

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 resetting 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 \pm 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

```
1  abcdefghij
2  abcdefghijk
3  @
4
5  12345678901
6  1234567890
7
8  12345.3
9  123456.3
10
11 1.12345
12 1.123456
13
14
15 123
16 0123
17 01.2
18 01.2E2
19
20 1230
21 1.2
22 1.20
23 1.20E-12
24
25 1.2E-10
26 1.2E-123
27 1.2E+5
28 1.2E+123
29
30
31 e#
32 3.4E+;
33 3.E4+;
34 34E+-;
35 E3.4+;
36
37 abcdefghijklmnopqrstuvwxyz0123456789abcdefghijklmnopqrstuvwxyz012345678
38 abcdefghijklmnopqrstuvwxyz0123456789abcdefghijklmnopqrstuvwxyz0123456789
```

Listing 2: bad_lex.list

```
1 1      abcdefghij
2 2      abcdefghijk
3 LEXERR: ID too long:      abcdefghijk
4 3      @
5 LEXERR: Unrecognized symbol:      @
6 4
7 5      12345678901
8 LEXERR: Int too long:      12345678901
9 6      1234567890
10 7
11 8      12345.3
12 9      123456.3
13 LEXERR: Mantissa too long:      123456.3
14 10
```

```

15 11      1.12345
16 12      1.123456
17 LEXERR: Fraction too long:          1.123456
18 13
19 14
20 15      123
21 16      0123
22 LEXERR: Leading zero:              0123
23 17      01.2
24 LEXERR: Leading zero:              01.2
25 18      01.2E2
26 LEXERR: Leading zero:              01.2E2
27 19
28 20      1230
29 21      1.2
30 22      1.20
31 LEXERR: Trailing zero:             1.20
32 23      1.20E-12
33 LEXERR: Trailing zero:             1.20E-12
34 24
35 25      1.2E-10
36 LEXERR: Trailing zero:             1.2E-10
37 26      1.2E-123
38 LEXERR: Exponent too long:         1.2E-123
39 27      1.2E+5
40 28      1.2E+123
41 LEXERR: Exponent too long:         1.2E+123
42 29
43 30
44 31      e#
45 LEXERR: Unrecognized symbol:        #
46 32      3.4E+;
47 LEXERR: No exponent:               3.4E+
48 33      3.E4+;
49 LEXERR: No fractional part:         3.E4
50 34      34E+-;
51 35      E3.4+;
52 36
53 37      abcdefghijklmnopqrstuvwxyz0123456789abcdefghijklmnopqrstuvwxyz012345678
54 LEXERR: ID too long:
      abcdefghijklmnopqrstuvwxyz0123456789abcdefghijklmnopqrstuvwxyz012345678
55 38
56 LEXERR: Line too long:

```

Listing 3: bad_lex.sym

```

1 0x55442b0      E3
2 0x5544010      E
3 0x5543a00      e
4 0x55410f0      abcdefghij

```

Listing 4: bad_lex.tok

```

1 1      abcdefghij      1      0x55410f0
2 2      abcdefghijk     99      1
3 3      @              99      2
4 5      12345678901     99      3
5 6      1234567890      6       1234567890
6 8      12345.3        5       12345
7 9      123456.3       99      4
8 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

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

```

38	4	real	21	0
39	4	;	33	59
40	5	var	23	0
41	5	c	1	0x5544350
42	5	:	32	58
43	5	integer	17	0
44	5	;	33	59
45	6	function	15	0
46	6	f2	1	0x5544820
47	6	(27	40
48	6	p	1	0x55449d0
49	6	:	32	58
50	6	integer	17	0
51	6)	28	41
52	6	:	32	58
53	6	integer	17	0
54	6	;	33	59
55	7	var	23	0
56	7	d	1	0x55450d0
57	7	:	32	58
58	7	real	21	0
59	7	;	33	59
60	8	begin	11	0
61	9	f2	1	0x5544820
62	9	:=	31	0
63	9	5	6	5
64	9	+	7	43
65	9	p	1	0x55449d0
66	9	;	33	59
67	10	if	16	0
68	10	p	1	0x55449d0
69	10	>	4	45
70	10	d	1	0x55450d0
71	10	then	22	0
72	11	if	16	0
73	11	p	1	0x55449d0
74	11	=	4	43
75	11	5	6	5
76	11	then	22	0
77	12	f2	1	0x5544820
78	12	:=	31	0
79	12	f	1	0x5546590
80	12	+	7	43
81	12	10	6	10
82	13	else	13	0
83	14	f2	1	0x5544820
84	14	:=	31	0
85	14	100	6	100
86	15	end	14	0
87	16	begin	11	0
88	17	f1	1	0x5543710
89	17	:=	31	0
90	17	f2	1	0x5544820
91	17	(27	40
92	17	a	1	0x5542960
93	17)	28	41
94	17	*	3	42
95	17	x	1	0x5543b90
96	18	end	14	0
97	19	function	15	0

98	19	f2	1	0x5544820
99	19	(27	40
100	19	a	1	0x5542960
101	19	:	32	58
102	19	real	21	0
103	19)	28	41
104	19	:	32	58
105	19	real	21	0
106	19	;	33	59
107	20	var	23	0
108	20	b	1	0x5542e30
109	20	:	32	58
110	20	integer	17	0
111	20	;	33	59
112	21	begin	11	0
113	22	f3	1	0x5548310
114	22	:=	31	0
115	22	10	6	10
116	22	*	3	42
117	22	f1	1	0x5543710
118	22	(27	40
119	22	a	1	0x5542960
120	22	,	34	44
121	22	2.3	5	2
122	22)	28	41
123	22	;	33	59
124	23	end	14	0
125	24	begin	11	0
126	25	f1	1	0x5543710
127	25	(27	40
128	25	5	6	5
129	25	,	34	44
130	25	3.2	5	3
131	25)	28	41
132	26	end	14	0
133	26	.	25	46
134	27	begin	11	0
135	29	end	14	0
136	30	EOF	0	0

gcd

Listing 9: gcd.pas

```
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.
```

Listing 10: gcd.list

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

Listing 11: gcd.sym

1	0x55455f0	write
2	0x5544f90	read
3	0x5544d40	out
4	0x55435b0	b
5	0x55432c0	a
6	0x5543110	gcd
7	0x5542ce0	y
8	0x55428b0	x
9	0x5542570	output
10	0x5542370	input
11	0x55421c0	example

