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* $\rightarrow$ **program** {{offset= 0}} **id** {{checkaddgreen(*id*.lex, TYPE_PGM)}} **(** *identifier_list* **)** **;** *declarations subprogram_declarations compound_statement* **.**

2.1.1 void *identifier_list* $\rightarrow$ **id** {{checkaddblue(*id*.lex, TYPE_IDLIST)}} *identifier_list′*

2.2.1 void *identifier_list′* $\rightarrow$ **, id** {{checkaddblue(*id*.lex, TYPE_IDLIST)}} *identifier_list′*

2.2.2 void *identifier_list′* $\rightarrow \epsilon$

3.1.1 void *declarations* $\rightarrow$ **var id :** *type* {{checkaddblue(*id*.lex, *type*.type, offset); offset += *type*.width}} **;** *declarations*

3.2.1 void *declarations* $\rightarrow \epsilon$

4.1 *type*.type *type* $\rightarrow$ *standard_type* {{*type*.type = *standard_type*.type; *type*.width = *standard_type*.width}}

4.2 *type*.type *type* $\rightarrow$ **array [num .. num] of** *standard_type* {{*type*.width = $(num_2 - num_1 + 1)$ * *standard_type*.width}}

type.type	←	*standard_type*.type
TYPE_AINT	if	TYPE_INT
TYPE_AREAL	if	TYPE_REAL
ERR	if	ERR
ERR*	otherwise	

5.1 *standard_type*.type *standard_type* $\rightarrow$ **integer** {{*standard_type*.type = TYPE_INT; *standard_type*.width = 4}}

5.2 *standard_type*.type *standard_type* $\rightarrow$ **real** {{*standard_type*.type = TYPE_REAL; *standard_type*.width = 8}}

6.1.1 void *subprogram_declarations* $\rightarrow$ *subprogram_declaration* **;** *subprogram_declarations*

6.2.1 void *subprogram_declarations* $\rightarrow \epsilon$

7 void *subprogram_declaration* $\rightarrow$ *subprogram_head declarations subprogram_declarations compound_statement* {{endgreenscope();}}

8 void *subprogram_head* $\rightarrow$ **function id** {{checkaddgreen(*id*.lex, TYPE_PLACEHOLDER)}} *arguments* **:** *standard_type* {{eye_stack.peek().args = *arguments*.str}} **;**

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

9.1 *arguments*.str *arguments* $\rightarrow$ **(** *parameter_list* **)** {{*arguments*.str = *parameter_list*.str}}

9.2 *arguments*.str *arguments* $\rightarrow \epsilon$ {{*arguments*.str = “”}}

10.1.1 *parameter_list*.str *parameter_list* $\rightarrow$ **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′* $\rightarrow$ **; 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′* $\rightarrow \epsilon$ {{*parameter_list′*.str = “”}}

11 void *compound_statement* $\rightarrow$ **begin** *optional_statements* **end**

12.1 void *optional_statements* $\rightarrow$ *statement_list*

12.2 void *optional_statements* $\rightarrow \epsilon$

13.1.1 void *statement_list* $\rightarrow$ *statement statement_list′*

13.2.1 void *statement_list′* $\rightarrow$ **;** *statement statement_list′*

13.2.2 void *statement_list′* $\rightarrow \epsilon$

14.1.1 void *statement* $\rightarrow$ *variable* **assignop** *expression*

statement.type	←	*variable*.type	*expression*.type
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* $\rightarrow$ *compound_statement*

14.3.1 void *statement* $\rightarrow$ **if** *expression* {{check(*expression*.type == TYPE_BOOL)}} **then** *statement statement′*

14.4.1 void *statement′* $\rightarrow$ **else** *statement*

14.4.2 void *statement′* $\rightarrow \epsilon$

14.5.1 void *statement* $\rightarrow$ **while** *expression* {{check(*expression*.type == TYPE_BOOL)}} **do** *statement*

15.1.1 *variable*.type *variable* $\rightarrow$ **id** {{*variable′*.i = gettype(*id*.lex)}} *variable′* {{*variable*.type = *variable′*.type}}

15.2.1 *variable′*.type *variable′* $\rightarrow$ [*expression*]

$\mathit{variable}'$.type	←	$\mathit{expression}$.type	$\mathit{variable}'$.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	¬TYPE_INT	
ERR*	if		¬TYPE_AINT *and* ¬TYPE_AREAL

15.2.2 $\mathit{variable}'$.type $\mathit{variable}' \to \epsilon$ $\{\{\mathit{variable}'.\text{type} = \mathit{variable}'.\text{i}\}\}$
16.1.1 $\mathit{expression\_list}$.str $\mathit{expression\_list} \to \mathit{expression}\ \mathit{expression\_list}'$
$\{\{\mathit{expression\_list}.\text{str} = \text{type2str}(\mathit{expression}.\text{type})\ .\ \mathit{expression\_list}'.\text{str}\}\}$
16.2.1 $\mathit{expression\_list}'$.str $\mathit{expression\_list}' \to \textbf{,}\ \mathit{expression}\ \mathit{expression\_list}'$
$\{\{\mathit{expression\_list}'.\text{str} = \text{type2str}(\mathit{expression}.\text{type})\ .\ \mathit{expression\_list}_1{}'.\text{str}\}\}$
16.2.2 $\mathit{expression\_list}'$.str $\mathit{expression\_list}' \to \epsilon$ $\{\{\mathit{expression\_list}'.\text{str} = \text{“”}\}\}$
17.1.1 $\mathit{expression}$.type $\mathit{expression} \to \mathit{simple\_expression}$ $\{\{\mathit{expression}'.\text{i} = \mathit{simple\_expression}.\text{type}\}\}$
$\mathit{expression}'$ $\{\{\mathit{expression}.\text{type} = \mathit{expression}'.\text{type}\}\}$
17.2.1 $\mathit{expression}'$.type $\mathit{expression}' \to \epsilon$ $\{\{\mathit{expression}'.\text{type} = \mathit{expression}'.\text{i}\}\}$
17.2.2 $\mathit{expression}'$.type $\mathit{expression}' \to$ **relop** $\mathit{simple\_expression}$

$\mathit{expression}'$.type	←	$\mathit{simple\_expression}$.type	$\mathit{expression}'$.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 $\mathit{simple\_expression}$.type $\mathit{simple\_expression} \to \mathit{term}$ $\{\{\mathit{simple\_expression}'.\text{i} = \mathit{term}.\text{type}\}\}$
$\mathit{simple\_expression}'$ $\{\{\mathit{simple\_expression}.\text{type} = \mathit{simple\_expression}'.\text{type}\}\}$
18.2.1 $\mathit{simple\_expression}$.type $\mathit{simple\_expression} \to \mathit{sign}\ \mathit{term}$
$\{\{\text{ERR* if } \mathit{term}.\text{type} \notin \{\text{TYPE\_REAL,TYPE\_INT,ERR}\}\ \}\}$
$\{\{\mathit{simple\_expression}'.\text{i} = \mathit{term}.\text{type}\}\}$ $\mathit{simple\_expression}'$ $\{\{\mathit{simple\_expression}.\text{type} = \mathit{simple\_expression}'.\text{type}\}\}$
18.3.1 $\mathit{simple\_expression}'$.type $\mathit{simple\_expression}' \to$ **addop** $\mathit{term}\ \mathit{simple\_expression}'$
$\{\{\mathit{simple\_expression}'.\text{type} = \mathit{simple\_expression}'_1.\text{type}\}\}$

$\mathit{simple\_expression}'_1$.i	←	$\mathit{simple\_expression}'$.i	**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 $\mathit{simple\_expression}'$.type $\mathit{simple\_expression}' \to \epsilon$ $\{\{\mathit{simple\_expression}'.\text{type} = \mathit{simple\_expression}'.\text{i}\}\}$
19.1.1 term.type $\mathit{term} \to \mathit{factor}$ $\{\{\mathit{term}'.\text{i} = \mathit{factor}.\text{type}\}\}$ term' $\{\{\mathit{term}.\text{type} = \mathit{term}'.\text{type}\}\}$
19.2.1 term'.type $\mathit{term}' \to$ **mulop** $\mathit{factor}\ \mathit{term}'$ $\{\{\mathit{term}.\text{type} = \mathit{term}'.\text{type}\}\}$

term'_1.i	←	term'.i	**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 $\mathit{term}' \to \epsilon$ $\{\{\mathit{term}'.\text{type} = \mathit{term}'.\text{i}\}\}$

20.1.1 $factor$.type *factor* → **id** {{$factor'$.i = gettype(id.lex)}} *factor′* {{$factor$.type = $factor'$.type }}
20.2.1 $factor'$.type *factor′* → [*expression*]

$factor'$.type	←	$expression$.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	¬TYPE_INT	
ERR*	if		¬TYPE_AINT *and* ¬TYPE_AREAL

20.2.2 $factor'$.type *factor′* → ϵ {{$factor'$.type = $factor'$.i if *declared and* ∈ {TYPE_INT, TYPE_REAL} }}
20.3.1 $factor'$.type *factor′* → (*expression_list*)
{{$factor'$.type = funtype˙to˙scalar($factor'$.i); check($expression\_list$.str == get˙args($factor'$.i)) }}

$factor'$.type	←	$factor'$.i
ERR*	if	Undeclared
ERR*	if	¬TYPE_FINT *and* ¬TYPE_FREAL
TYPE_INT	if	TYPE_FINT
TYPE_REAL	if	TYPE_FREAL

20.4.1 $factor$.type *factor* → **num** {{$factor$.type = **num**.type }}
20.5.1 $factor$.type *factor* → (*expression*) {{$factor$.type = $expression$.type }}
20.6.1 $factor$.type *factor* → **not** *factor* {{$factor$.type = $factor_1$.type *unless* $factor_1$.type ∉ {TYPE_BOOL,ERR} }}
21.1 void *sign* → **+**
21.2 void *sign* → **-**

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

```
abcdefghij
abcdefghijk
@

12345678901
1234567890

12345.3
123456.3

1.12345
1.123456


123
0123
01.2
01.2E2

1230
1.2
1.20
1.20E-12

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


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

abcdefghijklmnopqrstuvwxyz0123456789abcdefghijklmnopqrstuvwxyz012345678
abcdefghijklmnopqrstuvwxyz0123456789abcdefghijklmnopqrstuvwxyz0123456789
```

Listing 2: bad_lex.list

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

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

Listing 3: bad_lex.tok

```
1       abcdefghij        1     0x55420f0
2       abcdefghijk       99    1
3       @         99    2
5       12345678901       99    3
6       1234567890        6     1234567890
8       12345.3   5     12345
9       123456.3          99    4
11      1.12345   5     1
12      1.123456          99    5
15      123       6     123
16      0123      99    6
17      01.2      99    6
18      01.2E2    99    6
20      1230      6     1230
```

```
21      1.2      5     1
22      1.20     99    7
23      1.20E-12       99    7
25      1.2E-10  99    7
26      1.2E-123       99    9
27      1.2E+5   5     120000
28      1.2E+123       99    9
31      e        1     0x5544a00
31      #        99    2
32      3.4E+    99    10
32      ;        33    59
33      3.E4     99    11
33      +        7     56
33      ;        33    59
34      34       6     34
34      E        1     0x5545010
34      +        7     56
34      -        7     57
34      ;        33    59
35      E3       1     0x55452b0
35      .        25    46
35      4        6     4
35      +        7     56
35      ;        33    59
37      abcdefghijklmnopqrstuvwxyz0123456789abcdefghijklmnopqrstuvwxyz012345678    99
       1
39      EOF      0     0
```

bad_syn

Listing 4: bad_syn.pas

```
program test(input, output);
        var a : integer;
        var b : array[0..a] of real;
        function f1(a : integer, x : int) : real ;
                var c : integer;
                function f2(p : integer) : integer ;
                var d : real;
                begin
                        f2 := 5 + p;
                        if p > d then
                                if p = 5 then
                                        f2 := f + 10
                        else
                                f2 := 100
                end
        begin
                f1 := f2(a) * x
        end
        function f2(a : real) : real ;
        var b : integer;
        begin
                f3 := 10 * f1(a, 2.3);
        end
begin
        f1(5, 3.2)
end.
begin

end
```

