

# **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  $\epsilon$  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  $program \rightarrow \text{program id ( identifier\_list ) ; declarations subprogram\_declarations compound\_statement .}$
- 2.1  $identifier\_list \rightarrow \text{id}$
- 2.2  $identifier\_list \rightarrow identifier\_list , \text{id}$
- 3.1  $declarations \rightarrow declarations \text{ var id : type ;}$
- 3.2  $declarations \rightarrow \epsilon$
- 4.1  $type \rightarrow standard\_type$
- 4.2  $type \rightarrow \text{array [ num .. num ] of standard\_type}$
- 5.1  $standard\_type \rightarrow \text{integer}$
- 5.2  $standard\_type \rightarrow \text{real}$
- 6.1  $subprogram\_declarations \rightarrow subprogram\_declarations subprogram\_declaration ;$
- 6.2  $subprogram\_declarations \rightarrow \epsilon$
- 7  $subprogram\_declaration \rightarrow subprogram\_head declarations subprogram\_declarations compound\_statement$
- 8  $subprogram\_head \rightarrow \text{function id arguments : standard\_type ;}$
- 9.1  $arguments \rightarrow ( parameter\_list )$
- 9.2  $arguments \rightarrow \epsilon$
- 10.1  $parameter\_list \rightarrow \text{id : type}$
- 10.2  $parameter\_list \rightarrow parameter\_list ; \text{id : type}$
- 11  $compound\_statement \rightarrow \text{begin optional\_statements end}$
- 12.1  $optional\_statements \rightarrow statement\_list$
- 12.2  $optional\_statements \rightarrow \epsilon$
- 13.1  $statement\_list \rightarrow statement$
- 13.2  $statement\_list \rightarrow statement\_list ; statement$
- 14.1  $statement \rightarrow variable \text{ assignop expression}$
- 14.2  $statement \rightarrow compound\_statement$
- 14.3  $statement \rightarrow \text{if expression then statement}$
- 14.4  $statement \rightarrow \text{if expression then statement else statement}$
- 14.5  $statement \rightarrow \text{while expression do statement}$
- 15.1  $variable \rightarrow \text{id}$
- 15.2  $variable \rightarrow \text{id [ expression ]}$
- 16.1  $expression\_list \rightarrow expression$
- 16.2  $expression\_list \rightarrow expression\_list , expression$
- 17.1  $expression \rightarrow simple\_expression$
- 17.2  $expression \rightarrow simple\_expression \text{ relop simple\_expression}$
- 18.1  $simple\_expression \rightarrow term$
- 18.2  $simple\_expression \rightarrow \text{sign term}$
- 18.3  $simple\_expression \rightarrow simple\_expression \text{ addop term}$
- 19.1  $term \rightarrow factor$
- 19.2  $term \rightarrow term \text{ mulop factor}$
- 20.1  $factor \rightarrow \text{id}$
- 20.2  $factor \rightarrow \text{id [ expression ]}$
- 20.3  $factor \rightarrow \text{id ( expression\_list )}$
- 20.4  $factor \rightarrow \text{num}$
- 20.5  $factor \rightarrow ( expression )$
- 20.6  $factor \rightarrow \text{not factor}$
- 21.1  $sign \rightarrow +$
- 21.2  $sign \rightarrow -$

