CS 4013: Compiler Construction
Project 2

Benjamin James

November 28, 2017

Introduction

Project 2 consists of "massaging" the Pascal grammar into a $LL(1)$ grammar. Using the grammar, $first$ and $follow$'s were generated and put into a parse table, from which a recursive descent parser was made. The parser checks the syntax of the source program and outputs a parse tree which can be decorated later. Any errors (lexical or syntax) encountered are deposited in the listing file under the appropriate line.

Methodology

There were 6 steps to the process, discussed in Aho et al.[**?**]

- Find left recursion
- Remove ϵ in left-recursive productions
- Remove left recursion
- Left factor the grammar to make it a $LL(1)$ grammar
- Compute the $first$ and $follow$ sets for every production
- Create the parse table

Initial grammar

1 *program* → **program id** (*identifier_list*) ; *declarations subprogram_declarations compound_statement* .
2.1 *identifier_list* → **id**
2.2 *identifier_list* → *identifier_list* , **id**
3.1 *declarations* → *declarations* **var id :** *type* ;
3.2 *declarations* → ϵ
4.1 *type* → *standard_type*
4.2 *type* → **array** [**num .. num**] **of** *standard_type*
5.1 *standard_type* → **integer**
5.2 *standard_type* → **real**
6.1 *subprogram_declarations* → *subprogram_declarations subprogram_declaration* ;
6.2 *subprogram_declarations* → ϵ
7 *subprogram_declaration* → *subprogram_head declarations subprogram_declarations compound_statement*
8 *subprogram_head* → **function id** *arguments* **:** *standard_type* ;
9.1 *arguments* → (*parameter_list*)
9.2 *arguments* → ϵ
10.1 *parameter_list* → **id :** *type*
10.2 *parameter_list* → *parameter_list* **; id :** *type*
11 *compound_statement* → **begin** *optional_statements* **end**
12.1 *optional_statements* → *statement_list*
12.2 *optional_statements* → ϵ
13.1 *statement_list* → *statement*
13.2 *statement_list* → *statement_list* ; *statement*
14.1 *statement* → *variable* **assignop** *expression*
14.2 *statement* → *compound_statement*
14.3 *statement* → **if** *expression* **then** *statement*
14.4 *statement* → **if** *expression* **then** *statement* **else** *statement*
14.5 *statement* → **while** *expression* **do** *statement*
15.1 *variable* → **id**
15.2 *variable* → **id** [*expression*]
16.1 *expression_list* → *expression*
16.2 *expression_list* → *expression_list* , *expression*
17.1 *expression* → *simple_expression*
17.2 *expression* → *simple_expression* **relop** *simple_expression*
18.1 *simple_expression* → *term*
18.2 *simple_expression* → *sign term*
18.3 *simple_expression* → *simple_expression* **addop** *term*
19.1 *term* → *factor*
19.2 *term* → *term* **mulop** *factor*
20.1 *factor* → **id**
20.2 *factor* → **id** [*expression*]
20.3 *factor* → **id** (*expression_list*)
20.4 *factor* → **num**
20.5 *factor* → (*expression*)
20.6 *factor* → **not** *factor*
21.1 *sign* → **+**
21.2 *sign* → **-**

Removal of Epsilon Left Recursion

1 *program* → **program id** (*identifier_list*) ; *declarations subprogram_declarations compound_statement* .
2.1 *identifier_list* → **id**
2.2 *identifier_list* → *identifier_list* , **id**
3.1.1 *declarations* → **var id :** *type* ; *declarations*
3.2.1 *declarations* → ϵ
4.1 *type* → *standard_type*
4.2 *type* → **array** [**num .. num**] **of** *standard_type*
5.1 *standard_type* → **integer**
5.2 *standard_type* → **real**
6.1.1 *subprogram_declarations* → *subprogram_declaration* ; *subprogram_declarations*
6.2.1 *subprogram_declarations* → ϵ
7 *subprogram_declaration* → *subprogram_head declarations subprogram_declarations compound_statement*
8 *subprogram_head* → **function id** *arguments* **:** *standard_type* ;
9.1 *arguments* → (*parameter_list*)
9.2 *arguments* → ϵ
10.1 *parameter_list* → **id :** *type*
10.2 *parameter_list* → *parameter_list* ; **id :** *type*
11 *compound_statement* → **begin** *optional_statements* **end**
12.1 *optional_statements* → *statement_list*
12.2 *optional_statements* → ϵ
13.1 *statement_list* → *statement*
13.2 *statement_list* → *statement_list* ; *statement*
14.1 *statement* → *variable* **assignop** *expression*
14.2 *statement* → *compound_statement*
14.3 *statement* → **if** *expression* **then** *statement*
14.4 *statement* → **if** *expression* **then** *statement* **else** *statement*
14.5 *statement* → **while** *expression* **do** *statement*
15.1 *variable* → **id**
15.2 *variable* → **id** [*expression*]
16.1 *expression_list* → *expression*
16.2 *expression_list* → *expression_list* , *expression*
17.1 *expression* → *simple_expression*
17.2 *expression* → *simple_expression* **relop** *simple_expression*
18.1 *simple_expression* → *term*
18.2 *simple_expression* → *sign term*
18.3 *simple_expression* → *simple_expression* **addop** *term*
19.1 *term* → *factor*
19.2 *term* → *term* **mulop** *factor*
20.1 *factor* → **id**
20.2 *factor* → **id** [*expression*]
20.3 *factor* → **id** (*expression_list*)
20.4 *factor* → **num**
20.5 *factor* → (*expression*)
20.6 *factor* → **not** *factor*
21.1 *sign* → **+**
21.2 *sign* → **-**