Listing 5: bad_syn.list

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

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

Listing 6: bad_syn.sym

```
a       0       integer
b       4       array of real
c       0       integer
d       0       real
```

Listing 7: bad_syn.tok

```
1       program   20    0
1       test      1     0x5543220
1       (         27    40
1       input     1     0x55433d0
1       ,         34    44
1       output    1     0x55435d0
1       )         28    41
1       ;         33    59
2       var       23    0
2       a         1     0x5543960
2       :         32    58
2       integer   17    0
2       ;         33    59
3       var       23    0
3       b         1     0x5543e30
3       :         32    58
3       array     10    0
3       [         29    91
3       0         6     0
3       ..        26    26
3       a         1     0x5543960
3       ]         30    93
3       of        19    0
3       real      21    0
3       ;         33    59
4       function        15    0
4       f1        1     0x5544710
4       (         27    40
4       a         1     0x5543960
4       :         32    58
4       integer   17    0
4       ,         34    44
4       x         1     0x5544b90
4       :         32    58
4       int       1     0x5544de0
4       )         28    41
4       :         32    58
4       real      21    0
4       ;         33    59
5       var       23    0
5       c         1     0x5545350
5       :         32    58
5       integer   17    0
5       ;         33    59
6       function        15    0
6       f2        1     0x5545820
6       (         27    40
6       p         1     0x55459d0
6       :         32    58
6       integer   17    0
6       )         28    41
6       :         32    58
6       integer   17    0
6       ;         33    59
7       var       23    0
7       d         1     0x55460d0
7       :         32    58
7       real      21    0
7       ;         33    59
8       begin     11    0
9       f2        1     0x5545820
9       :=        31    0
9       5         6     5
9       +         7     56
9       p         1     0x55459d0
```

```
9         ;         33    59
10        if        16    0
10        p         1     0x55459d0
10        >         4     45
10        d         1     0x55460d0
10        then      22    0
11        if        16    0
11        p         1     0x55459d0
11        =         4     43
11        5         6     5
11        then      22    0
12        f2        1     0x5545820
12        :=        31    0
12        f         1     0x5547590
12        +         7     56
12        10        6     10
13        else      13    0
14        f2        1     0x5545820
14        :=        31    0
14        100       6     100
15        end       14    0
16        begin     11    0
17        f1        1     0x5544710
17        :=        31    0
17        f2        1     0x5545820
17        (         27    40
17        a         1     0x5543960
17        )         28    41
17        *         3     50
17        x         1     0x5544b90
18        end       14    0
19        function        15    0
19        f2        1     0x5545820
19        (         27    40
19        a         1     0x5543960
19        :         32    58
19        real      21    0
19        )         28    41
19        :         32    58
19        real      21    0
19        ;         33    59
20        var       23    0
20        b         1     0x5543e30
20        :         32    58
20        integer   17    0
20        ;         33    59
21        begin     11    0
22        f3        1     0x5549310
22        :=        31    0
22        10        6     10
22        *         3     50
22        f1        1     0x5544710
22        (         27    40
22        a         1     0x5543960
22        ,         34    44
22        2.3       5     2
22        )         28    41
22        ;         33    59
23        end       14    0
24        begin     11    0
25        f1        1     0x5544710
25        (         27    40
25        5         6     5
25        ,         34    44
25        3.2       5     3
25        )         28    41
26        end       14    0
26        .         25    46
27        begin     11    0
29        end       14    0
30        EOF       0     0
```

bad_sem

Listing 8: bad_sem.pas

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

begin
        out := read(x, y);
        out := write(gcd(x, y));
        out := 0
end.
```

Listing 9: bad_sem.list

```
1       program example(input, output);
2       var x: integer; var y: integer;
3       function gcd(a:integer; b: integer): integer;
4       begin
5               if b = 0 then gcd := a
6               else gcd := gcd(b, a mod b)
7       end;
8
9       begin
10              out := read(x, y);
SEMERR: Variable "out" was not declared in this scope
SEMERR: Function "read" not declared in this scope
11              out := write(gcd(x, y));
SEMERR: Variable "out" was not declared in this scope
SEMERR: Function "write" not declared in this scope
12              out := 0
SEMERR: Variable "out" was not declared in this scope
13      end.
```

Listing 10: bad_sem.sym

```
x       0       integer
y       4       integer
```

Listing 11: bad_sem.tok

```
1       program    20     0
1       example    1      0x55431c0
1       (          27     40
1       input      1      0x5543370
1       ,          34     44
1       output     1      0x5543570
1       )          28     41
1       ;          33     59
2       var        23     0
2       x          1      0x55438b0
2       :          32     58
2       integer    17     0
2       ;          33     59
2       var        23     0
2       y          1      0x5543ce0
2       :          32     58
2       integer    17     0
2       ;          33     59
3       function          15     0
3       gcd        1      0x5544110
3       (          27     40
3       a          1      0x55442c0
3       :          32     58
3       integer    17     0
3       ;          33     59
3       b          1      0x55445b0
3       :          32     58
3       integer    17     0
3       )          28     41
3       :          32     58
3       integer    17     0
```

```
3         ;         33     59
4         begin     11     0
5         if        16     0
5         b         1      0x55445b0
5         =         4      43
5         0         6      0
5         then      22     0
5         gcd       1      0x5544110
5         :=        31     0
5         a         1      0x55442c0
6         else      13     0
6         gcd       1      0x5544110
6         :=        31     0
6         gcd       1      0x5544110
6         (         27     40
6         b         1      0x55445b0
6         ,         34     44
6         a         1      0x55442c0
6         mod       3      53
6         b         1      0x55445b0
6         )         28     41
7         end       14     0
7         ;         33     59
9         begin     11     0
10        out       1      0x5545d40
10        :=        31     0
10        read      1      0x5545f90
10        (         27     40
10        x         1      0x55438b0
10        ,         34     44
10        y         1      0x5543ce0
10        )         28     41
10        ;         33     59
11        out       1      0x5545d40
11        :=        31     0
11        write     1      0x55465f0
11        (         27     40
11        gcd       1      0x5544110
11        (         27     40
11        x         1      0x55438b0
11        ,         34     44
11        y         1      0x5543ce0
11        )         28     41
11        )         28     41
11        ;         33     59
12        out       1      0x5545d40
12        :=        31     0
12        0         6      0
13        end       14     0
13        .         25     46
14        EOF       0      0
```

cor34

Listing 12: cor34.pas

```

program test (input, output);
  var a : integer;
  var b : real;
  var c : array [1..2] of integer;

  function fun1(x:integer; y:real;
                  z:array [1..2] of integer;
                  q: real) : integer;
    var d: integer;
    begin
      a:= 2;
      z[a] := 4;
      c[2] := 3;
      fun1 := c[1]
     end;

   function fun2(x: integer; y: integer) : real;
     var e: real;

     function fun3(n: integer; z: real) : integer;
       var e: integer;
       begin
         a:= e;
         e:= c[e];
         fun3 := 3
       end;

     begin
       a:= fun1(x, e, c, b);
       x:= fun3(c[1], e);
       e := e + 4.44;
       a:= (a mod y) div x;
       while ((a >= 4) and ((b <= e)
                       or (not (a = c[a])))) do
         begin
           a:= c[a] + 1
         end;
       fun2 := 2.5
     end;

begin
  b:= fun2(c[4], c[5]);
  b:= fun2(c[4],2);
  if (a < 2) then a:= 1 else a := a + 2;
  if (b > 4.2) then a := c[a]
end.
```

Listing 13: cor34.list

```
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.
```

Listing 14: cor34.sym

```
a         0         integer
b         4         real
c         12        array of integer
d         0         integer
e         0         real
e         0         integer
```

Listing 15: cor34.tok

```
2         program    20       0
2         test       1        0x5543210
2         (          27       40
2         input      1        0x5543410
2         ,          34       44
2         output     1        0x5543610
2         )          28       41
2         ;          33       59
3         var        23       0
3         a          1        0x55439a0
3         :          32       58
3         integer    17       0
3         ;          33       59
4         var        23       0
4         b          1        0x5543e70
4         :          32       58
4         real       21       0
4         ;          33       59
5         var        23       0
5         c          1        0x5544340
5         :          32       58
5         array      10       0
5         [          29       91
5         1          6        1
5         ..         26       26
5         2          6        2
5         ]          30       93
5         of         19       0
5         integer    17       0
5         ;          33       59
7         function            15      0
7         fun1       1        0x5544cc0
7         (          27       40
7         x          1        0x5544e70
7         :          32       58
7         integer    17       0
7         ;          33       59
7         y          1        0x5545160
7         :          32       58
7         real       21       0
7         ;          33       59
8         z          1        0x55454b0
8         :          32       58
8         array      10       0
8         [          29       91
8         1          6        1
8         ..         26       26
8         2          6        2
8         ]          30       93
8         of         19       0
8         integer    17       0
8         ;          33       59
9         q          1        0x5545c60
9         :          32       58
9         real       21       0
9         )          28       41
9         :          32       58
9         integer    17       0
9         ;          33       59
10        var        23       0
10        d          1        0x55462c0
10        :          32       58
10        integer    17       0
10        ;          33       59
11        begin      11       0
12        a          1        0x55439a0
12        :=         31       0
12        2          6        2
12        ;          33       59
13        z          1        0x55454b0
13        [          29       91
13        a          1        0x55439a0
13        ]          30       93
13        :=         31       0
13        4          6        4
13        ;          33       59
14        c          1        0x5544340
14        [          29       91
14        2          6        2
14        ]          30       93
14        :=         31       0
14        3          6        3
14        ;          33       59
15        fun1       1        0x5544cc0
15        :=         31       0
15        c          1        0x5544340
15        [          29       91
15        1          6        1
15        ]          30       93
16        end        14       0
16        ;          33       59
18        function            15      0
18        fun2       1        0x5547a00
18        (          27       40
18        x          1        0x5544e70
18        :          32       58
18        integer    17       0
18        ;          33       59
18        y          1        0x5545160
18        :          32       58
18        integer    17       0
18      )           28      41
18      :           32      58
18      real        21      0
18      ;           33      59
19      var         23      0
19      e           1       0x55483d0
19      :           32      58
19      real        21      0
19      ;           33      59
21      function            15     0
21      fun3        1       0x55488a0
21      (           27      40
21      n           1       0x5548a50
21      :           32      58
21      integer     17      0
21      ;           33      59
21      z           1       0x55454b0
21      :           32      58
21      real        21      0
21      )           28      41
21      :           32      58
21      integer     17      0
21      ;           33      59
22      var         23      0
22      e           1       0x55483d0
22      :           32      58
22      integer     17      0
22      ;           33      59
23      begin       11      0
24      a           1       0x55439a0
24      :=          31      0
24      e           1       0x55483d0
24      ;           33      59
25      e           1       0x55483d0
25      :=          31      0
25      c           1       0x5544340
25      [           29      91
25      e           1       0x55483d0
25      ]           30      93
25      ;           33      59
26      fun3        1       0x55488a0
26      :=          31      0
26      3           6       3
27      end         14      0
27      ;           33      59
29      begin       11      0
30      a           1       0x55439a0
30      :=          31      0
30      fun1        1       0x5544cc0
30      (           27      40
30      x           1       0x5544e70
30      ,           34      44
30      e           1       0x55483d0
30      ,           34      44
30      c           1       0x5544340
30      ,           34      44
30      b           1       0x5543e70
30      )           28      41
30      ;           33      59
31      x           1       0x5544e70
31      :=          31      0
31      fun3        1       0x55488a0
31      (           27      40
31      c           1       0x5544340
31      [           29      91
31      1           6       1
31      ]           30      93
31      ,           34      44
31      e           1       0x55483d0
31      )           28      41
31      ;           33      59
32      e           1       0x55483d0
32      :=          31      0
32      e           1       0x55483d0
32      +           7       56
32      4.44        5       4
32      ;           33      59
33      a           1       0x55439a0
33      :=          31      0
33      (           27      40
33      a           1       0x55439a0
33      mod         3       53
33      y           1       0x5545160
33      )           28      41
33      div         3       52
33      x           1       0x5544e70
33      ;           33      59
34      while       24      0
34      (           27      40
34      (           27      40
34      a           1       0x55439a0
34      >=          4       46
34      4           6       4
34      )           28      41
34      and         3       54
34      (           27      40
34      (           27      40
34      b           1       0x5543e70
34      <=          4       42
34      e           1       0x55483d0
34      )           28      41
35      or          2       55
35      (           27      40
35      not         18      0
35      (           27      40
35      a           1       0x55439a0
35      =           4       43
35      c           1       0x5544340
35      [           29      91
35      a           1       0x55439a0
35      ]           30      93
35      )           28      41
35      )           28      41
35      )           28      41
35      )           28      41
35      do          12      0
36      begin       11      0
37      a           1       0x55439a0
37      :=          31      0
37      c           1       0x5544340
```

