

CS 4013: Compiler Construction

Project 3

Benjamin James

December 14, 2017

Introduction

Project 3 consists of “decorating” the Pascal grammar to create a semantic analyzer. This analyzer performs declarations processing and checking for types and scope according to Pascal’s grammar, disallowing all mixed mode expressions. All errors found are deposited into the listing file. Using the recursive descent parser completed in Project 2, Left-Attributed Definitions were put in-place on the productions to allow one-pass parsing.

Methodology

The grammar was decorated with the following steps, as enumerated in Aho et al.[1]

The scope checking follows the rules of Pascal, which allows for nested subprograms but disallows duplicate names. Parameters are also checked, and Pascal’s function return values, which assigns an expression to the function name, are also implemented.

Error recovery is implemented so that an error is reported only once, and using Project 2’s recursive descent parser, skips only as many tokens as necessary.

1 void *program* → **program** {{offset= 0}} **id** {{checkaddgreen(id.lex, TYPE_PGM)}} (*identifier_list*) ;
declarations subprogram_declarations compound_statement .
2.1.1 void *identifier_list* → **id** {{checkaddblue(id.lex, TYPE_IDLIST)}} *identifier_list'*
2.2.1 void *identifier_list'* → , **id** {{checkaddblue(id.lex, TYPE_IDLIST)}} *identifier_list'*
2.2.2 void *identifier_list'* → ϵ
3.1.1 void *declarations* → **var id : type** {{checkaddblue(id.lex, type.type, offset); offset += type.width}} ; *declarations*
3.2.1 void *declarations* → ϵ
4.1 *type.type type* → *standard_type* {{type.type = standard_type.type; type.width = standard_type.width}}
4.2 *type.type type* → **array [num .. num] of standard_type**
{{type.width = (num₂ - num₁ + 1) * standard_type.width}}

<i>type.type</i>	←	<i>standard_type.type</i>
TYPE_AINT	if	TYPE_INT
TYPE_AREAL	if	TYPE_REAL
ERR	if	ERR
ERR*	otherwise	

5.1 *standard_type.type standard_type* → **integer** {{standard_type.type = TYPE_INT; standard_type.width = 4}}
5.2 *standard_type.type standard_type* → **real** {{standard_type.type = TYPE_REAL; standard_type.width = 8}}
6.1.1 void *subprogram_declarations* → *subprogram_declaration* ; *subprogram_declarations*
6.2.1 void *subprogram_declarations* → ϵ
7 void *subprogram_declaration* → *subprogram_head declarations*
subprogram_declarations compound_statement {{endgreenscope();}}
8 void *subprogram_head* → **function id** {{checkaddgreen(id.lex, TYPE_PLACEHOLDER)}}
arguments : standard_type {{eye_stack.peek().args = arguments.str}} ;

<i>eye_stack.peek().type</i>	←	<i>standard_type.type</i>
TYPE_FINT	if	TYPE_INT
TYPE_FREAL	if	TYPE_REAL
ERR	if	ERR
ERR*	otherwise	

9.1 *arguments.str arguments* → (*parameter_list*) {{arguments.str = parameter_list.str}}
9.2 *arguments.str arguments* → ϵ {{arguments.str = ""}}
10.1.1 *parameter_list.str parameter_list* → **id : type** {{checkaddblue(id.lex, type.type, 0); }}
parameter_list' {{parameter_list.str = type2str(type.type) . parameter_list'.str}}
10.2.1 *parameter_list'.str parameter_list'* → ; **id : type** {{checkaddblue(id.lex, type.type, 0); }}
parameter_list' {{parameter_list'.str = type2str(type.type) . parameter_list_1'.str}}
10.2.2 *parameter_list'.str parameter_list'* → ϵ {{parameter_list'.str = ""}}
11 void *compound_statement* → **begin optional_statements end**
12.1 void *optional_statements* → *statement_list*
12.2 void *optional_statements* → ϵ
13.1.1 void *statement_list* → *statement statement_list'*
13.2.1 void *statement_list'* → ; *statement statement_list'*
13.2.2 void *statement_list'* → ϵ
14.1.1 void *statement* → *variable assignop expression*

<i>statement.type</i>	←	<i>variable.type</i>	<i>expression.type</i>
ERR*	if	Undeclared	
ERR	if	ERR	
ERR	if		ERR
VOID	if	TYPE_INT	TYPE_INT
VOID	if	TYPE_FINT	TYPE_INT
VOID	if	TYPE_REAL	TYPE_REAL
VOID	if	TYPE_FREAL	TYPE_REAL
ERR*	otherwise		

14.2.1 void *statement* → *compound_statement*
14.3.1 void *statement* → **if expression** {{check(expression.type == TYPE_BOOL)}} **then statement statement'**
14.4.1 void *statement'* → **else statement**
14.4.2 void *statement'* → ϵ
14.5.1 void *statement* → **while expression** {{check(expression.type == TYPE_BOOL)}} **do statement**
15.1.1 *variable.type variable* → **id** {{variable'.i = gettype(id.lex)}} *variable'* {{variable.type = variable'.type}}
15.2.1 *variable'.type variable'* → [*expression*]

$variable'.type$	\leftarrow	$expression.type$	$variable'.i$
ERR*	if		Undeclared
TYPE_INT	if	TYPE_INT	TYPE_AINT
TYPE_REAL	if	TYPE_INT	TYPE_AREAL
ERR	if	ERR	
ERR	if		ERR
ERR*	if	\neg TYPE_INT	
ERR*	if		\neg TYPE_AINT and \neg TYPE_AREAL

15.2.2 $variable'.type \ variable' \rightarrow \epsilon \{ \{ variable'.type = variable'.i \} \}$

16.1.1 $expression_list.str \ expression_list \rightarrow expression \ expression_list'$
 $\{ \{ expression_list.str = type2str(expression.type) . expression_list'.str \} \}$

16.2.1 $expression_list'.str \ expression_list' \rightarrow , \ expression \ expression_list'$
 $\{ \{ expression_list'.str = type2str(expression.type) . expression_list_1'.str \} \}$

16.2.2 $expression_list'.str \ expression_list' \rightarrow \epsilon \{ \{ expression_list'.str = "" \} \}$

17.1.1 $expression.type \ expression \rightarrow simple_expression \ \{ \{ expression'.i = simple_expression.type \} \}$
 $expression' \ \{ \{ expression.type = expression'.type \} \}$

17.2.1 $expression'.type \ expression' \rightarrow \epsilon \{ \{ expression'.type = expression'.i \} \}$

17.2.2 $expression'.type \ expression' \rightarrow \mathbf{relop} \ simple_expression$

$expression'.type$	\leftarrow	$simple_expression.type$	$expression'.i$
TYPE_BOOL	if	TYPE_INT	TYPE_INT
TYPE_BOOL	if	TYPE_REAL	TYPE_REAL
ERR	if	ERR	
ERR	if		ERR
ERR*	otherwise		

18.1.1 $simple_expression.type \ simple_expression \rightarrow term \ \{ \{ simple_expression'.i = term.type \} \}$
 $simple_expression' \ \{ \{ simple_expression.type = simple_expression'.type \} \}$

18.2.1 $simple_expression.type \ simple_expression \rightarrow sign \ term$

$\{ \{ ERR* \text{ if } term.type \notin \{ TYPE_REAL, TYPE_INT, ERR \} \} \}$

$\{ \{ simple_expression'.i = term.type \} \} \ simple_expression' \ \{ \{ simple_expression.type = simple_expression'.type \} \}$

18.3.1 $simple_expression'.type \ simple_expression' \rightarrow \mathbf{addop} \ term \ simple_expression'$

$\{ \{ simple_expression'.type = simple_expression'_1.type \} \}$

$simple_expression'_1.i$	\leftarrow	$simple_expression'.i$	$\mathbf{addop}.attr$	$term.type$
TYPE_INT	if	TYPE_INT	+	TYPE_INT
TYPE_INT	if	TYPE_INT	-	TYPE_INT
TYPE_REAL	if	TYPE_REAL	+	TYPE_REAL
TYPE_REAL	if	TYPE_REAL	-	TYPE_REAL
TYPE_BOOL	if	TYPE_BOOL	or	TYPE_BOOL
ERR	if	ERR		
ERR	if			ERR
ERR*	otherwise			

18.3.2 $simple_expression'.type \ simple_expression' \rightarrow \epsilon \{ \{ simple_expression'.type = simple_expression'.i \} \}$

19.1.1 $term.type \ term \rightarrow factor \ \{ \{ term'.i = factor.type \} \} \ term' \ \{ \{ term.type = term'.type \} \}$

19.2.1 $term'.type \ term' \rightarrow \mathbf{mulop} \ factor \ term' \ \{ \{ term.type = term'.type \} \}$

$term'_1.i$	\leftarrow	$term'.i$	$\mathbf{mulop}.attr$	$factor.type$
TYPE_INT	if	TYPE_INT	*	TYPE_INT
TYPE_REAL	if	TYPE_REAL	*	TYPE_REAL
TYPE_REAL	if	TYPE_REAL	/	TYPE_REAL
TYPE_INT	if	TYPE_INT	div	TYPE_INT
TYPE_INT	if	TYPE_INT	mod	TYPE_INT
TYPE_BOOL	if	TYPE_BOOL	and	TYPE_BOOL
ERR	if	ERR		
ERR	if			ERR
ERR*	otherwise			

19.2.2 $term'.type \ term' \rightarrow \epsilon \{ \{ term'.type = term'.i \} \}$

20.1.1 $factor.type \ factor \rightarrow \mathbf{id} \ \{\{factor'.i = \text{gettype}(id.lex)\}\} \ factor' \ \{\{factor.type = factor'.type\}\}$

20.2.1 $factor'.type \ factor' \rightarrow [\ expression]$

$factor'.type$	\leftarrow	$expression.type$	$factor'.i$
ERR*	if		Undeclared
TYPE_INT	if	TYPE_INT	TYPE_AINT
TYPE_REAL	if	TYPE_INT	TYPE_AREAL
ERR	if	ERR	
ERR	if		ERR
ERR*	if	\neg TYPE_INT	
ERR*	if		\neg TYPE_AINT and \neg TYPE_AREAL

20.2.2 $factor'.type \ factor' \rightarrow \epsilon \ \{\{factor'.type = factor'.i \text{ if declared and } \in \{\text{TYPE_INT}, \text{TYPE_REAL}\}\}\}$

20.3.1 $factor'.type \ factor' \rightarrow (\ expression_list)$

$\{\{factor'.type = \text{funtype}'\text{to}'\text{scalar}(factor'.i); \text{check}(expression_list.str == \text{get}'\text{args}(factor'.i))\}\}$

$factor'.type$	\leftarrow	$factor'.i$
ERR*	if	Undeclared
ERR*	if	\neg TYPE_FINT and \neg TYPE_FREAL
TYPE_INT	if	TYPE_FINT
TYPE_REAL	if	TYPE_FREAL

20.4.1 $factor.type \ factor \rightarrow \mathbf{num} \ \{\{factor.type = \mathbf{num}.type\}\}$

20.5.1 $factor.type \ factor \rightarrow (\ expression) \ \{\{factor.type = expression.type\}\}$

20.6.1 $factor.type \ factor \rightarrow \mathbf{not} \ factor \ \{\{factor.type = factor_1.type \text{ unless } factor_1.type \notin \{\text{TYPE_BOOL}, \text{ERR}\}\}\}$

21.1 void $sign \rightarrow +$

21.2 void $sign \rightarrow -$

Implementation

The main implementation problem was creating the scope checking data structure. As discussed in class, there are two types of nodes, sterile (blue) nodes and nonsterile (green) nodes. Each function is a green node because it can have children (other functions). Each declaration puts the new node at the end of the doubly linked list, and the special “eye” pointer points to that node. Each green node added is pushed onto a “green stack”, and whenever a scope ends, the current green node’s *next* are assigned to its *child* pointer, which hide its scope from the global scope, and it is popped off of the green stack. This effectively manages the scope. When a token is looked up with *gettype()*, it is looked up from the eye until the *program* node. Each green node also has program arguments as a string so that they could be matched with an *expression_list* when the function is called.

Discussion and Conclusions

Implementing this project was very tedious and required lots of checking and verification, since the next step involves the previous. Many test cases help verify this.

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 SYNERR: Expected "program" but found identifier "abcdefghij"
3 2      abcdefghijk
4 LEXERR: ID too long:      abcdefghijk
5 3      @
6 LEXERR: Unrecognized symbol:      @
7 4
8 5      12345678901
9 LEXERR: Int too long:      12345678901
10 6      1234567890
11 7
12 8      12345.3
13 9      123456.3
14 LEXERR: Mantissa too long:      123456.3
```

```

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

```

Listing 3: bad_lex.tok

1	1	abcdefghij	1	0x55420f0
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	0x5544a00	
23	31	#	99	2	
24	32	3.4E+	99	10	
25	32	;	33	59	
26	33	3.E4	99	11	
27	33	+	7	56	
28	33	;	33	59	
29	34	34	6	34	
30	34	E	1	0x5545010	
31	34	+	7	56	
32	34	-	7	57	
33	34	;	33	59	
34	35	E3	1	0x55452b0	
35	35	.	25	46	
36	35	4	6	4	
37	35	+	7	56	
38	35	;	33	59	
39	37	abcdefghijklmnopqrstuvwxyz0123456789abcdefghijklmnopqrstuvwxyz012345678			99
		1			
40	39	EOF	0	0	

bad_syn

Listing 4: bad_syn.pas

```
1 program test(input, output);
2   var a : integer;
3   var b : array[0..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 5: bad_syn.list

```
1 1      program test(input, output);
2 2      var a : integer;
3 3      var b : array[0..a] of real;
4 SEMERR: Arrays must start at 1
5 SYNERR: Expected "NUM_INTEGER" but found identifier "a"
6 4      function f1(a : integer, x : int) : real ;
7 SYNERR: Expected one of ";" or ")" but found ","
8 5      var c : integer;
9 6      function f2(p : integer) : integer ;
10 7      var d : real;
11 8      begin
12 9          f2 := 5 + p;
13 10         if p > d then
14 SEMERR: Invalid operands to >: integer and real
15 11             if p = 5 then
16 12                 f2 := f + 10
17 SEMERR: Variable "f" was not declared in this scope
18 13             else
19 14                 f2 := 100
20 15         end
21 16     begin
22 SYNERR: Expected ";" but found "begin"
23 17         f1 := f2(a) * x
24 SYNERR: Expected one of "begin" or "function" but found identifier "f1"
25 18     end
```

```

26 19          function f2(a : real) : real ;
27 20          var b : integer;
28 21          begin
29 22              f3 := 10 * f1(a, 2.3);
30 SEMERR: Variable "f3" was not declared in this scope
31 SEMERR: Function args for "f1" do not match: expected (integer) but got (integer, real
   )
32 23          end
33 SYNERR: Expected one of "begin", "identifier", "if", or "while" but found "end"
34 24          begin
35 SYNERR: Expected ";" but found "begin"
36 25              f1(5, 3.2)
37 SYNERR: Expected one of "begin" or "function" but found identifier "f1"
38 26          end.
39 27          begin
40 28
41 29          end
42 SYNERR: Expected "." but found "EOF"

```

Listing 6: bad_syn.sym

1	a	0	integer
2	b	4	array of real
3	c	0	integer
4	d	0	real

Listing 7: bad_syn.tok

				33	4	x	1	0x5544b90
				34	4	:	32	58
1	1	program	20	0	35	4	int	1
2	1	test	1	0x5543220	36	4)	28
3	1	(27	40	37	4	:	32
4	1	input	1	0x55433d0	38	4	real	21
5	1	,	34	44	39	4	;	33
6	1	output	1	0x55435d0	40	5	var	23
7	1)	28	41	41	5	c	1
8	1	;	33	59	42	5	:	32
9	2	var	23	0	43	5	integer	17
10	2	a	1	0x5543960	44	5	;	33
11	2	:	32	58	45	6	function	15
12	2	integer	17	0	46	6	f2	1
13	2	;	33	59	47	6	(27
14	3	var	23	0	48	6	p	1
15	3	b	1	0x5543e30	49	6	:	32
16	3	:	32	58	50	6	integer	17
17	3	array	10	0	51	6)	28
18	3	[29	91	52	6	:	32
19	3	0	6	0	53	6	integer	17
20	3	..	26	26	54	6	;	33
21	3	a	1	0x5543960	55	7	var	23
22	3]	30	93	56	7	d	1
23	3	of	19	0	57	7	:	32
24	3	real	21	0	58	7	real	21
25	3	;	33	59	59	7	;	33
26	4	function		15	60	8	begin	11
27	4	f1	1	0x5544710	61	9	f2	1
28	4	(27	40	62	9	:=	31
29	4	a	1	0x5543960	63	9	5	6
30	4	:	32	58	64	9	+	7
31	4	integer	17	0	65	9	p	1
32	4	,	34	44				