Removal of Left Recursion

1 *program* → **program id** (*identifier_list*) ; *declarations subprogram_declarations compound_statement* .
2.1.1 *identifier_list* → **id** *identifier˙list′*
2.2.1 *identifier_list′* → , **id** *identifier_list′*
2.2.2 *identifier_list′* → ϵ
3.1.1 *declarations* → **var id :** *type* ; *declarations*
3.2.1 *declarations* → ϵ
4.1 *type* → *standard_type*
4.2 *type* → **array** [**num .. num**] **of** *standard_type*
5.1 *standard_type* → **integer**
5.2 *standard_type* → **real**
6.1.1 *subprogram_declarations* → *subprogram_declaration* ; *subprogram_declarations*
6.2.1 *subprogram_declarations* → ϵ
7 *subprogram_declaration* → *subprogram_head declarations subprogram_declarations compound_statement*
8 *subprogram_head* → **function id** *arguments* **:** *standard_type* ;
9.1 *arguments* → (*parameter_list*)
9.2 *arguments* → ϵ
10.1.1 *parameter_list* → **id :** *type parameter_list′*
10.2.1 *parameter_list′* → ; **id :** *type parameter_list′*
10.2.2 *parameter_list′* → *epsilon*
11 *compound_statement* → **begin** *optional_statements* **end**
12.1 *optional_statements* → *statement_list*
12.2 *optional_statements* → ϵ
13.1.1 *statement_list* → *statement statement_list′*
13.2.1 *statement_list′* → ; *statement statement_list′*
13.2.2 *statement_list′* → ϵ
14.1 *statement* → *variable* **assignop** *expression*
14.2 *statement* → *compound_statement*
14.3 *statement* → **if** *expression* **then** *statement*
14.4 *statement* → **if** *expression* **then** *statement* **else** *statement*
14.5 *statement* → **while** *expression* **do** *statement*
15.1 *variable* → **id**
15.2 *variable* → **id** [*expression*]
16.1.1 *expression_list* → *expression expression_list′*
16.2.1 *expression_list′* → , *expression expression_list′*
16.2.2 *expression_list′* → ϵ
17.1 *expression* → *simple_expression*
17.2 *expression* → *simple_expression* **relop** *simple_expression*
18.1.1 *simple_expression* → *term simple_expression′*
18.2.1 *simple_expression* → *sign term simple_expression′*
18.3.1 *simple_expression′* → **addop** *term simple_expression′*
18.3.2 *simple_expression′* → *epsilon*
19.1.1 *term* → *factor term′*
19.2.1 *term′* → **mulop** *factor term′*
19.2.2 *term′* → ϵ
20.1 *factor* → **id**
20.2 *factor* → **id** [*expression*]
20.3 *factor* → **id** (*expression_list*)
20.4 *factor* → **num**
20.5 *factor* → (*expression*)
20.6 *factor* → **not** *factor*
21.1 *sign* → +
21.2 *sign* → -

Removal of Left Factoring

1 *program* → **program id** (*identifier_list*) ; *declarations subprogram_declarations compound_statement* .
2.1.1 *identifier_list* → **id** *identifier˙list′*
2.2.1 *identifier_list′* → , **id** *identifier_list′*
2.2.2 *identifier_list′* → ϵ
3.1.1 *declarations* → **var id :** *type* ; *declarations*
3.2.1 *declarations* → ϵ
4.1 *type* → *standard_type*
4.2 *type* → **array** [**num .. num**] **of** *standard_type*
5.1 *standard_type* → **integer**
5.2 *standard_type* → **real**
6.1.1 *subprogram_declarations* → *subprogram_declaration* ; *subprogram_declarations*
6.2.1 *subprogram_declarations* → ϵ
7 *subprogram_declaration* → *subprogram_head declarations subprogram_declarations compound_statement*
8 *subprogram_head* → **function id** *arguments* **:** *standard_type* ;
9.1 *arguments* → (*parameter_list*)
9.2 *arguments* → ϵ
10.1.1 *parameter_list* → **id :** *type parameter_list′*
10.2.1 *parameter_list′* → **; id :** *type parameter_list′*
10.2.2 *parameter_list′* → *epsilon*
11 *compound_statement* → **begin** *optional_statements* **end**
12.1 *optional_statements* → *statement_list*
12.2 *optional_statements* → ϵ
13.1.1 *statement_list* → *statement statement_list′*
13.2.1 *statement_list′* → ; *statement statement_list′*
13.2.2 *statement_list′* → ϵ
14.1.1 *statement* → *variable* **assignop** *expression*
14.2.1 *statement* → *compound_statement*
14.3.1 *statement* → **if** *expression* **then** *statement statement′*
14.4.1 *statement′* → **else** *statement*
14.4.2 *statement′* → ϵ
14.5.1 *statement* → **while** *expression* **do** *statement*
15.1.1 *variable* → **id** *variable′*
15.2.1 *variable′* → [*expression*]
15.2.2 *variable′* → ϵ
16.1.1 *expression_list* → *expression expression_list′*
16.2.1 *expression_list′* → , *expression expression_list′*
16.2.2 *expression_list′* → ϵ
17.1.1 *expression* → *simple_expression expression′*
17.2.1 *expression′* → ϵ 17.2.2 *expression′* → **relop** *simple_expression*
18.1.1 *simple_expression* → *term simple_expression′*
18.2.1 *simple_expression* → *sign term simple_expression′*
18.3.1 *simple_expression′* → **addop** *term simple_expression′*
18.3.2 *simple_expression′* → *epsilon*
19.1.1 *term* → *factor term′*
19.2.1 *term′* → **mulop** *factor term′*
19.2.2 *term′* → ϵ
20.1.1 *factor* → **id** *factor′*
20.2.1 *factor′* → [*expression*]
20.2.2 *factor′* → ϵ
20.3.1 *factor′* → (*expression_list*)
20.4.1 *factor* → **num**
20.5.1 *factor* → (*expression*)
20.6.1 *factor* → **not** *factor*
21.1 *sign* → +
21.2 *sign* → -