## 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 →  $\epsilon$ 
  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 →  $\epsilon$ 
  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 →  $\epsilon$ 
  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 →  $\epsilon$ 
  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  $program \rightarrow \text{program id ( identifier\_list ) ; declarations subprogram\_declarations compound\_statement .}$ 
  - 2.1.1  $identifier\_list \rightarrow \text{id identifier\_list'}$
  - 2.2.1  $identifier\_list' \rightarrow \text{, id identifier\_list'}$
  - 2.2.2  $identifier\_list' \rightarrow \epsilon$
  - 3.1.1  $declarations \rightarrow \text{var id : type ; declarations}$
  - 3.2.1  $declarations \rightarrow \epsilon$
  - 4.1  $type \rightarrow \text{standard\_type}$
  - 4.2  $type \rightarrow \text{array [ num .. num ] of standard\_type}$
  - 5.1  $standard\_type \rightarrow \text{integer}$
  - 5.2  $standard\_type \rightarrow \text{real}$
  - 6.1.1  $subprogram\_declarations \rightarrow \text{subprogram\_declaration ; subprogram\_declarations}$
  - 6.2.1  $subprogram\_declarations \rightarrow \epsilon$
  - 7  $subprogram\_declaration \rightarrow \text{subprogram\_head declarations subprogram\_declarations compound\_statement}$
  - 8  $subprogram\_head \rightarrow \text{function id arguments : standard\_type ;}$ 
    - 9.1  $arguments \rightarrow \text{( parameter\_list )}$
    - 9.2  $arguments \rightarrow \epsilon$
    - 10.1.1  $parameter\_list \rightarrow \text{id : type parameter\_list'}$
    - 10.2.1  $parameter\_list' \rightarrow \text{; id : type parameter\_list'}$
    - 10.2.2  $parameter\_list' \rightarrow \epsilon$
  - 11  $compound\_statement \rightarrow \text{begin optional\_statements end}$ 
    - 12.1  $optional\_statements \rightarrow \text{statement\_list}$
    - 12.2  $optional\_statements \rightarrow \epsilon$
    - 13.1.1  $statement\_list \rightarrow \text{statement statement\_list'}$
    - 13.2.1  $statement\_list' \rightarrow \text{; statement statement\_list'}$
    - 13.2.2  $statement\_list' \rightarrow \epsilon$
    - 14.1  $statement \rightarrow \text{variable assignop expression}$
    - 14.2  $statement \rightarrow \text{compound\_statement}$
    - 14.3  $statement \rightarrow \text{if expression then statement}$
    - 14.4  $statement \rightarrow \text{if expression then statement else statement}$
    - 14.5  $statement \rightarrow \text{while expression do statement}$
    - 15.1  $variable \rightarrow \text{id}$
    - 15.2  $variable \rightarrow \text{id [ expression ]}$
    - 16.1.1  $expression\_list \rightarrow \text{expression expression\_list'}$
    - 16.2.1  $expression\_list' \rightarrow \text{, expression expression\_list'}$
    - 16.2.2  $expression\_list' \rightarrow \epsilon$
    - 17.1  $expression \rightarrow \text{simple\_expression}$
    - 17.2  $expression \rightarrow \text{simple\_expression relop simple\_expression}$
    - 18.1.1  $simple\_expression \rightarrow \text{term simple\_expression'}$
    - 18.2.1  $simple\_expression \rightarrow \text{sign term simple\_expression'}$
    - 18.3.1  $simple\_expression' \rightarrow \text{addop term simple\_expression'}$
    - 18.3.2  $simple\_expression' \rightarrow \epsilon$
    - 19.1.1  $term \rightarrow \text{factor term'}$
    - 19.2.1  $term' \rightarrow \text{mulop factor term'}$
    - 19.2.2  $term' \rightarrow \epsilon$
    - 20.1  $factor \rightarrow \text{id}$
    - 20.2  $factor \rightarrow \text{id [ expression ]}$
    - 20.3  $factor \rightarrow \text{id ( expression\_list )}$
    - 20.4  $factor \rightarrow \text{num}$
    - 20.5  $factor \rightarrow \text{( expression )}$
    - 20.6  $factor \rightarrow \text{not factor}$
    - 21.1  $sign \rightarrow \text{+}$
    - 21.2  $sign \rightarrow \text{-}$