66	9	;	33	59	102	19	real	21	0
67	10	if	16	0	103	19)	28	41
68	10	p	1	0x55459d0	104	19	:	32	58
69	10	>	4	45	105	19	real	21	0
70	10	d	1	0x55460d0	106	19	;	33	59
71	10	then	22	0	107	20	var	23	0
72	11	if	16	0	108	20	b	1	0x5543e30
73	11	p	1	0x55459d0	109	20	:	32	58
74	11	=	4	43	110	20	integer	17	0
75	11	5	6	5	111	20	;	33	59
76	11	then	22	0	112	21	begin	11	0
77	12	f2	1	0x5545820	113	22	f3	1	0x5549310
78	12	:=	31	0	114	22	:=	31	0
79	12	f	1	0x5547590	115	22	10	6	10
80	12	+	7	56	116	22	*	3	50
81	12	10	6	10	117	22	f1	1	0x5544710
82	13	else	13	0	118	22	(27	40
83	14	f2	1	0x5545820	119	22	a	1	0x5543960
84	14	:=	31	0	120	22	,	34	44
85	14	100	6	100	121	22	2.3	5	2
86	15	end	14	0	122	22)	28	41
87	16	begin	11	0	123	22	;	33	59
88	17	f1	1	0x5544710	124	23	end	14	0
89	17	:=	31	0	125	24	begin	11	0
90	17	f2	1	0x5545820	126	25	f1	1	0x5544710
91	17	(27	40	127	25	(27	40
92	17	a	1	0x5543960	128	25	5	6	5
93	17)	28	41	129	25	,	34	44
94	17	*	3	50	130	25	3.2	5	3
95	17	x	1	0x5544b90	131	25)	28	41
96	18	end	14	0	132	26	end	14	0
97	19	function		15 0	133	26	.	25	46
98	19	f2	1	0x5545820	134	27	begin	11	0
99	19	(27	40	135	29	end	14	0
100	19	a	1	0x5543960	136	30	EOF	0	0
101	19	:	32	58					

bad_sem

Listing 8: bad_sem.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    out := 0
13 end.
```

Listing 9: bad_sem.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 SEMERR: Variable "out" was not declared in this scope
12 SEMERR: Function "read" not declared in this scope
13 11         out := write(gcd(x, y));
14 SEMERR: Variable "out" was not declared in this scope
15 SEMERR: Function "write" not declared in this scope
16 12         out := 0
17 SEMERR: Variable "out" was not declared in this scope
18 13      end.
```

Listing 10: bad_sem.sym

```

1 x      0      integer
2 y      4      integer
```

Listing 11: bad_sem.tok

1	1	program	20	0	16	2	:	32	58
2	1	example	1	0x55431c0	17	2	integer	17	0
3	1	(27	40	18	2	;	33	59
4	1	input	1	0x5543370	19	3	function	15	0
5	1	,	34	44	20	3	gcd	1	0x5544110
6	1	output	1	0x5543570	21	3	(27	40
7	1)	28	41	22	3	a	1	0x55442c0
8	1	;	33	59	23	3	:	32	58
9	2	var	23	0	24	3	integer	17	0
10	2	x	1	0x55438b0	25	3	;	33	59
11	2	:	32	58	26	3	b	1	0x55445b0
12	2	integer	17	0	27	3	:	32	58
13	2	;	33	59	28	3	integer	17	0
14	2	var	23	0	29	3)	28	41
15	2	y	1	0x5543ce0	30	3	:	32	58
					31	3	integer	17	0

32	3	;	33	59	58	10	read	1	0x5545f90
33	4	begin	11	0	59	10	(27	40
34	5	if	16	0	60	10	x	1	0x55438b0
35	5	b	1	0x55445b0	61	10	,	34	44
36	5	=	4	43	62	10	y	1	0x5543ce0
37	5	0	6	0	63	10)	28	41
38	5	then	22	0	64	10	;	33	59
39	5	gcd	1	0x5544110	65	11	out	1	0x5545d40
40	5	:=	31	0	66	11	:=	31	0
41	5	a	1	0x55442c0	67	11	write	1	0x55465f0
42	6	else	13	0	68	11	(27	40
43	6	gcd	1	0x5544110	69	11	gcd	1	0x5544110
44	6	:=	31	0	70	11	(27	40
45	6	gcd	1	0x5544110	71	11	x	1	0x55438b0
46	6	(27	40	72	11	,	34	44
47	6	b	1	0x55445b0	73	11	y	1	0x5543ce0
48	6	,	34	44	74	11)	28	41
49	6	a	1	0x55442c0	75	11)	28	41
50	6	mod	3	53	76	11	;	33	59
51	6	b	1	0x55445b0	77	12	out	1	0x5545d40
52	6)	28	41	78	12	:=	31	0
53	7	end	14	0	79	12	0	6	0
54	7	;	33	59	80	13	end	14	0
55	9	begin	11	0	81	13	.	25	46
56	10	out	1	0x5545d40	82	14	EOF	0	0
57	10	:=	31	0					

```
1
2 program test (input, output);
3   var a : integer;
4   var b : real;
5   var c : array [1..2] of integer;
6
7   function fun1(x:integer; y:real;
8               z:array [1..2] of integer;
9               q: real) : integer;
10      var d: integer;
11      begin
12         a:= 2;
13         z[a] := 4;
14         c[2] := 3;
15         fun1 := c[1]
16      end;
17
18   function fun2(x: integer; y: integer) : real;
19      var e: real;
20
21   function fun3(n: integer; z: real) : integer;
22      var e: integer;
23      begin
24         a:= e;
25         e:= c[e];
26         fun3 := 3
27      end;
28
29      begin
30         a:= fun1(x, e, c, b);
31         x:= fun3(c[1], e);
32         e := e + 4.44;
33         a:= (a mod y) div x;
34         while ((a >= 4) and ((b <= e)
35                             or (not (a = c[a])))) do
36             begin
37                 a:= c[a] + 1
38             end;
39         fun2 := 2.5
40      end;
41
42 begin
43   b:= fun2(c[4], c[5]);
44   b:= fun2(c[4], 2);
45   if (a < 2) then a:= 1 else a := a + 2;
46   if (b > 4.2) then a := c[a]
47 end.
```

```

1  1
2  2    program test (input, output);
3  3      var a : integer;
4  4      var b : real;
5  5      var c : array [1..2] of integer;
6  6
7  7      function fun1(x:integer; y:real;
8  8          z:array [1..2] of integer;
9  9          q: real) : integer;
10 10      var d: integer;
11 11      begin
12 12          a:= 2;
13 13          z[a] := 4;
14 14          c[2] := 3;
15 15          fun1 := c[1]
16 16      end;
17 17
18 18      function fun2(x: integer; y: integer) : real;
19 19      var e: real;
20 20
21 21      function fun3(n: integer; z: real) : integer;
22 22      var e: integer;
23 23      begin
24 24          a:= e;
25 25          e:= c[e];
26 26          fun3 := 3
27 27      end;
28 28
29 29      begin
30 30          a:= fun1(x, e, c, b);
31 31          x:= fun3(c[1], e);
32 32          e := e + 4.44;
33 33          a:= (a mod y) div x;
34 34          while ((a >= 4) and ((b <= e)
35 35              or (not (a = c[a])))) do
36 36              begin
37 37                  a:= c[a] + 1
38 38              end;
39 39          fun2 := 2.5
40 40      end;
41 41
42 42      begin
43 43          b:= fun2(c[4], c[5]);
44 44          b:= fun2(c[4],2);
45 45          if (a < 2) then a:= 1 else a := a + 2;
46 46          if (b > 4.2) then a := c[a]
47 47      end.

```

1	a	0	integer
2	b	4	real
3	c	12	array of integer
4	d	0	integer
5	e	0	real
6	e	0	integer

Listing 15: cor34.tok

				51	8	integer	17	0	
1	2	program	20	0	52	8	;	33	59
2	2	test	1	0x5543210	53	9	q	1	0x5545c60
3	2	(27	40	54	9	:	32	58
4	2	input	1	0x5543410	55	9	real	21	0
5	2	,	34	44	56	9)	28	41
6	2	output	1	0x5543610	57	9	:	32	58
7	2)	28	41	58	9	integer	17	0
8	2	;	33	59	59	9	;	33	59
9	3	var	23	0	60	10	var	23	0
10	3	a	1	0x55439a0	61	10	d	1	0x55462c0
11	3	:	32	58	62	10	:	32	58
12	3	integer	17	0	63	10	integer	17	0
13	3	;	33	59	64	10	;	33	59
14	4	var	23	0	65	11	begin	11	0
15	4	b	1	0x5543e70	66	12	a	1	0x55439a0
16	4	:	32	58	67	12	:=	31	0
17	4	real	21	0	68	12	2	6	2
18	4	;	33	59	69	12	;	33	59
19	5	var	23	0	70	13	z	1	0x55454b0
20	5	c	1	0x5544340	71	13	[29	91
21	5	:	32	58	72	13	a	1	0x55439a0
22	5	array	10	0	73	13]	30	93
23	5	[29	91	74	13	:=	31	0
24	5	1	6	1	75	13	4	6	4
25	5	..	26	26	76	13	;	33	59
26	5	2	6	2	77	14	c	1	0x5544340
27	5]	30	93	78	14	[29	91
28	5	of	19	0	79	14	2	6	2
29	5	integer	17	0	80	14]	30	93
30	5	;	33	59	81	14	:=	31	0
31	7	function		15	82	14	3	6	3
32	7	fun1	1	0x5544cc0	83	14	;	33	59
33	7	(27	40	84	15	fun1	1	0x5544cc0
34	7	x	1	0x5544e70	85	15	:=	31	0
35	7	:	32	58	86	15	c	1	0x5544340
36	7	integer	17	0	87	15	[29	91
37	7	;	33	59	88	15	1	6	1
38	7	y	1	0x5545160	89	15]	30	93
39	7	:	32	58	90	16	end	14	0
40	7	real	21	0	91	16	;	33	59
41	7	;	33	59	92	18	function		15
42	8	z	1	0x55454b0	93	18	fun2	1	0x5547a00
43	8	:	32	58	94	18	(27	40
44	8	array	10	0	95	18	x	1	0x5544e70
45	8	[29	91	96	18	:	32	58
46	8	1	6	1	97	18	integer	17	0
47	8	..	26	26	98	18	;	33	59
48	8	2	6	2	99	18	y	1	0x5545160
49	8]	30	93	100	18	:	32	58
50	8	of	19	0	101	18	integer	17	0

102	18)	28	41	162	31	:=	31	0
103	18	:	32	58	163	31	fun3	1	0x55488a0
104	18	real	21	0	164	31	(27	40
105	18	;	33	59	165	31	c	1	0x5544340
106	19	var	23	0	166	31	[29	91
107	19	e	1	0x55483d0	167	31	1	6	1
108	19	:	32	58	168	31]	30	93
109	19	real	21	0	169	31	,	34	44
110	19	;	33	59	170	31	e	1	0x55483d0
111	21	function		15 0	171	31)	28	41
112	21	fun3	1	0x55488a0	172	31	;	33	59
113	21	(27	40	173	32	e	1	0x55483d0
114	21	n	1	0x5548a50	174	32	:=	31	0
115	21	:	32	58	175	32	e	1	0x55483d0
116	21	integer	17	0	176	32	+	7	56
117	21	;	33	59	177	32	4.44	5	4
118	21	z	1	0x55454b0	178	32	;	33	59
119	21	:	32	58	179	33	a	1	0x55439a0
120	21	real	21	0	180	33	:=	31	0
121	21)	28	41	181	33	(27	40
122	21	:	32	58	182	33	a	1	0x55439a0
123	21	integer	17	0	183	33	mod	3	53
124	21	;	33	59	184	33	y	1	0x5545160
125	22	var	23	0	185	33)	28	41
126	22	e	1	0x55483d0	186	33	div	3	52
127	22	:	32	58	187	33	x	1	0x5544e70
128	22	integer	17	0	188	33	;	33	59
129	22	;	33	59	189	34	while	24	0
130	23	begin	11	0	190	34	(27	40
131	24	a	1	0x55439a0	191	34	(27	40
132	24	:=	31	0	192	34	a	1	0x55439a0
133	24	e	1	0x55483d0	193	34	>=	4	46
134	24	;	33	59	194	34	4	6	4
135	25	e	1	0x55483d0	195	34)	28	41
136	25	:=	31	0	196	34	and	3	54
137	25	c	1	0x5544340	197	34	(27	40
138	25	[29	91	198	34	(27	40
139	25	e	1	0x55483d0	199	34	b	1	0x5543e70
140	25]	30	93	200	34	<=	4	42
141	25	;	33	59	201	34	e	1	0x55483d0
142	26	fun3	1	0x55488a0	202	34)	28	41
143	26	:=	31	0	203	35	or	2	55
144	26	3	6	3	204	35	(27	40
145	27	end	14	0	205	35	not	18	0
146	27	;	33	59	206	35	(27	40
147	29	begin	11	0	207	35	a	1	0x55439a0
148	30	a	1	0x55439a0	208	35	=	4	43
149	30	:=	31	0	209	35	c	1	0x5544340
150	30	fun1	1	0x5544cc0	210	35	[29	91
151	30	(27	40	211	35	a	1	0x55439a0
152	30	x	1	0x5544e70	212	35]	30	93
153	30	,	34	44	213	35)	28	41
154	30	e	1	0x55483d0	214	35)	28	41
155	30	,	34	44	215	35)	28	41
156	30	c	1	0x5544340	216	35)	28	41
157	30	,	34	44	217	35	do	12	0
158	30	b	1	0x5543e70	218	36	begin	11	0
159	30)	28	41	219	37	a	1	0x55439a0
160	30	;	33	59	220	37	:=	31	0
161	31	x	1	0x5544e70	221	37	c	1	0x5544340

222	37	[29	91	259	44	2	6	2
223	37	a	1	0x55439a0	260	44)	28	41
224	37]	30	93	261	44	;	33	59
225	37	+	7	56	262	45	if	16	0
226	37	1	6	1	263	45	(27	40
227	38	end	14	0	264	45	a	1	0x55439a0
228	38	;	33	59	265	45	<	4	41
229	39	fun2	1	0x5547a00	266	45	2	6	2
230	39	:=	31	0	267	45)	28	41
231	39	2.5	5	2	268	45	then	22	0
232	40	end	14	0	269	45	a	1	0x55439a0
233	40	;	33	59	270	45	:=	31	0
234	42	begin	11	0	271	45	1	6	1
235	43	b	1	0x5543e70	272	45	else	13	0
236	43	:=	31	0	273	45	a	1	0x55439a0
237	43	fun2	1	0x5547a00	274	45	:=	31	0
238	43	(27	40	275	45	a	1	0x55439a0
239	43	c	1	0x5544340	276	45	+	7	56
240	43	[29	91	277	45	2	6	2
241	43	4	6	4	278	45	;	33	59
242	43]	30	93	279	46	if	16	0
243	43	,	34	44	280	46	(27	40
244	43	c	1	0x5544340	281	46	b	1	0x5543e70
245	43	[29	91	282	46	>	4	45
246	43	5	6	5	283	46	4.2	5	4
247	43]	30	93	284	46)	28	41
248	43)	28	41	285	46	then	22	0
249	43	;	33	59	286	46	a	1	0x55439a0
250	44	b	1	0x5543e70	287	46	:=	31	0
251	44	:=	31	0	288	46	c	1	0x5544340
252	44	fun2	1	0x5547a00	289	46	[29	91
253	44	(27	40	290	46	a	1	0x55439a0
254	44	c	1	0x5544340	291	46]	30	93
255	44	[29	91	292	47	end	14	0
256	44	4	6	4	293	47	.	25	46
257	44]	30	93	294	48	EOF	0	0
258	44	,	34	44					

Appendix II: Sample Inputs and Outputs

Listing 16: 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     return f == NULL ? -1 : 0;
56 }
```

```
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 17: 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 18: 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 #define TOKEN_TIMES 50
60 #define TOKEN_RDIV 51
61 #define TOKEN_IDIV 52
62 #define TOKEN_MOD 53
63 #define TOKEN_AND 54

```

```

64
65 #define TOKEN_OR 55
66 #define TOKEN_PLUS 56
67 #define TOKEN_MINUS 57
68
69 #define LEXERR 99
70
71 #define TOKEN_WHITESPACE 1024
72 #define TOKEN_NEWLINE 1025
73
74 #define LEXERR_ID_TOO_LONG 1
75 #define LEXERR_UNREC_SYM 2
76 #define LEXERR_INT_TOO_LONG 3
77 #define LEXERR_MANTIS_TOO_LONG 4
78 #define LEXERR_FRAC_TOO_LONG 5
79 #define LEXERR_LEADING_ZERO 6
80 #define LEXERR_TRAILING_ZERO 7
81 #define LEXERR_LINE_TOO_LONG 8
82 #define LEXERR_EXP_TOO_LONG 9
83 #define LEXERR_NO_EXP 10
84 #define LEXERR_NO_FRAC 11
85
86
87
88 #define ID_STRLEN 10
89
90 #ifndef LINELEN
91 #define LINELEN 72
92 #endif
93
94 /* forward declarations */
95 typedef struct machine *machine_t;
96 typedef struct lex_state *lex_state_t;
97
98 #endif

```

Listing 19: 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 **
13     to_write)
14 {
15     char *loc, *buf;
16     if (sdup(f, &buf) < 0) {
17         return -1;
18     }
19     *to_write = buf;
20     loc = strrchr(buf, '.');
21     if (loc) {
22         *loc = 0;
23     } 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 20: 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 21: 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",
33                    line, lexeme, t.type, t.val.ptr);
34 } else {
35     return fprintf(f, "%d\t%s\t %d\t%d\n",
36                    line, lexeme, t.type, t.val.attr);
37 }
38 }
39
40
41 const char *token2str(int token)
42 {
43     switch (token) {
44     case TOKEN_ADDOP: {
45         return "ADDOP";
46     }
47     case TOKEN_ARRAY: return "array";
48     case TOKEN_ASSIGN: return "!=";
49     case TOKEN_BEGIN: return "begin";
50     case TOKEN_COLON: return ":";
51     case TOKEN_COMMA: return ",";
52     case TOKEN_DO: return "do";
53     case TOKEN_ELLIPSIS: return "...";
54     case TOKEN_ELSE: return "else";
55     case TOKEN_END: return "end";
56     case TOKEN_EOF: return "EOF";
57     case TOKEN_FUNCTION: return "function";
58     case TOKEN_ID: return "identifier";
59     case TOKEN_IF: return "if";
60     case TOKEN_INTEGER: return "integer";
61     case TOKEN_LBRACKET: return "[";
62     case TOKEN_LPAREN: return "(";
63     case TOKEN_MULOP: {
64         return "MULOP";
65     }
66     case TOKEN_NOT: return "not";
67     case TOKEN_NUM_INTEGER: return "NUM_INTEGER";
68     case TOKEN_NUM_REAL: return "NUM_REAL";
69     case TOKEN_OF: return "of";
70     case TOKEN_PERIOD: return ".";
71     case TOKEN_PROGRAM: return "program";
72     case TOKEN_RBRACKET: return "]";
73     case TOKEN_REAL: return "real";
74     case TOKEN_RELOP: {
75         return "RELOP";
76     }
77     case TOKEN_RPAREN: return ")";
78     case TOKEN_SEMICOLON: return ";";
79     case TOKEN_SIGN: return "+ or -";
80     case TOKEN_THEN: return "then";
81     case TOKEN_VAR: return "var";
82     case TOKEN_WHILE: return "while";
83     case LEXERR: return "LEXERR";
84     }
85     return "UNKNOWN";
86 }
87
88 const char* relop2str(int attr)
89 {
90     switch (attr) {