Listing 12: gcd.tok

1	1	program	20	0
2	1	example	1	0x55421c0
3	1	(27	40
4	1	input	1	0x5542370
5	1	,	34	44
6	1	output	1	0x5542570
7	1)	28	41
8	1	;	33	59
9	2	var	23	0
10	2	x	1	0x55428b0
11	2	:	32	58
12	2	integer	17	0

13	2	;	33	59
14	2	var	23	0
15	2	y	1	0x5542ce0
16	2	:	32	58
17	2	integer	17	0
18	2	;	33	59
19	3	function	15	0
20	3	gcd	1	0x5543110
21	3	(27	40
22	3	a	1	0x55432c0
23	3	:	32	58
24	3	integer	17	0
25	3	;	33	59
26	3	b	1	0x55435b0
27	3	:	32	58
28	3	integer	17	0
29	3)	28	41
30	3	:	32	58
31	3	integer	17	0
32	3	;	33	59
33	4	begin	11	0
34	5	if	16	0
35	5	b	1	0x55435b0
36	5	=	4	43
37	5	0	6	0
38	5	then	22	0
39	5	gcd	1	0x5543110
40	5	:=	31	0
41	5	a	1	0x55432c0
42	6	else	13	0
43	6	gcd	1	0x5543110
44	6	:=	31	0
45	6	gcd	1	0x5543110
46	6	(27	40
47	6	b	1	0x55435b0
48	6	,	34	44
49	6	a	1	0x55432c0
50	6	mod	3	37
51	6	b	1	0x55435b0
52	6)	28	41
53	7	end	14	0
54	7	;	33	59
55	9	begin	11	0
56	10	out	1	0x5544d40
57	10	:=	31	0
58	10	read	1	0x5544f90
59	10	(27	40
60	10	x	1	0x55428b0
61	10	,	34	44
62	10	y	1	0x5542ce0
63	10)	28	41
64	10	;	33	59
65	11	out	1	0x5544d40
66	11	:=	31	0
67	11	write	1	0x55455f0
68	11	(27	40
69	11	gcd	1	0x5543110
70	11	(27	40
71	11	x	1	0x55428b0
72	11	,	34	44

73	11	y	1	0x5542ce0
74	11)	28	41
75	11)	28	41
76	12	end	14	0
77	12	.	25	46
78	14	EOF	0	0

Listing 13: test.pas

```

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.
```

Listing 14: test.list

```

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

Listing 15: test.sym

1	0x55479d0	out
2	0x5546420	f3
3	0x5544fb0	d
4	0x55448b0	p
5	0x5544700	f2
6	0x5544230	c
7	0x5543b30	x
8	0x55436b0	f1

9	0x5542dd0	b
10	0x5542900	a
11	0x5542570	output
12	0x5542370	input
13	0x55421c0	test

Listing 16: test.tok

1	1	program	20	0
2	1	test	1	0x55421c0
3	1	(27	40
4	1	input	1	0x5542370
5	1	,	34	44
6	1	output	1	0x5542570
7	1)	28	41
8	1	;	33	59
9	2	var	23	0
10	2	a	1	0x5542900
11	2	:	32	58
12	2	integer	17	0
13	2	;	33	59
14	3	var	23	0
15	3	b	1	0x5542dd0
16	3	:	32	58
17	3	array	10	0
18	3	[29	91
19	3	1	6	1
20	3	..	26	26
21	3	5	6	5
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	0x55436b0
28	4	(27	40
29	4	a	1	0x5542900
30	4	:	32	58
31	4	integer	17	0
32	4	;	33	59
33	4	x	1	0x5543b30
34	4	:	32	58
35	4	real	21	0
36	4)	28	41
37	4	:	32	58
38	4	real	21	0
39	4	;	33	59
40	5	var	23	0
41	5	c	1	0x5544230
42	5	:	32	58
43	5	integer	17	0
44	5	;	33	59
45	6	function	15	0
46	6	f2	1	0x5544700
47	6	(27	40
48	6	p	1	0x55448b0
49	6	:	32	58
50	6	integer	17	0
51	6)	28	41
52	6	:	32	58
53	6	integer	17	0

54	6	;	33	59
55	7	var	23	0
56	7	d	1	0x5544fb0
57	7	:	32	58
58	7	real	21	0
59	7	;	33	59
60	8	begin	11	0
61	9	f2	1	0x5544700
62	9	:=	31	0
63	9	5	6	5
64	9	+	7	43
65	9	p	1	0x55448b0
66	10	end	14	0
67	10	;	33	59
68	11	begin	11	0
69	12	f1	1	0x55436b0
70	12	:=	31	0
71	12	f2	1	0x5544700
72	12	(27	40
73	12	a	1	0x5542900
74	12)	28	41
75	12	*	3	42
76	12	x	1	0x5543b30
77	13	end	14	0
78	13	;	33	59
79	14	function	15	0
80	14	f3	1	0x5546420
81	14	(27	40
82	14	a	1	0x5542900
83	14	:	32	58
84	14	real	21	0
85	14)	28	41
86	14	:	32	58
87	14	real	21	0
88	14	;	33	59
89	15	var	23	0
90	15	b	1	0x5542dd0
91	15	:	32	58
92	15	integer	17	0
93	15	;	33	59
94	16	begin	11	0
95	17	f3	1	0x5546420
96	17	:=	31	0
97	17	10	6	10
98	17	*	3	42
99	17	f1	1	0x55436b0
100	17	(27	40
101	17	a	1	0x5542900
102	17	,	34	44
103	17	2.3	5	2
104	17)	28	41
105	18	end	14	0
106	18	;	33	59
107	19	begin	11	0
108	20	out	1	0x55479d0
109	20	:=	31	0
110	20	f3	1	0x5546420
111	20	(27	40
112	20	5.3	5	5
113	20)	28	41

114	21	end	14	0
115	21	.	25	46
116	23	EOF	0	0

Appendix II: Program Listings

Listing 17: common/io.c