## Removal of Left Factoring

1  $program \rightarrow \text{program id ( identifier\_list ) ; declarations subprogram\_declarations compound\_statement .}$   
2.1.1  $identifier\_list \rightarrow \text{id identifier\_list'}$   
2.2.1  $identifier\_list' \rightarrow \text{, id identifier\_list'}$   
2.2.2  $identifier\_list' \rightarrow \epsilon$   
3.1.1  $declarations \rightarrow \text{var id : type ; declarations}$   
3.2.1  $declarations \rightarrow \epsilon$   
4.1  $type \rightarrow \text{standard\_type}$   
4.2  $type \rightarrow \text{array [ num .. num ] of standard\_type}$   
5.1  $standard\_type \rightarrow \text{integer}$   
5.2  $standard\_type \rightarrow \text{real}$   
6.1.1  $subprogram\_declarations \rightarrow \text{subprogram\_declaration ; subprogram\_declarations}$   
6.2.1  $subprogram\_declarations \rightarrow \epsilon$   
7  $subprogram\_declaration \rightarrow \text{subprogram\_head declarations subprogram\_declarations compound\_statement}$   
8  $subprogram\_head \rightarrow \text{function id arguments : standard\_type ;}$   
9.1  $arguments \rightarrow \text{( parameter\_list )}$   
9.2  $arguments \rightarrow \epsilon$   
10.1.1  $parameter\_list \rightarrow \text{id : type parameter\_list'}$   
10.2.1  $parameter\_list' \rightarrow \text{; id : type parameter\_list'}$   
10.2.2  $parameter\_list' \rightarrow \epsilon$   
11  $compound\_statement \rightarrow \text{begin optional\_statements end}$   
12.1  $optional\_statements \rightarrow \text{statement\_list}$   
12.2  $optional\_statements \rightarrow \epsilon$   
13.1.1  $statement\_list \rightarrow \text{statement statement\_list'}$   
13.2.1  $statement\_list' \rightarrow \text{; statement statement\_list'}$   
13.2.2  $statement\_list' \rightarrow \epsilon$   
14.1.1  $statement \rightarrow \text{variable assignop expression}$   
14.2.1  $statement \rightarrow \text{compound\_statement}$   
14.3.1  $statement \rightarrow \text{if expression then statement statement'}$   
14.4.1  $statement' \rightarrow \text{else statement}$   
14.4.2  $statement' \rightarrow \epsilon$   
14.5.1  $statement \rightarrow \text{while expression do statement}$   
15.1.1  $variable \rightarrow \text{id variable'}$   
15.2.1  $variable' \rightarrow \text{[ expression ]}$   
15.2.2  $variable' \rightarrow \epsilon$   
16.1.1  $expression\_list \rightarrow \text{expression expression\_list'}$   
16.2.1  $expression\_list' \rightarrow \text{, expression expression\_list'}$   
16.2.2  $expression\_list' \rightarrow \epsilon$   
17.1.1  $expression \rightarrow \text{simple\_expression expression'}$   
17.2.1  $expression' \rightarrow \epsilon$  17.2.2  $expression' \rightarrow \text{relop simple\_expression}$   
18.1.1  $simple\_expression \rightarrow \text{term simple\_expression'}$   
18.2.1  $simple\_expression \rightarrow \text{sign term simple\_expression'}$   
18.3.1  $simple\_expression' \rightarrow \text{addop term simple\_expression'}$   
18.3.2  $simple\_expression' \rightarrow \epsilon$   
19.1.1  $term \rightarrow \text{factor term'}$   
19.2.1  $term' \rightarrow \text{mulop factor term'}$   
19.2.2  $term' \rightarrow \epsilon$   
20.1.1  $factor \rightarrow \text{id factor'}$   
20.2.1  $factor' \rightarrow \text{[ expression ]}$   
20.2.2  $factor' \rightarrow \epsilon$   
20.3.1  $factor' \rightarrow \text{( expression\_list )}$   
20.4.1  $factor \rightarrow \text{num}$   
20.5.1  $factor \rightarrow \text{( expression )}$   
20.6.1  $factor \rightarrow \text{not factor}$   
21.1  $sign \rightarrow \text{+}$   
21.2  $sign \rightarrow \text{-}$