```

```

91     case TOKEN_LT: return "<";
92     case TOKEN_LEQ: return "<=";
93     case TOKEN_EQ: return "=";
94     case TOKEN_NEQ: return "<>";
95     case TOKEN_GT: return ">";
96     case TOKEN_GEQ: return ">=";
97     default: return "Unknown relop";
98     }
99 }

```

Listing 22: 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 const char* token2str(int token);
31 const char* relop2str(int attr);
32 #endif

```

Listing 23: 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 int idres_insert(struct idres **list, char* lexeme
26     , struct token token)
27 {
28     int ret = 0;
29     struct idres *root = malloc(sizeof(*root));
30     root->lexeme = lexeme;
31     root->type = 0;
32     root->token = token;
33     root->next = *list;
34     *list = root;
35     return ret;
36 }
37 int idres_add_rw(struct idres **list, char*
38     c_lexeme, int type, int attr)
39 {
40     char *lexeme;
41     struct token tok;
42     if (sdup(c_lexeme, &lexeme) < 0) {
43         return -1;
44     }
45     token_add(&tok, type, attr);
46     return idres_insert(list, lexeme, tok);
47 }
48 int idres_add_id(struct idres **list, char*
49     c_lexeme)
50 {
51     char *lexeme;
52     struct token tok;
53     if (sdup(c_lexeme, &lexeme) < 0) {
54         return -1;
55     }
56     token_id(&tok, lexeme);
57     return idres_insert(list, lexeme, tok);
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 int idres_lookup(struct idres **list, void* ptr,
72     struct idres **ret)
73 {
74     struct idres *node = *list;
75     while (node != NULL) {
76         if (ptr == node->token.val.ptr) {
77             *ret = node;
78             return 0;
79         }
80         node = node->next;
81     }
82     return -1;
83 }
84 int idres_find(struct idres *node, char *lexeme,
85     struct idres **ret)
86 {
87     while (node != NULL) {
88         if (!strcmp(lexeme, node->lexeme)) {
89             *ret = node;
90             return 0;
91         }
92         node = node->next;
93     }
94     return -1;
95 }
96 int idres_search(struct idres **list, char* lexeme
97     , struct idres **ret)
98 {
99     return idres_find(*list, lexeme, ret);
100 }
101 int idres_clean(struct idres **list)
102 {
103     while (*list != NULL) {
104         struct idres *prev = *list;
105         *list = prev->next;
106         free(prev->lexeme);
107         free(prev);
108     }
109     return 0;
110 }
111 int idres_read(const char *filename, struct idres
112     **list)
113 {
114     FILE* f = fopen(filename, "r");
115     void* addr = NULL;
116     long count;
117     char *lexeme = malloc(ID_STRLEN + 1);
118     /* strlen(lexeme) guaranteed to be ID_STRLEN
119        */
120     for (count = 0; fscanf(f, "0x%p\t%s\n", &addr,
121         lexeme) == 2; count++) {
122         idres_add_id_attr(list, lexeme, addr);
123     }
124     free(lexeme);
125     fclose(f);
126     /*return idres_balance(list, count);*/
127     return 0;

```



```

124 }
16
17
18
19 Listing 24: common/idres.h
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039

```

```

22 int idres_add_rw(struct idres **list, char* lexeme
    , int token, int attr);
23 int idres_add_id(struct idres **list, char* lexeme
    );
24
25 int idres_search(struct idres **list, char* lexeme
    , 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 25: lexer/main.c
1  /* -*- C -*-                                54
2      *                                       55
3      * main.c                               56
4      *                                       57
5      * Author: Benjamin T James            58
6      */                                    59
7                                           60
8  #include "state.h"                        61
9  #include "defs.h"                        62
10 #include "lexerr.h"                      63
11                                           64
12 int handle_line(struct lex_state *s, int line_no)
13 {
14     char *lexeme = NULL;
15     struct machine state;

```

Listing 26: lexer/state.c

```

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
17             \n");
18         return -1;
19     }
20     while (fscanf(st->res_word, "%s\t%d\t%d\n",
21         lexeme, &tok, &attr) != EOF) {
22         idres_add_rw(&st->rwords, lexeme, tok,
23             attr);
24     }
25     free(lexeme);
26     return 0;
27 }
28
29 int state_init(const char *source, const char *
30     res_word,
31     int line_len, struct lex_state *st)
32 {
33     if (init_buf(&st->buf, line_len) < 0) {
34         return -1;
35     }
36     if (open_file(source, "list", &st->list) < 0)
37     {
38         return -1;
39     }
40     if (open_file(source, "tok", &st->token) < 0)
41     {
42         return -1;
43     }
44     if (open_file(source, "sym", &st->sym) < 0) {
45         return -1;
46     }
47     st->res_word = fopen(res_word, "r");
48     if (st->res_word == NULL) {
49         fprintf(stderr, "Could not open file \"%s
50             \n\", res_word);
51         return -1;
52     }
53     st->source = fopen(source, "r");
54     if (st->source == NULL) {
55         fprintf(stderr, "Could not open file \"%s
56             \n\", source);
57         return -1;
58     }
59     st->rwords = NULL;
60     st->ids = NULL;
61     if (resword_init(st) < 0) {
62         return -1;
63     }
64 }

```

```

55     st->machines = NULL;
56     if (machine_init(&st->machines) < 0) {
57         return -1;
58     }
59     return 0;
60 }
61
62 int state_cleanup(struct lex_state *s)
63 {
64     free_buf(&s->buf);
65     fclose(s->source);
66     fclose(s->res_word);
67     fclose(s->sym);
68     fclose(s->list);
69     fclose(s->token);
70     idres_clean(&s->rwords);
71     idres_clean(&s->ids);
72     machine_clean(&s->machines);
73     return 0;
74 }

```

Listing 27: 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);

```

```

38 #endif
39
40 Listing 28: lexer/fsm.c
41
42 /* -*- C -*-
43  *
44  * fsm.c
45  *
46  * Author: Benjamin T James
47  */
48
49 #include <ctype.h>
50 #include <string.h>
51 #include "fsm.h"
52 #include "defs.h"
53 #include "util.h"
54
55 int digit_plus(struct machine *m)
56 {
57     int len = 1;
58     if (!isdigit(*m->f)) {
59         return 0;
60     }
61     m->f++;
62     while (isdigit(*m->f)) {
63         m->f++;
64         len++;
65     }
66     return len;
67 }
68
69 int fsm_relop(struct machine *m, struct lex_state
70 *ls)
71 {
72     if (*m->f == '>') {
73         m->f++;
74         if (*m->f == '=') {
75             token_add(&m->tok, TOKEN_RELOP,
76                     TOKEN_GEQ);
77         } else {
78             m->f--;
79             token_add(&m->tok, TOKEN_RELOP,
80                     TOKEN_GT);
81         }
82     } else if (*m->f == '=') {
83         token_add(&m->tok, TOKEN_RELOP, TOKEN_EQ);
84     } else if (*m->f == '<') {
85         m->f++;
86         if (*m->f == '=') {
87             token_add(&m->tok, TOKEN_RELOP,
88                     TOKEN_LEQ);
89         } else if (*m->f == '>') {
90             token_add(&m->tok, TOKEN_RELOP,
91                     TOKEN_NEQ);
92         } else {
93             m->f--;
94             token_add(&m->tok, TOKEN_RELOP,
95                     TOKEN_LT);
96         }
97     } else {
98         return 0;
99     }
100 }
101
102 }
103
104 return 1;
105
106 }
107
108 int fsm_integer(struct machine *m, struct
109 lex_state *ls)
110 {
111     char *lexeme;
112     int result, len;
113     if (!digit_plus(m)) {
114         return 0;
115     }
116     m->f--;
117     if (get_str(m->f, m->b, &lexeme) < 0) {
118         return -1;
119     }
120     len = strlen(lexeme);
121     if (len > 10) {
122         token_add(&m->tok, LEXERR,
123                 LEXERR_INT_TOO_LONG);
124     } else if ((lexeme[0] == '0' && len > 1)
125             || (lexeme[0] == '-' && lexeme[1] == '0'
126                 && len > 2)) {
127         token_add(&m->tok, LEXERR,
128                 LEXERR_LEADING_ZERO);
129     } else {
130         result = strtol(lexeme, NULL, 10);
131         token_add(&m->tok, TOKEN_NUM_INTEGER,
132                 result);
133     }
134     free(lexeme);
135     return 1;
136 }
137
138 int fsm_real(struct machine *m, struct lex_state *
139 ls)
140 {
141     char *lexeme;
142     double result;
143     int len, mantis_len = 0, frac_len = 0;
144     mantis_len = digit_plus(m);
145     if (mantis_len == 0 || *m->f != '.') {
146         return 0;
147     }
148     m->f++;
149     frac_len = digit_plus(m);
150     if (frac_len == 0) {
151         return 0;
152     }
153     m->f--;
154     if (get_str(m->f, m->b, &lexeme) < 0) {
155         return -1;
156     }
157     len = strlen(lexeme);
158     if (mantis_len > 5) {
159         token_add(&m->tok, LEXERR,
160                 LEXERR_MANTIS_TOO_LONG);
161     } else if (frac_len > 5) {
162         token_add(&m->tok, LEXERR,
163                 LEXERR_FRAC_TOO_LONG);
164     }
165 }

```

```

104     } else if (lexeme[0] == '0' || (lexeme[0] == '
105         -' && lexeme[1] == '0')) {
106         token_add(&m->tok, LEXERR,
107             LEXERR_LEADING_ZERO);
108     } else if (lexeme[len-1] == '0') {
109         token_add(&m->tok, LEXERR,
110             LEXERR_TRAILING_ZERO);
111     } else {
112         result = strtod(lexeme, NULL);
113         token_add(&m->tok, TOKEN_NUM_REAL, (int)
114             result);
115     }
116     free(lexeme);
117     return 1;
118 }
119 int fsm_long_real(struct machine *m, struct
120     lex_state *ls)
121 {
122     char *lexeme;
123     double result;
124     int len, tz = 0, mantis_len = 0, frac_len = 0,
125     exp_len = 0;
126     mantis_len = digit_plus(m);
127     if (mantis_len == 0 || *m->f != '.') {
128         return 0;
129     }
130     m->f++;
131     frac_len = digit_plus(m);
132     if (/* frac_len == 0 || */ *m->f != 'E') {
133         return 0;
134     }
135     if (get_str(m->f - 1, m->b, &lexeme) < 0) {
136         return -1;
137     }
138     if (lexeme[strlen(lexeme) - 1] == '0') {
139         tz = 1; /* set trailing zero flag to 1 */
140     }
141     free(lexeme);
142     m->f++;
143     if (*m->f == '-' || *m->f == '+') {
144         m->f++;
145     }
146     exp_len = digit_plus(m);
147     /* if (exp_len == 0) { */
148     /* /* err */ */
149     /* return 0; */
150     /* } */
151     m->f--;
152     if (get_str(m->f, m->b, &lexeme) < 0) {
153         return -1;
154     }
155     len = strlen(lexeme);
156     if (frac_len == 0) {
157         token_add(&m->tok, LEXERR, LEXERR_NO_FRAC)
158         ;
159     } else if (exp_len == 0) {
160         token_add(&m->tok, LEXERR, LEXERR_NO_EXP);
161     } else if (mantis_len > 5) {
162         token_add(&m->tok, LEXERR,
163             LEXERR_MANTIS_TOO_LONG);
164     } else if (frac_len > 5) {
165         token_add(&m->tok, LEXERR,
166             LEXERR_FRAC_TOO_LONG);
167     } else if (exp_len > 2) {
168         token_add(&m->tok, LEXERR,
169             LEXERR_EXP_TOO_LONG);
170     } else if (lexeme[0] == '0' || (lexeme[0] == '
171         -' && lexeme[1] == '0')) {
172         token_add(&m->tok, LEXERR,
173             LEXERR_LEADING_ZERO);
174     } else if (tz || lexeme[len-1] == '0') {
175         token_add(&m->tok, LEXERR,
176             LEXERR_TRAILING_ZERO);
177     } else {
178         result = strtod(lexeme, NULL);
179         token_add(&m->tok, TOKEN_NUM_REAL, (int)
180             result);
181     }
182     free(lexeme);
183     return 1;
184 }
185 int fsm_addop(struct machine *m, struct lex_state
186     *ls)
187 {
188     if (*m->f == '+') {
189         token_add(&m->tok, TOKEN_SIGN, TOKEN_PLUS)
190         ;
191         return 1;
192     } else if (*m->f == '-') {
193         token_add(&m->tok, TOKEN_SIGN, TOKEN_MINUS)
194         );
195         return 1;
196     } else if (*m->f == 'o') {
197         m->f++;
198         if (*m->f == 'r') {
199             token_add(&m->tok, TOKEN_ADDOP,
200                 TOKEN_OR);
201             return 1;
202         }
203     }
204     return 0;
205 }
206 int fsm_mulop(struct machine *m, struct lex_state
207     *ls)
208 {
209     if (*m->f == '*') {
210         token_add(&m->tok, TOKEN_MULOP,
211             TOKEN_TIMES);
212         return 1;
213     } else if (*m->f == '/') {
214         token_add(&m->tok, TOKEN_MULOP, TOKEN_RDIV)
215         );
216         return 1;
217     } else if (*m->f == 'd') {
218         m->f++;
219         if (*m->f == 'i' && m->f++ && *m->f == 'v'
220             ) {
221             token_add(&m->tok, TOKEN_MULOP,

```

```

201         TOKEN_IDIV);
202     return 1;
203 }
204 } else if (*m->f == 'm') {
205     m->f++;
206     if (*m->f == 'o' && m->f++ && *m->f == 'd') {
207         token_add(&m->tok, TOKEN_MULOP,
208             TOKEN_MOD);
209         return 1;
210     }
211 } else if (*m->f == 'a') {
212     m->f++;
213     if (*m->f == 'n' && m->f++ && *m->f == 'd') {
214         token_add(&m->tok, TOKEN_MULOP,
215             TOKEN_AND);
216         return 1;
217     }
218 }
219
220 int fsm_catchall(struct machine *m, struct
221     lex_state *ls)
222 {
223     switch (*m->f) {
224     case '[':
225         token_add(&m->tok, TOKEN_LBRACKET, *m->f);
226         return 1;
227     case ']':
228         token_add(&m->tok, TOKEN_RBRACKET, *m->f);
229         return 1;
230     case '(':
231         token_add(&m->tok, TOKEN_LPAREN, *m->f);
232         return 1;
233     case ')':
234         token_add(&m->tok, TOKEN_RPAREN, *m->f);
235         return 1;
236     case ',':
237         token_add(&m->tok, TOKEN_COMMA, *m->f);
238         return 1;
239     case ';':
240         token_add(&m->tok, TOKEN_SEMICOLON, *m->f);
241         ;
242         return 1;
243     default:
244         break;
245     }
246     if (*m->f == '.') {
247         m->f++;
248         if (*m->f == '.') {
249             token_add(&m->tok, TOKEN_ELLIPSIS,
250                 TOKEN_ELLIPSIS);
251         } else {
252             m->f--;
253             token_add(&m->tok, TOKEN_PERIOD, '.');
254             return 1;
255         }
256     } else if (*m->f == ':') {
257         m->f++;
258         if (*m->f == '=') {
259             token_add(&m->tok, TOKEN_ASSIGN, 0);
260         } else {
261             m->f--;
262             token_add(&m->tok, TOKEN_COLON, ':');
263         }
264         return 1;
265     }
266     return 0;
267 }
268
269 int fsm_idres(struct machine *m, struct lex_state
270     *ls)
271 {
272     if (isalpha(*m->f)) {
273         int len;
274         char *lexeme;
275         struct idres *result;
276         m->f++;
277         while (isalnum(*m->f)) {
278             m->f++;
279         }
280         m->f--;
281
282         if (get_str(m->f, m->b, &lexeme) < 0) {
283             return -1;
284         }
285         len = strlen(lexeme);
286         if (len > ID_STRLEN) {
287             m->tok.type = LEXERR;
288             m->tok.is_id = 0;
289             m->tok.val.attr = LEXERR_ID_TOO_LONG;
290         } else if (idres_search(&ls->rwords,
291             lexeme, &result) == 0) {
292             m->tok = result->token;
293         } else if (idres_search(&ls->ids, lexeme,
294             &result) == 0) {
295             m->tok = result->token;
296         } else {
297             idres_add_id(&ls->ids, lexeme);
298             m->tok = ls->ids->token;
299         }
300         free(lexeme);
301         return 1;
302     }
303     return 0;
304 }
305
306 int fsm_unrecognized_symbol(struct machine *m,
307     struct lex_state *ls)
308 {
309     m->tok.type = LEXERR;
310     m->tok.val.attr = LEXERR_UNREC_SYM;
311     return 1;
312 }
313
314 int fsm_newline(struct machine *m, struct
315     lex_state *ls)
316 {
317     if (*m->f == '\r') {

```