First and Follow

ID	Production	First	E?	Follow
1	program	**program**		**$**
2.1.1	identifier_list	**id**		)
2.2.1	identifier_list′	**,**		)
2.2.2	identifier_list′	ϵ	y	)
3.1.1	declarations	**var**		**function begin**
3.2.1	declarations	ϵ	y	**function begin**
4.1	type	**integer real**		**;)**
4.2	type	**array**		**;)**
5.1	standard_type	**integer**		**;)**
5.2	standard_type	**real**		**;)**
6.1.1	subprogram_declarations	**function**		**begin**
6.2.1	subprogram_declarations	ϵ	y	**begin**
7.1	subprogram_declaration	**function**		**;**
8	subprogram_head	**function**		**var function begin**
9.1	arguments	**(**		**:**
9.2	arguments	ϵ	y	**:**
10.1.1	parameter_list	**id**		)
10.2.1	parameter_list′	**;**		)
10.2.2	parameter_list′	ϵ	y	)
11	compound_statement	**begin**		**. ; end else**
12.1	optional_statements	**id begin if while**		**end**
12.2	optional_statements	ϵ	y	**end**
13.1.1	statement_list	**id begin if while**		**end**
13.2.1	statement_list′	**;**		**end**
13.2.2	statement_list′	ϵ	y	**end**
14.1.1	statement	**id**		**end else ;**
14.2.1	statement	**begin**		**end else ;**
14.3.1	statement	**if**		**end else ;**
14.4.1	statement′	**else**		**end else ;**
14.4.2	statement′	ϵ	y	**end else ;**
14.5.1	statement	**while**		**end else ;**
15.1.1	variable	**id**		**assignop**
15.2.1	variable′	**[**		**assignop**
15.2.2	variable′	ϵ	y	**assignop**
16.1.1	expression_list	**id num (not - +**		)
16.2.1	expression_list′	**,**		)
16.2.2	expression_list′	ϵ	y	)
17.1.1	expression	**id num (not - +**		**end else ; then do] ,)**
17.2.1	expression′	ϵ	y	**end else ; then do] ,)**
17.2.2	expression′	**relop**		**end else ; then do] ,)**
18.1.1	simple_expression	**id num (not**		**end else ; then do] ,) relop**
18.2.1	simple_expression	**- +**		**end else ; then do] ,) relop**
18.3.1	simple_expression′	**addop**		**end else ; then do] ,) relop**
18.3.2	simple_expression′	ϵ	y	**end else ; then do] ,) relop**
19.1.1	term	**id num (not**		**end else ; then do] ,) relop addop**
19.2.1	term′	**mulop**		**end else ; then do] ,) relop addop**
19.2.2	term′	ϵ	y	**end else ; then do] ,) relop addop**
20.1.1	factor	**id**		**end else ; then do] ,) relop addop mulop**
20.2.1	factor′	**[**		**end else ; then do] ,) relop addop mulop**
20.2.2	factor′	ϵ	y	**end else ; then do] ,) relop addop mulop**
20.3.1	factor′	**(**		**end else ; then do] ,) relop addop mulop**
20.4.1	factor	**num**		**end else ; then do] ,) relop addop mulop**
20.5.1	factor	**(**		**end else ; then do] ,) relop addop mulop**
20.6.1	factor	**not**		**end else ; then do] ,) relop addop mulop**
21.1	sign	**-**		**id num (not**
21.2	sign	**+**		**id num (not**

Parse Table

ID('s)	Productions	-	+	:	;	,	'	..	(	)	[	]	addop	array	assignop	begin	do	else	end	function	id	if	integer	mulop	not	num	program	real	relop	then	var	while
1	program																										1					
2.1	identifier_list																				2.1.1											
2.2	identifier_list'				2.2.1				2.2.2							3.2.1																
3	declarations																			3.2.1										3.1.1	
4	type													4.2									4.1					4.1				
5	standard_type																						5.1					5.2				
6	subprogram_declarations															6.2.1				6.1.1												
7	subprogram_declaration																			7.1												
8	subprogram_head																			8												
9	arguments			9.2				9.1																								
10.1	parameter_list																				10.1.1											
10.2	parameter_list'				10.2.1					10.2.2																						
11	compound_statement															11																
12	optional_statements															12.1			12.2		12.1	12.1										12.1
13.1	statement_list															13.1.1					13.1.1	13.1.1										13.1.1
13.2	statement_list'				13.2.1														13.2.2													
14.1 14.2 14.3 14.5	statement															14.2.1		14.4.1 14.4.2			14.1.1	14.3.1										14.5.1
14.4	statement'				14.4.2														14.4.2													
15.1	variable																				15.1.1											
15.2	variable'										15.2.1				15.2.2																	
16.1	expression_list	16.1.1	16.1.1						16.1.1												16.1.1				16.1.1	16.1.1						
16.2	expression_list'					16.2.1				16.2.2																						
17.1	expression	17.1.1	17.1.1						17.1.1												17.1.1				17.1.1	17.1.1						
17.2	expression'				17.2.1	17.2.1				17.2.1		17.2.1					17.2.1	17.2.1	17.2.1									17.2.2	17.2.1			
18.1 18.2	simple_expression	18.2.1	18.2.1						18.1.1												18.1.1				18.1.1	18.1.1						
18.3	simple_expression'				18.3.2	18.3.2				18.3.2		18.3.2	18.3.1				18.3.2	18.3.2	18.3.2									18.3.2	18.3.2			
19.1	term								19.1.1												19.1.1				19.1.1	19.1.1						
19.2	term'				19.2.2	19.2.2				19.2.2		19.2.2	19.2.2				19.2.2	19.2.2	19.2.2					19.2.1				19.2.2	19.2.2			
20.1 20.4	20.5 20.6	factor							20.5.1												20.1.1				20.6.1	20.4.1						
20.2	20.3	factor'				20.2.2	20.2.2		20.3.1	20.3.2	20.3.2	20.2.1	20.2.2	20.2.2			20.2.2	20.2.2	20.2.2					20.2.2				20.2.2	20.2.2			
21	sign	21.1	21.2																													

Implementation

The parser is a recursive descent $LL(1)$ parser. Each production has its own file and function. Each production's function contains an array of the expected set and the sync set. The expected set is the $first$ unless $first = \{\epsilon\}$, in that case $first = follow$. The sync set is the $follow \cup \{\$\}$.