```
37          [           29      91
37          a           1       0x55439a0
37          ]           30      93
37          +           7       56
37          1           6       1
38          end         14      0
38          ;           33      59
39          fun2        1       0x5547a00
39          :=          31      0
39          2.5         5       2
40          end         14      0
40          ;           33      59
42          begin       11      0
43          b           1       0x5543e70
43          :=          31      0
43          fun2        1       0x5547a00
43          (           27      40
43          c           1       0x5544340
43          [           29      91
43          4           6       4
43          ]           30      93
43          ,           34      44
43          c           1       0x5544340
43          [           29      91
43          5           6       5
43          ]           30      93
43          )           28      41
43          ;           33      59
44          b           1       0x5543e70
44          :=          31      0
44          fun2        1       0x5547a00
44          (           27      40
44          c           1       0x5544340
44          [           29      91
44          4           6       4
44          ]           30      93
44          ,           34      44
44          2           6       2
44          )           28      41
44          ;           33      59
45          if          16      0
45          (           27      40
45          a           1       0x55439a0
45          <           4       41
45          2           6       2
45          )           28      41
45          then        22      0
45          a           1       0x55439a0
45          :=          31      0
45          1           6       1
45          else        13      0
45          a           1       0x55439a0
45          :=          31      0
45          a           1       0x55439a0
45          +           7       56
45          2           6       2
45          ;           33      59
46          if          16      0
46          (           27      40
46          b           1       0x5543e70
46          >           4       45
46          4.2         5       4
46          )           28      41
46          then        22      0
46          a           1       0x55439a0
46          :=          31      0
46          c           1       0x5544340
46          [           29      91
46          a           1       0x55439a0
46          ]           30      93
47          end         14      0
47          .           25      46
48          EOF         0       0
```

Appendix II: Sample Inputs and Outputs

Listing 16: common/io.c

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

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

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

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

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

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

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

Listing 17: common/io.h

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

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

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

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

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

#endif
```

Listing 18: common/defs.h

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

#ifndef DEFS_H
#define DEFS_H

#define NOPRINT 1024

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

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


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



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

#define TOKEN_TIMES 50
#define TOKEN_RDIV 51
#define TOKEN_IDIV 52
#define TOKEN_MOD 53
#define TOKEN_AND 54

#define TOKEN_OR 55
#define TOKEN_PLUS 56
#define TOKEN_MINUS 57

#define LEXERR 99

#define TOKEN_WHITESPACE 1024
#define TOKEN_NEWLINE 1025

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



#define ID_STRLEN 10

#ifndef LINELEN
#define LINELEN 72
#endif

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

#endif
```

Listing 19: common/util.c

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

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

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

```c
        return -1;
    }
    return 0;
}

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

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

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

Listing 20: common/util.h

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

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


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

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

#endif
```

Listing 21: common/token.c

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

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

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

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

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


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

const char* relop2str(int attr)
{
    switch (attr) {
    case TOKEN_LT: return "<";
    case TOKEN_LEQ: return "<=";
    case TOKEN_EQ: return "=";
    case TOKEN_NEQ: return "<>";
    case TOKEN_GT: return ">";
    case TOKEN_GEQ: return ">=";
    default: return "Unknown relop";
    }
}
```

Listing 22: common/token.h

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

#ifndef TOKEN_H
#define TOKEN_H

#include <stdio.h>

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

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

int token_id(struct token *t, char *ptr);
int token_add(struct token *t, int type, int attr)
    ;
int token_println(FILE *f, int line, const char *
    lexeme, struct token t);
const char* token2str(int token);
const char* relop2str(int attr);
#endif
```

Listing 23: common/idres.c

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

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

```

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

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

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

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

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

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

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

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

Listing 24: common/idres.h

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

#ifndef IDRES_H
#define IDRES_H

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

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

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

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

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

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

Listing 25: lexer/main.c

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

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

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

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

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

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

        handle_line(&s, line);

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

Listing 26: lexer/state.c

```
/* -*- C -*-
 *
```

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

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

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

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

Listing 27: lexer/state.h

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

#ifndef STATE_H
#define STATE_H

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

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

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

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

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

```
#endif
```

Listing 28: lexer/fsm.c

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

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

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

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

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

    len = strlen(lexeme);
    if (mantis_len > 5) {
        token_add(&m->tok, LEXERR,
            LEXERR_MANTIS_TOO_LONG);
    } else if (frac_len > 5) {
        token_add(&m->tok, LEXERR,
            LEXERR_FRAC_TOO_LONG);
```

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

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

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

int fsm_catchall(struct machine *m, struct lex_state *ls)
{
    switch (*m->f) {
    case '[':
        token_add(&m->tok, TOKEN_LBRACKET, *m->f);
        return 1;
    case ']':
        token_add(&m->tok, TOKEN_RBRACKET, *m->f);
        return 1;
    case '(':
        token_add(&m->tok, TOKEN_LPAREN, *m->f);
        return 1;
    case ')':
        token_add(&m->tok, TOKEN_RPAREN, *m->f);
        return 1;
    case ',':
        token_add(&m->tok, TOKEN_COMMA, *m->f);
        return 1;
    case ';':
        token_add(&m->tok, TOKEN_SEMICOLON, *m->f);
        return 1;
    default:
        break;
    }
    if (*m->f == '.') {
        m->f++;
        if (*m->f == '.') {
            token_add(&m->tok, TOKEN_ELLIPSIS, TOKEN_ELLIPSIS);
        } else {
            m->f--;
            token_add(&m->tok, TOKEN_PERIOD, '.');
        }
        return 1;
    } else if (*m->f == ':') {
        m->f++;
        if (*m->f == '=') {
            token_add(&m->tok, TOKEN_ASSIGN, 0);
        } else {
            m->f--;
            token_add(&m->tok, TOKEN_COLON, ':');
        }
        return 1;
    }
    return 0;
}

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


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

}
```

Listing 29: lexer/fsm.h

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

#ifndef FSM_H
#define FSM_H

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

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

Listing 30: lexer/lexerr.c

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

#include "lexerr.h"

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

Listing 31: lexer/lexerr.h

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

#ifndef LEXERR_H
#define LEXERR_H

#include "machine.h"

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

#endif
```

Listing 32: lexer/machine.c

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

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

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

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

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

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

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

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

Listing 33: lexer/machine.h

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

#ifndef MACHINE_H
#define MACHINE_H

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

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

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

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

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

int machine_clean(struct machine **list);

#endif
```

Listing 34: parser/main.c

```
#include <stdio.h>
#include "parser.h"
#include "tokenizer.h"

int main(int argc, char **argv)
{
    struct parser parser;
    if (argc != 6) {
        fprintf(stderr, "Usage: %s symbols tokens
            listing_in address_out listing_out\n",
            *argv);
        return -1;
    }
    parser_init(&parser, argv[1], argv[2], argv
        [3], argv[4], argv[5]);
/* idres_print(stdout, &parser.symbols);*/
    parse(&parser);
    parser_cleanup(&parser);
    return 0;
}
```

Listing 35: parser/tokenizer.h

```
#ifndef TOKENIZER_H
#define TOKENIZER_H

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

int next_token(struct parser *p);

int parser_listing(struct parser *p, unsigned
    lineno);
#endif
```

Listing 36: parser/tokenizer.c

```
#include "tokenizer.h"
#include <string.h>
#include <stdlib.h>

int next_token(struct parser *p)
{
    unsigned lineno;
    int attr;
    if (fscanf(p->f_token, "%u", &lineno) == EOF)
        {
        fprintf(stderr, "warning: EOF token may
            not be included in token file\n");
        return -1;
    }
    fscanf(p->f_token, "%s", p->buffer);
    fscanf(p->f_token, "%d", &p->token.type);
    if (p->token.type == TOKEN_ID) {
        struct idres *tok = NULL;
        if (idres_search(&p->symbols, p->buffer, &
            tok) == -1) {
            fprintf(stderr, "idres can't find
                symbol %s\n", p->buffer);
            return -1;
        }
        p->token = tok->token;
        fscanf(p->f_token, "%s\n", p->buffer);
    } else {
        fscanf(p->f_token, "%d\n", &attr);
        p->token.is_id = 0;
        p->token.val.attr = attr;
        if (!strcmp(token2str(p->token.type), "
            UNKNOWN")) {
            fprintf(p->syn_list, "SYNERR: Unknown
                token encountered: %d\n", attr);
        }
    }
    parser_listing(p, lineno);

    return p->token.type;
}

int g_temp = 0;
int parser_listing(struct parser *p, unsigned
    new_tok_lineno)
{
    int c;
        unsigned cur_tok_lineno = p->tok_line;
    if (cur_tok_lineno == new_tok_lineno) {
        return 0;
    } else if (g_temp == 0) {
        g_temp = 1;
        /* return 0; */
    } else {
        g_temp = 0;
    }
    p->tok_line = new_tok_lineno;
    while (p->list_line < p->tok_line) {
        int fret = fscanf(p->lex_list, "%s", p->
            buffer);
        if (fret == -1) { /* EOF reached */
            p->list_line = p->tok_line;
            break;
        }
        if (strcmp(p->buffer, "LEXERR:")) {
            char *ptr = NULL;
            p->list_line = strtoul(p->buffer, &ptr,
                10);
            if (*ptr) {
                fprintf(p->syn_list, "Listing file
                    invalid format, fret %d\n",
                    fret);
                exit(EXIT_FAILURE);
            }
        }
        fprintf(p->syn_list, "%s", p->buffer);
        while ((c = fgetc(p->lex_list)) != '\n') {
            if (c == EOF) {
                p->list_line = p->tok_line;
                break;
            }
            fputc(c, p->syn_list);
        }
        fputc('\n', p->syn_list);
    }
    return 0;
}
```

Listing 37: parser/parser.h

```
#ifndef PARSER_H
#define PARSER_H
#include <stdio.h>
#include "idres.h"
#include "defs.h"
#include "gb.h"
#include "types.h"

struct parser {
    struct idres *symbols;
    FILE* f_token;
    FILE* lex_list;
    FILE* syn_list;
    FILE* addr;
    char *buffer;
    unsigned tok_line, list_line;
    struct gb_state gbs;
    struct token token;
};

int parser_init(struct parser *p, const char*
    symbol_file, const char* token_file, const
    char* list_in, const char* addr_out, const
    char* list_out);

int parse(struct parser *p);

int match(struct parser *p, int tok, int *ret);

int sync(struct parser *p, int* sync_set, int
    sync_size);
void expected_found(struct parser *p, int *
    expected, int size);
int parser_cleanup(struct parser *p);

char* parser_get_tok_lex(struct parser *p,
            struct token t_id);

#endif
```