## First and Follow

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

# Parse Table

ID(s)	Productions	.	+	:	;	,	•	..	(	)	[	]	addop	array	assignop	begin	do	else	end	function	id	if	integer	mulp	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															3.1.1	
3	declarations																			3.2.1													
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	14.4.2		14.1.1	14.3.1									14.5.1		
14.4	statement/					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	18.3.2	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					19.2.1				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	20.2.2	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	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

---

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

---

Listing 2: bad\_lex.list

---

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

```

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

```

---

Listing 3: bad\_lex.sym

---

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

---

Listing 4: bad\_lex.tok

---

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

8	11	1.12345	5	1	
9	12	1.123456		99	5
10	15	123	6	123	
11	16	0123	99	6	
12	17	01.2	99	6	
13	18	01.2E2	99	6	
14	20	1230	6	1230	
15	21	1.2	5	1	
16	22	1.20	99	7	
17	23	1.20E-12		99	7
18	25	1.2E-10	99	7	
19	26	1.2E-123		99	9
20	27	1.2E+5	5	120000	
21	28	1.2E+123		99	9
22	31	e	1	0x5543a00	
23	31	#	99	2	
24	32	3.4E+	99	10	
25	32	;	33	59	
26	33	3.E4	99	11	
27	33	+	7	43	
28	33	;	33	59	
29	34	34	6	34	
30	34	E	1	0x5544010	
31	34	+	7	43	
32	34	-	7	45	
33	34	;	33	59	
34	35	E3	1	0x55442b0	
35	35	.	25	46	
36	35	4	6	4	
37	35	+	7	43	
38	35	;	33	59	
39	37	abcdefghijklmnopqrstuvwxyz0123456789abcdefghijklmnopqrstuvwxyz012345678			99
		1			
40	39	EOF	0	0	

---

Listing 5: bad\_syn.pas

---

```

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

```

---

Listing 6: bad\_syn.list

---

```

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

```

```

26 22          f3 := 10 * f1(a, 2.3);
27 23          end
28 SYNERR: Expected one of "begin", "identifier", "if", or "while" but found "end"
29 24      begin
30 SYNERR: Expected ";" but found "begin"
31 25          f1(5, 3.2)
32 SYNERR: Expected one of "begin" or "function" but found identifier "f1"
33 26      end.
34 27      begin
35 28
36 29      end
37 SYNERR: Expected "." but found "EOF"

```

---

Listing 7: bad\_syn.sym

---

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

---

Listing 8: bad\_syn.tok

---

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

---

30	4	:	32	58
31	4	integer	17	0
32	4	,	34	44
33	4	x	1	0x5543b90
34	4	:	32	58
35	4	int	1	0x5543de0
36	4	)	28	41
37	4	:	32	58
38	4	real	21	0
39	4	;	33	59
40	5	var	23	0
41	5	c	1	0x5544350
42	5	:	32	58
43	5	integer	17	0
44	5	;	33	59
45	6	function	15	0
46	6	f2	1	0x5544820
47	6	(	27	40
48	6	p	1	0x55449d0
49	6	:	32	58
50	6	integer	17	0
51	6	)	28	41
52	6	:	32	58
53	6	integer	17	0
54	6	;	33	59
55	7	var	23	0
56	7	d	1	0x55450d0
57	7	:	32	58
58	7	real	21	0
59	7	;	33	59
60	8	begin	11	0
61	9	f2	1	0x5544820
62	9	:=	31	0
63	9	5	6	5
64	9	+	7	43
65	9	p	1	0x55449d0
66	9	;	33	59
67	10	if	16	0
68	10	p	1	0x55449d0
69	10	>	4	45
70	10	d	1	0x55450d0
71	10	then	22	0
72	11	if	16	0
73	11	p	1	0x55449d0
74	11	=	4	43
75	11	5	6	5
76	11	then	22	0
77	12	f2	1	0x5544820
78	12	:=	31	0
79	12	f	1	0x5546590
80	12	+	7	43
81	12	10	6	10
82	13	else	13	0
83	14	f2	1	0x5544820
84	14	:=	31	0
85	14	100	6	100
86	15	end	14	0
87	16	begin	11	0
88	17	f1	1	0x5543710
89	17	:=	31	0