Other special functions:

- *match*() - Check if the current token is the matched token, error if it isn't, then get the next token unless the current token is EOF.
- *next_token*() - Get the next token from the token file
- *expected_found*() - Output the syntax error: "expected A, found B"
- *sync*() - Call *next_token*() until the token is in the sync set

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.

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.sym

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

Listing 4: bad_lex.tok

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

```
11      1.12345   5     1
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     0x5543a00
31      #         99    2
32      3.4E+     99    10
32      ;         33    59
33      3.E4      99    11
33      +         7     43
33      ;         33    59
34      34        6     34
34      E         1     0x5544010
34      +         7     43
34      -         7     45
34      ;         33    59
35      E3        1     0x55442b0
35      .         25    46
35      4         6     4
35      +         7     43
35      ;         33    59
37      abcdefghijklmnopqrstuvwxyz0123456789abcdefghijklmnopqrstuvwxyz012345678   99
        1
39      EOF       0     0
```

bad_syn

Listing 5: bad_syn.pas

```
program test(input, output);
        var a : integer;
        var b : array[1..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 6: bad_syn.list

```
1       program test(input, output);
2               var a : integer;
3               var b : array[1..a] of real;
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
11                                      if p = 5 then
12                                              f2 := f + 10
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);
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 7: bad_syn.sym

```
0x5548310       f3
0x5546590       f
0x55450d0       d
0x55449d0       p
0x5544820       f2
0x5544350       c
0x5543de0       int
0x5543b90       x
0x5543710       f1
0x5542e30       b
0x5542960       a
0x55425d0       output
0x55423d0       input
0x5542220       test
```

Listing 8: bad_syn.tok

```
1       program   20    0
1       test      1     0x5542220
1       (         27    40
1       input     1     0x55423d0
1       ,         34    44
1       output    1     0x55425d0
1       )         28    41
1       ;         33    59
2       var       23    0
2       a         1     0x5542960
2       :         32    58
2       integer   17    0
2       ;         33    59
3       var       23    0
3       b         1     0x5542e30
3       :         32    58
3       array     10    0
3       [         29    91
3       1         6     1
3       ..        26    26
3       a         1     0x5542960
3       ]         30    93
3       of        19    0
3       real      21    0
3       ;         33    59
4       function          15    0
4       f1        1     0x5543710
4       (         27    40
4       a         1     0x5542960
```

```
4       :           32      58
4       integer     17      0
4       ,           34      44
4       x           1       0x5543b90
4       :           32      58
4       int         1       0x5543de0
4       )           28      41
4       :           32      58
4       real        21      0
4       ;           33      59
5       var         23      0
5       c           1       0x5544350
5       :           32      58
5       integer     17      0
5       ;           33      59
6       function            15      0
6       f2          1       0x5544820
6       (           27      40
6       p           1       0x55449d0
6       :           32      58
6       integer     17      0
6       )           28      41
6       :           32      58
6       integer     17      0
6       ;           33      59
7       var         23      0
7       d           1       0x55450d0
7       :           32      58
7       real        21      0
7       ;           33      59
8       begin       11      0
9       f2          1       0x5544820
9       :=          31      0
9       5           6       5
9       +           7       43
9       p           1       0x55449d0
9       ;           33      59
10      if          16      0
10      p           1       0x55449d0
10      >           4       45
10      d           1       0x55450d0
10      then        22      0
11      if          16      0
11      p           1       0x55449d0
11      =           4       43
11      5           6       5
11      then        22      0
12      f2          1       0x5544820
12      :=          31      0
12      f           1       0x5546590
12      +           7       43
12      10          6       10
13      else        13      0
14      f2          1       0x5544820
14      :=          31      0
14      100         6       100
15      end         14      0
16      begin       11      0
17      f1          1       0x5543710
17      :=          31      0
```

```
17      f2          1     0x5544820
17      (           27    40
17      a           1     0x5542960
17      )           28    41
17      *           3     42
17      x           1     0x5543b90
18      end         14    0
19      function            15    0
19      f2          1     0x5544820
19      (           27    40
19      a           1     0x5542960
19      :           32    58
19      real        21    0
19      )           28    41
19      :           32    58
19      real        21    0
19      ;           33    59
20      var         23    0
20      b           1     0x5542e30
20      :           32    58
20      integer     17    0
20      ;           33    59
21      begin       11    0
22      f3          1     0x5548310
22      :=          31    0
22      10          6     10
22      *           3     42
22      f1          1     0x5543710
22      (           27    40
22      a           1     0x5542960
22      ,           34    44
22      2.3         5     2
22      )           28    41
22      ;           33    59
23      end         14    0
24      begin       11    0
25      f1          1     0x5543710
25      (           27    40
25      5           6     5
25      ,           34    44
25      3.2         5     3
25      )           28    41
26      end         14    0
26      .           25    46
27      begin       11    0
29      end         14    0
30      EOF         0     0
```

gcd

Listing 9: gcd.pas

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

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

Listing 10: gcd.list

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

Listing 11: gcd.sym

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

Listing 12: gcd.tok

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

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

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

test

Listing 13: test.pas

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

Listing 14: test.list

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

Listing 15: test.sym

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

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

Listing 16: test.tok

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

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

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

Appendix II: Sample Inputs and Outputs

Listing 17: common/io.c

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

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

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

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

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

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

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

Listing 18: common/io.h

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

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

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

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

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

#endif
```

Listing 19: common/defs.h

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

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

#ifndef DEFS_H
#define DEFS_H

#define NOPRINT 1024

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

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


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



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


#define LEXERR 99

#define TOKEN_WHITESPACE 1024
#define TOKEN_NEWLINE 1025

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



#define ID_STRLEN 10

#ifndef LINELEN
#define LINELEN 72
#endif

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

#endif
```