```

309     m->f++;
310     if (*m->f == '\n') {
311         m->tok.type = TOKEN_NEWLINE;
312         return 1;
313     }
314     } else if (*m->f == '\n') {
315         m->f++;
316         m->tok.type = TOKEN_NEWLINE;
317         return 1;
318     }
319     return 0;
320 }
321 int fsm_whitespace(struct machine *m, struct
lex_state *ls)
322 {
323     if (*m->f == ' ' || *m->f == '\t') {
324         m->f++;
325         while (*m->f == ' ' || *m->f == '\t') {
326             m->f++;
327         }
328         m->f--;
329         m->tok.type = TOKEN_WHITESPACE;
330         return 1;
331     }
332     return 0;
333 }
334 }

```

Listing 29: 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,
struct lex_state *ls);
15  int fsm_whitespace(struct machine *m, struct
lex_state *ls);
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

```

Listing 30: 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:
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     fprintf(listing, "\t\t%s\n", lexeme);
51     return 0;

```

```

52 }
38 machine_add(list, fsm_catchall);
39 machine_add(list, fsm_relop);
40
41 Listing 31: lexer/lexerr.h
42
43 /* -*- C -*-
44 *
45 * lexerr.h
46 *
47 * Author: Benjamin T James
48 */
49
50 #ifndef LEXERR_H
51 #define LEXERR_H
52
53 #include "machine.h"
54
55 int print_error(FILE* listing, int err, const char
56 *lexeme);
57
58 #endif

```

```

41 Listing 32: lexer/machine.c
42
43 /* -*- C -*-
44 *
45 * machine.c
46 *
47 * Author: Benjamin T James
48 */
49
50 #include "machine.h"
51 #include "fsm.h"
52 #include "util.h"
53
54 int machine_iter(struct lex_state *ls, struct
55 machine *state, char **out_str)
56 {
57     int ret;
58     struct machine *m = ls->machines;
59     for (; m != NULL; m = m->next) {
60         m->b = state->f;
61         m->f = m->b;
62         m->tok.is_id = 0;
63         ret = m->call(m, ls);
64         if (ret == 1) {
65             state->tok = m->tok;
66             state->f = m->f + 1;
67             state->b = state->f;
68             return get_str(m->f, m->b, out_str);
69         } else if (ret == -1) {
70             return -1;
71         }
72     }
73     return -1;
74 }
75
76 int machine_init(struct machine **list)
77 {
78     machine_add(list, fsm_unrecognized_symbol);
79     machine_add(list, fsm_newline);
80 }

```

```

41 Listing 33: lexer/machine.h
42
43 /* -*- C -*-
44 *
45 * machine.h
46 *
47 * Author: Benjamin T James
48 */
49
50 #ifndef MACHINE_H
51 #define MACHINE_H
52
53 #include "defs.h"
54 #include "token.h"
55 #include "state.h"
56
57 struct machine {
58     /* returns 1 on success, 0 on failure */
59     int (*call)(struct machine *m, lex_state_t ls)
60     ;
61 }

```

```

18
19     char *f;
20     char *b;
21     struct token tok;
22     struct machine *next;
23 };
24
25 /* interface for the lexer */
26 int machine_iter(lex_state_t ls, struct machine *
    state, char **out_str);
27
28 int machine_init(struct machine **list);
29 int machine_add(struct machine **list,
30     int (*func)(struct machine *m, struct
        lex_state *ls));
31
32 int machine_clean(struct machine **list);
33
34 #endif

```

Listing 34: parser/main.c

```

1 #include <stdio.h>
2 #include "parser.h"
3 #include "tokenizer.h"
4
5 int main(int argc, char **argv)
6 {
7     struct parser parser;
8     if (argc != 6) {
9         fprintf(stderr, "Usage: %s symbols tokens
        listing_in address_out listing_out\n",
            *argv);
10        return -1;
11    }
12    parser_init(&parser, argv[1], argv[2], argv
        [3], argv[4], argv[5]);
13    /* idres_print(stdout, &parser.symbols); */
14    parse(&parser);
15    parser_cleanup(&parser);
16    return 0;
17 }

```

Listing 35: parser/tokenizer.h

```

1 #ifndef TOKENIZER_H
2 #define TOKENIZER_H
3
4 #include "token.h"
5 #include "defs.h"
6 #include "parser.h"
7
8 int next_token(struct parser *p);
9
10 int parser_listing(struct parser *p, unsigned
    lineno);
11 #endif

```

Listing 36: parser/tokenizer.c

```

1 #include "tokenizer.h"

```

```

2 #include <string.h>
3 #include <stdlib.h>
4
5 int next_token(struct parser *p)
6 {
7     unsigned lineno;
8     int attr;
9     if (fscanf(p->f_token, "%u", &lineno) == EOF)
10    {
11        fprintf(stderr, "warning: EOF token may
        not be included in token file\n");
12        return -1;
13    }
14    fscanf(p->f_token, "%s", p->buffer);
15    fscanf(p->f_token, "%d", &p->token.type);
16    if (p->token.type == TOKEN_ID) {
17        struct idres *tok = NULL;
18        if (idres_search(&p->symbols, p->buffer, &
            tok) == -1) {
19            fprintf(stderr, "idres can't find
        symbol %s\n", p->buffer);
20            return -1;
21        }
22        p->token = tok->token;
23        fscanf(p->f_token, "%s\n", p->buffer);
24    } else {
25        fscanf(p->f_token, "%d\n", &attr);
26        p->token.is_id = 0;
27        p->token.val.attr = attr;
28        if (!strcmp(token2str(p->token.type), "
        UNKNOWN")) {
29            fprintf(p->syn_list, "SYNERR: Unknown
        token encountered: %d\n", attr);
30        }
31    }
32    parser_listing(p, lineno);
33
34    return p->token.type;
35 }
36
37 int g_temp = 0;
38 int parser_listing(struct parser *p, unsigned
    new_tok_lineno)
39 {
40     int c;
41     unsigned cur_tok_lineno = p->tok_line;
42     if (cur_tok_lineno == new_tok_lineno) {
43         return 0;
44     } else if (g_temp == 0) {
45         g_temp = 1;
46         /* return 0; */
47     } else {
48         g_temp = 0;
49     }
50     p->tok_line = new_tok_lineno;
51     while (p->list_line < p->tok_line) {
52         int fret = fscanf(p->lex_list, "%s", p->
            buffer);
53         if (fret == -1) { /* EOF reached */
54             p->list_line = p->tok_line;

```



```

54         break;
55     }
56     if (strcmp(p->buffer, "LEXERR:")) {
57         char *ptr = NULL;
58         p->list_line = strtoul(p->buffer, &ptr,
59                               10);
60         if (*ptr) {
61             fprintf(p->syn_list, "Listing file
62                 invalid format, fret %d\n",
63                 fret);
64             exit(EXIT_FAILURE);
65         }
66         fprintf(p->syn_list, "%s", p->buffer);
67         while ((c = fgetc(p->lex_list)) != '\n') {
68             if (c == EOF) {
69                 p->list_line = p->tok_line;
70                 break;
71             }
72             fputc(c, p->syn_list);
73         }
74         fputc('\n', p->syn_list);
75     }
76     return 0;
77 }

```

Listing 37: parser/parser.h

```

1  #ifndef PARSER_H
2  #define PARSER_H
3  #include <stdio.h>
4  #include "idres.h"
5  #include "defs.h"
6  #include "gb.h"
7  #include "types.h"
8
9  struct parser {
10     struct idres *symbols;
11     FILE* f_token;
12     FILE* lex_list;
13     FILE* syn_list;
14     FILE* addr;
15     char *buffer;
16     unsigned tok_line, list_line;
17     struct gb_state gbs;
18     struct token token;
19 };
20
21 int parser_init(struct parser *p, const char*
22     symbol_file, const char* token_file, const
23     char* list_in, const char* addr_out, const
24     char* list_out);
25
26 int parse(struct parser *p);
27
28 int match(struct parser *p, int tok, int *ret);
29
30 int sync(struct parser *p, int* sync_set, int
31     sync_size);
32
33 void expected_found(struct parser *p, int *
34     expected, int size);

```

```

29 int parser_cleanup(struct parser *p);
30
31 char* parser_get_tok_lex(struct parser *p,
32     struct token t_id);
33
34 #endif

```

Listing 38: parser/parser.c

```

1  #include "defs.h"
2  #include "parser.h"
3  #include "tokenizer.h"
4  #include "../prod/program.h"
5
6  int parse(struct parser *p)
7  {
8     next_token(p);
9     program(p);
10    return match(p, TOKEN_EOF, NULL);
11 }
12
13 int match(struct parser *p, int tok, int *ret)
14 {
15     if (p->token.type == tok) {
16         if (tok == TOKEN_EOF) {
17             return 0;
18         } else {
19             printf("Matched token %s\n", token2str(
20                 tok));
21             next_token(p);
22             return 0;
23         }
24     } else {
25         expected_found(p, &tok, 1);
26         next_token(p);
27         if (ret) {
28             *ret = -1;
29         }
30         return -1;
31     }
32 }
33
34 int parser_init(struct parser *p, const char*
35     symbol_file, const char* token_file, const
36     char* list_in, const char* addr_out, const
37     char* list_out)
38 {
39     p->symbols = NULL;
40     if (idres_read(symbol_file, &p->symbols) ==
41         -1) {
42         return -1;
43     }
44     p->f_token = fopen(token_file, "r");
45     if (p->f_token == NULL) {
46         fprintf(stderr, "Could not open file \"%s
47             \n\", token_file);
48         return -1;
49     }
50     p->addr = fopen(addr_out, "w");
51     if (p->addr == NULL) {
52         fprintf(stderr, "Could not open file \"%s
53             \n\", addr_out);

```

```

46     return -1;
47 }
48 p->buffer = malloc(LINELEN+1);
49 if (p->buffer == NULL) {
50     fprintf(stderr, "Could allocate memory\n");
51     ;
52     return -1;
53 }
54 p->lex_list = fopen(list_in, "r");
55 if (p->lex_list == NULL) {
56     fprintf(stderr, "Could not open file \"%s\n\"", list_in);
57     return -1;
58 }
59 p->syn_list = fopen(list_out, "w");
60 if (p->syn_list == NULL) {
61     fprintf(stderr, "Could not open file \"%s\n\"", list_out);
62     return -1;
63 }
64 p->tok_line = 0;
65 p->list_line = 0;
66 return gb_state_init(&p->gbs);
67 }
68 void expected_found(struct parser *p, int*
69     expected, int size)
70 {
71     int i;
72     fprintf(p->syn_list, "SYNERR: Expected ");
73     if (size > 1) {
74         fprintf(p->syn_list, "one of ");
75     }
76     for (i = 0; i < size - 1; i++) {
77         fprintf(p->syn_list, "\"%s\"", token2str(
78             expected[i]));
79         if (size > 2) {
80             fputc(',', p->syn_list);
81         }
82         fputc(' ', p->syn_list);
83     }
84     if (size > 1) {
85         fprintf(p->syn_list, "or ");
86     }
87     fprintf(p->syn_list, "\"%s\" ", token2str(
88         expected[size-1]));
89     if (p->token.is_id) {
90         struct idres *ret = NULL;
91         idres_lookup(&p->symbols, p->token.val.ptr,
92             &ret);
93         if (ret == NULL) {
94             fprintf(stderr, "symbol not found\n");
95             exit(1);
96         }
97         fprintf(p->syn_list, "but found identifier
98             \"%s\"", ret->lexeme);
99     } else {
100         fprintf(p->syn_list, "but found \"%s\"",
101             token2str(p->token.type));
102     }
103 }
104 }
105
106 int sync(struct parser *p, int* sync_set, int
107     sync_size)
108 {
109     int i, found = 0;
110     while (!found) {
111         for (i = 0; i < sync_size; i++) {
112             if (p->token.type == sync_set[i]) {
113                 return 0;
114             }
115         }
116         if (p->token.type == TOKEN_EOF) {
117             return 0;
118         }
119         next_token(p);
120     }
121     return 0;
122 }
123
124 int parser_cleanup(struct parser *p)
125 {
126     int c;
127     while ((c = fgetc(p->lex_list)) != EOF) {
128         fputc(c, p->syn_list);
129     }
130     free(p->buffer);
131     /* gb_print_all(p->gbs.cur_eye, p->addr, 4);
132     */
133     gb_state_free(&p->gbs);
134     fclose(p->addr);
135     fclose(p->f_token);
136     fclose(p->lex_list);
137     fclose(p->syn_list);
138     idres_clean(&p->symbols);
139     return 0;
140 }
141
142 char* parser_get_tok_lex(struct parser *p,
143     struct token t_id)
144 {
145     struct idres *ret = NULL;
146     if (!t_id.is_id
147         || idres_lookup(&p->symbols, t_id.val.ptr,
148             &ret) == -1
149         || ret == NULL) {
150         fprintf(stderr, "INTERNAL ERROR: symbol
151             table"
152             " does not match token: \"%s\"",
153             token2str(t_id.type));
154         exit(EXIT_FAILURE);
155     }
156     return ret->lexeme;
157 }
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999

```

Listing 39: parser/prod/arguments.h

```

1 /* -*- C -*-
2  *

```

```

3  * arguments.h
4  *
5  * Author: Benjamin T James
6  */
7
8  #ifndef ARGUMENTS_H
9  #define ARGUMENTS_H
10
11 #include "../parser.h"
12
13 int arguments(struct parser *p, char **args);
14
15 #endif

```

Listing 40: parser/prod/arguments.c

```

1  /* -*- C -*-
2  *
3  * arguments.c
4  *
5  * Author: Benjamin T James
6  */
7
8  #include "arguments.h"
9  #include "parameter_list.h"
10
11 int arguments(struct parser *p, char **args)
12 {
13     int sync_set[] = {TOKEN_COLON};
14     int expected[] = {TOKEN_COLON, TOKEN_LPAREN};
15     const int sync_size = sizeof(sync_set)/sizeof
16         (*sync_set);
17     const int expected_size = sizeof(expected)/
18         sizeof(*expected);
19     int ret = TYPE_VOID;
20     switch (p->token.type) {
21     case TOKEN_LPAREN:
22         match(p, TOKEN_LPAREN, &ret);
23         err_propagate(parameter_list(p, args), &
24             ret);
25         match(p, TOKEN_RPAREN, &ret);
26         break;
27     case TOKEN_COLON:
28         *args = calloc(1, 1);
29         break;
30     default:
31         ret = TYPE_ERR;
32         expected_found(p, expected, expected_size)
33             ;
34         sync(p, sync_set, sync_size);
35     }
36     return ret;
37 }

```

Listing 41: parser/prod/compound_statement.h

```

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

```

```

6  */
7
8  #ifndef COMPOUND_STATEMENT_H
9  #define COMPOUND_STATEMENT_H
10
11 #include "../parser.h"
12
13 int compound_statement(struct parser *p);
14
15 #endif

```

Listing 42: parser/prod/compound_statement.c

```

1  /* -*- C -*-
2  *
3  * compound_statement.c
4  *
5  * Author: Benjamin T James
6  */
7
8  #include "compound_statement.h"
9  #include "optional_statements.h"
10
11 int compound_statement(struct parser *p)
12 {
13     int sync_set[] = {TOKEN_PERIOD,
14         TOKEN_SEMICOLON, TOKEN_END, TOKEN_ELSE};
15     int expected[] = {TOKEN_BEGIN};
16     const int sync_size = sizeof(sync_set)/sizeof
17         (*sync_set);
18     const int expected_size = sizeof(expected)/
19         sizeof(*expected);
20     int ret = TYPE_VOID;
21     switch (p->token.type) {
22     case TOKEN_BEGIN:
23         match(p, TOKEN_BEGIN, &ret);
24         err_propagate(optional_statements(p), &ret
25             );
26         match(p, TOKEN_END, &ret);
27         break;
28     default:
29         ret = TYPE_ERR;
30         expected_found(p, expected, expected_size)
31             ;
32         sync(p, sync_set, sync_size);
33     }
34     return ret;
35 }

```

Listing 43: parser/prod/declarations.h

```

1  /* -*- C -*-
2  *
3  * declarations.h
4  *
5  * Author: Benjamin T James
6  */
7
8  #ifndef DECLARATIONS_H
9  #define DECLARATIONS_H
10

```

```

11 #include "../parser.h"
12
13 int declarations(struct parser *p);
14
15 #endif

```

Listing 44: parser/prod/declarations.c