Listing 38: parser/parser.c

```
#include "defs.h"
#include "parser.h"
#include "tokenizer.h"
#include "./prod/program.h"

int parse(struct parser *p)
{
    next_token(p);
    program(p);
    return match(p, TOKEN_EOF, NULL);
}

int match(struct parser *p, int tok, int *ret)
{
    if (p->token.type == tok) {
        if (tok == TOKEN_EOF) {
            return 0;
        } else {
            printf("Matched token %s\n", token2str(
                tok));
            next_token(p);
            return 0;
        }
    } else {
        expected_found(p, &tok, 1);
        next_token(p);
        if (ret) {
            *ret = -1;
        }
        return -1;
    }
}
int parser_init(struct parser *p, const char*
    symbol_file, const char* token_file, const
    char* list_in, const char* addr_out, const
    char* list_out)
{
    p->symbols = NULL;
    if (idres_read(symbol_file, &p->symbols) ==
        -1) {
        return -1;
    }
    p->f_token = fopen(token_file, "r");
    if (p->f_token == NULL) {
        fprintf(stderr, "Could not open file \"%s
            \"\n", token_file);
        return -1;
    }
    p->addr = fopen(addr_out, "w");
    if (p->addr == NULL) {
        fprintf(stderr, "Could not open file \"%s
            \"\n", addr_out);
        return -1;
    }
    p->buffer = malloc(LINELEN+1);
    if (p->buffer == NULL) {
        fprintf(stderr, "Could allocate memory\n")
            ;
        return -1;
    }
    p->lex_list = fopen(list_in, "r");
    if (p->lex_list == NULL) {
        fprintf(stderr, "Could not open file \"%s
            \"\n", list_in);
        return -1;
    }
    p->syn_list = fopen(list_out, "w");
    if (p->syn_list == NULL) {
        fprintf(stderr, "Could not open file \"%s
            \"\n", list_out);
        return -1;
    }
    p->tok_line = 0;
    p->list_line = 0;
    return gb_state_init(&p->gbs);
}

void expected_found(struct parser *p, int*
    expected, int size)
{
    int i;
    fprintf(p->syn_list, "SYNERR: Expected ");
    if (size > 1) {
        fprintf(p->syn_list, "one of ");
    }
    for (i = 0; i < size - 1; i++) {
        fprintf(p->syn_list, "\"%s\"", token2str(
            expected[i]));
        if (size > 2) {
            fputc(',', p->syn_list);
        }
        fputc(' ', p->syn_list);
    }
    if (size > 1) {
        fprintf(p->syn_list, "or ");
    }
    fprintf(p->syn_list, "\"%s\" ", token2str(
        expected[size-1]));
    if (p->token.is_id) {
        struct idres *ret = NULL;
        idres_lookup(&p->symbols, p->token.val.ptr
            , &ret);
        if (ret == NULL) {
            fprintf(stderr, "symbol not found\n");
            exit(1);
        }
        fprintf(p->syn_list, "but found identifier
             \"%s\"\n", ret->lexeme);
    } else {
        fprintf(p->syn_list, "but found \"%s\"\n",
             token2str(p->token.type));
    }
}

int sync(struct parser *p, int* sync_set, int
    sync_size)
{
    int i, found = 0;
    while (!found) {
        for (i = 0; i < sync_size; i++) {
            if (p->token.type == sync_set[i]) {
                return 0;
            }
        }
        if (p->token.type == TOKEN_EOF) {
            return 0;
        }
        next_token(p);
    }
    return 0;
}

int parser_cleanup(struct parser *p)
{
    int c;
    while ((c = fgetc(p->lex_list)) != EOF) {
        fputc(c, p->syn_list);
    }
    free(p->buffer);
    /* gb_print_all(p->gbs.cur_eye, p->addr, 4);
        */
    gb_state_free(&p->gbs);
    fclose(p->addr);
    fclose(p->f_token);
    fclose(p->lex_list);
    fclose(p->syn_list);
    idres_clean(&p->symbols);

    return 0;
}


char* parser_get_tok_lex(struct parser *p,
            struct token t_id)
{
    struct idres *ret = NULL;
    if (!t_id.is_id
        || idres_lookup(&p->symbols, t_id.val.ptr,
             &ret) == -1
        || ret == NULL) {
        fprintf(stderr, "INTERNAL ERROR: symbol
            table"
            " does not match token: \"%s\"\n",
                token2str(t_id.type));
        exit(EXIT_FAILURE);
    }
    return ret->lexeme;
}
```

Listing 39: parser/prod/arguments.h

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

#ifndef ARGUMENTS_H
#define ARGUMENTS_H

#include "../parser.h"

int arguments(struct parser *p, char **args);

#endif
```

Listing 40: parser/prod/arguments.c

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

#include "arguments.h"
#include "parameter_list.h"

int arguments(struct parser *p, char **args)
{
    int sync_set[] = {TOKEN_COLON};
    int expected[] = {TOKEN_COLON, TOKEN_LPAREN};
    const int sync_size = sizeof(sync_set)/sizeof
        (*sync_set);
    const int expected_size = sizeof(expected)/
        sizeof(*expected);
    int ret = TYPE_VOID;
    switch (p->token.type) {
    case TOKEN_LPAREN:
        match(p, TOKEN_LPAREN, &ret);
        err_propogate(parameter_list(p, args), &
            ret);
        match(p, TOKEN_RPAREN, &ret);
        break;
    case TOKEN_COLON:
        *args = calloc(1, 1);
        break;
    default:
        ret = TYPE_ERR;
        expected_found(p, expected, expected_size)
            ;
        sync(p, sync_set, sync_size);
    }
    return ret;
}
```

Listing 41: parser/prod/compound_statement.h

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

#ifndef COMPOUND_STATEMENT_H
#define COMPOUND_STATEMENT_H

#include "../parser.h"

int compound_statement(struct parser *p);

#endif
```

Listing 42: parser/prod/compound_statement.c

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

#include "compound_statement.h"
#include "optional_statements.h"

int compound_statement(struct parser *p)
{
    int sync_set[] = {TOKEN_PERIOD,
        TOKEN_SEMICOLON, TOKEN_END, TOKEN_ELSE};
    int expected[] = {TOKEN_BEGIN};
    const int sync_size = sizeof(sync_set)/sizeof
        (*sync_set);
    const int expected_size = sizeof(expected)/
        sizeof(*expected);
    int ret = TYPE_VOID;
    switch (p->token.type) {
    case TOKEN_BEGIN:
        match(p, TOKEN_BEGIN, &ret);
        err_propogate(optional_statements(p), &ret
            );
        match(p, TOKEN_END, &ret);
        break;
    default:
        ret = TYPE_ERR;
        expected_found(p, expected, expected_size)
            ;
        sync(p, sync_set, sync_size);
    }
    return ret;
}
```

Listing 43: parser/prod/declarations.h

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

#ifndef DECLARATIONS_H
#define DECLARATIONS_H

```

```c
#include "../parser.h"

int declarations(struct parser *p);

#endif
```

Listing 44: parser/prod/declarations.c

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

#include "declarations.h"
#include "type.h"
int declarations(struct parser *p)
{
    int sync_set[] = {TOKEN_BEGIN, TOKEN_FUNCTION
        };
    int expected[] = {TOKEN_BEGIN, TOKEN_FUNCTION,
         TOKEN_VAR};
    const int sync_size = sizeof(sync_set)/sizeof
        (*sync_set);
    const int expected_size = sizeof(expected)/
        sizeof(*expected);
    struct token t_id;
    int ret = TYPE_VOID;
    char *lexeme = NULL;
    switch (p->token.type) {
    case TOKEN_VAR:
        match(p, TOKEN_VAR, &ret);
            t_id = p->token;
        match(p, TOKEN_ID, &ret);
        match(p, TOKEN_COLON, &ret);
        {
            int offset, var_type, width;
            lexeme = parser_get_tok_lex(p, t_id);
            var_type = type(p, &width);
            err_propogate(var_type, &ret);
            get_offset(&p->gbs, &offset);
            int bret = check_add_blue(&p->gbs,
                lexeme, var_type, offset);
            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;
                } else {
                    fprintf(p->addr, "%s\t%d\t%s\n",
                         lexeme, offset, strtype(
                        var_type));
                }
            }
            set_offset(&p->gbs, offset + width);
        }
        match(p, TOKEN_SEMICOLON, &ret);
        err_propogate(declarations(p), &ret);
        break;
    case TOKEN_BEGIN:
    case TOKEN_FUNCTION:
        break;
    default:
        ret = TYPE_ERR;
        expected_found(p, expected, expected_size)
            ;
        sync(p, sync_set, sync_size);
    }
    return ret;
}
```

Listing 45: parser/prod/expression.h

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

#ifndef EXPRESSION_H
#define EXPRESSION_H

#include "../parser.h"

int expression(struct parser *p);

#endif
```

Listing 46: parser/prod/expression.c

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

#include "expression.h"
#include "expression_prime.h"
#include "simple_expression.h"

int expression(struct parser *p)
{
    int sync_set[] = {TOKEN_DO, TOKEN_ELSE,
        TOKEN_END,
              TOKEN_THEN, TOKEN_RPAREN,
              TOKEN_RBRACKET, TOKEN_COMMA,
              TOKEN_SEMICOLON};
    int expected[] = {TOKEN_ID, TOKEN_NUM_INTEGER,
              TOKEN_NUM_REAL, TOKEN_LPAREN,
              TOKEN_NOT, TOKEN_SIGN};
    const int sync_size = sizeof(sync_set)/sizeof
        (*sync_set);
    const int expected_size = sizeof(expected)/
        sizeof(*expected);
    int ret = TYPE_VOID;

    switch (p->token.type) {
    case TOKEN_ID:
    case TOKEN_NUM_INTEGER:
    case TOKEN_NUM_REAL:
    case TOKEN_LPAREN:
    case TOKEN_NOT:
    case TOKEN_SIGN:
        ret = simple_expression(p);
        ret = expression_prime(p, ret);
        break;
    default:
        ret = TYPE_ERR;
        expected_found(p, expected, expected_size)
            ;
        sync(p, sync_set, sync_size);
    }
    return ret;
}
```

Listing 47: parser/prod/expression_prime.h

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

#ifndef EXPRESSION_PRIME_H
#define EXPRESSION_PRIME_H

#include "../parser.h"

int expression_prime(struct parser *p, int type);

#endif
```

Listing 48: parser/prod/expression_prime.c

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

#include "expression_prime.h"
#include "simple_expression.h"

int expression_prime(struct parser *p, int type)
{
    int sync_set[] = {TOKEN_DO, TOKEN_ELSE,
        TOKEN_END,
              TOKEN_THEN, TOKEN_RPAREN,
                  TOKEN_RBRACKET,
              TOKEN_COMMA, TOKEN_SEMICOLON};
    int expected[] = {TOKEN_RELOP, TOKEN_DO,
        TOKEN_ELSE,
              TOKEN_END, TOKEN_THEN, TOKEN_RPAREN,
              TOKEN_RBRACKET, TOKEN_COMMA,
                  TOKEN_SEMICOLON};
    const int sync_size = sizeof(sync_set)/sizeof
        (*sync_set);
    const int expected_size = sizeof(expected)/
        sizeof(*expected);
    int ret = type;
    switch (p->token.type) {
    case TOKEN_RELOP: {
        int type2;
        struct token t_relop = p->token;
        match(p, TOKEN_RELOP, &ret);
        type2 = simple_expression(p);
        if (type != TYPE_ERR && type2 != TYPE_ERR)
             {
            if (type != type2) {
                fprintf(p->syn_list, "SEMERR:
                    Invalid operands to %s: %s and
                    %s\n",
                    relop2str(t_relop.val.attr),
                    strtype(type),
                    strtype(type2));
                ret = TYPE_ERR;
            } else {
                ret = TYPE_BOOL;
            }
        } else {
            ret = TYPE_ERR;
        }
        break;
    }
    case TOKEN_DO:
    case TOKEN_ELSE:
    case TOKEN_END:
    case TOKEN_THEN:
    case TOKEN_RPAREN:
    case TOKEN_RBRACKET:
    case TOKEN_COMMA:
    case TOKEN_SEMICOLON:
        ret = type;
        break;
    default:
        ret = TYPE_ERR;
        expected_found(p, expected, expected_size)
            ;
        sync(p, sync_set, sync_size);
    }
    return ret;
}
```

Listing 49: parser/prod/expression_list.h

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

#ifndef EXPRESSION_LIST_H
#define EXPRESSION_LIST_H

```

```
#include "../parser.h"

int expression_list(struct parser *p, char **args)
    ;