Listing 20: common/util.c

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

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

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

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

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

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

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

Listing 21: common/util.h

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

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


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

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

#endif
```

Listing 22: common/token.c

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

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

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

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

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

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

Listing 23: common/token.h

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

#ifndef TOKEN_H
#define TOKEN_H

#include <stdio.h>

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

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

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

Listing 24: common/idres.c

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

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

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

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

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

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

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

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

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

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

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

Listing 25: common/idres.h

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

#ifndef IDRES_H
#define IDRES_H

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

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

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

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

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

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

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

Listing 26: lexer/main.c

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

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

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

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

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

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

        handle_line(&s, line);

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

Listing 27: lexer/state.c

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

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

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

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

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

Listing 28: lexer/state.h

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

#ifndef STATE_H
#define STATE_H

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

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

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

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

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

Listing 29: lexer/fsm.c

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

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

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

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

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

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

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

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

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

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

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

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

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



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

}
```

Listing 30: lexer/fsm.h

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

#ifndef FSM_H
#define FSM_H

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

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

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

Listing 31: lexer/lexerr.c

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

#include "lexerr.h"

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

Listing 32: lexer/lexerr.h

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

#ifndef LEXERR_H
#define LEXERR_H

#include "machine.h"

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

#endif
```

Listing 33: lexer/machine.c

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

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

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

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

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

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

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

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

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

Listing 34: lexer/machine.h

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

#ifndef MACHINE_H
#define MACHINE_H

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

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

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

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

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

int machine_clean(struct machine **list);

#endif
```

Listing 35: parser/main.c

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

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

Listing 36: 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 37: 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 parser_listing(struct parser *p, unsigned
    lineno)
{
    int c;
        unsigned cur_tok_lineno = p->tok_line;
    if (cur_tok_lineno == lineno) {
        return 0;
    }
    p->tok_line = 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 38: parser/parser.h

```
#ifndef PARSER_H
#define PARSER_H
#include <stdio.h>
#include "idres.h"
#include "defs.h"
struct parser {
    struct idres *symbols;
    FILE* f_token;
    FILE* lex_list;
    FILE* syn_list;
    char *buffer;
    unsigned tok_line, list_line;
    struct token token;
};

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

int parse(struct parser *p);

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

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);
#endif
```

Listing 39: 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);
}

int match(struct parser *p, int tok)
{
    if (p->token.type == tok) {
        if (tok == TOKEN_EOF) {
            return 0;
        } else {
            printf("Matched token %s\n", token2str(
                tok));
            return next_token(p);
        }
    } else {
        expected_found(p, &tok, 1);
        return next_token(p);
/*  return -1;*/
    }
}
int parser_init(struct parser *p, const char*
    symbol_file, const char* token_file, const
    char* list_in, 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->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 0;
}

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);
    fclose(p->f_token);
    fclose(p->lex_list);
    fclose(p->syn_list);
    idres_clean(&p->symbols);
    return 0;
}
```

Listing 40: parser/prod/arguments.h

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

#ifndef ARGUMENTS_H
#define ARGUMENTS_H

#include "../parser.h"

void arguments(struct parser *p);

#endif
```

Listing 41: parser/prod/arguments.c

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

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

void arguments(struct parser *p)
{
    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);

    switch (p->token.type) {
    case TOKEN_LPAREN:
        match(p, TOKEN_LPAREN);
        parameter_list(p);
        match(p, TOKEN_RPAREN);
        break;
    case TOKEN_COLON:
        break;
    default:
        expected_found(p, expected, expected_size)
            ;
        sync(p, sync_set, sync_size);
    }
}
```

Listing 42: parser/prod/compound_statement.h

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

#ifndef COMPOUND_STATEMENT_H
#define COMPOUND_STATEMENT_H

#include "../parser.h"

void compound_statement(struct parser *p);

#endif
```

Listing 43: parser/prod/compound_statement.c

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

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

void 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);

    switch (p->token.type) {
    case TOKEN_BEGIN:
        match(p, TOKEN_BEGIN);
        optional_statements(p);
        match(p, TOKEN_END);
        break;
    default:
        expected_found(p, expected, expected_size)
            ;
```

```
        sync(p, sync_set, sync_size);
    }
}
```

Listing 44: parser/prod/declarations.h

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

#ifndef DECLARATIONS_H
#define DECLARATIONS_H

#include "../parser.h"

void declarations(struct parser *p);

#endif
```

Listing 45: parser/prod/declarations.c

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

#include "declarations.h"
#include "type.h"
void 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);

    switch (p->token.type) {
    case TOKEN_VAR:
        match(p, TOKEN_VAR);
        match(p, TOKEN_ID);
        match(p, TOKEN_COLON);
        type(p);
        match(p, TOKEN_SEMICOLON);
        declarations(p);
        break;
    case TOKEN_BEGIN:
    case TOKEN_FUNCTION:
        break;
    default:
        expected_found(p, expected, expected_size);
        sync(p, sync_set, sync_size);
    }
}
```

Listing 46: parser/prod/expression.h

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

#ifndef EXPRESSION_H
#define EXPRESSION_H

#include "../parser.h"

void expression(struct parser *p);

#endif
```

Listing 47: parser/prod/expression.c

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

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

void 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);

    switch (p->token.type) {
    case TOKEN_ID:
    case TOKEN_NUM_INTEGER:
    case TOKEN_NUM_REAL:
    case TOKEN_LPAREN:
    case TOKEN_NOT:
    case TOKEN_SIGN:
        simple_expression(p);
        expression_prime(p);
        break;
    default:
        expected_found(p, expected, expected_size)
            ;
```

```
        sync(p, sync_set, sync_size);
    }
}
```

Listing 48: parser/prod/expression_prime.h

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

#ifndef EXPRESSION_PRIME_H
#define EXPRESSION_PRIME_H

#include "../parser.h"

void expression_prime(struct parser *p);

#endif
```

Listing 49: parser/prod/expression_prime.c

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

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

void expression_prime(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_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);

    switch (p->token.type) {
    case TOKEN_RELOP:
        match(p, TOKEN_RELOP);
        simple_expression(p);
        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:
        break;
    default:
        expected_found(p, expected, expected_size)
            ;
        sync(p, sync_set, sync_size);
    }