```

1  /* -*- C -*-
2  *
3  * declarations.c
4  *
5  * Author: Benjamin T James
6  */
7
8 #include "declarations.h"
9 #include "type.h"
10 int declarations(struct parser *p)
11 {
12     int sync_set[] = {TOKEN_BEGIN, TOKEN_FUNCTION
13     };
14     int expected[] = {TOKEN_BEGIN, TOKEN_FUNCTION,
15     TOKEN_VAR};
16     const int sync_size = sizeof(sync_set)/sizeof
17     (*sync_set);
18     const int expected_size = sizeof(expected)/
19     sizeof(*expected);
20     struct token t_id;
21     int ret = TYPE_VOID;
22     char *lexeme = NULL;
23     switch (p->token.type) {
24     case TOKEN_VAR:
25         match(p, TOKEN_VAR, &ret);
26         t_id = p->token;
27         match(p, TOKEN_ID, &ret);
28         match(p, TOKEN_COLON, &ret);
29         {
30             int offset, var_type, width;
31             lexeme = parser_get_tok_lex(p, t_id);
32             var_type = type(p, &width);
33             err_propagate(var_type, &ret);
34             get_offset(&p->gbs, &offset);
35             int bret = check_add_blue(&p->gbs,
36             lexeme, var_type, offset);
37             if (lexeme != NULL) {
38                 if (bret == -1) {
39                     fprintf(p->syn_list,
40                     "SEMERR: Identifier \"%s\"
41                     already declared\n",
42                     lexeme);
43                     ret = TYPE_ERR;
44                 } else if (bret < 0) {
45                     ret = TYPE_ERR;
46                 } else {
47                     fprintf(p->addr, "%s\t%d\t%s\n",
48                     lexeme, offset, strtype(
49                     var_type));
50                 }
51             }
52             set_offset(&p->gbs, offset + width);
53         }
54     }
55 }

```

```

46     match(p, TOKEN_SEMICOLON, &ret);
47     err_propagate(declarations(p), &ret);
48     break;
49     case TOKEN_BEGIN:
50     case TOKEN_FUNCTION:
51         break;
52     default:
53         ret = TYPE_ERR;
54         expected_found(p, expected, expected_size)
55         ;
56         sync(p, sync_set, sync_size);
57     }
58     return ret;
59 }

```

Listing 45: parser/prod/expression.h

```

1  /* -*- C -*-
2  *
3  * expression.h
4  *
5  * Author: Benjamin T James
6  */
7
8 #ifndef EXPRESSION_H
9 #define EXPRESSION_H
10
11 #include "../parser.h"
12
13 int expression(struct parser *p);
14
15 #endif

```

Listing 46: parser/prod/expression.c

```

1  /* -*- C -*-
2  *
3  * expression.c
4  *
5  * Author: Benjamin T James
6  */
7
8 #include "expression.h"
9 #include "expression_prime.h"
10 #include "simple_expression.h"
11
12 int expression(struct parser *p)
13 {
14     int sync_set[] = {TOKEN_DO, TOKEN_ELSE,
15     TOKEN_END,
16     TOKEN_THEN, TOKEN_RPAREN,
17     TOKEN_RBACKET, TOKEN_COMMA,
18     TOKEN_SEMICOLON};
19     int expected[] = {TOKEN_ID, TOKEN_NUM_INTEGER,
20     TOKEN_NUM_REAL, TOKEN_LPAREN,
21     TOKEN_NOT, TOKEN_SIGN};
22     const int sync_size = sizeof(sync_set)/sizeof
23     (*sync_set);
24     const int expected_size = sizeof(expected)/
25     sizeof(*expected);
26     int ret = TYPE_VOID;

```

```

24
25     switch (p->token.type) {
26     case TOKEN_ID:
27     case TOKEN_NUM_INTEGER:
28     case TOKEN_NUM_REAL:
29     case TOKEN_LPAREN:
30     case TOKEN_NOT:
31     case TOKEN_SIGN:
32         ret = simple_expression(p);
33         ret = expression_prime(p, ret);
34         break;
35     default:
36         ret = TYPE_ERR;
37         expected_found(p, expected, expected_size)
38             ;
39         sync(p, sync_set, sync_size);
40     }
41     return ret;

```

Listing 47: parser/prod/expression_prime.h

```

1  /* -*- C -*-
2  *
3  * expression_prime.h
4  *
5  * Author: Benjamin T James
6  */
7
8  #ifndef EXPRESSION_PRIME_H
9  #define EXPRESSION_PRIME_H
10
11  #include "../parser.h"
12
13  int expression_prime(struct parser *p, int type);
14
15  #endif

```

Listing 48: parser/prod/expression_prime.c

```

1  /* -*- C -*-
2  *
3  * expression_prime.c
4  *
5  * Author: Benjamin T James
6  */
7
8  #include "expression_prime.h"
9  #include "simple_expression.h"
10
11  int expression_prime(struct parser *p, int type)
12  {
13      int sync_set[] = {TOKEN_DO, TOKEN_ELSE,
14                       TOKEN_END,
15                       TOKEN_THEN, TOKEN_RPAREN,
16                       TOKEN_RBRACKET,
17                       TOKEN_COMMA, TOKEN_SEMICOLON};
18      int expected[] = {TOKEN_RELOP, TOKEN_DO,
19                       TOKEN_ELSE,
20                       TOKEN_END, TOKEN_THEN, TOKEN_RPAREN,
21                       TOKEN_RBRACKET, TOKEN_COMMA,

```

```

19         TOKEN_SEMICOLON};
20     const int sync_size = sizeof(sync_set)/sizeof
21         (*sync_set);
22     const int expected_size = sizeof(expected)/
23         sizeof(*expected);
24     int ret = type;
25     switch (p->token.type) {
26     case TOKEN_RELOP: {
27         int type2;
28         struct token t_relop = p->token;
29         match(p, TOKEN_RELOP, &ret);
30         type2 = simple_expression(p);
31         if (type != TYPE_ERR && type2 != TYPE_ERR)
32             {
33                 if (type != type2) {
34                     fprintf(p->syn_list, "SEMERR:
35                         Invalid operands to %s: %s and
36                         %s\n",
37                         relop2str(t_relop.val.attr),
38                         strtype(type),
39                         strtype(type2));
40                     ret = TYPE_ERR;
41                 } else {
42                     ret = TYPE_BOOL;
43                 }
44             } else {
45                 ret = TYPE_ERR;
46             }
47         break;
48     }
49     case TOKEN_DO:
50     case TOKEN_ELSE:
51     case TOKEN_END:
52     case TOKEN_THEN:
53     case TOKEN_RPAREN:
54     case TOKEN_RBRACKET:
55     case TOKEN_COMMA:
56     case TOKEN_SEMICOLON:
57         ret = type;
58         break;
59     default:
60         ret = TYPE_ERR;
61         expected_found(p, expected, expected_size)
62             ;
63         sync(p, sync_set, sync_size);
64     }
65     return ret;
66 }

```

Listing 49: parser/prod/expression_list.h

```

1  /* -*- C -*-
2  *
3  * expression_list.h
4  *
5  * Author: Benjamin T James
6  */
7
8  #ifndef EXPRESSION_LIST_H
9  #define EXPRESSION_LIST_H
10

```

<pre> 11 #include "../parser.h" 12 13 int expression_list(struct parser *p, char **args) 14 ; 15 #endif </pre>	<pre> 7 8 #ifndef EXPRESSION_LIST_PRIME_H 9 #define EXPRESSION_LIST_PRIME_H 10 11 #include "../parser.h" 12 13 int expression_list_prime(struct parser *p, char 14 **args); 15 #endif </pre>
--	--

Listing 50: parser/prod/expression_list.c

```

1 /* -*- C -*-
2 *
3 * expression_list.c
4 *
5 * Author: Benjamin T James
6 */
7
8 #include "expression_list.h"
9 #include "expression_list_prime.h"
10 #include "expression.h"
11 int expression_list(struct parser *p, char **args)
12 {
13     int sync_set[] = {TOKEN_RPAREN};
14     int expected[] = {TOKEN_ID, TOKEN_NUM_INTEGER,
15         TOKEN_NUM_REAL, TOKEN_LPAREN,
16         TOKEN_NOT, TOKEN_SIGN};
17     const int sync_size = sizeof(sync_set)/sizeof
18         (*sync_set);
19     const int expected_size = sizeof(expected)/
20         sizeof(*expected);
21     int ret = TYPE_VOID;
22     switch (p->token.type) {
23     case TOKEN_ID:
24     case TOKEN_NUM_INTEGER:
25     case TOKEN_NUM_REAL:
26     case TOKEN_LPAREN:
27     case TOKEN_NOT:
28     case TOKEN_SIGN: {
29         int type = expression(p);
30         char *arg = malloc(2);
31         sprintf(arg, "%c", type);
32         *args = arg;
33         ret = expression_list_prime(p, args);
34         break;
35     }
36     default:
37         ret = TYPE_ERR;
38         expected_found(p, expected, expected_size)
39             ;
40         sync(p, sync_set, sync_size);
41     }
42     return ret;
43 }

```

Listing 52: parser/prod/expression_list_prime.c

```

1 /* -*- C -*-
2 *
3 * expression_list_prime.c
4 *
5 * Author: Benjamin T James
6 */
7
8 #include "expression_list_prime.h"
9 #include "expression.h"
10 #include <string.h>
11 int expression_list_prime(struct parser *p, char
12     **args)
13 {
14     int sync_set[] = {TOKEN_RPAREN};
15     int expected[] = {TOKEN_COMMA};
16     const int sync_size = sizeof(sync_set)/sizeof
17         (*sync_set);
18     const int expected_size = sizeof(expected)/
19         sizeof(*expected);
20     int ret = TYPE_VOID;
21     switch (p->token.type) {
22     case TOKEN_COMMA:
23         match(p, TOKEN_COMMA, &ret);
24         int type = expression(p);
25         char *arg = malloc(strlen(*args) + 2);
26         sprintf(arg, "%s%c", *args, type);
27         free(*args);
28         *args = arg;
29         ret = expression_list_prime(p, args);
30         break;
31     case TOKEN_RPAREN:
32         break;
33     default:
34         ret = TYPE_ERR;
35         expected_found(p, expected, expected_size)
36             ;
37         sync(p, sync_set, sync_size);
38     }
39     return ret;
40 }

```

Listing 51: parser/prod/expression_list_prime.h

```

1 /* -*- C -*-
2 *
3 * expression_list_prime.h
4 *
5 * Author: Benjamin T James
6 */

```

Listing 53: parser/prod/factor.h

```

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

```

```

6  */
7
8  #ifndef FACTOR_H
9  #define FACTOR_H
10
11 #include "../parser.h"
12
13 int factor(struct parser *p);
14
15 #endif

```

Listing 54: parser/prod/factor.c

```

1  /* -*- C -*-
2  *
3  * factor.c
4  *
5  * Author: Benjamin T James
6  */
7
8 #include "factor.h"
9 #include "expression.h"
10 #include "factor_prime.h"
11
12 int factor(struct parser *p)
13 {
14     int sync_set[] = {TOKEN_MULOP, TOKEN_ADDOP,
15                       TOKEN_SIGN, TOKEN_RELOP,
16                       TOKEN_DO, TOKEN_ELSE, TOKEN_END,
17                       TOKEN_THEN, TOKEN_LPAREN,
18                       TOKEN_RBACKET, TOKEN_COMMA,
19                       TOKEN_SEMICOLON};
20     int expected[] = {TOKEN_ID, TOKEN_NUM_INTEGER,
21                      TOKEN_NUM_REAL, TOKEN_LPAREN,
22                      TOKEN_NOT};
23     const int sync_size = sizeof(sync_set)/sizeof
24         (*sync_set);
25     const int expected_size = sizeof(expected)/
26         sizeof(*expected);
27     int ret = TYPE_VOID;
28     switch (p->token.type) {
29     case TOKEN_ID: {
30         struct token t_id = p->token;
31         char *lexeme = parser_get_tok_lex(p, t_id)
32             ;
33         int type;
34         match(p, TOKEN_ID, &ret);
35         type = factor_prime(p, lexeme);
36         if (ret != TYPE_ERR) {
37             ret = type;
38         }
39         break;
40     }
41     case TOKEN_NUM_INTEGER:
42         match(p, TOKEN_NUM_INTEGER, &ret);
43         ret = TYPE_INT;
44         break;
45     case TOKEN_NUM_REAL:
46         match(p, TOKEN_NUM_REAL, &ret);
47         ret = TYPE_REAL;
48         break;

```

```

45     case TOKEN_LPAREN: {
46         int type;
47         match(p, TOKEN_LPAREN, &ret);
48         type = expression(p);
49         match(p, TOKEN_RPAREN, &ret);
50         if (ret != TYPE_ERR) {
51             ret = type;
52         }
53         break;
54     }
55     case TOKEN_NOT: {
56         int type;
57         match(p, TOKEN_NOT, &ret);
58         type = factor(p);
59         if (type != TYPE_ERR && type != TYPE_BOOL)
60             {
61                 fprintf(p->syn_list, "SEMERR: Invalid
62                     operand to \"not\": %s\n",
63                     strttype(type));
64                 type = TYPE_ERR;
65             }
66         if (ret != TYPE_ERR) {
67             ret = type;
68         }
69         break;
70     }
71     default:
72         ret = TYPE_ERR;
73         expected_found(p, expected, expected_size)
74             ;
75         sync(p, sync_set, sync_size);
76     }
77     return ret;
78 }

```

Listing 55: parser/prod/factor_prime.h

```

1  /* -*- C -*-
2  *
3  * factor_prime.h
4  *
5  * Author: Benjamin T James
6  */
7
8 #ifndef FACTOR_PRIME_H
9 #define FACTOR_PRIME_H
10
11 #include "../parser.h"
12
13 int factor_prime(struct parser *p, char *id_lex);
14
15 #endif

```

Listing 56: parser/prod/factor_prime.c

```

1  /* -*- C -*-
2  *
3  * factor_prime.c
4  *
5  * Author: Benjamin T James
6  */

```

```

7
8 #include "factor_prime.h"
9 #include "expression.h"
10 #include "expression_list.h"
11 #include <string.h>
12 int fp_array(struct parser *p, char *id_lex);
13 int fp_function(struct parser *p, char *id_lex);
14 int fp_id(struct parser *p, char *id_lex);
15
16 int factor_prime(struct parser *p, char *id_lex)
17 {
18     int sync_set[] = {TOKEN_MULOP, TOKEN_ADDOP,
19                       TOKEN_SIGN, TOKEN_RELOP,
20                       TOKEN_DO, TOKEN_ELSE, TOKEN_END,
21                       TOKEN_THEN, TOKEN_RPAREN,
22                       TOKEN_RBRACKET, TOKEN_COMMA,
23                       TOKEN_SEMICOLON};
24     int expected[] = {TOKEN_LBRACKET, TOKEN_LPAREN,
25                      , TOKEN_MULOP,
26                      TOKEN_ADDOP, TOKEN_SIGN, TOKEN_RELOP,
27                      TOKEN_DO, TOKEN_ELSE, TOKEN_END,
28                      TOKEN_THEN, TOKEN_RPAREN,
29                      TOKEN_RBRACKET, TOKEN_COMMA,
30                      TOKEN_SEMICOLON};
31     const int sync_size = sizeof(sync_set)/sizeof
32     (*sync_set);
33     const int expected_size = sizeof(expected)/
34     sizeof(*expected);
35     int ret = TYPE_VOID;
36     /*
37         if (lexeme == NULL && < 0) {
38             fprintf(p->syn_list, "SEMERR: Symbol
39             \"%s\" was not declared in this
40             scope\n", lexeme);
41             ret = TYPE_ERR;
42         }
43     */
44     switch (p->token.type) {
45     case TOKEN_LBRACKET: {
46         ret = fp_array(p, id_lex);
47         break;
48     }
49     case TOKEN_LPAREN: { /* turn FPINT -> INT */
50         ret = fp_function(p, id_lex);
51         break;
52     }
53     case TOKEN_MULOP:
54     case TOKEN_SIGN: /* used as addition/
55                       subtraction here */
56     case TOKEN_ADDOP:
57     case TOKEN_RELOP:
58     case TOKEN_DO:
59     case TOKEN_ELSE:
60     case TOKEN_END:
61     case TOKEN_THEN:
62     case TOKEN_RPAREN:
63     case TOKEN_RBRACKET:
64     case TOKEN_COMMA:
65         case TOKEN_SEMICOLON: /* ID */ {
66             ret = fp_id(p, id_lex);
67             break;
68         }
69     default:
70         ret = TYPE_ERR;
71         expected_found(p, expected, expected_size)
72         ;
73         sync(p, sync_set, sync_size);
74     }
75     return ret;
76 }
77
78 int fp_array(struct parser *p, char *id_lex)
79 {
80     int ret = TYPE_VOID;
81     int expr_type, atype, scalar_type, type;
82     if (gettype(&p->gbs, id_lex, &atype) < 0) {
83         fprintf(p->syn_list, "SEMERR: Variable \"%s\"
84         was not declared in this scope\n",
85         id_lex);
86         ret = TYPE_ERR;
87     }
88     scalar_type = ARRAY_TO_SCALAR(atype);
89     if (scalar_type == TYPE_ERR) {
90         fprintf(p->syn_list, "SEMERR: Attempted
91         indexing of non-array type %s\n",
92         strtype(atype));
93         ret = TYPE_ERR;
94     } else {
95         type = scalar_type;
96     }
97     match(p, TOKEN_LBRACKET, &ret);
98     expr_type = expression(p);
99     if (expr_type != TYPE_ERR && expr_type !=
100     TYPE_INT) {
101         fprintf(p->syn_list, "SEMERR: Array index
102         must be an integer (found %s)\n",
103         strtype(expr_type));
104         ret = TYPE_ERR;
105     }
106     match(p, TOKEN_RBRACKET, &ret);
107     if (ret != TYPE_ERR) {
108         ret = type;
109     }
110     return ret;
111 }
112
113 int fp_function(struct parser *p, char *id_lex)
114 {
115     char *args = NULL;
116     int ret = TYPE_VOID;
117     match(p, TOKEN_LPAREN, &ret);
118     err_propagate(expression_list(p, &args), &ret)
119     ;
120     match(p, TOKEN_RPAREN, &ret);
121     struct gb *gbret = NULL;
122     find_green(p->gbs.cur_eye, id_lex, &gbret);
123     if (gbret == NULL) {
124         fprintf(p->syn_list, "SEMERR: Function \"%s\"

```