```
1  /* -*- C -*-
2  *
3  * io.c
4  *
5  * Author: Benjamin T James
6  */
7
8  #include <stdlib.h>
9  #include <stddef.h>
10 #include "defs.h"
11 #include "io.h"
12 #include "util.h"
13
14 int read_line(struct line *buf, FILE *f)
15 {
16     int c, ret = 0;
17     unsigned offset = 0;
18     buf->len = 0;
19     while ((c = getc(f)) != EOF) {
20         if (offset == buf->alloc) {
21             buf->len += offset;
22             offset = 0;
23             buf->err = LEXERR_LINE_TOO_LONG;
24         }
25         buf->buf[offset++] = c;
26         if (c == '\n') {
27             buf->buf[offset] = '\0';
28             break;
29         }
30     }
31     buf->len += offset;
32     if (c == EOF) {
33         ret = -1;
34     }
35     return ret;
36 }
37
38 int open_file(const char* src_file, const char*
39 ext, FILE** out)
40 {
41     char *out_name;
42     FILE *f;
43     if (get_out_file(src_file, ext, &out_name) <
44         0) {
45         return -1;
46     }
47     f = fopen(out_name, "w");
48     *out = f;
49     free(out_name);
50     if (f == NULL) {
51         fprintf(stderr, "Could not open file \"%s
52         \"\n", out_name);
53         return -1;
54     }
55 }
```

```
55
56 int init_buf(struct line* l, size_t alloc)
57 {
58     l->buf = malloc(alloc + 1);
59     if (l->buf == NULL) {
60         fprintf(stderr, "Could not allocate
61         resources\n");
62         return -1;
63     }
64     l->buf[alloc] = 0;
65     l->alloc = alloc;
66     l->err = 0;
67     l->len = 0;
68     return 0;
69 }
70
71 int free_buf(struct line *l)
72 {
73     if (l->buf != NULL) {
74         free(l->buf);
75     }
76     return 0;
77 }
```

Listing 18: common/io.h

```
1  /* -*- C -*-
2  *
3  * io.h
4  *
5  * Author: Benjamin T James
6  */
7
8  #ifndef IO_H
9  #define IO_H
10 #include <stdio.h>
11 #include <stddef.h>
12
13 struct line {
14     char *buf;
15     int len;
16     size_t alloc;
17     int err;
18 };
19
20 int open_file(const char* src_file, const char*
21 ext, FILE** out);
22 int init_buf(struct line* l, size_t alloc);
23 int free_buf(struct line* l);
24
25 int read_line(struct line* l, FILE *f);
26
27 #endif
```

Listing 19: common/defs.h

```
1  /* -*- C -*-
2  *
3  * defs.h
```

```

4  *
5  * Author: Benjamin T James
6  */
7
8  #ifndef DEFS_H
9  #define DEFS_H
10
11 #define NOPRINT 1024
12
13 #define TOKEN_EOF 0
14 #define TOKEN_ID 1
15 #define TOKEN_ADDOP 2
16 #define TOKEN_MULOP 3
17 #define TOKEN_RELOP 4
18 #define TOKEN_NUM_REAL 5
19 #define TOKEN_NUM_INTEGER 6
20 #define TOKEN_SIGN 7
21
22 #define TOKEN_ARRAY 10
23 #define TOKEN_BEGIN 11
24 #define TOKEN_DO 12
25 #define TOKEN_ELSE 13
26 #define TOKEN_END 14
27 #define TOKEN_FUNCTION 15
28 #define TOKEN_IF 16
29 #define TOKEN_INTEGER 17
30 #define TOKEN_NOT 18
31 #define TOKEN_OF 19
32 #define TOKEN_PROGRAM 20
33 #define TOKEN_REAL 21
34 #define TOKEN_THEN 22
35 #define TOKEN_VAR 23
36 #define TOKEN_WHILE 24
37
38
39 #define TOKEN_PERIOD 25
40 #define TOKEN_ELLIPSIS 26
41 #define TOKEN_LPAREN 27
42 #define TOKEN_RPAREN 28
43 #define TOKEN_LBRACKET 29
44 #define TOKEN_RBRACKET 30
45 #define TOKEN_ASSIGN 31
46 #define TOKEN_COLON 32
47 #define TOKEN_SEMICOLON 33
48 #define TOKEN_COMMA 34
49
50
51
52 #define TOKEN_LT 41
53 #define TOKEN_LEQ 42
54 #define TOKEN_EQ 43
55 #define TOKEN_NEQ 44
56 #define TOKEN_GT 45
57 #define TOKEN_GEQ 46
58
59
60 #define LEXERR 99
61
62 #define TOKEN_WHITESPACE 1024
63 #define TOKEN_NEWLINE 1025

```

```

64
65 #define LEXERR_ID_TOO_LONG 1
66 #define LEXERR_UNREC_SYM 2
67 #define LEXERR_INT_TOO_LONG 3
68 #define LEXERR_MANTIS_TOO_LONG 4
69 #define LEXERR_FRAC_TOO_LONG 5
70 #define LEXERR_LEADING_ZERO 6
71 #define LEXERR_TRAILING_ZERO 7
72 #define LEXERR_LINE_TOO_LONG 8
73 #define LEXERR_EXP_TOO_LONG 9
74 #define LEXERR_NO_EXP 10
75 #define LEXERR_NO_FRAC 11
76
77
78
79 #define ID_STRLEN 10
80
81 #ifndef LINELEN
82 #define LINELEN 72
83 #endif
84
85 /* forward declarations */
86 typedef struct machine *machine_t;
87 typedef struct lex_state *lex_state_t;
88
89 #endif

```

Listing 20: common/util.c

```

1  /* -*- C -*-
2  *
3  * util.c
4  *
5  * Author: Benjamin T James
6  */
7
8  #include "util.h"
9  #include <stdlib.h>
10 #include <string.h>
11
12 int get_file_without_ext(const char* f, char **
    to_write)
13 {
14     char *loc, *buf;
15     if (sdup(f, &buf) < 0) {
16         return -1;
17     }
18     *to_write = buf;
19     loc = strrchr(buf, '.');
20     if (loc) {
21         *loc = 0;
22     } else {
23         return -1;
24     }
25     return 0;
26 }
27
28 int get_out_file(const char* in_file, const char*
    extension, char **out_file)
29 {
30     char *str, *buf;

```

```

31     int i, ext_len, total_len;
32     if (get_file_without_ext(in_file, &str) < 0) {
33         fprintf(stderr, "File \"%s\" must have an
           extension\n", in_file);
34         return -1;
35     }
36     i = strlen(str);
37     ext_len = strlen(extension) + 1; /* for
           decimal */
38     total_len = i + ext_len;
39     buf = malloc(total_len + 1); /* for null
           terminator */
40     if (buf == NULL) {
41         fprintf(stderr, "Could not allocate
           resources\n");
42         return -1;
43     }
44     buf[total_len] = 0;
45     sprintf(buf, "%s.%s", str, extension);
46     free(str);
47     *out_file = buf;
48     return 0;
49 }
50
51 int get_str(char *f, char *b, char **ret)
52 {
53     int len = (f - b) + 1;
54     char *buf = malloc(len + 1);
55     if (buf == NULL) {
56         fprintf(stderr, "Could not allocate
           resources\n");
57         return -1;
58     }
59     memcpy(buf, b, len);
60     buf[len] = 0;
61     *ret = buf;
62     return 0;
63 }
64
65 int sdup(const char* s, char **ret)
66 {
67     int len = strlen(s);
68     char *buf = malloc(len + 1);
69     if (buf == NULL) {
70         fprintf(stderr, "Could not allocate
           resources\n");
71         return -1;
72     }
73     buf[len] = 0;
74     memcpy(buf, s, len);
75     *ret = buf;
76     return 0;
77 }

```

Listing 21: common/util.h