}
```

Listing 50: parser/prod/expression_list.h

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

#ifndef EXPRESSION_LIST_H
#define EXPRESSION_LIST_H

#include "../parser.h"

void expression_list(struct parser *p);

#endif
```

Listing 51: 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"
void expression_list(struct parser *p)
{
    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);

    switch (p->token.type) {
    case TOKEN_ID:
    case TOKEN_NUM_INTEGER:
    case TOKEN_NUM_REAL:
    case TOKEN_LPAREN:
    case TOKEN_NOT:
    case TOKEN_SIGN:
        expression(p);
```

```
        expression_list_prime(p);
        break;
    default:
        expected_found(p, expected, expected_size)
            ;
        sync(p, sync_set, sync_size);
    }
}
```

Listing 52: 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"

void expression_list_prime(struct parser *p);

#endif
```

Listing 53: parser/prod/expression_list_prime.c

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

#include "expression_list_prime.h"
#include "expression.h"

void expression_list_prime(struct parser *p)
{
    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);

    switch (p->token.type) {
    case TOKEN_COMMA:
        match(p, TOKEN_COMMA);
        expression(p);
        expression_list_prime(p);
        break;
    case TOKEN_RPAREN:
        break;
    default:
        expected_found(p, expected, expected_size)
            ;
        sync(p, sync_set, sync_size);
    }

}
```

Listing 54: parser/prod/factor.h

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

#ifndef FACTOR_H
#define FACTOR_H

#include "../parser.h"

void factor(struct parser *p);

#endif
```

Listing 55: parser/prod/factor.c

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

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

void 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);

    switch (p->token.type) {
    case TOKEN_ID:
        match(p, TOKEN_ID);
        factor_prime(p);
        break;
    case TOKEN_NUM_INTEGER:
        match(p, TOKEN_NUM_INTEGER);
        break;
    case TOKEN_NUM_REAL:
        match(p, TOKEN_NUM_REAL);
        break;
```

```
    case TOKEN_LPAREN:
        match(p, TOKEN_LPAREN);
        expression(p);
        match(p, TOKEN_RPAREN);
        break;
    case TOKEN_NOT:
        match(p, TOKEN_NOT);
        factor(p);
        break;
    default:
        expected_found(p, expected, expected_size)
            ;
        sync(p, sync_set, sync_size);
    }

}
```

Listing 56: parser/prod/factor_prime.h

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

#ifndef FACTOR_PRIME_H
#define FACTOR_PRIME_H

#include "../parser.h"

void factor_prime(struct parser *p);

#endif
```

Listing 57: parser/prod/factor_prime.c

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

#include "factor_prime.h"
#include "expression.h"
#include "expression_list.h"

void factor_prime(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_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);

    switch (p->token.type) {
    case TOKEN_LBRACKET:
        match(p, TOKEN_LBRACKET);
        expression(p);
        match(p, TOKEN_RBRACKET);
        break;
    case TOKEN_LPAREN:
        match(p, TOKEN_LPAREN);
        expression_list(p);
        match(p, TOKEN_RPAREN);
        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:
        break;
    default:
        expected_found(p, expected, expected_size)
            ;
        sync(p, sync_set, sync_size);
    }
}
```

Listing 58: parser/prod/identifier_list.h

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

#ifndef IDENTIFIER_LIST_H
#define IDENTIFIER_LIST_H

#include "../parser.h"

void identifier_list(struct parser *p);

#endif
```

Listing 59: parser/prod/identifier_list.c

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

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

#include "identifier_list.h"
#include "identifier_list_prime.h"
void 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);

    switch (p->token.type) {
    case TOKEN_ID:
        match(p, TOKEN_ID);
        identifier_list_prime(p);
        break;
    default:
        expected_found(p, expected, expected_size)
            ;
        sync(p, sync_set, sync_size);
    }
}
```

Listing 60: parser/prod/identifier_list_prime.h

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

#ifndef IDENTIFIER_LIST_PRIME_H
#define IDENTIFIER_LIST_PRIME_H

#include "../parser.h"

void identifier_list_prime(struct parser *p);

#endif
```

Listing 61: parser/prod/identifier_list_prime.c

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

#include "identifier_list_prime.h"

void 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);

    switch (p->token.type) {
    case TOKEN_COMMA:
        match(p, TOKEN_COMMA);
        match(p, TOKEN_ID);
        identifier_list_prime(p);
        break;
    case TOKEN_RPAREN:
        break;
    default:
        expected_found(p, expected, expected_size)
            ;
        sync(p, sync_set, sync_size);
    }
}
```

Listing 62: parser/prod/optional_statements.h

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

#ifndef OPTIONAL_STATEMENTS_H
#define OPTIONAL_STATEMENTS_H

#include "../parser.h"

void optional_statements(struct parser *p);

#endif
```

Listing 63: parser/prod/optional_statements.c

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

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

void 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);

    switch (p->token.type) {
```

```
    case TOKEN_BEGIN:
    case TOKEN_ID:
    case TOKEN_IF:
    case TOKEN_WHILE:
        statement_list(p);
        break;
    case TOKEN_END:
        break;
    default:
        expected_found(p, expected, expected_size)
            ;
        sync(p, sync_set, sync_size);
    }
}
```

Listing 64: parser/prod/parameter_list.h

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

#ifndef PARAMETER_LIST_H
#define PARAMETER_LIST_H

#include "../parser.h"

void parameter_list(struct parser *p);

#endif
```

Listing 65: parser/prod/parameter_list.c

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

#include "parameter_list.h"
#include "type.h"
#include "parameter_list_prime.h"
void parameter_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);

    switch (p->token.type) {
    case TOKEN_ID:
        match(p, TOKEN_ID);
        match(p, TOKEN_COLON);
        type(p);
        parameter_list_prime(p);
        break;
    default:
        expected_found(p, expected, expected_size)
            ;
        sync(p, sync_set, sync_size);
    }
}
```

Listing 66: 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"

void parameter_list_prime(struct parser *p);

#endif
```

