ) {
            token_add(&m->tok, TOKEN_MULOP, '&');
            return 1;
        }
    }
    return 0;
}

int fsm_catchall(struct machine *m, struct
    lex_state *ls)
{
    switch (*m->f) {
    case '[':
        token_add(&m->tok, TOKEN_LBRACKET, *m->f);
        return 1;
    case ']':
        token_add(&m->tok, TOKEN_RBRACKET, *m->f);
        return 1;
    case '(':
        token_add(&m->tok, TOKEN_LPAREN, *m->f);
        return 1;
    case ')':
        token_add(&m->tok, TOKEN_RPAREN, *m->f);
        return 1;
    case ',':
```

```
        token_add(&m->tok, TOKEN_COMMA, *m->f);
        return 1;
    case ';':
        token_add(&m->tok, TOKEN_SEMICOLON, *m->f);
        return 1;
    default:
        break;
    }
    if (*m->f == '.') {
        m->f++;
        if (*m->f == '.') {
            token_add(&m->tok, TOKEN_ELLIPSIS,
                TOKEN_ELLIPSIS);
        } else {
            m->f--;
            token_add(&m->tok, TOKEN_PERIOD, '.');
        }
        return 1;
    } else if (*m->f == ':') {
        m->f++;
        if (*m->f == '=') {
            token_add(&m->tok, TOKEN_ASSIGN, 0);
        } else {
            m->f--;
            token_add(&m->tok, TOKEN_COLON, ':');
        }
        return 1;
    }
    return 0;
}

int fsm_idres(struct machine *m, struct lex_state
    *ls)
{
    if (isalpha(*m->f)) {
        int len;
        char *lexeme;
        struct idres *result;
        m->f++;
        while (isalnum(*m->f)) {
            m->f++;
        }
        m->f--;



        if (get_str(m->f, m->b, &lexeme) < 0) {
            return -1;
        }
        len = strlen(lexeme);
        if (len > ID_STRLEN) {
            m->tok.type = LEXERR;
            m->tok.is_id = 0;
            m->tok.val.attr = LEXERR_ID_TOO_LONG;
        } else if (idres_search(&ls->rwords,
            lexeme, &result) == 0) {
            m->tok = result->token;
        } else if (idres_search(&ls->ids, lexeme,
            &result) == 0) {
            m->tok = result->token;
        } else {
            idres_add_id(&ls->ids, lexeme);
            m->tok = ls->ids->token;
        }
        free(lexeme);
        return 1;
    }
    return 0;
}
int fsm_unrecognized_symbol(struct machine *m,
    struct lex_state *ls)
{
    m->tok.type = LEXERR;
    m->tok.val.attr = LEXERR_UNREC_SYM;
    return 1;
}
int fsm_newline(struct machine *m, struct
    lex_state *ls)
{
    if (*m->f == '\n') {
        m->f++;
        m->tok.type = TOKEN_NEWLINE;
        return 1;
    }
    return 0;
}
int fsm_whitespace(struct machine *m, struct
    lex_state *ls)
{
    if (*m->f == ' ' || *m->f == '\t') {
        m->f++;
        while (*m->f == ' ' || *m->f == '\t') {
            m->f++;
        }
        m->f--;
        m->tok.type = TOKEN_WHITESPACE;
        return 1;
    }
    return 0;

}
```

Listing 30: lexer/fsm.h

```
/* -*- C -*-
 *
 * fsm.h
 *
 * Author: Benjamin T James
 */

#ifndef FSM_H
#define FSM_H

#include "machine.h"
#include "state.h"

int fsm_unrecognized_symbol(struct machine *m,
    struct lex_state *ls);
int fsm_whitespace(struct machine *m, struct
    lex_state *ls);
```

```
int fsm_newline(struct machine *m, struct
    lex_state *ls);
int fsm_idres(struct machine *m, struct lex_state
    *ls);
int fsm_relop(struct machine *m, struct lex_state
    *ls);
int fsm_addop(struct machine *m, struct lex_state
    *ls);
int fsm_mulop(struct machine *m, struct lex_state
    *ls);
int fsm_catchall(struct machine *m, struct
    lex_state *ls);
int fsm_integer(struct machine *m, struct
    lex_state *ls);
int fsm_real(struct machine *m, struct lex_state *
    ls);
int fsm_long_real(struct machine *m, struct
    lex_state *ls);
#endif
```

Listing 31: lexer/lexerr.c