```

1  /* -*- C -*-
2  *
3  * util.h
4  *
5  * Author: Benjamin T James

```

```

6  */
7
8  #ifndef UTIL_H
9  #define UTIL_H
10 #include <stdio.h>
11
12
13 int get_out_file(const char* in_file, const char*
           extension, char **out_file);
14 int get_file_without_ext(const char* f, char **
           to_write);
15 int get_str(char *f, char *b, char **ret);
16
17 int sdup(const char* s, char **ret);
18
19 #endif

```

Listing 22: common/token.c

```

1  /* -*- C -*-
2  *
3  * token.c
4  *
5  * Author: Benjamin T James
6  */
7
8  #include "token.h"
9  #include "defs.h"
10
11 int token_id(struct token *t, char *ptr)
12 {
13     t->is_id = 1;
14     t->type = TOKEN_ID;
15     t->val.ptr = ptr;
16     return 0;
17 }
18
19 int token_add(struct token *t, int type, int attr)
20 {
21     t->is_id = 0;
22     t->type = type;
23     t->val.attr = attr;
24     return 0;
25 }
26
27 int token_println(FILE* f, int line, const char *
           lexeme, struct token t)
28 {
29     if (t.type & NOPRINT) {
30         return 0;
31     } else if (t.is_id) {
32         return fprintf(f, "%d\t%s\t %d\t%p\n",
           line, lexeme, t.type, t.val.ptr);
33     } else {
34         return fprintf(f, "%d\t%s\t %d\t%d\n",
           line, lexeme, t.type, t.val.attr);
35     }
36 }
37
38 }
39
40

```

```

41 char *token2str(int token)
42 {
43     switch (token) {
44         case TOKEN_ADDOP: return "ADDOP";
45         case TOKEN_ARRAY: return "array";
46         case TOKEN_ASSIGN: return "!=";
47         case TOKEN_BEGIN: return "begin";
48         case TOKEN_COLON: return ":";
49         case TOKEN_COMMA: return ",";
50         case TOKEN_DO: return "do";
51         case TOKEN_ELLIPSIS: return "...";
52         case TOKEN_ELSE: return "else";
53         case TOKEN_END: return "end";
54         case TOKEN_EOF: return "EOF";
55         case TOKEN_FUNCTION: return "function";
56         case TOKEN_ID: return "identifier";
57         case TOKEN_IF: return "if";
58         case TOKEN_INTEGER: return "integer";
59         case TOKEN_LBRACKET: return "[";
60         case TOKEN_LPAREN: return "(";
61         case TOKEN_MULOP: return "MULOP";
62         case TOKEN_NOT: return "not";
63         case TOKEN_NUM_INTEGER: return "NUM_INTEGER";
64         case TOKEN_NUM_REAL: return "NUM_REAL";
65         case TOKEN_OF: return "of";
66         case TOKEN_PERIOD: return ".";
67         case TOKEN_PROGRAM: return "program";
68         case TOKEN_RBRACKET: return "]";
69         case TOKEN_REAL: return "real";
70         case TOKEN_RELOP: return "RELOP";
71         case TOKEN_RPAREN: return ")";
72         case TOKEN_SEMICOLON: return ";";
73         case TOKEN_SIGN: return "+ or -";
74         case TOKEN_THEN: return "then";
75         case TOKEN_VAR: return "var";
76         case TOKEN_WHILE: return "while";
77         case LEXERR: return "LEXERR";
78     }
79     return "UNKNOWN";
80 }

```

Listing 23: common/token.h

```

1  /* -*- C -*-
2  *
3  * token.h
4  *
5  * Author: Benjamin T James
6  */
7
8  #ifndef TOKEN_H
9  #define TOKEN_H
10
11 #include <stdio.h>
12
13 union tok_val {
14     int attr;
15     void *ptr;
16 };
17
18 struct token {

```

```

19     int type;
20     /* char *lex; */
21     unsigned is_id : 1;
22     union tok_val val;
23 };
24
25 int token_id(struct token *t, char *ptr);
26 int token_add(struct token *t, int type, int attr)
27 ;
28 int token_println(FILE *f, int line, const char *
29     lexeme, struct token t);
30 char* token2str(int token);
31 #endif

```

Listing 24: common/idres.c

```

1  /* -*- C -*-
2  *
3  * idres.c
4  *
5  * Author: Benjamin T James
6  */
7
8  #include "idres.h"
9  #include "defs.h"
10 #include "util.h"
11 #include <math.h>
12 #include <stdlib.h>
13 #include <string.h>
14
15 int idres_print(FILE* f, struct idres **list)
16 {
17     struct idres *node = *list;
18     while (node != NULL) {
19         fprintf(f, "%p\t%s\n", node->token.val.ptr
20             , node->lexeme);
21         node = node->next;
22     }
23     return 0;
24 }
25
26 int idres_insert(struct idres **list, char* lexeme
27     , struct token token)
28 {
29     int ret = 0;
30     struct idres *root = malloc(sizeof(*root));
31     root->lexeme = lexeme;
32     root->type = 0;
33     root->token = token;
34     root->next = *list;
35     *list = root;
36     return ret;
37 }
38
39 int idres_add_rw(struct idres **list, char*
40     c_lexeme, int type, int attr)
41 {
42     char *lexeme;
43     struct token tok;
44     if (sdup(c_lexeme, &lexeme) < 0) {
45         return -1;
46     }
47 }

```