90	17	f2	1	0x5544820
91	17	(	27	40
92	17	a	1	0x5542960
93	17	)	28	41
94	17	*	3	42
95	17	x	1	0x5543b90
96	18	end	14	0
97	19	function	15	0
98	19	f2	1	0x5544820
99	19	(	27	40
100	19	a	1	0x5542960
101	19	:	32	58
102	19	real	21	0
103	19	)	28	41
104	19	:	32	58
105	19	real	21	0
106	19	;	33	59
107	20	var	23	0
108	20	b	1	0x5542e30
109	20	:	32	58
110	20	integer	17	0
111	20	;	33	59
112	21	begin	11	0
113	22	f3	1	0x5548310
114	22	:=	31	0
115	22	10	6	10
116	22	*	3	42
117	22	f1	1	0x5543710
118	22	(	27	40
119	22	a	1	0x5542960
120	22	,	34	44
121	22	2.3	5	2
122	22	)	28	41
123	22	;	33	59
124	23	end	14	0
125	24	begin	11	0
126	25	f1	1	0x5543710
127	25	(	27	40
128	25	5	6	5
129	25	,	34	44
130	25	3.2	5	3
131	25	)	28	41
132	26	end	14	0
133	26	.	25	46
134	27	begin	11	0
135	29	end	14	0
136	30	EOF	0	0

---



## gcd

Listing 9: gcd.pas

---

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

---

Listing 10: gcd.list

---

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

---

Listing 11: gcd.sym

---

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

---

Listing 12: gcd.tok

---

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

---

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

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

---

Listing 13: test.pas

---

```

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

---

Listing 14: test.list

---

```

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

---

Listing 15: test.sym

---

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

---

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

Listing 16: test.tok

1	1	program	20	0
2	1	test	1	0x55421c0
3	1	(	27	40
4	1	input	1	0x5542370
5	1	,	34	44
6	1	output	1	0x5542570
7	1	)	28	41
8	1	;	33	59
9	2	var	23	0
10	2	a	1	0x5542900
11	2	:	32	58
12	2	integer	17	0
13	2	;	33	59
14	3	var	23	0
15	3	b	1	0x5542dd0
16	3	:	32	58
17	3	array	10	0
18	3	[	29	91
19	3	1	6	1
20	3	..	26	26
21	3	5	6	5
22	3	]	30	93
23	3	of	19	0
24	3	real	21	0
25	3	;	33	59
26	4	function		15 0
27	4	f1	1	0x55436b0
28	4	(	27	40
29	4	a	1	0x5542900
30	4	:	32	58
31	4	integer	17	0
32	4	;	33	59
33	4	x	1	0x5543b30
34	4	:	32	58
35	4	real	21	0
36	4	)	28	41
37	4	:	32	58
38	4	real	21	0
39	4	;	33	59
40	5	var	23	0
41	5	c	1	0x5544230
42	5	:	32	58
43	5	integer	17	0
44	5	;	33	59
45	6	function		15 0
46	6	f2	1	0x5544700
47	6	(	27	40
48	6	p	1	0x55448b0
49	6	:	32	58
50	6	integer	17	0
51	6	)	28	41
52	6	:	32	58
53	6	integer	17	0

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

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

---

## Appendix II: Sample Inputs and Outputs

Listing 17: common/io.c

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

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

Listing 18: common/io.h

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

Listing 19: common/defs.h

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



```

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

```

```

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

```

---

Listing 20: common/util.c

---

```

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

```

```

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

```

Listing 21: common/util.h

```

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

```

```

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

```

Listing 22: common/token.c

```

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

```

```

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

```

Listing 23: common/token.h

```

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

```

```

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

```

Listing 24: common/idres.c

```

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

```

```

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

```

---

Listing 25: common/idres.h

---

```

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

```

```

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

```

Listing 26: lexer/main.c

```

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