```

        s\" not declared in this scope\n",
        id_lex);
112     if (args) {
113         free(args);
114     }
115     return ret;
116 }
117 char *fargs = gbret->n.g.arglist;
118 ret = FUNC_TO_SCALAR(gbret->n.g.type);
119 if (strcmp(args, fargs)) {
120     char *exp = param2str(fargs);
121     char *got = param2str(args);
122     fprintf(p->syn_list, "SEMERR: Function
        args for \"%s\" do not match: expected
        (%s) but got (%s)\n",
123         id_lex,
124         exp,
125         got);
126     ret = TYPE_ERR;
127     free(exp);
128     free(got);
129 }
130 if (args) {
131     free(args);
132 }
133 return ret;
134 }
135
136 int fp_id(struct parser *p, char *id_lex)
137 {
138     int type = TYPE_ERR;
139     if (id_lex != NULL &&
140         gettype(&p->gbs, id_lex, &type) < 0) {
141         fprintf(p->syn_list, "SEMERR: Variable \"%s\"
        s\" was not declared in this scope\n",
        id_lex);
142         type = TYPE_ERR;
143     }
144     return type;
145 }

```

Listing 57: parser/prod/identifier_list.h

```

1  /* -*- C -*-
2  *
3  * identifier_list.h
4  *
5  * Author: Benjamin T James
6  */
7
8  #ifndef IDENTIFIER_LIST_H
9  #define IDENTIFIER_LIST_H
10
11  #include "../parser.h"
12
13  int identifier_list(struct parser *p);
14
15  #endif

```

Listing 58: parser/prod/identifier_list.c

```

1  /* -*- C -*-
2  *
3  * identifier_list.c
4  *
5  * Author: Benjamin T James
6  */
7
8  #include "identifier_list.h"
9  #include "identifier_list_prime.h"
10 int identifier_list(struct parser *p)
11 {
12     int sync_set[] = {TOKEN_RPAREN};
13     int expected[] = {TOKEN_ID};
14     const int sync_size = sizeof(sync_set)/sizeof
        (*sync_set);
15     const int expected_size = sizeof(expected)/
        sizeof(*expected);
16     struct token t_id;
17     int offset, ret = TYPE_VOID;
18     char *lexeme;
19     switch (p->token.type) {
20     case TOKEN_ID:
21         t_id = p->token;
22         match(p, TOKEN_ID, &ret);
23         {
24             lexeme = parser_get_tok_lex(p, t_id);
25             get_offset(&p->gbs, &offset);
26             int bret = check_add_blue(&p->gbs,
                lexeme, TYPE_PGMPARAM, offset);
27             if (bret == -1) {
28                 fprintf(p->syn_list,
                "SEMERR: Identifier \"%s\"
                already declared\n",
                lexeme);
29                 ret = TYPE_ERR;
30             } else if (bret < 0) {
31                 /* err already reported */
32                 ret = TYPE_ERR;
33             }
34             offset += type_width(TYPE_PGMPARAM);
35             set_offset(&p->gbs, offset);
36         }
37     }
38     err_propagate(identifier_list_prime(p), &
        ret);
39     break;
40     default:
41         ret = TYPE_ERR;
42         expected_found(p, expected, expected_size)
43             ;
44         sync(p, sync_set, sync_size);
45     }
46     return ret;
47 }

```

Listing 59: parser/prod/identifier_list_prime.h

```

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

```

```

6  */
7
8  #ifndef IDENTIFIER_LIST_PRIME_H
9  #define IDENTIFIER_LIST_PRIME_H
10
11 #include "../parser.h"
12
13 int identifier_list_prime(struct parser *p);
14
15 #endif

```

Listing 60: parser/prod/identifier_list_prime.c

```

1  /* -*- C -*-
2  *
3  * identifier_list_prime.c
4  *
5  * Author: Benjamin T James
6  */
7
8  #include "identifier_list_prime.h"
9
10 int identifier_list_prime(struct parser *p)
11 {
12     int sync_set[] = {TOKEN_RPAREN};
13     int expected[] = {TOKEN_COMMA, TOKEN_RPAREN};
14     const int sync_size = sizeof(sync_set)/sizeof
15         (*sync_set);
16     const int expected_size = sizeof(expected)/
17         sizeof(*expected);
18     struct token t_id;
19     int offset, ret = TYPE_VOID;
20     char *lexeme;
21     switch (p->token.type) {
22     case TOKEN_COMMA:
23         match(p, TOKEN_COMMA, &ret);
24         t_id = p->token;
25         match(p, TOKEN_ID, &ret);
26         {
27             lexeme = parser_get_tok_lex(p, t_id);
28             get_offset(&p->gbs, &offset);
29             int bret = check_add_blue(&p->gbs,
30                 lexeme, TYPE_PGMPARAM, offset);
31             if (bret == -1) {
32                 fprintf(p->syn_list,
33                     "SEMERR: Identifier \"%s\"
34                     already declared\n",
35                     lexeme);
36                 ret = TYPE_ERR;
37             } else if (bret < 0) {
38                 /* err already reported */
39                 ret = TYPE_ERR;
40             }
41             offset += type_width(TYPE_PGMPARAM);
42             set_offset(&p->gbs, offset);
43         }
44         err_propagate(identifier_list_prime(p), &
45             ret);
46         break;
47     case TOKEN_RPAREN:
48         break;

```

```

44     default:
45         ret = TYPE_ERR;
46         expected_found(p, expected, expected_size)
47             ;
48         sync(p, sync_set, sync_size);
49     }
50     return ret;
51 }

```

Listing 61: parser/prod/optional_statements.h

```

1  /* -*- C -*-
2  *
3  * optional_statements.h
4  *
5  * Author: Benjamin T James
6  */
7
8  #ifndef OPTIONAL_STATEMENTS_H
9  #define OPTIONAL_STATEMENTS_H
10
11 #include "../parser.h"
12
13 int optional_statements(struct parser *p);
14
15 #endif

```

Listing 62: parser/prod/optional_statements.c

```

1  /* -*- C -*-
2  *
3  * optional_statements.c
4  *
5  * Author: Benjamin T James
6  */
7
8  #include "optional_statements.h"
9  #include "statement_list.h"
10
11 int optional_statements(struct parser *p)
12 {
13     int sync_set[] = {TOKEN_END};
14     int expected[] = {TOKEN_BEGIN, TOKEN_ID,
15         TOKEN_IF, TOKEN_WHILE};
16     const int sync_size = sizeof(sync_set)/sizeof
17         (*sync_set);
18     const int expected_size = sizeof(expected)/
19         sizeof(*expected);
20     int ret = TYPE_VOID;
21     switch (p->token.type) {
22     case TOKEN_BEGIN:
23     case TOKEN_ID:
24     case TOKEN_IF:
25     case TOKEN_WHILE:
26         err_propagate(statement_list(p), &ret);
27         break;
28     case TOKEN_END:
29         break;
30     default:
31         ret = TYPE_ERR;

```

```

29     expected_found(p, expected, expected_size) 33
        ;                                         34
30     sync(p, sync_set, sync_size);
31 }                                             35
32 return ret;                                 36
33 }                                           37

```

Listing 63: parser/prod/parameter_list.h

```

1  /* -*- C -*-                                40
2  *                                           41
3  * parameter_list.h                         42
4  *                                           43
5  * Author: Benjamin T James                 44
6  */                                         45
7
8  #ifndef PARAMETER_LIST_H                    46
9  #define PARAMETER_LIST_H                    47
10
11 #include "../parser.h"                      48
12
13 int parameter_list(struct parser *p, char **args); 49
14
15 #endif                                     50

```

Listing 64: parser/prod/parameter_list.c

```

1  /* -*- C -*-                                56
2  *                                           57
3  * parameter_list.c                         58
4  *                                           59
5  * Author: Benjamin T James                 60
6  */                                         61
7
8  #include <string.h>                          62
9  #include "parameter_list.h"                 63
10 #include "type.h"                           64
11 #include "parameter_list_prime.h"            65
12
13 #define INTBUF 10                           66
14
15 int parameter_list(struct parser *p, char **args)
16 {
17     int sync_set[] = {TOKEN_RPAREN};
18     int expected[] = {TOKEN_ID};
19     const int sync_size = sizeof(sync_set)/sizeof
        (*sync_set);
20     const int expected_size = sizeof(expected)/
        sizeof(*expected);
21     struct token t_id;
22     int ret = TYPE_VOID;
23     int width, var_type, offset;
24     char *lexeme = NULL;
25     switch (p->token.type) {
26     case TOKEN_ID:
27         t_id = p->token;
28         match(p, TOKEN_ID, &ret);
29         match(p, TOKEN_COLON, &ret);
30         lexeme = parser_get_tok_lex(p, t_id);
31         var_type = type(p, &width);
32         err_propagate(var_type, &ret);

```

```

        get_offset(&p->gbs, &offset);
        int bret = check_add_blue(&p->gbs, lexeme,
            var_type, 0);
        if (lexeme != NULL) {
            if (bret == -1) {
                fprintf(p->syn_list,
                    "SEMERR: Identifier \"%s\"
                    already declared\n",
                    lexeme);
                ret = TYPE_ERR;
            } else if (bret < 0) {
                ret = TYPE_ERR;
            }
        }
        /* set_offset(&p->gbs, offset + width); */
        err_propagate(parameter_list_prime(p, args
            ), &ret);
        int len = INTBUF + 1;
        if (*args != NULL) {
            len += strlen(*args);
        }
        char *next_str = malloc(len);
        *next_str = (char)var_type;
        next_str[1] = 0;
        if (*args != NULL) {
            strcpy(next_str+1, *args);
            free(*args);
        }
        *args = next_str;
        break;
    default:
        ret = TYPE_ERR;
        expected_found(p, expected, expected_size)
            ;
        sync(p, sync_set, sync_size);
    }
    return ret;
}

```

Listing 65: parser/prod/parameter_list_prime.h

```

1  /* -*- C -*-                                1
2  *                                           2
3  * parameter_list_prime.h                  3
4  *                                           4
5  * Author: Benjamin T James                 5
6  */                                         6
7
8  #ifndef PARAMETER_LIST_PRIME_H              8
9  #define PARAMETER_LIST_PRIME_H              9
10
11 #include "../parser.h"                      10
12
13 int parameter_list_prime(struct parser *p, char ** 11
        args);
14
15 #endif                                     12

```

Listing 66: parser/prod/parameter_list_prime.c

```

1  /* -*- C -*-                                1

```

```

2  *
3  * parameter_list_prime.c
4  *
5  * Author: Benjamin T James
6  */
7
8  #include <string.h>
9  #include "parameter_list_prime.h"
10 #include "type.h"
11
12 #define INTBUF 10
13
14 int parameter_list_prime(struct parser *p, char **
    args)
15 {
16     int sync_set[] = {TOKEN_RPAREN};
17     int expected[] = {TOKEN_SEMICOLON,
18         TOKEN_RPAREN};
19     const int sync_size = sizeof(sync_set)/sizeof
20         (*sync_set);
21     const int expected_size = sizeof(expected)/
22         sizeof(*expected);
23     struct token t_id;
24     int ret = TYPE_VOID;
25     int width, var_type, offset;
26     char *lexeme = NULL;
27     switch (p->token.type) {
28     case TOKEN_SEMICOLON:
29         match(p, TOKEN_SEMICOLON, &ret);
30         t_id = p->token;
31         match(p, TOKEN_ID, &ret);
32         match(p, TOKEN_COLON, &ret);
33         lexeme = parser_get_tok_lex(p, t_id);
34         var_type = type(p, &width);
35         err_propagate(var_type, &ret);
36
37         get_offset(&p->gbs, &offset);
38         int bret = check_add_blue(&p->gbs, lexeme,
39             var_type, 0);
40         if (lexeme != NULL) {
41             if (bret == -1) {
42                 fprintf(p->syn_list,
43                     "SEMERR: Identifier \"%s\"
44                     already declared\n",
45                     lexeme);
46                 ret = TYPE_ERR;
47             } else if (bret < 0) {
48                 ret = TYPE_ERR;
49             }
50         }
51         /* set_offset(&p->gbs, offset + width); */
52         err_propagate(parameter_list_prime(p, args
53             ), &ret);
54         int len = INTBUF + strlen(*args) + 1;
55         char *next_str = malloc(len);
56         *next_str = (char)var_type;
57         strcpy(next_str+1, *args);
58         free(*args);
59         *args = next_str;
60         break;

```

```

55     case TOKEN_RPAREN:
56         *args = calloc(1,1);
57         break;
58     default:
59         ret = TYPE_ERR;
60         expected_found(p, expected, expected_size)
61             ;
62         sync(p, sync_set, sync_size);
63     }
64     return ret;

```

Listing 67: parser/prod/program.h

```

1  /* -*- C -*-
2  *
3  * program.h
4  *
5  * Author: Benjamin T James
6  */
7
8  #ifndef PROGRAM_H
9  #define PROGRAM_H
10
11 #include "../parser.h"
12
13 int program(struct parser *p);
14
15 #endif

```

Listing 68: parser/prod/program.c

```

1  /* -*- C -*-
2  *
3  * program.c
4  *
5  * Author: Benjamin T James
6  */
7
8  #include "program.h"
9  #include "identifier_list.h"
10 #include "declarations.h"
11 #include "subprogram_declarations.h"
12 #include "compound_statement.h"
13
14 int program(struct parser *p)
15 {
16     int expected = TOKEN_PROGRAM;
17     struct token t_id;
18     char *lexeme = NULL;
19     int ret = TYPE_VOID;
20     switch (p->token.type) {
21     case TOKEN_PROGRAM:
22         match(p, TOKEN_PROGRAM, &ret);
23         t_id = p->token;
24         match(p, TOKEN_ID, &ret);
25         lexeme = parser_get_tok_lex(p, t_id);
26         if (lexeme != NULL) {
27             if (check_add_green(&p->gbs, lexeme,
28                 TYPE_PGMNAME) == -1) {

```

```

28         fprintf(p->syn_list, "SEMERR: Scope 10 int sign(struct parser *p)
           \"%s\" already exists\n", 11 {
29         lexeme); 12     int sync_set[] = {TOKEN_ID, TOKEN_NUM_INTEGER,
30     } 13         TOKEN_NUM_REAL, TOKEN_LPAREN,
31 } 14         TOKEN_NOT};
32 15     int expected[] = {TOKEN_SIGN};
33 match(p, TOKEN_LPAREN, &ret); 16     const int sync_size = sizeof(sync_set)/sizeof
34 err_propagate(identifier_list(p), &ret); 17         (*sync_set);
35 match(p, TOKEN_RPAREN, &ret); 18     const int expected_size = sizeof(expected)/
36 match(p, TOKEN_SEMICOLON, &ret); 19         sizeof(*expected);
37 err_propagate(declarations(p), &ret); 20     int ret = TYPE_VOID;
38 err_propagate(subprogram_declarations(p), 21     switch (p->token.type) {
           &ret); 22     case TOKEN_SIGN:
39 err_propagate(compound_statement(p), &ret) 23         match(p, TOKEN_SIGN, &ret);
           ; 24         break;
40 match(p, TOKEN_PERIOD, &ret); 25     default:
41 if (end_green(&p->gbs) == -1) { 26         ret = TYPE_ERR;
42     fprintf(p->syn_list, "SEMERR: Scope for 27         expected_found(p, expected, expected_size)
           \"program\" ended but never 28         ;
           started\n"); 29         sync(p, sync_set, sync_size);
43     ret = TYPE_ERR; 30     }
44 } 31     return ret;
45 break; 32 }
46 default: 33
47     ret = TYPE_ERR; 34
48     expected_found(p, &expected, 1); 35
49     sync(p, NULL, 0); 36
50 } 37
51 return ret; 38
52 } 39

```

Listing 69: parser/prod/sign.h

```

1  /* -*- C -*-
2  *
3  * sign.h
4  *
5  * Author: Benjamin T James
6  */
7
8  #ifndef SIGN_H
9  #define SIGN_H
10
11  #include "../parser.h"
12
13  int sign(struct parser *p);
14
15  #endif

```

Listing 70: parser/prod/sign.c

```

1  /* -*- C -*-
2  *
3  * sign.c
4  *
5  * Author: Benjamin T James
6  */
7
8  #include "sign.h"
9

```

Listing 71: parser/prod/simple_expression.h

```

1  /* -*- C -*-
2  *
3  * simple_expression.h
4  *
5  * Author: Benjamin T James
6  */
7
8  #ifndef SIMPLE_EXPRESSION_H
9  #define SIMPLE_EXPRESSION_H
10
11  #include "../parser.h"
12
13  int simple_expression(struct parser *p);
14
15  #endif

```

Listing 72: parser/prod/simple_expression.c

```

1  /* -*- C -*-
2  *
3  * simple_expression.c
4  *
5  * Author: Benjamin T James
6  */
7
8  #include "simple_expression.h"
9  #include "term.h"
10 #include "simple_expression_prime.h"
11 #include "sign.h"
12
13 int simple_expression(struct parser *p)
14 {
15     int sync_set[] = {TOKEN_RELOP, TOKEN_DO,
16                     TOKEN_ELSE,
17                     TOKEN_END, TOKEN_THEN, TOKEN_RPAREN,

```