```

43     token_add(&tok, type, attr);
44     return idres_insert(list, lexeme, tok);
45 }
46
47 int idres_add_id(struct idres **list, char*
48     c_lexeme)
49 {
50     char *lexeme;
51     struct token tok;
52     if (sdup(c_lexeme, &lexeme) < 0) {
53         return -1;
54     }
55     token_id(&tok, lexeme);
56     return idres_insert(list, lexeme, tok);
57 }
58
59 int idres_add_id_attr(struct idres **list, char*
60     c_lexeme, char* attr)
61 {
62     char *lexeme;
63     struct token tok;
64     if (sdup(c_lexeme, &lexeme) < 0) {
65         return -1;
66     }
67     token_id(&tok, lexeme);
68     tok.val.ptr = attr;
69     return idres_insert(list, lexeme, tok);
70 }
71
72 int idres_lookup(struct idres **list, void* ptr,
73     struct idres **ret)
74 {
75     struct idres *node = *list;
76     while (node != NULL) {
77         if (ptr == node->token.val.ptr) {
78             *ret = node;
79             return 0;
80         }
81         node = node->next;
82     }
83     return -1;
84 }
85
86 int idres_find(struct idres *node, char *lexeme,
87     struct idres **ret)
88 {
89     while (node != NULL) {
90         if (!strcmp(lexeme, node->lexeme)) {
91             *ret = node;
92             return 0;
93         }
94         node = node->next;
95     }
96     return -1;
97 }
98
99 int idres_search(struct idres **list, char* lexeme,
100     struct idres **ret)
101 {
102     return idres_find(*list, lexeme, ret);
103 }
104
105 int idres_clean(struct idres **list)
106 {
107     while (*list != NULL) {
108         struct idres *prev = *list;
109         *list = prev->next;
110         free(prev->lexeme);
111         free(prev);
112     }
113     return 0;
114 }
115
116 int idres_read(const char *filename, struct idres
117     **list)
118 {
119     FILE* f = fopen(filename, "r");
120     void* addr = NULL;
121     long count;
122     char *lexeme = malloc(ID_STRLEN + 1);
123     /* strlen(lexeme) guaranteed to be ID_STRLEN
124        */
125     for (count = 0; fscanf(f, "0x%p\t%s\n", &addr,
126         lexeme) == 2; count++) {
127         idres_add_id_attr(list, lexeme, addr);
128     }
129     free(lexeme);
130     fclose(f);
131     /*return idres_balance(list, count);*/
132     return 0;
133 }

```

Listing 25: common/idres.h

```

1  /* -*- C -*-
2  *
3  * idres.h
4  *
5  * Author: Benjamin T James
6  */
7
8  #ifndef IDRES_H
9  #define IDRES_H
10
11 #include <stdlib.h>
12 #include "token.h"
13 #include "io.h"
14
15 struct idres {
16     char *lexeme;
17     int type;
18     struct token token;
19     struct idres *next;
20 };
21
22 int idres_add_rw(struct idres **list, char* lexeme
23     , int token, int attr);
24 int idres_add_id(struct idres **list, char* lexeme
25     );
26
27 int idres_search(struct idres **list, char* lexeme
28     , struct idres **ret);

```

```

26 int idres_lookup(struct idres **list, void* ptr,
    struct idres **ret);
27 int idres_clean(struct idres **list);
28 int idres_print(FILE* f, struct idres **list);
29
30 int idres_read(const char *filename, struct idres
    **list);
31
32 int idres_add_id_attr(struct idres **list, char*
    lexeme, char* attr);
33 #endif

```

Listing 26: lexer/main.c

```

1  /* -*- C -*-
2  *
3  * main.c
4  *
5  * Author: Benjamin T James
6  */
7
8 #include "state.h"
9 #include "defs.h"
10 #include "lexerr.h"
11
12 int handle_line(struct lex_state *s, int line_no)
13 {
14     char *lexeme = NULL;
15     struct machine state;
16     state.f = s->buf.buf;
17     state.b = state.f;
18     state.tok.type = 0;
19     if (s->buf.err == LEXERR_LINE_TOO_LONG) {
20         print_error(s->list, s->buf.err, s->buf.
            buf);
21     }
22     while (state.tok.type != TOKEN_NEWLINE) {
23         int ret = machine_iter(s, &state, &lexeme)
            ;
24         if (ret < 0) {
25             fprintf(stderr, "Machine not found\n");
26             return -1;
27         }
28         if (state.tok.type == LEXERR) {
29             print_error(s->list, state.tok.val.attr
                , lexeme);
30         }
31         token_println(s->token, line_no, lexeme,
            state.tok);
32         free(lexeme);
33         lexeme = NULL;
34     }
35     return 0;
36 }
37
38 int main(int argc, char **argv)
39 {
40     struct lex_state s;
41     struct token tok_eof;
42     int line = 1;
43     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

```

1  /* -*- C -*-
2  *
3  * state.c
4  *
5  * Author: Benjamin T James
6  */
7
8 #include "state.h"
9 #include "util.h"
10
11 int resword_init(struct lex_state *st)
12 {
13     int tok, attr;
14     char *lexeme = malloc(st->buf.alloc);
15     if (lexeme == NULL) {
16         fprintf(stderr, "Could not allocate memory
            \n");
17         return -1;
18     }
19     while (fscanf(st->res_word, "%s\t%d\t%d\n",
        lexeme, &tok, &attr) != EOF) {
20         idres_add_rw(&st->rwords, lexeme, tok,
            attr);
21     }
22     free(lexeme);
23     return 0;
24 }
25
26 int state_init(const char *source, const char *
    res_word,
27               int line_len, struct lex_state *st)
28 {
29     if (init_buf(&st->buf, line_len) < 0) {
30         return -1;
31     }

```

```

31     if (open_file(source, "list", &st->list) < 0)
32     {
33         return -1;
34     }
35     if (open_file(source, "tok", &st->token) < 0)
36     {
37         return -1;
38     }
39     st->res_word = fopen(res_word, "r");
40     if (st->res_word == NULL) {
41         fprintf(stderr, "Could not open file \"%s
42             \\\"\\n\", res_word);
43         return -1;
44     }
45     st->source = fopen(source, "r");
46     if (st->source == NULL) {
47         fprintf(stderr, "Could not open file \"%s
48             \\\"\\n\", source);
49         return -1;
50     }
51     st->rwords = NULL;
52     st->ids = NULL;
53     if (resword_init(st) < 0) {
54         return -1;
55     }
56     st->machines = NULL;
57     if (machine_init(&st->machines) < 0) {
58         return -1;
59     }
60     return 0;
61 }
62
63 int state_cleanup(struct lex_state *s)
64 {
65     free_buf(&s->buf);
66     fclose(s->source);
67     fclose(s->res_word);
68     fclose(s->sym);
69     fclose(s->list);
70     fclose(s->token);
71     idres_clean(&s->rwords);
72     idres_clean(&s->ids);
73     machine_clean(&s->machines);
74     return 0;
75 }