#endif
```

Listing 50: parser/prod/expression_list.c

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

#include "expression_list.h"
#include "expression_list_prime.h"
#include "expression.h"
int expression_list(struct parser *p, char **args)
{
    int sync_set[] = {TOKEN_RPAREN};
    int expected[] = {TOKEN_ID, TOKEN_NUM_INTEGER,
              TOKEN_NUM_REAL, TOKEN_LPAREN,
              TOKEN_NOT, TOKEN_SIGN};
    const int sync_size = sizeof(sync_set)/sizeof
        (*sync_set);
    const int expected_size = sizeof(expected)/
        sizeof(*expected);
    int ret = TYPE_VOID;
    switch (p->token.type) {
    case TOKEN_ID:
    case TOKEN_NUM_INTEGER:
    case TOKEN_NUM_REAL:
    case TOKEN_LPAREN:
    case TOKEN_NOT:
    case TOKEN_SIGN: {
        int type = expression(p);
        char *arg = malloc(2);
        sprintf(arg, "%c", type);
        *args = arg;
        ret = expression_list_prime(p, args);
        break;
    }
    default:
        ret = TYPE_ERR;
        expected_found(p, expected, expected_size)
            ;
        sync(p, sync_set, sync_size);
    }
    return ret;
}
```

Listing 51: parser/prod/expression_list_prime.h

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

#ifndef EXPRESSION_LIST_PRIME_H
#define EXPRESSION_LIST_PRIME_H

#include "../parser.h"

int expression_list_prime(struct parser *p, char
    **args);

#endif
```

Listing 52: parser/prod/expression_list_prime.c

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

#include "expression_list_prime.h"
#include "expression.h"
#include <string.h>
int expression_list_prime(struct parser *p, char
    **args)
{
    int sync_set[] = {TOKEN_RPAREN};
    int expected[] = {TOKEN_COMMA};
    const int sync_size = sizeof(sync_set)/sizeof
        (*sync_set);
    const int expected_size = sizeof(expected)/
        sizeof(*expected);
    int ret = TYPE_VOID;
    switch (p->token.type) {
    case TOKEN_COMMA:
        match(p, TOKEN_COMMA, &ret);
        int type = expression(p);
        char *arg = malloc(strlen(*args) + 2);
        sprintf(arg, "%s%c", *args, type);
        free(*args);
        *args = arg;
        ret = expression_list_prime(p, args);
        break;
    case TOKEN_RPAREN:
        break;
    default:
        ret = TYPE_ERR;
        expected_found(p, expected, expected_size)
            ;
        sync(p, sync_set, sync_size);
    }
    return ret;
}
```

Listing 53: parser/prod/factor.h

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

```c
 */

#ifndef FACTOR_H
#define FACTOR_H

#include "../parser.h"

int factor(struct parser *p);

#endif
```

Listing 54: parser/prod/factor.c

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

#include "factor.h"
#include "expression.h"
#include "factor_prime.h"

int factor(struct parser *p)
{
    int sync_set[] = {TOKEN_MULOP, TOKEN_ADDOP,
        TOKEN_SIGN, TOKEN_RELOP,
              TOKEN_DO, TOKEN_ELSE, TOKEN_END,
              TOKEN_THEN, TOKEN_RPAREN,
              TOKEN_RBRACKET, TOKEN_COMMA,
              TOKEN_SEMICOLON};
    int expected[] = {TOKEN_ID, TOKEN_NUM_INTEGER,
              TOKEN_NUM_REAL, TOKEN_LPAREN,
              TOKEN_NOT};
    const int sync_size = sizeof(sync_set)/sizeof
        (*sync_set);
    const int expected_size = sizeof(expected)/
        sizeof(*expected);
    int ret = TYPE_VOID;
    switch (p->token.type) {
    case TOKEN_ID: {
        struct token t_id = p->token;
        char *lexeme = parser_get_tok_lex(p, t_id)
            ;
        int type;
        match(p, TOKEN_ID, &ret);
        type = factor_prime(p, lexeme);
        if (ret != TYPE_ERR) {
            ret = type;
        }
        break;
    }
    case TOKEN_NUM_INTEGER:
        match(p, TOKEN_NUM_INTEGER, &ret);
        ret = TYPE_INT;
        break;
    case TOKEN_NUM_REAL:
        match(p, TOKEN_NUM_REAL, &ret);
        ret = TYPE_REAL;
        break;
    case TOKEN_LPAREN: {
        int type;
        match(p, TOKEN_LPAREN, &ret);
        type = expression(p);
        match(p, TOKEN_RPAREN, &ret);
        if (ret != TYPE_ERR) {
            ret = type;
        }
        break;
    }
    case TOKEN_NOT: {
        int type;
        match(p, TOKEN_NOT, &ret);
        type = factor(p);
        if (type != TYPE_ERR && type != TYPE_BOOL)
            {
            fprintf(p->syn_list, "SEMERR: Invalid
                operand to \"not\": %s\n",
                strtype(type));
            type = TYPE_ERR;
        }
        if (ret != TYPE_ERR) {
            ret = type;
        }
        break;
    }
    default:
        ret = TYPE_ERR;
        expected_found(p, expected, expected_size)
            ;
        sync(p, sync_set, sync_size);
    }
    return ret;
}
```

Listing 55: parser/prod/factor_prime.h

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

#ifndef FACTOR_PRIME_H
#define FACTOR_PRIME_H

#include "../parser.h"

int factor_prime(struct parser *p, char *id_lex);

#endif
```

Listing 56: parser/prod/factor_prime.c

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

```c

#include "factor_prime.h"
#include "expression.h"
#include "expression_list.h"
#include <string.h>
int fp_array(struct parser *p, char *id_lex);
int fp_function(struct parser *p, char *id_lex);
int fp_id(struct parser *p, char *id_lex);

int factor_prime(struct parser *p, char *id_lex)
{
   int sync_set[] = {TOKEN_MULOP, TOKEN_ADDOP,
      TOKEN_SIGN, TOKEN_RELOP,
          TOKEN_DO, TOKEN_ELSE, TOKEN_END,
          TOKEN_THEN, TOKEN_RPAREN,
          TOKEN_RBRACKET, TOKEN_COMMA,
          TOKEN_SEMICOLON};
   int expected[] = {TOKEN_LBRACKET, TOKEN_LPAREN
      , TOKEN_MULOP,
          TOKEN_ADDOP, TOKEN_SIGN, TOKEN_RELOP,
          TOKEN_DO, TOKEN_ELSE, TOKEN_END,
          TOKEN_THEN, TOKEN_RPAREN,
          TOKEN_RBRACKET, TOKEN_COMMA,
          TOKEN_SEMICOLON};
   const int sync_size = sizeof(sync_set)/sizeof
      (*sync_set);
   const int expected_size = sizeof(expected)/
      sizeof(*expected);
   int ret = TYPE_VOID;
   /*
         if (lexeme == NULL && < 0) {
         fprintf(p->syn_list, "SEMERR: Symbol
            \"%s\" was not declared in this
            scope\n", lexeme);
         ret = TYPE_ERR;
      }

    */
   switch (p->token.type) {
   case TOKEN_LBRACKET: {
      ret = fp_array(p, id_lex);
      break;
   }
   case TOKEN_LPAREN: { /* turn FPINT -> INT */
      ret = fp_function(p, id_lex);

      break;
   }
   case TOKEN_MULOP:
   case TOKEN_SIGN: /* used as addition/
      subtraction here */
   case TOKEN_ADDOP:
   case TOKEN_RELOP:
   case TOKEN_DO:
   case TOKEN_ELSE:
   case TOKEN_END:
   case TOKEN_THEN:
   case TOKEN_RPAREN:
   case TOKEN_RBRACKET:
   case TOKEN_COMMA:
   case TOKEN_SEMICOLON: /* ID */ {
      ret = fp_id(p, id_lex);
      break;
   }
   default:
      ret = TYPE_ERR;
      expected_found(p, expected, expected_size)
         ;
      sync(p, sync_set, sync_size);
   }
   return ret;
}

int fp_array(struct parser *p, char *id_lex)
{
   int ret = TYPE_VOID;
   int expr_type, atype, scalar_type, type;
   if (gettype(&p->gbs, id_lex, &atype) < 0) {
      fprintf(p->syn_list, "SEMERR: Variable \"%
         s\" was not declared in this scope\n",
         id_lex);
      ret = TYPE_ERR;
   }
   scalar_type = ARRAY_TO_SCALAR(atype);
   if (scalar_type == TYPE_ERR) {
      fprintf(p->syn_list, "SEMERR: Attempted
         indexing of non-array type %s\n",
         strtype(atype));
      ret = TYPE_ERR;
   } else {
      type = 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 = type;
   }
   return ret;
}

int fp_function(struct parser *p, char *id_lex)
{
   char *args = NULL;
   int ret = TYPE_VOID;
   match(p, TOKEN_LPAREN, &ret);
   err_propogate(expression_list(p, &args), &ret)
      ;
   match(p, TOKEN_RPAREN, &ret);
   struct gb *gbret = NULL;
   find_green(p->gbs.cur_eye, id_lex, &gbret);
   if (gbret == NULL) {
      fprintf(p->syn_list, "SEMERR: Function \"%
            s\" not declared in this scope\n",
            id_lex);
        if (args) {
            free(args);
        }
        return ret;
    }
    char *fargs = gbret->n.g.arglist;
    ret = FUNC_TO_SCALAR(gbret->n.g.type);
    if (strcmp(args, fargs)) {
        char *exp = param2str(fargs);
        char *got = param2str(args);
        fprintf(p->syn_list, "SEMERR: Function
            args for \"%s\" do not match: expected
            (%s) but got (%s)\n",
            id_lex,
            exp,
            got);
        ret = TYPE_ERR;
        free(exp);
        free(got);
    }
    if (args) {
        free(args);
    }
    return ret;
}

int fp_id(struct parser *p, char *id_lex)
{
    int type = TYPE_ERR;
    if (id_lex != NULL &&
        gettype(&p->gbs, id_lex, &type) < 0) {
        fprintf(p->syn_list, "SEMERR: Variable \"%
            s\" was not declared in this scope\n",
            id_lex);
        type = TYPE_ERR;
    }
    return type;
}
```

Listing 57: parser/prod/identifier_list.h

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

#ifndef IDENTIFIER_LIST_H
#define IDENTIFIER_LIST_H

#include "../parser.h"

int identifier_list(struct parser *p);

#endif
```

Listing 58: parser/prod/identifier_list.c

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

#include "identifier_list.h"
#include "identifier_list_prime.h"
int identifier_list(struct parser *p)
{
    int sync_set[] = {TOKEN_RPAREN};
    int expected[] = {TOKEN_ID};
    const int sync_size = sizeof(sync_set)/sizeof
        (*sync_set);
    const int expected_size = sizeof(expected)/
        sizeof(*expected);
    struct token t_id;
    int offset, ret = TYPE_VOID;
    char *lexeme;
    switch (p->token.type) {
    case TOKEN_ID:
        t_id = p->token;
        match(p, TOKEN_ID, &ret);
        {
            lexeme = parser_get_tok_lex(p, t_id);
            get_offset(&p->gbs, &offset);
            int bret = check_add_blue(&p->gbs,
                lexeme, TYPE_PGMPARAM, offset);
            if (bret == -1) {
                fprintf(p->syn_list,
                    "SEMERR: Identifier \"%s\"
                        already declared\n",
                    lexeme);
                ret = TYPE_ERR;
            } else if (bret < 0) {
                /* err already reported */
                ret = TYPE_ERR;
            }
            offset += type_width(TYPE_PGMPARAM);
            set_offset(&p->gbs, offset);
        }
        err_propogate(identifier_list_prime(p), &
            ret);
        break;
    default:
        ret = TYPE_ERR;
        expected_found(p, expected, expected_size)
            ;
        sync(p, sync_set, sync_size);
    }
    return ret;
}
```

Listing 59: parser/prod/identifier_list_prime.h

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

```c
 */

#ifndef IDENTIFIER_LIST_PRIME_H
#define IDENTIFIER_LIST_PRIME_H

#include "../parser.h"

int identifier_list_prime(struct parser *p);

#endif
```