Listing 67: parser/prod/parameter_list_prime.c

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

#include "parameter_list_prime.h"
#include "type.h"

void parameter_list_prime(struct parser *p)
{
    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);

    switch (p->token.type) {
    case TOKEN_SEMICOLON:
        match(p, TOKEN_SEMICOLON);
        match(p, TOKEN_ID);
        match(p, TOKEN_COLON);
        type(p);
        parameter_list_prime(p);
        break;
    case TOKEN_RPAREN:
        break;
    default:
        expected_found(p, expected, expected_size)
            ;
        sync(p, sync_set, sync_size);
```

```
    }
}
```

Listing 68: parser/prod/program.h

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

#ifndef PROGRAM_H
#define PROGRAM_H

#include "../parser.h"

void program(struct parser *p);

#endif
```

Listing 69: 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"
void program(struct parser *p)
{
    int expected = TOKEN_PROGRAM;

    switch (p->token.type) {
    case TOKEN_PROGRAM:
        match(p, TOKEN_PROGRAM);
        match(p, TOKEN_ID);
        match(p, TOKEN_LPAREN);
        identifier_list(p);
        match(p, TOKEN_RPAREN);
        match(p, TOKEN_SEMICOLON);
        declarations(p);
        subprogram_declarations(p);
        compound_statement(p);
        match(p, TOKEN_PERIOD);
        break;
    default:
        expected_found(p, &expected, 1);
        sync(p, NULL, 0);
    }
}
```

Listing 70: parser/prod/sign.h

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

#ifndef SIGN_H
#define SIGN_H

#include "../parser.h"

void sign(struct parser *p);

#endif
```

Listing 71: parser/prod/sign.c

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

#include "sign.h"

void 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);

    switch (p->token.type) {
    case TOKEN_SIGN:
        match(p, TOKEN_SIGN);
        break;
    default:
        expected_found(p, expected, expected_size)
            ;
        sync(p, sync_set, sync_size);
    }
}
```

Listing 72: parser/prod/simple_expression.h

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

#ifndef SIMPLE_EXPRESSION_H
#define SIMPLE_EXPRESSION_H

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

```

void simple_expression(struct parser *p);

#endif
```

Listing 73: parser/prod/simple_expression.c

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

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

void 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);

    switch (p->token.type) {
    case TOKEN_SIGN:
        sign(p);
        term(p);
        simple_expression_prime(p);
        break;
    case TOKEN_ID:
    case TOKEN_NUM_INTEGER:
    case TOKEN_NUM_REAL:
    case TOKEN_LPAREN:
    case TOKEN_NOT:
        term(p);
        simple_expression_prime(p);
        break;
    default:
        expected_found(p, expected, expected_size)
            ;
        sync(p, sync_set, sync_size);
    }
}
```

Listing 74: 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"

void simple_expression_prime(struct parser *p);

#endif
```

Listing 75: parser/prod/simple_expression_prime.c

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

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

void simple_expression_prime(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_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);

    switch (p->token.type) {
    case TOKEN_SIGN: /* used as addition/
        subtraction here */
        match(p, TOKEN_SIGN);
        term(p);
        simple_expression_prime(p);
        break;
    case TOKEN_ADDOP:
        match(p, TOKEN_ADDOP);
        term(p);
        simple_expression_prime(p);
        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:
        break;
    default:
        expected_found(p, expected, expected_size);
        sync(p, sync_set, sync_size);
    }
}
```

Listing 76: parser/prod/standard_type.h

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

#ifndef STANDARD_TYPE_H
#define STANDARD_TYPE_H

#include "../parser.h"

void standard_type(struct parser *p);

#endif
```

Listing 77: parser/prod/standard_type.c

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

#include "standard_type.h"

void 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);

    switch (p->token.type) {
    case TOKEN_INTEGER:
        match(p, TOKEN_INTEGER);
        break;
    case TOKEN_REAL:
        match(p, TOKEN_REAL);
        break;
    default:
        expected_found(p, expected, expected_size);
        sync(p, sync_set, sync_size);
    }
}
```

Listing 78: parser/prod/statement.h

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

#ifndef STATEMENT_H
#define STATEMENT_H

#include "../parser.h"

void statement(struct parser *p);

#endif
```

Listing 79: 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"

void 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);

    switch (p->token.type) {
    case TOKEN_BEGIN:
        compound_statement(p);
        break;
    case TOKEN_ID:
        variable(p);
        match(p, TOKEN_ASSIGN);
        expression(p);
        break;
    case TOKEN_IF:
        match(p, TOKEN_IF);
        expression(p);
        match(p, TOKEN_THEN);
        statement(p);
        statement_prime(p);
```

```
        break;
    case TOKEN_WHILE:
        match(p, TOKEN_WHILE);
        expression(p);
        match(p, TOKEN_DO);
        statement(p);
        break;
    default:
        expected_found(p, expected, expected_size)
            ;
        sync(p, sync_set, sync_size);
    }
}
```

Listing 80: parser/prod/statement_prime.h

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

#ifndef STATEMENT_PRIME_H
#define STATEMENT_PRIME_H

#include "../parser.h"

void statement_prime(struct parser *p);

#endif
```

Listing 81: parser/prod/statement_prime.c

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

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

void 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);

    switch (p->token.type) {
    case TOKEN_ELSE:
        match(p, TOKEN_ELSE);
        statement(p);
    case TOKEN_SEMICOLON:
    case TOKEN_END:
        break;
    default:
        expected_found(p, expected, expected_size)
            ;
        sync(p, sync_set, sync_size);
    }
}
```

Listing 82: parser/prod/statement_list.h

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

#ifndef STATEMENT_LIST_H
#define STATEMENT_LIST_H

#include "../parser.h"

void statement_list(struct parser *p);

#endif
```

Listing 83: 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"
void 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);

    switch (p->token.type) {
    case TOKEN_BEGIN:
    case TOKEN_ID:
    case TOKEN_IF:
    case TOKEN_WHILE:
        statement(p);
        statement_list_prime(p);
        break;
    case TOKEN_END:
        break;
    default:
        expected_found(p, expected, expected_size)
            ;
```

```
        sync(p, sync_set, sync_size);
    }
}
```