```

17     TOKEN_RBRACKET, TOKEN_COMMA,
18     TOKEN_SEMICOLON};
19 int expected[] = {TOKEN_ID, TOKEN_NUM_INTEGER,
20     TOKEN_NUM_REAL, TOKEN_LPAREN,
21     TOKEN_NOT, TOKEN_SIGN};
22 const int sync_size = sizeof(sync_set)/sizeof
    (*sync_set);
23 const int expected_size = sizeof(expected)/
    sizeof(*expected);
24 int ret = TYPE_VOID;
25 switch (p->token.type) {
26 case TOKEN_SIGN: {
27     int term_type;
28     sign(p);
29     term_type = term(p);
30     err_propagate(term_type, &ret);
31     if (term_type != TYPE_ERR && term_type !=
        TYPE_INT && term_type != TYPE_REAL) {
32         fprintf(p->syn_list, "Cannot apply a
            sign to type %s\n",
33             strtype(ret));
34         ret = TYPE_ERR;
35     }
36     term_type = simple_expression_prime(p,
        term_type);
37     if (ret != TYPE_ERR) {
38         ret = term_type;
39     }
40     break;
41 }
42 case TOKEN_ID:
43 case TOKEN_NUM_INTEGER:
44 case TOKEN_NUM_REAL:
45 case TOKEN_LPAREN:
46 case TOKEN_NOT:
47     ret = term(p);
48     ret = simple_expression_prime(p, ret);
49     break;
50 default:
51     ret = TYPE_ERR;
52     expected_found(p, expected, expected_size)
        ;
53     sync(p, sync_set, sync_size);
54 }
55 return ret;
56 }

```

Listing 73: parser/prod/simple-expression_prime.h

```

1  /* -*- C -*-
2  *
3  * simple_expression_prime.h
4  *
5  * Author: Benjamin T James
6  */
7
8 #ifndef SIMPLE_EXPRESSION_PRIME_H
9 #define SIMPLE_EXPRESSION_PRIME_H
10
11 #include "../parser.h"
12

```

```

13 int simple_expression_prime(struct parser *p, int
    type);
14
15 #endif

```

Listing 74: parser/prod/simple-expression_prime.c

```

1  /* -*- C -*-
2  *
3  * simple_expression_prime.c
4  *
5  * Author: Benjamin T James
6  */
7
8 #include "simple_expression_prime.h"
9 #include "term.h"
10
11 int simple_expression_prime(struct parser *p, int
    type)
12 {
13     int sync_set[] = {TOKEN_RELOP, TOKEN_DO,
        TOKEN_ELSE,
14         TOKEN_END, TOKEN_THEN, TOKEN_RPAREN,
        TOKEN_RBRACKET, TOKEN_COMMA,
15         TOKEN_SEMICOLON};
16     int expected[] = {TOKEN_SIGN, TOKEN_ADDOP,
        TOKEN_RELOP,
17         TOKEN_DO, TOKEN_ELSE, TOKEN_END,
        TOKEN_THEN, TOKEN_RPAREN,
18         TOKEN_RBRACKET, TOKEN_COMMA,
        TOKEN_SEMICOLON};
19     const int sync_size = sizeof(sync_set)/sizeof
        (*sync_set);
20     const int expected_size = sizeof(expected)/
        sizeof(*expected);
21     int ret = TYPE_VOID;
22     struct token t_id;
23     switch (p->token.type) {
24 case TOKEN_SIGN: { /* used as addition/
        subtraction here */
25     int term_type;
26     t_id = p->token;
27     match(p, TOKEN_SIGN, &ret);
28     term_type = term(p);
29     if (type != TYPE_ERR && term_type !=
        TYPE_ERR) {
30         if (type != term_type ||
31             (type != TYPE_INT
32              && type != TYPE_REAL) ||
33             (term_type != TYPE_INT
34              && term_type != TYPE_REAL)) {
35             fprintf(p->syn_list, "SEMErr:
                Invalid operands to \"%c\": %s
                and %s\n",
36                 TOKEN_MINUS == t_id.val.attr ? '
                    -' : '+',
37                 strtype(type),
38                 strtype(term_type));
39             ret = TYPE_ERR;
40         }
41     }
42 }
43 }
44 }

```

```

45     ret = simple_expression_prime(p, term_type 15 #endif
46     );
47     break;
48 }
49 case TOKEN_ADDOP: { /* OR */
50     int term_type;
51     match(p, TOKEN_ADDOP, &ret);
52
53     term_type = term(p);
54     if (type != TYPE_ERR && type != TYPE_BOOL
55         && term_type != TYPE_ERR && term_type
56         != TYPE_BOOL) {
57         fprintf(p->syn_list, "SEMERR: Invalid
58             operands to OR: %s and %s\n",
59             strtype(type),
60             strtype(term_type));
61         ret = TYPE_ERR;
62     }
63     term_type = simple_expression_prime(p,
64         term_type);
65     if (ret != TYPE_ERR) {
66         ret = term_type;
67     }
68     break;
69 }
70 case TOKEN_RELOP:
71 case TOKEN_DO:
72 case TOKEN_ELSE:
73 case TOKEN_END:
74 case TOKEN_THEN:
75 case TOKEN_RPAREN:
76 case TOKEN_RBRACKET:
77 case TOKEN_COMMA:
78 case TOKEN_SEMICOLON:
79     ret = type;
80     break;
81 default:
82     ret = TYPE_ERR;
83     expected_found(p, expected, expected_size)
84     ;
85     sync(p, sync_set, sync_size);
86 }
87 return ret;
88 }

```

Listing 75: parser/prod/standard_type.h

```

1  /* -*- C -*-
2  *
3  * standard_type.h
4  *
5  * Author: Benjamin T James
6  */
7
8  #ifndef STANDARD_TYPE_H
9  #define STANDARD_TYPE_H
10
11  #include "../parser.h"
12
13  int standard_type(struct parser *p);
14

```

Listing 76: parser/prod/standard_type.c

```

1  /* -*- C -*-
2  *
3  * standard_type.c
4  *
5  * Author: Benjamin T James
6  */
7
8  #include "standard_type.h"
9
10 int standard_type(struct parser *p)
11 {
12     int sync_set[] = {TOKEN_SEMICOLON,
13         TOKEN_RPAREN};
14     int expected[] = {TOKEN_INTEGER, TOKEN_REAL};
15     const int sync_size = sizeof(sync_set)/sizeof
16         (*sync_set);
17     const int expected_size = sizeof(expected)/
18         sizeof(*expected);
19     int ret = TYPE_VOID;
20     switch (p->token.type) {
21     case TOKEN_INTEGER:
22         ret = TYPE_INT;
23         match(p, TOKEN_INTEGER, &ret);
24         break;
25     case TOKEN_REAL:
26         ret = TYPE_REAL;
27         match(p, TOKEN_REAL, &ret);
28         break;
29     default:
30         ret = TYPE_ERR;
31         expected_found(p, expected, expected_size)
32         ;
33         sync(p, sync_set, sync_size);
34     }
35     return ret;
36 }

```

Listing 77: parser/prod/statement.h

```

1  /* -*- C -*-
2  *
3  * statement.h
4  *
5  * Author: Benjamin T James
6  */
7
8  #ifndef STATEMENT_H
9  #define STATEMENT_H
10
11  #include "../parser.h"
12
13  int statement(struct parser *p);
14
15  #endif

```

Listing 78: parser/prod/statement.c

```

1  /* -*- C -*-

```

```

2  *
3  * statement.c
4  *
5  * Author: Benjamin T James
6  */
7
8  #include "statement.h"
9  #include "compound_statement.h"
10 #include "variable.h"
11 #include "expression.h"
12 #include "statement_prime.h"
13
14 int statement(struct parser *p)
15 {
16     int sync_set[] = {TOKEN_END, TOKEN_ELSE,
17                       TOKEN_SEMICOLON};
18     int expected[] = {TOKEN_BEGIN, TOKEN_ID,
19                       TOKEN_IF, TOKEN_WHILE};
20     const int sync_size = sizeof(sync_set)/sizeof
21     (*sync_set);
22     const int expected_size = sizeof(expected)/
23     sizeof(*expected);
24     int ret = TYPE_VOID;
25     switch (p->token.type) {
26     case TOKEN_BEGIN:
27         err_propagate(compound_statement(p), &ret);
28         ;
29         break;
30     case TOKEN_ID: {
31         int var_type, expr_type;
32         var_type = variable(p);
33         err_propagate(var_type, &ret);
34         match(p, TOKEN_ASSIGN, &ret);
35         expr_type = expression(p);
36         err_propagate(expr_type, &ret);
37         if (var_type != expr_type &&
38             var_type != TYPE_ERR &&
39             expr_type != TYPE_ERR) {
40             fprintf(p->syn_list, "SEMERR: Cannot
41                 assign type \"%s\" to variable of
42                 type \"%s\"\\n", strtype(expr_type),
43                 strtype(var_type));
44             ret = TYPE_ERR;
45         }
46         break;
47     }
48     case TOKEN_IF: {
49         int expr_type;
50         match(p, TOKEN_IF, &ret);
51         expr_type = expression(p);
52         err_propagate(expr_type, &ret);
53         if (expr_type != TYPE_BOOL && expr_type !=
54             TYPE_ERR) {
55             fprintf(p->syn_list, "SEMERR: Argument
56                 to \"if\" must be a boolean
57                 expression\\n");
58             ret = TYPE_ERR;
59         }
60         match(p, TOKEN_THEN, &ret);
61         err_propagate(statement(p), &ret);
62     }
63     case TOKEN_WHILE: {
64         int expr_type;
65         match(p, TOKEN_WHILE, &ret);
66         expr_type = expression(p);
67         err_propagate(expr_type, &ret);
68         if (expr_type != TYPE_BOOL && expr_type !=
69             TYPE_ERR) {
70             fprintf(p->syn_list, "SEMERR: Argument
71                 to \"while\" must be a boolean
72                 expression\\n");
73             ret = TYPE_ERR;
74         }
75         match(p, TOKEN_DO, &ret);
76         err_propagate(statement(p), &ret);
77         break;
78     }
79     default:
80         ret = TYPE_ERR;
81         expected_found(p, expected, expected_size);
82         ;
83         sync(p, sync_set, sync_size);
84     }
85     return ret;
86 }

```

Listing 79: parser/prod/statement_prime.h

```

1  /* -*- C -*-
2  *
3  * statement_prime.h
4  *
5  * Author: Benjamin T James
6  */
7
8  #ifndef STATEMENT_PRIME_H
9  #define STATEMENT_PRIME_H
10
11 #include "../parser.h"
12
13 int statement_prime(struct parser *p);
14
15 #endif

```

Listing 80: parser/prod/statement_prime.c

```

1  /* -*- C -*-
2  *
3  * statement_prime.c
4  *
5  * Author: Benjamin T James
6  */
7
8  #include "statement_prime.h"
9  #include "statement.h"
10
11 int statement_prime(struct parser *p)
12 {

```


<pre> 13 int sync_set[] = {TOKEN_ELSE, TOKEN_END, 14 TOKEN_SEMICOLON}; 15 const int sync_size = sizeof(sync_set)/sizeof 16 (*sync_set); 17 const int expected_size = sizeof(expected)/ 18 sizeof(*expected); 19 int ret = TYPE_VOID; 20 switch (p->token.type) { 21 case TOKEN_ELSE: 22 match(p, TOKEN_ELSE, &ret); 23 statement(p); 24 case TOKEN_SEMICOLON: 25 case TOKEN_END: 26 break; 27 default: 28 ret = TYPE_ERR; 29 expected_found(p, expected, expected_size) 30 ; 31 sync(p, sync_set, sync_size); 32 } 33 return ret; 34 }</pre>	<pre> 15 const int sync_size = sizeof(sync_set)/sizeof 16 (*sync_set); 17 const int expected_size = sizeof(expected)/ 18 sizeof(*expected); 19 int ret = TYPE_VOID; 20 switch (p->token.type) { 21 case TOKEN_BEGIN: 22 case TOKEN_ID: 23 case TOKEN_IF: 24 case TOKEN_WHILE: 25 err_propagate(statement(p), &ret); 26 err_propagate(statement_list_prime(p), & 27 ret); 28 break; 29 case TOKEN_END: 30 break; 31 default: 32 ret = TYPE_ERR; 33 expected_found(p, expected, expected_size) 34 ; 35 sync(p, sync_set, sync_size); 36 } 37 return ret; 38 }</pre>
---	--

Listing 81: parser/prod/statement_list.h

```

1  /* -*- C -*-
2  *
3  * statement_list.h
4  *
5  * Author: Benjamin T James
6  */
7
8  #ifndef STATEMENT_LIST_H
9  #define STATEMENT_LIST_H
10
11 #include "../parser.h"
12
13 int statement_list(struct parser *p);
14
15 #endif
```

Listing 83: parser/prod/statement_list_prime.h

```

1  /* -*- C -*-
2  *
3  * statement_list_prime.h
4  *
5  * Author: Benjamin T James
6  */
7
8  #ifndef STATEMENT_LIST_PRIME_H
9  #define STATEMENT_LIST_PRIME_H
10
11 #include "../parser.h"
12
13 int statement_list_prime(struct parser *p);
14
15 #endif
```

Listing 82: parser/prod/statement_list.c

```

1  /* -*- C -*-
2  *
3  * statement_list.c
4  *
5  * Author: Benjamin T James
6  */
7
8  #include "statement_list.h"
9  #include "statement_list_prime.h"
10 #include "statement.h"
11 int statement_list(struct parser *p)
12 {
13     int sync_set[] = {TOKEN_END};
14     int expected[] = {TOKEN_BEGIN, TOKEN_ID,
15         TOKEN_IF, TOKEN_WHILE};
```

Listing 84: parser/prod/statement_list_prime.c

```

1  /* -*- C -*-
2  *
3  * statement_list_prime.c
4  *
5  * Author: Benjamin T James
6  */
7
8  #include "statement_list_prime.h"
9  #include "statement.h"
10
11 int statement_list_prime(struct parser *p)
12 {
13     int sync_set[] = {TOKEN_END};
14     int expected[] = {TOKEN_SEMICOLON, TOKEN_END};
15     const int sync_size = sizeof(sync_set)/sizeof
16         (*sync_set);
```

```

16     const int expected_size = sizeof(expected)/
17         sizeof(*expected);
18     int ret = TYPE_VOID;
19     switch (p->token.type) {
20     case TOKEN_SEMICOLON:
21         match(p, TOKEN_SEMICOLON, &ret);
22         err_propagate(statement(p), &ret);
23         err_propagate(statement_list_prime(p), &ret);
24         break;
25     case TOKEN_END:
26         break;
27     default:
28         ret = TYPE_ERR;
29         expected_found(p, expected, expected_size)
30         ;
31         sync(p, sync_set, sync_size);
32     }
33     return ret;
34 }

```

Listing 85: parser/prod/subprogram_declaration.h

```

1  /* -*- C -*-
2  *
3  * subprogram_declaration.h
4  *
5  * Author: Benjamin T James
6  */
7
8  #ifndef SUBPROGRAM_DECLARATION_H
9  #define SUBPROGRAM_DECLARATION_H
10
11 #include "../parser.h"
12
13 int subprogram_declaration(struct parser *p);
14
15 #endif

```

Listing 86: parser/prod/subprogram_declaration.c

```

1  /* -*- C -*-
2  *
3  * subprogram_declaration.c
4  *
5  * Author: Benjamin T James
6  */
7
8  #include "subprogram_declaration.h"
9  #include "subprogram_head.h"
10 #include "declarations.h"
11 #include "subprogram_declarations.h"
12 #include "compound_statement.h"
13
14 int subprogram_declaration(struct parser *p)
15 {
16     int sync_set[] = {TOKEN_SEMICOLON};
17     int expected[] = {TOKEN_FUNCTION};
18     const int sync_size = sizeof(sync_set)/sizeof
        (*sync_set);

```

```

19     const int expected_size = sizeof(expected)/
20         sizeof(*expected);
21     int ret = TYPE_VOID;
22     switch (p->token.type) {
23     case TOKEN_FUNCTION:
24         err_propagate(subprogram_head(p), &ret);
25         err_propagate(declarations(p), &ret);
26         err_propagate(subprogram_declarations(p),
27             &ret);
28         err_propagate(compound_statement(p), &ret)
29         ;
30         /* if (check_green_set(&p->gbs) == -1) {
31             /*
32             /* fprintf(p->syn_list, "SEMERR: Function
33                 return value not set\n"); */
34             /* ret = -1; */
35             /* } */
36         if (end_green(&p->gbs) == -1) {
37             fprintf(p->syn_list, "SEMERR: Scope for
38                 function ended but never started\n
39                 ");
40             ret = TYPE_ERR;
41         }
42         break;
43     default:
44         ret = TYPE_ERR;
45         expected_found(p, expected, expected_size)
46         ;
47         sync(p, sync_set, sync_size);
48     }
49     return ret;
50 }

```

Listing 87: parser/prod/subprogram_declarations.h

```

1  /* -*- C -*-
2  *
3  * subprogram_declarations.h
4  *
5  * Author: Benjamin T James
6  */
7
8  #ifndef SUBPROGRAM_DECLARATIONS_H
9  #define SUBPROGRAM_DECLARATIONS_H
10
11 #include "../parser.h"
12
13 int subprogram_declarations(struct parser *p);
14
15 #endif

```

Listing 88: parser/prod/subprogram_declarations.c