Listing 60: parser/prod/identifier_list_prime.c

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

#include "identifier_list_prime.h"

int identifier_list_prime(struct parser *p)
{
    int sync_set[] = {TOKEN_RPAREN};
    int expected[] = {TOKEN_COMMA, TOKEN_RPAREN};
    const int sync_size = sizeof(sync_set)/sizeof
        (*sync_set);
    const int expected_size = sizeof(expected)/
        sizeof(*expected);
    struct token t_id;
    int offset, ret = TYPE_VOID;
    char *lexeme;
    switch (p->token.type) {
    case TOKEN_COMMA:
        match(p, TOKEN_COMMA, &ret);
        t_id = p->token;
        match(p, TOKEN_ID, &ret);
        {
            lexeme = parser_get_tok_lex(p, t_id);
            get_offset(&p->gbs, &offset);
            int bret = check_add_blue(&p->gbs,
                lexeme, TYPE_PGMPARAM, offset);
            if (bret == -1) {
                fprintf(p->syn_list,
                    "SEMERR: Identifier \"%s\"
                        already declared\n",
                    lexeme);
                ret = TYPE_ERR;
            } else if (bret < 0) {
                /* err already reported */
                ret = TYPE_ERR;
            }
            offset += type_width(TYPE_PGMPARAM);
            set_offset(&p->gbs, offset);
        }
        err_propogate(identifier_list_prime(p), &
            ret);
        break;
    case TOKEN_RPAREN:
        break;
    default:
        ret = TYPE_ERR;
        expected_found(p, expected, expected_size)
            ;
        sync(p, sync_set, sync_size);
    }
    return ret;
}
```

Listing 61: parser/prod/optional_statements.h

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

#ifndef OPTIONAL_STATEMENTS_H
#define OPTIONAL_STATEMENTS_H

#include "../parser.h"

int optional_statements(struct parser *p);

#endif
```

Listing 62: parser/prod/optional_statements.c

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

#include "optional_statements.h"
#include "statement_list.h"

int optional_statements(struct parser *p)
{
    int sync_set[] = {TOKEN_END};
    int expected[] = {TOKEN_BEGIN, TOKEN_ID,
        TOKEN_IF, TOKEN_WHILE};
    const int sync_size = sizeof(sync_set)/sizeof
        (*sync_set);
    const int expected_size = sizeof(expected)/
        sizeof(*expected);
    int ret = TYPE_VOID;
    switch (p->token.type) {
    case TOKEN_BEGIN:
    case TOKEN_ID:
    case TOKEN_IF:
    case TOKEN_WHILE:
        err_propogate(statement_list(p), &ret);
        break;
    case TOKEN_END:
        break;
    default:
        ret = TYPE_ERR;
        expected_found(p, expected, expected_size)
            ;
        sync(p, sync_set, sync_size);
    }
    return ret;
}
```

Listing 63: parser/prod/parameter_list.h

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

#ifndef PARAMETER_LIST_H
#define PARAMETER_LIST_H

#include "../parser.h"

int parameter_list(struct parser *p, char **args);

#endif
```

Listing 64: parser/prod/parameter_list.c

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

#include <string.h>
#include "parameter_list.h"
#include "type.h"
#include "parameter_list_prime.h"

#define INTBUF 10

int parameter_list(struct parser *p, char **args)
{
    int sync_set[] = {TOKEN_RPAREN};
    int expected[] = {TOKEN_ID};
    const int sync_size = sizeof(sync_set)/sizeof
        (*sync_set);
    const int expected_size = sizeof(expected)/
        sizeof(*expected);
    struct token t_id;
    int ret = TYPE_VOID;
    int width, var_type, offset;
    char *lexeme = NULL;
    switch (p->token.type) {
    case TOKEN_ID:
        t_id = p->token;
        match(p, TOKEN_ID, &ret);
        match(p, TOKEN_COLON, &ret);
        lexeme = parser_get_tok_lex(p, t_id);
        var_type = type(p, &width);
        err_propogate(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_propogate(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

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

#ifndef PARAMETER_LIST_PRIME_H
#define PARAMETER_LIST_PRIME_H

#include "../parser.h"

int parameter_list_prime(struct parser *p, char **
    args);

#endif
```

Listing 66: parser/prod/parameter_list_prime.c

```
/* -*- C -*-
```

```
 *
 * parameter_list_prime.c
 *
 * Author: Benjamin T James
 */

#include <string.h>
#include "parameter_list_prime.h"
#include "type.h"

#define INTBUF 10

int parameter_list_prime(struct parser *p, char **
    args)
{
    int sync_set[] = {TOKEN_RPAREN};
    int expected[] = {TOKEN_SEMICOLON,
        TOKEN_RPAREN};
    const int sync_size = sizeof(sync_set)/sizeof
        (*sync_set);
    const int expected_size = sizeof(expected)/
        sizeof(*expected);
    struct token t_id;
    int ret = TYPE_VOID;
    int width, var_type, offset;
    char *lexeme = NULL;
    switch (p->token.type) {
    case TOKEN_SEMICOLON:
        match(p, TOKEN_SEMICOLON, &ret);
        t_id = p->token;
        match(p, TOKEN_ID, &ret);
        match(p, TOKEN_COLON, &ret);
        lexeme = parser_get_tok_lex(p, t_id);
        var_type = type(p, &width);
        err_propogate(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_propogate(parameter_list_prime(p, args
            ), &ret);
        int len = INTBUF + strlen(*args) + 1;
        char *next_str = malloc(len);
        *next_str = (char)var_type;
        strcpy(next_str+1, *args);
        free(*args);
        *args = next_str;
        break;
    case TOKEN_RPAREN:
        *args = calloc(1,1);
        break;
    default:
        ret = TYPE_ERR;
        expected_found(p, expected, expected_size)
            ;
        sync(p, sync_set, sync_size);
    }
    return ret;
}
```

Listing 67: parser/prod/program.h

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

#ifndef PROGRAM_H
#define PROGRAM_H

#include "../parser.h"

int program(struct parser *p);

#endif
```

Listing 68: parser/prod/program.c

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

#include "program.h"
#include "identifier_list.h"
#include "declarations.h"
#include "subprogram_declarations.h"
#include "compound_statement.h"

int program(struct parser *p)
{
    int expected = TOKEN_PROGRAM;
    struct token t_id;
    char *lexeme = NULL;
    int ret = TYPE_VOID;
    switch (p->token.type) {
    case TOKEN_PROGRAM:
        match(p, TOKEN_PROGRAM, &ret);
        t_id = p->token;
        match(p, TOKEN_ID, &ret);
        lexeme = parser_get_tok_lex(p, t_id);
        if (lexeme != NULL) {
            if (check_add_green(&p->gbs, lexeme,
                TYPE_PGMNAME) == -1) {
```

```c
                fprintf(p->syn_list, "SEMERR: Scope
                    \"%s\" already exists\n",
                    lexeme);
                ret = TYPE_ERR;
            }
        }

        match(p, TOKEN_LPAREN, &ret);
        err_propogate(identifier_list(p), &ret);
        match(p, TOKEN_RPAREN, &ret);
        match(p, TOKEN_SEMICOLON, &ret);
        err_propogate(declarations(p), &ret);
        err_propogate(subprogram_declarations(p),
            &ret);
        err_propogate(compound_statement(p), &ret)
            ;
        match(p, TOKEN_PERIOD, &ret);
        if (end_green(&p->gbs) == -1) {
            fprintf(p->syn_list, "SEMERR: Scope for
                \"program\" ended but never
                started\n");
            ret = TYPE_ERR;
        }
        break;
    default:
        ret = TYPE_ERR;
        expected_found(p, &expected, 1);
        sync(p, NULL, 0);
    }
    return ret;
}
```

Listing 69: parser/prod/sign.h

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

#ifndef SIGN_H
#define SIGN_H

#include "../parser.h"

int sign(struct parser *p);

#endif
```

Listing 70: parser/prod/sign.c

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

#include "sign.h"

int sign(struct parser *p)
{
    int sync_set[] = {TOKEN_ID, TOKEN_NUM_INTEGER,
            TOKEN_NUM_REAL, TOKEN_LPAREN,
            TOKEN_NOT};
    int expected[] = {TOKEN_SIGN};
    const int sync_size = sizeof(sync_set)/sizeof
        (*sync_set);
    const int expected_size = sizeof(expected)/
        sizeof(*expected);
    int ret = TYPE_VOID;
    switch (p->token.type) {
    case TOKEN_SIGN:
        match(p, TOKEN_SIGN, &ret);
        break;
    default:
        ret = TYPE_ERR;
        expected_found(p, expected, expected_size)
            ;
        sync(p, sync_set, sync_size);
    }
    return ret;
}
```

Listing 71: parser/prod/simple_expression.h

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

#ifndef SIMPLE_EXPRESSION_H
#define SIMPLE_EXPRESSION_H

#include "../parser.h"

int simple_expression(struct parser *p);

#endif
```

Listing 72: parser/prod/simple_expression.c

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

#include "simple_expression.h"
#include "term.h"
#include "simple_expression_prime.h"
#include "sign.h"

int simple_expression(struct parser *p)
{
    int sync_set[] = {TOKEN_RELOP, TOKEN_DO,
        TOKEN_ELSE,
            TOKEN_END, TOKEN_THEN, TOKEN_RPAREN,
            TOKEN_RBRACKET, TOKEN_COMMA,
            TOKEN_SEMICOLON};
    int expected[] = {TOKEN_ID, TOKEN_NUM_INTEGER,
            TOKEN_NUM_REAL, TOKEN_LPAREN,
            TOKEN_NOT, TOKEN_SIGN};
    const int sync_size = sizeof(sync_set)/sizeof
        (*sync_set);
    const int expected_size = sizeof(expected)/
        sizeof(*expected);
    int ret = TYPE_VOID;
    switch (p->token.type) {
    case TOKEN_SIGN: {
        int term_type;
        sign(p);
        term_type = term(p);
        err_propogate(term_type, &ret);
        if (term_type != TYPE_ERR && term_type !=
            TYPE_INT && term_type != TYPE_REAL) {
            fprintf(p->syn_list, "Cannot apply a
                sign to type %s\n",
                strtype(ret));
            ret = TYPE_ERR;
        }
            term_type = simple_expression_prime(p,
                term_type);
        if (ret != TYPE_ERR) {
            ret = term_type;
        }
        break;
    }
    case TOKEN_ID:
    case TOKEN_NUM_INTEGER:
    case TOKEN_NUM_REAL:
    case TOKEN_LPAREN:
    case TOKEN_NOT:
        ret = term(p);
        ret = simple_expression_prime(p, ret);
        break;
    default:
        ret = TYPE_ERR;
        expected_found(p, expected, expected_size)
            ;
        sync(p, sync_set, sync_size);
    }
    return ret;
}
```

Listing 73: parser/prod/simple_expression_prime.h

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

#ifndef SIMPLE_EXPRESSION_PRIME_H
#define SIMPLE_EXPRESSION_PRIME_H

#include "../parser.h"

int simple_expression_prime(struct parser *p, int
    type);

#endif
```

Listing 74: parser/prod/simple_expression_prime.c

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

#include "simple_expression_prime.h"
#include "term.h"

int simple_expression_prime(struct parser *p, int
    type)
{
    int sync_set[] = {TOKEN_RELOP, TOKEN_DO,
        TOKEN_ELSE,
            TOKEN_END, TOKEN_THEN, TOKEN_RPAREN,
            TOKEN_RBRACKET, TOKEN_COMMA,
            TOKEN_SEMICOLON};
    int expected[] = {TOKEN_SIGN, TOKEN_ADDOP,
        TOKEN_RELOP,
            TOKEN_DO, TOKEN_ELSE, TOKEN_END,
            TOKEN_THEN, TOKEN_RPAREN,
            TOKEN_RBRACKET, TOKEN_COMMA,
            TOKEN_SEMICOLON};
    const int sync_size = sizeof(sync_set)/sizeof
        (*sync_set);
    const int expected_size = sizeof(expected)/
        sizeof(*expected);
    int ret = TYPE_VOID;
    struct token t_id;
    switch (p->token.type) {
    case TOKEN_SIGN: {/* used as addition/
        subtraction here */
        int term_type;
        t_id = p->token;
        match(p, TOKEN_SIGN, &ret);
        term_type = term(p);
        if (type != TYPE_ERR && term_type !=
            TYPE_ERR) {
            if (type != term_type ||
                (type != TYPE_INT
                 && type != TYPE_REAL) ||
                (term_type != TYPE_INT
                 && term_type != TYPE_REAL)) {
                fprintf(p->syn_list, "SEMERR:
                    Invalid operands to \"%c\": %s
                    and %s\n",
                    TOKEN_MINUS == t_id.val.attr ? '
                        -' : '+',
                    strtype(type),
                    strtype(term_type));
                ret = TYPE_ERR;
            }
        }
        ret = simple_expression_prime(p, term_type
            );
        break;
    }
    case TOKEN_ADDOP: {/* OR */
        int term_type;
        match(p, TOKEN_ADDOP, &ret);

        term_type = term(p);
        if (type != TYPE_ERR && type != TYPE_BOOL
            && term_type != TYPE_ERR && term_type
            != TYPE_BOOL) {
            fprintf(p->syn_list, "SEMERR: Invalid
                operands to OR: %s and %s\n",
                strtype(type),
                strtype(term_type));
            ret = TYPE_ERR;
        }
        term_type = simple_expression_prime(p,
            term_type);
        if (ret != TYPE_ERR) {
            ret = term_type;
        }
        break;
    }
    case TOKEN_RELOP:
    case TOKEN_DO:
    case TOKEN_ELSE:
    case TOKEN_END:
    case TOKEN_THEN:
    case TOKEN_RPAREN:
    case TOKEN_RBRACKET:
    case TOKEN_COMMA:
    case TOKEN_SEMICOLON:
        ret = type;
        break;
    default:
        ret = TYPE_ERR;
        expected_found(p, expected, expected_size)
            ;
        sync(p, sync_set, sync_size);
    }
    return ret;
}
```

Listing 75: parser/prod/standard_type.h

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

#ifndef STANDARD_TYPE_H
#define STANDARD_TYPE_H

#include "../parser.h"

int standard_type(struct parser *p);

#endif
```