Listing 84: 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"

void statement_list_prime(struct parser *p);

#endif
```

Listing 85: parser/prod/statement_list_prime.c

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

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

void 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);

    switch (p->token.type) {
    case TOKEN_SEMICOLON:
        match(p, TOKEN_SEMICOLON);
        statement(p);
        statement_list_prime(p);
        break;
    case TOKEN_END:
        break;
    default:
        expected_found(p, expected, expected_size)
            ;
        sync(p, sync_set, sync_size);
    }
}
```

Listing 86: parser/prod/subprogram_declaration.h

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

#ifndef SUBPROGRAM_DECLARATION_H
#define SUBPROGRAM_DECLARATION_H

#include "../parser.h"

void subprogram_declaration(struct parser *p);

#endif
```

Listing 87: 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"

void 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);

    switch (p->token.type) {
    case TOKEN_FUNCTION:
        subprogram_head(p);
        declarations(p);
        subprogram_declarations(p);
        compound_statement(p);
        break;
    default:
        expected_found(p, expected, expected_size)
            ;
        sync(p, sync_set, sync_size);
    }
}
```

Listing 88: parser/prod/subprogram_declarations.h

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

```

#ifndef SUBPROGRAM_DECLARATIONS_H
#define SUBPROGRAM_DECLARATIONS_H

#include "../parser.h"

void subprogram_declarations(struct parser *p);

#endif
```

Listing 89: parser/prod/subprogram_declarations.c

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

void 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);

    switch (p->token.type) {
    case TOKEN_BEGIN:
        break;
    case TOKEN_FUNCTION:
        subprogram_declaration(p);
        match(p, TOKEN_SEMICOLON);
        subprogram_declarations(p);
        break;
    default:
        expected_found(p, expected, expected_size)
            ;
        sync(p, sync_set, sync_size);
    }
}
```

Listing 90: parser/prod/subprogram_head.h

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

#ifndef SUBPROGRAM_HEAD_H
#define SUBPROGRAM_HEAD_H

#include "../parser.h"

void subprogram_head(struct parser *p);

#endif
```

Listing 91: parser/prod/subprogram_head.c

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

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

void 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);

    switch (p->token.type) {
    case TOKEN_FUNCTION:
        match(p, TOKEN_FUNCTION);
        match(p, TOKEN_ID);
        arguments(p);
        match(p, TOKEN_COLON);
        standard_type(p);
        match(p, TOKEN_SEMICOLON);
        break;
    default:
        expected_found(p, expected, expected_size)
            ;
        sync(p, sync_set, sync_size);
    }
}
```

Listing 92: parser/prod/term.h

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

#ifndef TERM_H
#define TERM_H

#include "../parser.h"

void term(struct parser *p);

#endif
```

Listing 93: parser/prod/term.c

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

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

void 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);

    switch (p->token.type) {
    case TOKEN_ID:
    case TOKEN_NUM_INTEGER:
    case TOKEN_NUM_REAL:
    case TOKEN_LPAREN:
    case TOKEN_NOT:
        factor(p);
        term_prime(p);
        break;
    default:
        expected_found(p, expected, expected_size)
            ;
        sync(p, sync_set, sync_size);
    }
}
```

Listing 94: parser/prod/term_prime.h

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

#ifndef TERM_PRIME_H
#define TERM_PRIME_H

#include "../parser.h"

void term_prime(struct parser *p);

#endif
```

Listing 95: parser/prod/term_prime.c

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

#include "term_prime.h"
#include "factor.h"
void term_prime(struct parser *p)
{
    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);

    switch (p->token.type) {
    case TOKEN_MULOP:
        match(p, TOKEN_MULOP);
        factor(p);
        term_prime(p);
        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:
        break;
    default:
        expected_found(p, expected, expected_size)
            ;
        sync(p, sync_set, sync_size);
    }

}
```

Listing 96: parser/prod/type.h

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

#ifndef TYPE_H
#define TYPE_H

#include "../parser.h"

void type(struct parser *p);

#endif
```

Listing 97: parser/prod/type.c

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

#include "type.h"
#include "standard_type.h"
void type(struct parser *p)
{
    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);

    switch (p->token.type) {
    case TOKEN_INTEGER:
    case TOKEN_REAL:
        standard_type(p);
        break;
    case TOKEN_ARRAY:
        match(p, TOKEN_ARRAY);
        match(p, TOKEN_LBRACKET);
        match(p, TOKEN_NUM_INTEGER);
        match(p, TOKEN_ELLIPSIS);
        match(p, TOKEN_NUM_INTEGER);
        match(p, TOKEN_RBRACKET);
        match(p, TOKEN_OF);
        standard_type(p);
        break;
    default:
        expected_found(p, expected, expected_size)
            ;
        sync(p, sync_set, sync_size);
    }

}
```

Listing 98: parser/prod/variable.h

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

#ifndef VARIABLE_H
#define VARIABLE_H

#include "../parser.h"

void variable(struct parser *p);

#endif
```

Listing 99: parser/prod/variable.c

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

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

void 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);

    switch (p->token.type) {
    case TOKEN_ID:
        match(p, TOKEN_ID);
        variable_prime(p);
        break;
    default:
        expected_found(p, expected, expected_size)
            ;
        sync(p, sync_set, sync_size);
    }
}
```

Listing 100: parser/prod/variable_prime.h

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

#ifndef VARIABLE_PRIME_H
#define VARIABLE_PRIME_H
```

```

#include "../parser.h"

void variable_prime(struct parser *p);

#endif
```

Listing 101: parser/prod/variable_prime.c

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

#include "variable_prime.h"
#include "expression.h"
void variable_prime(struct parser *p)
{
    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);

    switch (p->token.type) {
    case TOKEN_LBRACKET:
        match(p, TOKEN_LBRACKET);
        expression(p);
        match(p, TOKEN_RBRACKET);
        break;
    case TOKEN_ASSIGN:
        break;
    default:
        expected_found(p, expected, expected_size)
            ;
        sync(p, sync_set, sync_size);
    }

}
```