```

```

    fprintf(stderr, "Usage: %s source
        reservedWordFile\n", *argv);
    return -1;
}

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

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

    handle_line(&s, line);

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

```

Listing 27: lexer/state.c

```

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

```

```

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

```

---

Listing 28: lexer/state.h

---

```

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

```

---

Listing 29: lexer/fsm.c

---

```

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

```

```

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

```

```

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

```



```

236     token_add(&m->tok, TOKEN_COMMA, *m->f);
237     return 1;
238 case ',':
239     token_add(&m->tok, TOKEN_SEMICOLON, *m->f);
240     ;
241     return 1;
242 default:
243     break;
244 }
245 if (*m->f == '.') {
246     m->f++;
247     if (*m->f == '.') {
248         token_add(&m->tok, TOKEN_ELLIPSIS,
249             TOKEN_ELLIPSIS);
250     } else {
251         m->f--;
252         token_add(&m->tok, TOKEN_PERIOD, '.');
253     }
254     return 1;
255 } else if (*m->f == ':') {
256     m->f++;
257     if (*m->f == '=') {
258         token_add(&m->tok, TOKEN_ASSIGN, 0);
259     } else {
260         m->f--;
261         token_add(&m->tok, TOKEN_COLON, ':');
262     }
263     return 1;
264 }
265 return 0;
266 int fsm_idres(struct machine *m, struct lex_state
267     *ls)
268 {
269     if (isalpha(*m->f)) {
270         int len;
271         char *lexeme;
272         struct idres *result;
273         m->f++;
274         while (isalnum(*m->f)) {
275             m->f++;
276         }
277         m->f--;
278
279         if (get_str(m->f, m->b, &lexeme) < 0) {
280             return -1;
281         }
282         len = strlen(lexeme);
283         if (len > ID_STRLEN) {
284             m->tok.type = LEXERR;
285             m->tok.is_id = 0;
286             m->tok.val.attr = LEXERR_ID_TOO_LONG;
287         } else if (idres_search(&ls->rwords,
288             lexeme, &result) == 0) {
289             m->tok = result->token;
290         } else if (idres_search(&ls->ids, lexeme,
291             &result) == 0) {
292             m->tok = result->token;
293         } else {
294             idres_add_id(&ls->ids, lexeme);
295             m->tok = ls->ids->token;
296         }
297         free(lexeme);
298         return 1;
299     }
300     return 0;
301 }
302 int fsm_unrecognized_symbol(struct machine *m,
303     struct lex_state *ls)
304 {
305     m->tok.type = LEXERR;
306     m->tok.val.attr = LEXERR_UNREC_SYM;
307     return 1;
308 }
309 int fsm_newline(struct machine *m, struct
310     lex_state *ls)
311 {
312     if (*m->f == '\n') {
313         m->f++;
314         m->tok.type = TOKEN_NEWLINE;
315         return 1;
316     }
317     return 0;
318 }
319 int fsm_whitespace(struct machine *m, struct
320     lex_state *ls)
321 {
322     if (*m->f == ' ' || *m->f == '\t') {
323         m->f++;
324         while (*m->f == ' ' || *m->f == '\t') {
325             m->f++;
326         }
327         m->f--;
328         m->tok.type = TOKEN_WHITESPACE;
329         return 1;
330     }
331     return 0;
332 }

```

---

Listing 30: lexer/fsm.h

---

```

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

```

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

Listing 31: lexer/lexerr.c

```

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

```

Listing 32: lexer/lexerr.h

```

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

```

Listing 33: lexer/machine.c

```

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