Listing 76: parser/prod/standard_type.c

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

#include "standard_type.h"

int standard_type(struct parser *p)
{
    int sync_set[] = {TOKEN_SEMICOLON,
        TOKEN_RPAREN};
    int expected[] = {TOKEN_INTEGER, TOKEN_REAL};
    const int sync_size = sizeof(sync_set)/sizeof
        (*sync_set);
    const int expected_size = sizeof(expected)/
        sizeof(*expected);
    int ret = TYPE_VOID;
    switch (p->token.type) {
    case TOKEN_INTEGER:
        ret = TYPE_INT;
        match(p, TOKEN_INTEGER, &ret);
        break;
    case TOKEN_REAL:
        ret = TYPE_REAL;
        match(p, TOKEN_REAL, &ret);
        break;
    default:
        ret = TYPE_ERR;
        expected_found(p, expected, expected_size)
            ;
        sync(p, sync_set, sync_size);
    }
    return ret;
}
```

Listing 77: parser/prod/statement.h

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

#ifndef STATEMENT_H
#define STATEMENT_H

#include "../parser.h"

int statement(struct parser *p);

#endif
```

Listing 78: parser/prod/statement.c

```
/* -*- C -*-
```

```
 *
 * statement.c
 *
 * Author: Benjamin T James
 */

#include "statement.h"
#include "compound_statement.h"
#include "variable.h"
#include "expression.h"
#include "statement_prime.h"

int statement(struct parser *p)
{
	int sync_set[] = {TOKEN_END, TOKEN_ELSE,
	    TOKEN_SEMICOLON};
	int expected[] = {TOKEN_BEGIN, TOKEN_ID,
	    TOKEN_IF, TOKEN_WHILE};
	const int sync_size = sizeof(sync_set)/sizeof
	    (*sync_set);
	const int expected_size = sizeof(expected)/
	    sizeof(*expected);
	int ret = TYPE_VOID;
	switch (p->token.type) {
	case TOKEN_BEGIN:
		err_propogate(compound_statement(p), &ret)
		    ;
		break;
	case TOKEN_ID: {
		int var_type, expr_type;
		var_type = variable(p);
		err_propogate(var_type, &ret);
		match(p, TOKEN_ASSIGN, &ret);
		expr_type = expression(p);
		err_propogate(expr_type, &ret);
		if (var_type != expr_type &&
			var_type != TYPE_ERR &&
			expr_type != TYPE_ERR) {
			fprintf(p->syn_list, "SEMERR: Cannot
			    assign type \"%s\" to variable of
			    type \"%s\"\n", strtype(expr_type),
			     strtype(var_type));
			ret = TYPE_ERR;
		}
		break;
	}
	case TOKEN_IF: {
		int expr_type;
		match(p, TOKEN_IF, &ret);
		expr_type = expression(p);
		err_propogate(expr_type, &ret);
		if (expr_type != TYPE_BOOL && expr_type !=
		     TYPE_ERR) {
			fprintf(p->syn_list, "SEMERR: Argument
			    to \"if\" must be a boolean
			    expression\n");
			ret = TYPE_ERR;
		}
		match(p, TOKEN_THEN, &ret);
		err_propogate(statement(p), &ret);
		err_propogate(statement_prime(p), &ret);
		break;
	}
	case TOKEN_WHILE: {
		int expr_type;
		match(p, TOKEN_WHILE, &ret);
		expr_type = expression(p);
		err_propogate(expr_type, &ret);
		if (expr_type != TYPE_BOOL && expr_type !=
		     TYPE_ERR) {
			fprintf(p->syn_list, "SEMERR: Argument
			    to \"while\" must be a boolean
			    expression\n");
			ret = TYPE_ERR;
		}
		match(p, TOKEN_DO, &ret);
		err_propogate(statement(p), &ret);
		break;
	}
	default:
		ret = TYPE_ERR;
		expected_found(p, expected, expected_size)
		    ;
		sync(p, sync_set, sync_size);
	}
	return ret;
}
```

Listing 79: parser/prod/statement_prime.h

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

#ifndef STATEMENT_PRIME_H
#define STATEMENT_PRIME_H

#include "../parser.h"

int statement_prime(struct parser *p);

#endif
```

Listing 80: parser/prod/statement_prime.c

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

#include "statement_prime.h"
#include "statement.h"

int statement_prime(struct parser *p)
{
    int sync_set[] = {TOKEN_ELSE, TOKEN_END,
        TOKEN_SEMICOLON};
    int expected[] = {TOKEN_ELSE, TOKEN_END,
        TOKEN_SEMICOLON};
    const int sync_size = sizeof(sync_set)/sizeof
        (*sync_set);
    const int expected_size = sizeof(expected)/
        sizeof(*expected);
    int ret = TYPE_VOID;
    switch (p->token.type) {
    case TOKEN_ELSE:
        match(p, TOKEN_ELSE, &ret);
        statement(p);
    case TOKEN_SEMICOLON:
    case TOKEN_END:
        break;
    default:
        ret = TYPE_ERR;
        expected_found(p, expected, expected_size)
            ;
        sync(p, sync_set, sync_size);
    }
    return ret;
}
```

Listing 81: parser/prod/statement_list.h

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

#ifndef STATEMENT_LIST_H
#define STATEMENT_LIST_H

#include "../parser.h"

int statement_list(struct parser *p);

#endif
```

Listing 82: parser/prod/statement_list.c

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

#include "statement_list.h"
#include "statement_list_prime.h"
#include "statement.h"
int statement_list(struct parser *p)
{
    int sync_set[] = {TOKEN_END};
    int expected[] = {TOKEN_BEGIN, TOKEN_ID,
        TOKEN_IF, TOKEN_WHILE};
    const int sync_size = sizeof(sync_set)/sizeof
        (*sync_set);
    const int expected_size = sizeof(expected)/
        sizeof(*expected);
    int ret = TYPE_VOID;
    switch (p->token.type) {
    case TOKEN_BEGIN:
    case TOKEN_ID:
    case TOKEN_IF:
    case TOKEN_WHILE:
        err_propogate(statement(p), &ret);
        err_propogate(statement_list_prime(p), &
            ret);
        break;
    case TOKEN_END:
        break;
    default:
        ret = TYPE_ERR;
        expected_found(p, expected, expected_size)
            ;
        sync(p, sync_set, sync_size);
    }
    return ret;
}
```

Listing 83: parser/prod/statement_list_prime.h

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

#ifndef STATEMENT_LIST_PRIME_H
#define STATEMENT_LIST_PRIME_H

#include "../parser.h"

int statement_list_prime(struct parser *p);

#endif
```

Listing 84: parser/prod/statement_list_prime.c

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

#include "statement_list_prime.h"
#include "statement.h"

int statement_list_prime(struct parser *p)
{
    int sync_set[] = {TOKEN_END};
    int expected[] = {TOKEN_SEMICOLON, TOKEN_END};
    const int sync_size = sizeof(sync_set)/sizeof
        (*sync_set);
    const int expected_size = sizeof(expected)/
        sizeof(*expected);
    int ret = TYPE_VOID;
    switch (p->token.type) {
    case TOKEN_SEMICOLON:
        match(p, TOKEN_SEMICOLON, &ret);
        err_propogate(statement(p), &ret);
        err_propogate(statement_list_prime(p), &
            ret);
        break;
    case TOKEN_END:
        break;
    default:
        ret = TYPE_ERR;
        expected_found(p, expected, expected_size)
            ;
        sync(p, sync_set, sync_size);
    }
    return ret;
}
```

Listing 85: parser/prod/subprogram_declaration.h

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

#ifndef SUBPROGRAM_DECLARATION_H
#define SUBPROGRAM_DECLARATION_H

#include "../parser.h"

int subprogram_declaration(struct parser *p);

#endif
```

Listing 86: parser/prod/subprogram_declaration.c

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

#include "subprogram_declaration.h"
#include "subprogram_head.h"
#include "declarations.h"
#include "subprogram_declarations.h"
#include "compound_statement.h"

int subprogram_declaration(struct parser *p)
{
    int sync_set[] = {TOKEN_SEMICOLON};
    int expected[] = {TOKEN_FUNCTION};
    const int sync_size = sizeof(sync_set)/sizeof
        (*sync_set);
    const int expected_size = sizeof(expected)/
        sizeof(*expected);
    int ret = TYPE_VOID;
    switch (p->token.type) {
    case TOKEN_FUNCTION:
        err_propogate(subprogram_head(p), &ret);
        err_propogate(declarations(p), &ret);
        err_propogate(subprogram_declarations(p),
            &ret);
        err_propogate(compound_statement(p), &ret)
            ;
        /* if (check_green_set(&p->gbs) == -1) {
            */
        /*  fprintf(p->syn_list, "SEMERR: Function
             return value not set\n"); */
        /*  ret = -1; */
        /* } */
        if (end_green(&p->gbs) == -1) {
            fprintf(p->syn_list, "SEMERR: Scope for
                 function ended but never started\n
                ");
            ret = TYPE_ERR;
        }
        break;
    default:
        ret = TYPE_ERR;
        expected_found(p, expected, expected_size)
            ;
        sync(p, sync_set, sync_size);
    }
    return ret;
}
```

Listing 87: parser/prod/subprogram_declarations.h

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

#ifndef SUBPROGRAM_DECLARATIONS_H
#define SUBPROGRAM_DECLARATIONS_H

#include "../parser.h"

int subprogram_declarations(struct parser *p);

#endif
```

Listing 88: parser/prod/subprogram_declarations.c

```
/* -*- C -*-
 *
 * subprogram_declarations.c
 *
 * Author: Benjamin T James
 */
#include "subprogram_declaration.h"
#include "subprogram_declarations.h"
```

```c

int subprogram_declarations(struct parser *p)
{
    int sync_set[] = {TOKEN_BEGIN};
    int expected[] = {TOKEN_BEGIN, TOKEN_FUNCTION
        };
    const int sync_size = sizeof(sync_set)/sizeof
        (*sync_set);
    const int expected_size = sizeof(expected)/
        sizeof(*expected);
    int ret = TYPE_VOID;
    switch (p->token.type) {
    case TOKEN_BEGIN:
        break;
    case TOKEN_FUNCTION:
        err_propogate(subprogram_declaration(p), &
            ret);
        match(p, TOKEN_SEMICOLON, &ret);
        err_propogate(subprogram_declarations(p),
            &ret);
        break;
    default:
        ret = TYPE_ERR;
        expected_found(p, expected, expected_size)
            ;
        sync(p, sync_set, sync_size);
    }
    return ret;
}
```

Listing 89: parser/prod/subprogram_head.h

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

#ifndef SUBPROGRAM_HEAD_H
#define SUBPROGRAM_HEAD_H

#include "../parser.h"

int subprogram_head(struct parser *p);

#endif
```