```

1  /* -*- C -*-
2  *
3  * subprogram_declarations.c
4  *
5  * Author: Benjamin T James
6  */
7  #include "subprogram_declaration.h"
8  #include "subprogram_declarations.h"

```

```

9
10 int subprogram_declarations(struct parser *p)
11 {
12     int sync_set[] = {TOKEN_BEGIN};
13     int expected[] = {TOKEN_BEGIN, TOKEN_FUNCTION};
14     const int sync_size = sizeof(sync_set)/sizeof
        (*sync_set);
15     const int expected_size = sizeof(expected)/
        sizeof(*expected);
16     int ret = TYPE_VOID;
17     switch (p->token.type) {
18     case TOKEN_BEGIN:
19         break;
20     case TOKEN_FUNCTION:
21         err_propagate(subprogram_declaration(p), &
            ret);
22         match(p, TOKEN_SEMICOLON, &ret);
23         err_propagate(subprogram_declarations(p),
            &ret);
24         break;
25     default:
26         ret = TYPE_ERR;
27         expected_found(p, expected, expected_size)
            ;
28         sync(p, sync_set, sync_size);
29     }
30     return ret;
31 }

```

Listing 89: parser/prod/subprogram_head.h

```

1 /* -*- C -*-
2  *
3  * subprogram_head.h
4  *
5  * Author: Benjamin T James
6  */
7
8 #ifndef SUBPROGRAM_HEAD_H
9 #define SUBPROGRAM_HEAD_H
10
11 #include "../parser.h"
12
13 int subprogram_head(struct parser *p);
14
15 #endif

```

Listing 90: parser/prod/subprogram_head.c

```

1 /* -*- C -*-
2  *
3  * subprogram_head.c
4  *
5  * Author: Benjamin T James
6  */
7
8 #include "subprogram_head.h"
9 #include "arguments.h"
10 #include "standard_type.h"
11

```

```

12 int subprogram_head(struct parser *p)
13 {
14     int sync_set[] = {TOKEN_BEGIN, TOKEN_FUNCTION,
        TOKEN_VAR};
15     int expected[] = {TOKEN_FUNCTION};
16     const int sync_size = sizeof(sync_set)/sizeof
        (*sync_set);
17     const int expected_size = sizeof(expected)/
        sizeof(*expected);
18     int ret = TYPE_VOID;
19     char *lexeme = NULL;
20     char *args = NULL;
21     struct token t_id;
22     int ret_type = TYPE_VOID;
23     switch (p->token.type) {
24     case TOKEN_FUNCTION:
25         match(p, TOKEN_FUNCTION, &ret);
26         t_id = p->token;
27         match(p, TOKEN_ID, &ret);
28         lexeme = parser_get_tok_lex(p, t_id);
29         if (lexeme != NULL) {
30             if (check_add_green(&p->gbs, lexeme,
                TYPE_PLACEHOLDER) == -1) {
31                 fprintf(p->syn_list, "SEMERR: Scope
                    \"%s\" already exists\n",
                        lexeme);
32                 ret = TYPE_ERR;
33             }
34         }
35         err_propagate(arguments(p, &args), &ret);
36         match(p, TOKEN_COLON, &ret);
37         ret_type = standard_type(p);
38         err_propagate(ret_type, &ret);
39         ret_type = SCALAR_TO_FUNC(ret_type);
40         set_peek_args_type(&p->gbs, ret_type, args
            );
41         free(args);
42         match(p, TOKEN_SEMICOLON, &ret);
43         break;
44     default:
45         ret = TYPE_ERR;
46         expected_found(p, expected, expected_size)
            ;
47         sync(p, sync_set, sync_size);
48     }
49     return ret;
50 }

```

Listing 91: parser/prod/term.h

```

1 /* -*- C -*-
2  *
3  * term.h
4  *
5  * Author: Benjamin T James
6  */
7
8 #ifndef TERM_H
9 #define TERM_H
10
11 #include "../parser.h"

```

```

12
13 int term(struct parser *p);
14
15 #endif

```

Listing 92: parser/prod/term.c

```

1  /* -*- C -*-
2  *
3  * term.c
4  *
5  * Author: Benjamin T James
6  */
7
8  #include "term.h"
9  #include "factor.h"
10 #include "term_prime.h"
11
12 int term(struct parser *p)
13 {
14     int sync_set[] = {TOKEN_ADDOP, TOKEN_SIGN,
15                       TOKEN_RELOP,
16                       TOKEN_DO, TOKEN_ELSE, TOKEN_END,
17                       TOKEN_THEN, TOKEN_LPAREN,
18                       TOKEN_RBRACKET, TOKEN_COMMA,
19                       TOKEN_SEMICOLON};
20     int expected[] = {TOKEN_ID, TOKEN_NUM_INTEGER,
21                      TOKEN_NUM_REAL, TOKEN_LPAREN,
22                      TOKEN_NOT};
23     const int sync_size = sizeof(sync_set)/sizeof
24         (*sync_set);
25     const int expected_size = sizeof(expected)/
26         sizeof(*expected);
27     int ret = TYPE_VOID;
28     switch (p->token.type) {
29     case TOKEN_ID:
30     case TOKEN_NUM_INTEGER:
31     case TOKEN_NUM_REAL:
32     case TOKEN_LPAREN:
33     case TOKEN_NOT:
34         ret = factor(p);
35         ret = term_prime(p, ret);
36         break;
37     default:
38         ret = TYPE_ERR;
39         expected_found(p, expected, expected_size)
40         ;
41         sync(p, sync_set, sync_size);
42     }
43     return ret;
44 }

```

Listing 93: parser/prod/term_prime.h

```

1  /* -*- C -*-
2  *
3  * term_prime.h
4  *
5  * Author: Benjamin T James
6  */
7

```

```

8  #ifndef TERM_PRIME_H
9  #define TERM_PRIME_H
10
11 #include "../parser.h"
12
13 int term_prime(struct parser *p, int type);
14
15 #endif

```

Listing 94: parser/prod/term_prime.c

```

1  /* -*- C -*-
2  *
3  * term_prime.c
4  *
5  * Author: Benjamin T James
6  */
7
8  #include "term_prime.h"
9  #include "factor.h"
10
11 int mulop(struct parser *p, int t1, int op, int t2
12     , int *p_ret);
13 int term_prime(struct parser *p, int type)
14 {
15     int sync_set[] = {TOKEN_SIGN, TOKEN_ADDOP,
16                       TOKEN_RELOP,
17                       TOKEN_DO, TOKEN_ELSE, TOKEN_END,
18                       TOKEN_THEN, TOKEN_LPAREN,
19                       TOKEN_RBRACKET, TOKEN_COMMA,
20                       TOKEN_SEMICOLON};
21     int expected[] = {TOKEN_MULOP, TOKEN_SIGN,
22                      TOKEN_ADDOP, TOKEN_RELOP,
23                      TOKEN_DO, TOKEN_ELSE, TOKEN_END,
24                      TOKEN_THEN, TOKEN_LPAREN,
25                      TOKEN_RBRACKET, TOKEN_COMMA,
26                      TOKEN_SEMICOLON};
27     const int sync_size = sizeof(sync_set)/sizeof
28         (*sync_set);
29     const int expected_size = sizeof(expected)/
30         sizeof(*expected);
31     int ret = TYPE_VOID;
32     switch (p->token.type) {
33     case TOKEN_MULOP: {
34         int ftype, op;
35         op = p->token.val.attr;
36         match(p, TOKEN_MULOP, &ret);
37         ftype = factor(p);
38         mulop(p, type, op, ftype, &ret);
39         ret = term_prime(p, ret);
40         break;
41     }
42     case TOKEN_SIGN: /* used as addition/
43                       subtraction here */
44     case TOKEN_ADDOP:
45     case TOKEN_RELOP:
46     case TOKEN_DO:
47     case TOKEN_ELSE:
48     case TOKEN_END:
49     case TOKEN_THEN:
50     case TOKEN_LPAREN:

```

45	case TOKEN_RBRACKET:	98	*p_ret = TYPE_INT;
46	case TOKEN_COMMA:	99	}
47	case TOKEN_SEMICOLON:	100	} else if (op == TOKEN_MOD) {
48	ret = type;	101	if (t1 != TYPE_INT t2 != TYPE_INT) {
49	break;	102	fprintf(p->syn_list, "SEMERR: Invalid
50	default:		operands to \"mod\": %s and %s\n",
51	ret = TYPE_ERR;	103	strtype(t1),
52	expected_found(p, expected, expected_size)	104	strtype(t2));
	;	105	*p_ret = TYPE_ERR;
53	sync(p, sync_set, sync_size);	106	} else {
54	}	107	*p_ret = TYPE_INT;
55	return ret;	108	}
56	}	109	} else {
57		110	fprintf(stderr, "unknown operation %d\n",
58	int mulop(struct parser *p, int t1, int op, int t2		op);
	, int *p_ret)	111	}
59	{	112	return *p_ret;
60	if (t1 == TYPE_ERR t2 == TYPE_ERR) {	113	}
61	*p_ret = TYPE_ERR;		
62	} else if (op == TOKEN_AND) {		
63	if (t1 != TYPE_BOOL t2 != TYPE_BOOL) {		
64	fprintf(p->syn_list, "SEMERR: Invalid	1	/* -*- C -*-
	operands to \"and\": %s and %s\n",	2	*
65	strtype(t1),	3	* type.h
66	strtype(t2));	4	*
67	*p_ret = TYPE_ERR;	5	* Author: Benjamin T James
68	} else {	6	*/
69	*p_ret = TYPE_BOOL;	7	
70	}	8	#ifndef TYPE_H
71	} else if (op == TOKEN_TIMES) {	9	#define TYPE_H
72	if (t1 == TYPE_INT && t2 == TYPE_INT) {	10	
73	*p_ret = TYPE_INT;	11	#include "../parser.h"
74	} else if (t1 == TYPE_REAL && t2 ==	12	
	TYPE_REAL) {	13	int type(struct parser *p, int *width);
75	*p_ret = TYPE_REAL;	14	
76	} else {	15	#endif
77	fprintf(p->syn_list, "SEMERR: Invalid		
	operations to \"*\": %s and %s\n",		
78	strtype(t1),		
79	strtype(t2));		
80	*p_ret = TYPE_ERR;	1	/* -*- C -*-
81	}	2	*
		3	* type.c
		4	*
82	} else if (op == TOKEN_RDIV) {	5	* Author: Benjamin T James
83	if (t1 != TYPE_REAL t2 != TYPE_REAL) {	6	*/
84	fprintf(p->syn_list, "SEMERR: Invalid	7	
	operands to \"/\": %s and %s\n",	8	#include "type.h"
85	strtype(t1),	9	#include "standard_type.h"
86	strtype(t2));	10	int type(struct parser *p, int *width)
87	*p_ret = TYPE_ERR;	11	{
88	} else {	12	int sync_set[] = {TOKEN_SEMICOLON,
89	*p_ret = TYPE_REAL;		TOKEN_RPAREN};
90	}	13	int expected[] = {TOKEN_INTEGER, TOKEN_REAL,
91	} else if (op == TOKEN_IDIV) {		TOKEN_ARRAY};
92	if (t1 != TYPE_INT t2 != TYPE_INT) {	14	const int sync_size = sizeof(sync_set)/sizeof
93	fprintf(p->syn_list, "SEMERR: Invalid		(*sync_set);
	operands to \"div\": %s and %s\n",	15	const int expected_size = sizeof(expected)/
94	strtype(t1),		sizeof(*expected);
95	strtype(t2));	16	int a_begin, a_end, a_len;
96	*p_ret = TYPE_ERR;	17	int ret = TYPE_VOID;
97	} else {	18	switch (p->token.type) {

Listing 95: parser/prod/type.h

Listing 96: parser/prod/type.c

```

19 case TOKEN_INTEGER:
20 case TOKEN_REAL:
21     ret = standard_type(p);
22     *width = type_width(ret);
23     break;
24 case TOKEN_ARRAY:
25     match(p, TOKEN_ARRAY, &ret);
26     match(p, TOKEN_LBRACKET, &ret);
27     a_begin = p->token.val.attr;
28     match(p, TOKEN_NUM_INTEGER, &ret);
29     if (a_begin != 1) {
30         fprintf(p->syn_list, "SEMERR: Arrays
31             must start at 1\n");
32         ret = TYPE_ERR;
33     }
34     match(p, TOKEN_ELLIPSIS, &ret);
35     a_end = p->token.val.attr;
36     match(p, TOKEN_NUM_INTEGER, &ret);
37     a_len = a_end - a_begin + 1;
38     if (a_end < a_begin) {
39         fprintf(p->syn_list, "SEMERR: Must have
40             a positive length array (Length is
41             %d)\n", a_len);
42         ret = TYPE_ERR;
43     }
44     match(p, TOKEN_RBRACKET, &ret);
45     match(p, TOKEN_OF, &ret);
46     ret = standard_type(p);
47     *width = type_width(ret) * a_len;
48     ret = SCALAR_TO_ARRAY(ret);
49     break;
50 default:
51     ret = TYPE_ERR;
52     expected_found(p, expected, expected_size)
53     ;
54     sync(p, sync_set, sync_size);
55 }
56 return ret;
57 }

```

Listing 97: parser/prod/variable.h

```

1 /* -*- C -*-
2 *
3 * variable.h
4 *
5 * Author: Benjamin T James
6 */

```

```

7
8 #ifndef VARIABLE_H
9 #define VARIABLE_H
10
11 #include "../parser.h"
12
13 int variable(struct parser *p);
14
15 #endif

```

Listing 98: parser/prod/variable.c

```

1 /* -*- C -*-

```

```

2 *
3 * variable.c
4 *
5 * Author: Benjamin T James
6 */
7
8 #include "variable.h"
9 #include "variable_prime.h"
10
11 int variable(struct parser *p)
12 {
13     int sync_set[] = {TOKEN_ASSIGN};
14     int expected[] = {TOKEN_ID};
15     const int sync_size = sizeof(sync_set)/sizeof
16         (*sync_set);
17     const int expected_size = sizeof(expected)/
18         sizeof(*expected);
19     int ret = TYPE_VOID, type = TYPE_ERR;
20     struct token t_id;
21     char *lexeme = NULL;
22     switch (p->token.type) {
23     case TOKEN_ID:
24         t_id = p->token;
25         match(p, TOKEN_ID, &ret);
26         lexeme = parser_get_tok_lex(p, t_id);
27         if (lexeme != NULL &&
28             gettype(&p->gbs, lexeme, &type) < 0) {
29             fprintf(p->syn_list, "SEMERR: Variable
30                 \"%s\" was not declared in this
31                 scope\n", lexeme);
32             ret = TYPE_ERR;
33         }
34         if (FUNC_TO_SCALAR(type) != TYPE_ERR) {
35             /* function return value */
36             type = FUNC_TO_SCALAR(type);
37         }
38         /* synthesized variable type */
39         ret = variable_prime(p, type);
40         break;
41     default:
42         ret = TYPE_ERR;
43         expected_found(p, expected, expected_size)
44         ;
45         sync(p, sync_set, sync_size);
46     }
47     return ret;
48 }

```

Listing 99: parser/prod/variable_prime.h

```

1 /* -*- C -*-
2 *
3 * variable_prime.h
4 *
5 * Author: Benjamin T James
6 */
7
8 #ifndef VARIABLE_PRIME_H
9 #define VARIABLE_PRIME_H
10
11 #include "../parser.h"

```

12		23	
13	int variable_prime(struct parser *p, int type);	24	
14		25	
15	#endif	26	
		27	
		28	
	Listing 100: parser/prod/variable_prime.c	29	
1	/* -*- C -*-	30	
2	*		
3	* variable_prime.c	31	
4	*		
5	* Author: Benjamin T James		
6	*/	32	
7		33	
8	#include "variable_prime.h"	34	
9	#include "expression.h"	35	
10	int variable_prime(struct parser *p, int type)	36	
11	{	37	
12	int sync_set[] = {TOKEN_ASSIGN};	38	
13	int expected[] = {TOKEN_LBRACKET, TOKEN_ASSIGN	39	
	};	40	
14	const int sync_size = sizeof(sync_set)/sizeof	41	
	(*sync_set);	42	
15	const int expected_size = sizeof(expected)/	43	
	sizeof(*expected);	44	
16	int ret = TYPE_VOID;	45	
17	switch (p->token.type) {	46	
18	case TOKEN_LBRACKET: {/* type array */		
19	int expr_type, scalar_type, rtype;	47	
20	scalar_type = ARRAY_TO_SCALAR(type);	48	
21	if (scalar_type == TYPE_ERR) {	49	
22	fprintf(p->syn_list, "SEMERR: Attempted	50	
	indexing of non-array type %s\n",	51	
	strtype(type));		

```

        rtype = TYPE_ERR;
    } else {
        rtype = scalar_type;
    }

    match(p, TOKEN_LBRACKET, &ret);
    expr_type = expression(p);
    if (expr_type != TYPE_ERR && expr_type !=
        TYPE_INT) {
        fprintf(p->syn_list, "SEMERR: Array
            index must be an integer (found %s)
            \n",
            strtype(expr_type));
        ret = TYPE_ERR;
    }
    match(p, TOKEN_RBRACKET, &ret);
    if (ret != TYPE_ERR) {
        ret = rtype;
    }
    break;
}

case TOKEN_ASSIGN:
    ret = type;
    break;
default:
    ret = TYPE_ERR;
    expected_found(p, expected, expected_size)
        ;
    sync(p, sync_set, sync_size);
}
return ret;

```