```
/* -*- C -*-
 *
 * lexerr.c
 *
 * Author: Benjamin T James
 */

#include "lexerr.h"

int print_error(FILE* listing, int err, const char
     *lexeme)
{
    fprintf(listing, "LEXERR:\t");
    switch (err) {
    case LEXERR_ID_TOO_LONG:
        fprintf(listing, "ID too long:");
        break;
    case LEXERR_UNREC_SYM:
        fprintf(listing, "Unrecognized symbol:");
        break;
    case LEXERR_INT_TOO_LONG:
        fprintf(listing, "Int too long:");
        break;
    case LEXERR_MANTIS_TOO_LONG:
        fprintf(listing, "Mantissa too long:");
        break;
    case LEXERR_FRAC_TOO_LONG:
        fprintf(listing, "Fraction too long:");
        break;
    case LEXERR_LEADING_ZERO:
        fprintf(listing, "Leading zero:");
        break;
    case LEXERR_TRAILING_ZERO:
        fprintf(listing, "Trailing zero:");
        break;
    case LEXERR_LINE_TOO_LONG:
        fprintf(listing, "Line too long:");
        break;
    case LEXERR_EXP_TOO_LONG:
        fprintf(listing, "Exponent too long:");
        break;
    case LEXERR_NO_EXP:
        fprintf(listing, "No exponent:");
        break;
    case LEXERR_NO_FRAC:
        fprintf(listing, "No fractional part:");
        break;
    default:
        fprintf(listing, "Unknown error %d:", err)
            ;
    }
    fprintf(listing, "\t\t%s\n", lexeme);
    return 0;
}
```

Listing 32: lexer/lexerr.h

```
/* -*- C -*-
 *
 * lexerr.h
 *
 * Author: Benjamin T James
 */

#ifndef LEXERR_H
#define LEXERR_H

#include "machine.h"

int print_error(FILE* listing, int err, const char
     *lexeme);

#endif
```

Listing 33: lexer/machine.c

```
/* -*- C -*-
 *
 * machine.c
 *
 * Author: Benjamin T James
 */

#include "machine.h"
#include "fsm.h"
#include "util.h"

int machine_iter(struct lex_state *ls, struct
    machine *state, char **out_str)
{
    int ret;
    struct machine *m = ls->machines;
    for (; m != NULL; m = m->next) {
        m->b = state->f;
        m->f = m->b;
        m->tok.is_id = 0;
            ret = m->call(m, ls);
        if (ret == 1) {
            state->tok = m->tok;
            state->f = m->f + 1;
```

```
            state->b = state->f;

            return get_str(m->f, m->b, out_str);
        } else if (ret == -1) {
            return -1;
        }
    }
    return -1;
}
int machine_init(struct machine **list)
{
    machine_add(list, fsm_unrecognized_symbol);
    machine_add(list, fsm_newline);

    machine_add(list, fsm_catchall);
    machine_add(list, fsm_relop);

    machine_add(list, fsm_integer);
    machine_add(list, fsm_real);
    machine_add(list, fsm_long_real);

    machine_add(list, fsm_idres);
    machine_add(list, fsm_addop);
    machine_add(list, fsm_mulop);

    machine_add(list, fsm_whitespace);
    return 0;
}
int machine_add(struct machine **list,
        int (*func)(struct machine *m, struct
            lex_state *ls))
{
    struct machine *m = malloc(sizeof(*m));
    if (m == NULL) {
        fprintf(stderr, "Unable to allocate
            resources\n");
        return -1;
    }
    m->call = func;
    m->next = *list;
    *list = m;
    return 0;
}
int machine_clean(struct machine **list)
{
    struct machine *head = *list;
    struct machine *tmp;
    while (head != NULL) {
        tmp = head;
        head = head->next;
        free(tmp);
    }
    return 0;
}
```

Listing 34: lexer/machine.h

```
/* -*- C -*-
 *
 * machine.h
 *
 * Author: Benjamin T James
 */

#ifndef MACHINE_H
#define MACHINE_H

#include "defs.h"
#include "token.h"
#include "state.h"

struct machine {
    /* returns 1 on success, 0 on failure */
    int (*call)(struct machine *m, lex_state_t ls)
        ;

    char *f;
    char *b;
    struct token tok;
    struct machine *next;
};

/* interface for the lexer */
int machine_iter(lex_state_t ls, struct machine *
    state, char **out_str);

int machine_init(struct machine **list);
int machine_add(struct machine **list,
        int (*func)(struct machine *m, struct
            lex_state *ls));

int machine_clean(struct machine **list);

#endif
```