Listing 90: parser/prod/subprogram_head.c

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

#include "subprogram_head.h"
#include "arguments.h"
#include "standard_type.h"

int subprogram_head(struct parser *p)
{
    int sync_set[] = {TOKEN_BEGIN, TOKEN_FUNCTION,
         TOKEN_VAR};
    int expected[] = {TOKEN_FUNCTION};
    const int sync_size = sizeof(sync_set)/sizeof
        (*sync_set);
    const int expected_size = sizeof(expected)/
        sizeof(*expected);
    int ret = TYPE_VOID;
    char *lexeme = NULL;
    char *args = NULL;
    struct token t_id;
    int ret_type = TYPE_VOID;
    switch (p->token.type) {
    case TOKEN_FUNCTION:
        match(p, TOKEN_FUNCTION, &ret);
        t_id = p->token;
        match(p, TOKEN_ID, &ret);
        lexeme = parser_get_tok_lex(p, t_id);
        if (lexeme != NULL) {
            if (check_add_green(&p->gbs, lexeme,
                TYPE_PLACEHOLDER) == -1) {
                fprintf(p->syn_list, "SEMERR: Scope
                     \"%s\" already exists\n",
                    lexeme);
                ret = TYPE_ERR;
            }
        }
        err_propogate(arguments(p, &args), &ret);
        match(p, TOKEN_COLON, &ret);
        ret_type = standard_type(p);
        err_propogate(ret_type, &ret);
        ret_type = SCALAR_TO_FUNC(ret_type);
        set_peek_args_type(&p->gbs, ret_type, args
            );
        free(args);
        match(p, TOKEN_SEMICOLON, &ret);
        break;
    default:
        ret = TYPE_ERR;
        expected_found(p, expected, expected_size)
            ;
        sync(p, sync_set, sync_size);
    }
    return ret;
}
```

Listing 91: parser/prod/term.h

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

#ifndef TERM_H
#define TERM_H

#include "../parser.h"
```

```

int term(struct parser *p);

#endif
```

Listing 92: parser/prod/term.c

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

#include "term.h"
#include "factor.h"
#include "term_prime.h"

int term(struct parser *p)
{
    int sync_set[] = {TOKEN_ADDOP, TOKEN_SIGN,
        TOKEN_RELOP,
              TOKEN_DO, TOKEN_ELSE, TOKEN_END,
              TOKEN_THEN, TOKEN_RPAREN,
              TOKEN_RBRACKET, TOKEN_COMMA,
              TOKEN_SEMICOLON};
    int expected[] = {TOKEN_ID, TOKEN_NUM_INTEGER,
              TOKEN_NUM_REAL, TOKEN_LPAREN,
              TOKEN_NOT};
    const int sync_size = sizeof(sync_set)/sizeof
        (*sync_set);
    const int expected_size = sizeof(expected)/
        sizeof(*expected);
    int ret = TYPE_VOID;
    switch (p->token.type) {
    case TOKEN_ID:
    case TOKEN_NUM_INTEGER:
    case TOKEN_NUM_REAL:
    case TOKEN_LPAREN:
    case TOKEN_NOT:
        ret = factor(p);
        ret = term_prime(p, ret);
        break;
    default:
        ret = TYPE_ERR;
        expected_found(p, expected, expected_size)
            ;
        sync(p, sync_set, sync_size);
    }
    return ret;
}
```

Listing 93: parser/prod/term_prime.h

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

#ifndef TERM_PRIME_H
#define TERM_PRIME_H

#include "../parser.h"

int term_prime(struct parser *p, int type);

#endif
```

Listing 94: parser/prod/term_prime.c

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

#include "term_prime.h"
#include "factor.h"

int mulop(struct parser *p, int t1, int op, int t2
    , int *p_ret);
int term_prime(struct parser *p, int type)
{
    int sync_set[] = {TOKEN_SIGN, TOKEN_ADDOP,
        TOKEN_RELOP,
              TOKEN_DO, TOKEN_ELSE, TOKEN_END,
              TOKEN_THEN, TOKEN_RPAREN,
              TOKEN_RBRACKET, TOKEN_COMMA,
              TOKEN_SEMICOLON};
    int expected[] = {TOKEN_MULOP, TOKEN_SIGN,
        TOKEN_ADDOP, TOKEN_RELOP,
              TOKEN_DO, TOKEN_ELSE, TOKEN_END,
              TOKEN_THEN, TOKEN_RPAREN,
              TOKEN_RBRACKET, TOKEN_COMMA,
              TOKEN_SEMICOLON};
    const int sync_size = sizeof(sync_set)/sizeof
        (*sync_set);
    const int expected_size = sizeof(expected)/
        sizeof(*expected);
    int ret = TYPE_VOID;
    switch (p->token.type) {
    case TOKEN_MULOP: {
        int ftype, op;
        op = p->token.val.attr;
        match(p, TOKEN_MULOP, &ret);
        ftype = factor(p);
        mulop(p, type, op, ftype, &ret);
        ret = term_prime(p, ret);
        break;
    }
    case TOKEN_SIGN: /* used as addition/
        subtraction here */
    case TOKEN_ADDOP:
    case TOKEN_RELOP:
    case TOKEN_DO:
    case TOKEN_ELSE:
    case TOKEN_END:
    case TOKEN_THEN:
    case TOKEN_RPAREN:
```

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

int mulop(struct parser *p, int t1, int op, int t2
    , int *p_ret)
{
    if (t1 == TYPE_ERR || t2 == TYPE_ERR) {
        *p_ret = TYPE_ERR;
    } else if (op == TOKEN_AND) {
        if (t1 != TYPE_BOOL || t2 != TYPE_BOOL) {
            fprintf(p->syn_list, "SEMERR: Invalid
                operands to \"and\": %s and %s\n",
                strtype(t1),
                strtype(t2));
                *p_ret = TYPE_ERR;
        } else {
            *p_ret = TYPE_BOOL;
        }
    } else if (op == TOKEN_TIMES) {
        if (t1 == TYPE_INT && t2 == TYPE_INT) {
            *p_ret = TYPE_INT;
        } else if (t1 == TYPE_REAL && t2 ==
            TYPE_REAL) {
            *p_ret = TYPE_REAL;
        } else {
            fprintf(p->syn_list, "SEMERR: Invalid
                operations to \"*\": %s and %s\n",
                strtype(t1),
                strtype(t2));
            *p_ret = TYPE_ERR;
        }
    } else if (op == TOKEN_RDIV) {
        if (t1 != TYPE_REAL || t2 != TYPE_REAL) {
            fprintf(p->syn_list, "SEMERR: Invalid
                operands to \"/\": %s and %s\n",
                strtype(t1),
                strtype(t2));
                *p_ret = TYPE_ERR;
        } else {
            *p_ret = TYPE_REAL;
        }
    } else if (op == TOKEN_IDIV) {
        if (t1 != TYPE_INT || t2 != TYPE_INT) {
            fprintf(p->syn_list, "SEMERR: Invalid
                operands to \"div\": %s and %s\n",
                strtype(t1),
                strtype(t2));
                *p_ret = TYPE_ERR;
        } else {
            *p_ret = TYPE_INT;
        }
    } else if (op == TOKEN_MOD) {
        if (t1 != TYPE_INT || t2 != TYPE_INT) {
            fprintf(p->syn_list, "SEMERR: Invalid
                operands to \"mod\": %s and %s\n",
                strtype(t1),
                strtype(t2));
                *p_ret = TYPE_ERR;
        } else {
            *p_ret = TYPE_INT;
        }
    } else {
        fprintf(stderr, "unknown operation %d\n",
            op);
    }
    return *p_ret;
}
```

Listing 95: parser/prod/type.h

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

#ifndef TYPE_H
#define TYPE_H

#include "../parser.h"

int type(struct parser *p, int *width);

#endif
```

Listing 96: parser/prod/type.c

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

#include "type.h"
#include "standard_type.h"
int type(struct parser *p, int *width)
{
    int sync_set[] = {TOKEN_SEMICOLON,
        TOKEN_RPAREN};
    int expected[] = {TOKEN_INTEGER, TOKEN_REAL,
        TOKEN_ARRAY};
    const int sync_size = sizeof(sync_set)/sizeof
        (*sync_set);
    const int expected_size = sizeof(expected)/
        sizeof(*expected);
    int a_begin, a_end, a_len;
    int ret = TYPE_VOID;
    switch (p->token.type) {
```

```
    case TOKEN_INTEGER:
    case TOKEN_REAL:
        ret = standard_type(p);
        *width = type_width(ret);
        break;
    case TOKEN_ARRAY:
        match(p, TOKEN_ARRAY, &ret);
        match(p, TOKEN_LBRACKET, &ret);
        a_begin = p->token.val.attr;
        match(p, TOKEN_NUM_INTEGER, &ret);
        if (a_begin != 1) {
            fprintf(p->syn_list, "SEMERR: Arrays
                must start at 1\n");
            ret = TYPE_ERR;
        }
        match(p, TOKEN_ELLIPSIS, &ret);
        a_end = p->token.val.attr;
        match(p, TOKEN_NUM_INTEGER, &ret);
        a_len = a_end - a_begin + 1;
        if (a_end < a_begin) {
            fprintf(p->syn_list, "SEMERR: Must have
                 a positive length array (Length is
                 %d)\n", a_len);
            ret = TYPE_ERR;
        }
        match(p, TOKEN_RBRACKET, &ret);
        match(p, TOKEN_OF, &ret);
        ret = standard_type(p);
        *width = type_width(ret) * a_len;
        ret = SCALAR_TO_ARRAY(ret);
        break;
    default:
        ret = TYPE_ERR;
        expected_found(p, expected, expected_size)
            ;
        sync(p, sync_set, sync_size);
    }
    return ret;
}
```

Listing 97: parser/prod/variable.h

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

#ifndef VARIABLE_H
#define VARIABLE_H

#include "../parser.h"

int variable(struct parser *p);

#endif
```

Listing 98: parser/prod/variable.c

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

#include "variable.h"
#include "variable_prime.h"

int variable(struct parser *p)
{
    int sync_set[] = {TOKEN_ASSIGN};
    int expected[] = {TOKEN_ID};
    const int sync_size = sizeof(sync_set)/sizeof
        (*sync_set);
    const int expected_size = sizeof(expected)/
        sizeof(*expected);
    int ret = TYPE_VOID, type = TYPE_ERR;
    struct token t_id;
    char *lexeme = NULL;
    switch (p->token.type) {
    case TOKEN_ID:
        t_id = p->token;
        match(p, TOKEN_ID, &ret);
        lexeme = parser_get_tok_lex(p, t_id);
        if (lexeme != NULL &&
            gettype(&p->gbs, lexeme, &type) < 0) {
            fprintf(p->syn_list, "SEMERR: Variable
                \"%s\" was not declared in this
                scope\n", lexeme);
            ret = TYPE_ERR;
        }
        if (FUNC_TO_SCALAR(type) != TYPE_ERR) {
            /* function return value */
            type = FUNC_TO_SCALAR(type);
        }
        /* synthesized variable type */
        ret = variable_prime(p, type);
        break;
    default:
        ret = TYPE_ERR;
        expected_found(p, expected, expected_size)
            ;
        sync(p, sync_set, sync_size);
    }
    return ret;
}
```

Listing 99: parser/prod/variable_prime.h

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

#ifndef VARIABLE_PRIME_H
#define VARIABLE_PRIME_H

#include "../parser.h"
```

```

int variable_prime(struct parser *p, int type);

#endif
```

Listing 100: parser/prod/variable_prime.c

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

#include "variable_prime.h"
#include "expression.h"
int variable_prime(struct parser *p, int type)
{
	int sync_set[] = {TOKEN_ASSIGN};
	int expected[] = {TOKEN_LBRACKET, TOKEN_ASSIGN
	    };
	const int sync_size = sizeof(sync_set)/sizeof
	    (*sync_set);
	const int expected_size = sizeof(expected)/
	    sizeof(*expected);
	int ret = TYPE_VOID;
	switch (p->token.type) {
	case TOKEN_LBRACKET: {/* type array */
		int expr_type, scalar_type, rtype;
		scalar_type = ARRAY_TO_SCALAR(type);
		if (scalar_type == TYPE_ERR) {
			fprintf(p->syn_list, "SEMERR: Attempted
			     indexing of non-array type %s\n",
			    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;

}
```