```

Listing 28: lexer/state.h

```

1  /* -*- C -*-
2  *
3  * state.h
4  *
5  * Author: Benjamin T James
6  */
7
8 #ifndef STATE_H
9 #define STATE_H
10
11 #include <stdio.h>
12 #include "defs.h"
13 #include "io.h"
14 #include "idres.h"
15 #include "machine.h"
16
17 struct lex_state {
18     /* inputs */
19     FILE* source;
20     FILE* res_word;
21
22     /* outputs */
23     FILE* sym;
24     FILE* list;
25     FILE* token;
26
27     /* lexer state */
28     struct line buf;
29     struct idres *rwords;
30     struct idres *ids;
31     machine_t machines;
32 };
33
34 int state_init(const char *source, const char *
35     res_word,
36     int line_len, struct lex_state *st);
37 int resword_init(struct lex_state *s);
38 int state_cleanup(struct lex_state *s);
39 #endif

```

Listing 29: lexer/fsm.c

```

1  /* -*- C -*-
2  *
3  * fsm.c
4  *
5  * Author: Benjamin T James
6  */
7
8 #include <ctype.h>
9 #include <string.h>
10 #include "fsm.h"
11 #include "defs.h"
12 #include "util.h"
13
14 int digit_plus(struct machine *m)
15 {
16     int len = 1;
17     if (!isdigit(*m->f)) {
18         return 0;
19     }
20     m->f++;
21     while (isdigit(*m->f)) {
22         m->f++;
23         len++;
24     }
25     return len;
26 }
27
28 int fsm_relop(struct machine *m, struct lex_state
29     *ls)

```

```

29 {
30     if (*m->f == '>') {
31         m->f++;
32         if (*m->f == '=') {
33             token_add(&m->tok, TOKEN_RELOP,
34                     TOKEN_GEQ);
35         } else {
36             m->f--;
37             token_add(&m->tok, TOKEN_RELOP,
38                     TOKEN_GT);
39         }
40     } else if (*m->f == '=') {
41         token_add(&m->tok, TOKEN_RELOP, TOKEN_EQ);
42     } else if (*m->f == '<') {
43         m->f++;
44         if (*m->f == '=') {
45             token_add(&m->tok, TOKEN_RELOP,
46                     TOKEN_LEQ);
47         } else if (*m->f == '>') {
48             token_add(&m->tok, TOKEN_RELOP,
49                     TOKEN_NEQ);
50         } else {
51             m->f--;
52             token_add(&m->tok, TOKEN_RELOP,
53                     TOKEN_LT);
54         }
55     } else {
56         return 0;
57     }
58     return 1;
59 }
60
61 int fsm_integer(struct machine *m, struct
62 lex_state *ls)
63 {
64     char *lexeme;
65     int result, len;
66     if (!digit_plus(m)) {
67         return 0;
68     }
69     m->f--;
70     if (get_str(m->f, m->b, &lexeme) < 0) {
71         return -1;
72     }
73     len = strlen(lexeme);
74     if (len > 10) {
75         token_add(&m->tok, LEXERR,
76                 LEXERR_INT_TOO_LONG);
77     } else if ((lexeme[0] == '0' && len > 1)
78                || (lexeme[0] == '-' && lexeme[1] == '0'
79                    && len > 2)) {
80         token_add(&m->tok, LEXERR,
81                 LEXERR_LEADING_ZERO);
82     } else {
83         result = strtol(lexeme, NULL, 10);
84         token_add(&m->tok, TOKEN_NUM_INTEGER,
85                 result);
86     }
87     free(lexeme);
88     return 1;
89 }
90
91 int fsm_real(struct machine *m, struct lex_state *
92 ls)
93 {
94     char *lexeme;
95     double result;
96     int len, mantis_len = 0, frac_len = 0;
97     mantis_len = digit_plus(m);
98     if (mantis_len == 0 || *m->f != '.') {
99         return 0;
100     }
101     m->f++;
102     frac_len = digit_plus(m);
103     if (frac_len == 0) {
104         return 0;
105     }
106     m->f--;
107     if (get_str(m->f, m->b, &lexeme) < 0) {
108         return -1;
109     }
110     len = strlen(lexeme);
111     if (mantis_len > 5) {
112         token_add(&m->tok, LEXERR,
113                 LEXERR_MANTIS_TOO_LONG);
114     } else if (frac_len > 5) {
115         token_add(&m->tok, LEXERR,
116                 LEXERR_FRAC_TOO_LONG);
117     } else if (lexeme[0] == '0' || (lexeme[0] == '
118         -' && lexeme[1] == '0')) {
119         token_add(&m->tok, LEXERR,
120                 LEXERR_LEADING_ZERO);
121     } else if (lexeme[len-1] == '0') {
122         token_add(&m->tok, LEXERR,
123                 LEXERR_TRAILING_ZERO);
124     } else {
125         result = strtod(lexeme, NULL);
126         token_add(&m->tok, TOKEN_NUM_REAL, (int)
127                 result);
128     }
129     free(lexeme);
130     return 1;
131 }
132
133 int fsm_long_real(struct machine *m, struct
134 lex_state *ls)
135 {
136     char *lexeme;
137     double result;
138     int len, tz = 0, mantis_len = 0, frac_len = 0,
139         exp_len = 0;
140     mantis_len = digit_plus(m);
141     if (mantis_len == 0 || *m->f != '.') {
142         return 0;
143     }
144     m->f++;
145     frac_len = digit_plus(m);
146     if (/* frac_len == 0 || */ *m->f != 'E') {
147         return 0;
148     }
149     if (get_str(m->f - 1, m->b, &lexeme) < 0) {

```



```

130     return -1;
131 }
132 if (lexeme[strlen(lexeme) - 1] == '0') {
133     tz = 1; /* set trailing zero flag to 1 */
134 }
135
136 free(lexeme);
137 m->f++;
138 if (*m->f == '-' || *m->f == '+') {
139     m->f++;
140 }
141 exp_len = digit_plus(m);
142 /* if (exp_len == 0) { */
143 /* /\* err *\*/
144 /* return 0; */
145 /* } */
146 m->f--;
147 if (get_str(m->f, m->b, &lexeme) < 0) {
148     return -1;
149 }
150 len = strlen(lexeme);
151 if (frac_len == 0) {
152     token_add(&m->tok, LEXERR, LEXERR_NO_FRAC);
153     ;
154 } else if (exp_len == 0) {
155     token_add(&m->tok, LEXERR, LEXERR_NO_EXP);
156 } else if (mantis_len > 5) {
157     token_add(&m->tok, LEXERR,
158         LEXERR_MANTIS_TOO_LONG);
159 } else if (frac_len > 5) {
160     token_add(&m->tok, LEXERR,
161         LEXERR_FRAC_TOO_LONG);
162 } else if (exp_len > 2) {
163     token_add(&m->tok, LEXERR,
164         LEXERR_EXP_TOO_LONG);
165 } else if (lexeme[0] == '0' || (lexeme[0] == '-'
166     && lexeme[1] == '0')) {
167     token_add(&m->tok, LEXERR,
168         LEXERR_LEADING_ZERO);
169 } else if (tz || lexeme[len-1] == '0') {
170     token_add(&m->tok, LEXERR,
171         LEXERR_TRAILING_ZERO);
172 } else {
173     result = strtod(lexeme, NULL);
174     token_add(&m->tok, TOKEN_NUM_REAL, (int)
175         result);
176 }
177 free(lexeme);
178 return 1;
179 }
180
181 int fsm_addop(struct machine *m, struct lex_state
182     *ls)
183 {
184     if (*m->f == '+') {
185         token_add(&m->tok, TOKEN_SIGN, '+');
186         return 1;
187     } else if (*m->f == '-') {
188         token_add(&m->tok, TOKEN_SIGN, '-');
189         return 1;
190     } else if (*m->f == 'o') {
191         m->f++;
192         if (*m->f == 'r') {
193             token_add(&m->tok, TOKEN_ADDOP, '|');
194             return 1;
195         }
196     }
197     return 0;
198 }
199
200 int fsm_mulop(struct machine *m, struct lex_state
201     *ls)
202 {
203     if (*m->f == '*') {
204         token_add(&m->tok, TOKEN_MULOP, '*');
205         return 1;
206     } else if (*m->f == '/') {
207         token_add(&m->tok, TOKEN_MULOP, '/');
208         return 1;
209     } else if (*m->f == 'd') {
210         m->f++;
211         if (*m->f == 'i' && m->f++ && *m->f == 'v'
212             ) {
213             token_add(&m->tok, TOKEN_MULOP, '/');
214             return 1;
215         }
216     } else if (*m->f == 'm') {
217         m->f++;
218         if (*m->f == 'o' && m->f++ && *m->f == 'd'
219             ) {
220             token_add(&m->tok, TOKEN_MULOP, '%');
221             return 1;
222         }
223     } else if (*m->f == 'a') {
224         m->f++;
225         if (*m->f == 'n' && m->f++ && *m->f == 'd'
226             ) {
227             token_add(&m->tok, TOKEN_MULOP, '&');
228             return 1;
229         }
230     }
231     return 0;
232 }
233
234 int fsm_catchall(struct machine *m, struct
235     lex_state *ls)
236 {
237     switch (*m->f) {
238     case '[':
239         token_add(&m->tok, TOKEN_LBRACKET, *m->f);
240         return 1;
241     case ']':
242         token_add(&m->tok, TOKEN_RBRACKET, *m->f);
243         return 1;
244     case '(':
245         token_add(&m->tok, TOKEN_LPAREN, *m->f);
246         return 1;
247     case ')':
248         token_add(&m->tok, TOKEN_RPAREN, *m->f);
249         return 1;
250     case ',':
251         token_add(&m->tok, TOKEN_COMMA, *m->f);
252         return 1;
253     }
254     return 0;
255 }

```

```

236     token_add(&m->tok, TOKEN_COMMA, *m->f); 291     } else {
237     return 1;                                292         idres_add_id(&ls->ids, lexeme);
238     case ',':                                293         m->tok = ls->ids->token;
239     token_add(&m->tok, TOKEN_SEMICOLON, *m->f) 294     }
240     ;                                         295     free(lexeme);
241     return 1;                                296     return 1;
242     break;                                   297     }
243 }                                             298     return 0;
244 if (*m->f == '.') {                          299 }
245     m->f++;
246     if (*m->f == '.') {                      300 int fsm_unrecognized_symbol(struct machine *m,
247         token_add(&m->tok, TOKEN_ELLIPSIS,    struct lex_state *ls)
248         TOKEN_ELLIPSIS);                    301 {
249     } else {                                302     m->tok.type = LEXERR;
250     m->f--;                                  303     m->tok.val.attr = LEXERR_UNREC_SYM;
251     token_add(&m->tok, TOKEN_PERIOD, '.');    304     return 1;
252     return 1;                                305 }
253 } else if (*m->f == ':') {                  306 int fsm_newline(struct machine *m, struct
254     m->f++;                                  lex_state *ls)
255     if (*m->f == '=') {                      307 {
256         token_add(&m->tok, TOKEN_ASSIGN, 0);  308     if (*m->f == '\n') {
257     } else {                                309         m->f++;
258         m->f--;                                  310         m->tok.type = TOKEN_NEWLINE;
259         token_add(&m->tok, TOKEN_COLON, ':');  311         return 1;
260     }                                         312     }
261     return 1;                                313     return 0;
262 }                                             314 }
263 return 0;                                   315 int fsm_whitespace(struct machine *m, struct
264 }                                             lex_state *ls)
265                                             316 {
266 int fsm_idres(struct machine *m, struct lex_state 317     if (*m->f == ' ' || *m->f == '\t') {
267     *ls)                                     318         m->f++;
268 {                                             319         while (*m->f == ' ' || *m->f == '\t') {
269     if (isalpha(*m->f)) {                    320             m->f++;
270         int len;                            321         }
271         char *lexeme;                        322         m->f--;
272         struct idres *result;                323         m->tok.type = TOKEN_WHITESPACE;
273         m->f++;                              324         return 1;
274         while (isalnum(*m->f)) {            325     }
275             m->f++;                          326     return 0;
276     }                                         327 }
277     m->f--;                                  328 }
278
279     if (get_str(m->f, m->b, &lexeme) < 0) {
280         return -1;
281     }
282     len = strlen(lexeme);
283     if (len > ID_STRLEN) {
284         m->tok.type = LEXERR;
285         m->tok.is_id = 0;
286         m->tok.val.attr = LEXERR_ID_TOO_LONG;
287     } else if (idres_search(&ls->rwords,
288         lexeme, &result) == 0) {
289         m->tok = result->token;
290     } else if (idres_search(&ls->ids, lexeme,
291         &result) == 0) {
292         m->tok = result->token;

```

Listing 30: lexer/fsm.h

```

1  /* -*- C -*-
2  *
3  * fsm.h
4  *
5  * Author: Benjamin T James
6  */
7
8  #ifndef FSM_H
9  #define FSM_H
10
11 #include "machine.h"
12 #include "state.h"
13
14 int fsm_unrecognized_symbol(struct machine *m,
15     struct lex_state *ls);
16
17 int fsm_whitespace(struct machine *m, struct
18     lex_state *ls);

```

<pre> 16 int fsm_newline(struct machine *m, struct lex_state *ls); 17 int fsm_idres(struct machine *m, struct lex_state *ls); 18 int fsm_relop(struct machine *m, struct lex_state *ls); 19 int fsm_addop(struct machine *m, struct lex_state *ls); 20 int fsm_mulop(struct machine *m, struct lex_state *ls); 21 int fsm_catchall(struct machine *m, struct lex_state *ls); 22 int fsm_integer(struct machine *m, struct lex_state *ls); 23 int fsm_real(struct machine *m, struct lex_state *ls); 24 int fsm_long_real(struct machine *m, struct lex_state *ls); 25 #endif </pre>	<pre> 39 fprintf(listing, "Exponent too long:"); 40 break; 41 case LEXERR_NO_EXP: 42 fprintf(listing, "No exponent:"); 43 break; 44 case LEXERR_NO_FRAC: 45 fprintf(listing, "No fractional part:"); 46 break; 47 default: 48 fprintf(listing, "Unknown error %d:", err) 49 ; 50 } 51 fprintf(listing, "\t\t%s\n", lexeme); 52 return 0; } </pre>
---	---

Listing 31: lexer/lexerr.c

```

1  /* -*- C -*-
2  *
3  * lexerr.c
4  *
5  * Author: Benjamin T James
6  */
7
8 #include "lexerr.h"
9
10 int print_error(FILE* listing, int err, const char
    *lexeme)
11 {
12     fprintf(listing, "LEXERR:\t");
13     switch (err) {
14     case LEXERR_ID_TOO_LONG:
15         fprintf(listing, "ID too long:");
16         break;
17     case LEXERR_UNREC_SYM:
18         fprintf(listing, "Unrecognized symbol:");
19         break;
20     case LEXERR_INT_TOO_LONG:
21         fprintf(listing, "Int too long:");
22         break;
23     case LEXERR_MANTIS_TOO_LONG:
24         fprintf(listing, "Mantissa too long:");
25         break;
26     case LEXERR_FRAC_TOO_LONG:
27         fprintf(listing, "Fraction too long:");
28         break;
29     case LEXERR_LEADING_ZERO:
30         fprintf(listing, "Leading zero:");
31         break;
32     case LEXERR_TRAILING_ZERO:
33         fprintf(listing, "Trailing zero:");
34         break;
35     case LEXERR_LINE_TOO_LONG:
36         fprintf(listing, "Line too long:");
37         break;
38     case LEXERR_EXP_TOO_LONG:

```

Listing 32: lexer/lexerr.h

```

1  /* -*- C -*-
2  *
3  * lexerr.h
4  *
5  * Author: Benjamin T James
6  */
7
8 #ifndef LEXERR_H
9 #define LEXERR_H
10
11 #include "machine.h"
12
13 int print_error(FILE* listing, int err, const char
    *lexeme);
14
15 #endif

```

Listing 33: lexer/machine.c

```

1  /* -*- C -*-
2  *
3  * machine.c
4  *
5  * Author: Benjamin T James
6  */
7
8 #include "machine.h"
9 #include "fsm.h"
10 #include "util.h"
11
12 int machine_iter(struct lex_state *ls, struct
    machine *state, char **out_str)
13 {
14     int ret;
15     struct machine *m = ls->machines;
16     for (; m != NULL; m = m->next) {
17         m->b = state->f;
18         m->f = m->b;
19         m->tok.is_id = 0;
20         ret = m->call(m, ls);
21         if (ret == 1) {
22             state->tok = m->tok;
23             state->f = m->f + 1;

```

```

24         state->b = state->f;
25
26         return get_str(m->f, m->b, out_str);
27     } else if (ret == -1) {
28         return -1;
29     }
30 }
31 return -1;
32 }
33 int machine_init(struct machine **list)
34 {
35     machine_add(list, fsm_unrecognized_symbol);
36     machine_add(list, fsm_newline);
37
38     machine_add(list, fsm_catchall);
39     machine_add(list, fsm_relop);
40
41     machine_add(list, fsm_integer);
42     machine_add(list, fsm_real);
43     machine_add(list, fsm_long_real);
44
45     machine_add(list, fsm_idres);
46     machine_add(list, fsm_addop);
47     machine_add(list, fsm_mulop);
48
49     machine_add(list, fsm_whitespace);
50     return 0;
51 }
52 int machine_add(struct machine **list,
53                 int (*func)(struct machine *m, struct
54                             lex_state *ls))
55 {
56     struct machine *m = malloc(sizeof(*m));
57     if (m == NULL) {
58         fprintf(stderr, "Unable to allocate
59             resources\n");
60         return -1;
61     }
62     m->call = func;
63     m->next = *list;
64     *list = m;
65     return 0;
66 }
67 int machine_clean(struct machine **list)
68 {
69     struct machine *head = *list;
70     struct machine *tmp;
71     while (head != NULL) {

```

```

70         tmp = head;
71         head = head->next;
72         free(tmp);
73     }
74     return 0;
75 }

```

Listing 34: lexer/machine.h

```

1  /* -*- C -*-
2  *
3  * machine.h
4  *
5  * Author: Benjamin T James
6  */
7
8  #ifndef MACHINE_H
9  #define MACHINE_H
10
11  #include "defs.h"
12  #include "token.h"
13  #include "state.h"
14
15  struct machine {
16      /* returns 1 on success, 0 on failure */
17      int (*call)(struct machine *m, lex_state_t ls)
18          ;
19
20      char *f;
21      char *b;
22      struct token tok;
23      struct machine *next;
24  };
25
26  /* interface for the lexer */
27  int machine_iter(lex_state_t ls, struct machine *
28                  state, char **out_str);
29
30  int machine_init(struct machine **list);
31  int machine_add(struct machine **list,
32                  int (*func)(struct machine *m, struct
33                              lex_state *ls));
34
35  int machine_clean(struct machine **list);
36
37  #endif

```