```

```

24         state->b = state->f;
25
26         return get_str(m->f, m->b, out_str);
27     } else if (ret == -1) {
28         return -1;
29     }
30 }
31 return -1;
32 }
33 int machine_init(struct machine **list)
34 {
35     machine_add(list, fsm_unrecognized_symbol);
36     machine_add(list, fsm_newline);
37
38     machine_add(list, fsm_catchall);
39     machine_add(list, fsm_relop);
40
41     machine_add(list, fsm_integer);
42     machine_add(list, fsm_real);
43     machine_add(list, fsm_long_real);
44
45     machine_add(list, fsm_idres);
46     machine_add(list, fsm_addop);
47     machine_add(list, fsm_mulop);
48
49     machine_add(list, fsm_whitespace);
50     return 0;
51 }
52 int machine_add(struct machine **list,
53                 int (*func)(struct machine *m, struct
54                             lex_state *ls))
55 {
56     struct machine *m = malloc(sizeof(*m));
57     if (m == NULL) {
58         fprintf(stderr, "Unable to allocate
59             resources\n");
60         return -1;
61     }
62     m->call = func;
63     m->next = *list;
64     *list = m;
65     return 0;
66 }
67 int machine_clean(struct machine **list)
68 {
69     struct machine *head = *list;
70     struct machine *tmp;
71     while (head != NULL) {
72         tmp = head;
73         head = head->next;
74         free(tmp);
75     }
76     return 0;
77 }

```

---

Listing 34: lexer/machine.h

```

1  /* -*- C -*-
2  *
3  * machine.h
4  *

```

```

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

```

---

Listing 35: parser/main.c

```

1 #include <stdio.h>
2 #include "parser.h"
3 #include "tokenizer.h"
4
5 int main(int argc, char **argv)
6 {
7     struct parser parser;
8     if (argc != 5) {
9         fprintf(stderr, "Usage: %s symbols tokens
10             listing_in listing_out\n", *argv);
11         return -1;
12     }
13     parser_init(&parser, argv[1], argv[2], argv
14                 [3], argv[4]);
15     /* idres_print(stdout, &parser.symbols); */
16     parse(&parser);
17     parser_cleanup(&parser);
18     return 0;
19 }

```

---

Listing 36: parser/tokenizer.h

```

1 #ifndef TOKENIZER_H
2 #define TOKENIZER_H
3

```

```

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

```

---

Listing 37: parser/tokenizer.c

---

```

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

```

```

43     p->tok_lineno = lineno;
44     while (p->list_lineno < p->tok_lineno) {
45         int fret = fscanf(p->lex_list, "%s", p->
            buffer);
46         if (fret == -1) { /* EOF reached */
47             p->list_lineno = p->tok_lineno;
48             break;
49         }
50         if (strcmp(p->buffer, "LEXERR:")) {
51             char *ptr = NULL;
52             p->list_lineno = strtoul(p->buffer, &ptr,
                10);
53             if (*ptr) {
54                 fprintf(p->syn_list, "Listing file
                    invalid format, fret %d\n",
                        fret);
55                 exit(EXIT_FAILURE);
56             }
57         }
58         fprintf(p->syn_list, "%s", p->buffer);
59         while ((c = fgetc(p->lex_list)) != '\n') {
60             if (c == EOF) {
61                 p->list_lineno = p->tok_lineno;
62                 break;
63             }
64             fputc(c, p->syn_list);
65         }
66         fputc('\n', p->syn_list);
67     }
68     return 0;
69 }

```

Listing 38: parser/parser.h

---

```

1  #ifndef PARSE_H
2  #define PARSE_H
3  #include <stdio.h>
4  #include "idres.h"
5  #include "defs.h"
6  struct parser {
7      struct idres *symbols;
8      FILE* f_token;
9      FILE* lex_list;
10     FILE* syn_list;
11     char *buffer;
12     unsigned tok_lineno, list_lineno;
13     struct token token;
14 };
15
16 int parser_init(struct parser *p, const char*
    symbol_file, const char* token_file, const
    char* list_in, const char* list_out);
17
18 int parse(struct parser *p);
19
20 int match(struct parser *p, int tok);
21
22 int sync(struct parser *p, int* sync_set, int
    sync_size);
23 void expected_found(struct parser *p, int *
    expected, int size);

```

```

24 int parser_cleanup(struct parser *p);
25 #endif

```

---

Listing 39: parser/parser.c

```

1  #include "defs.h"
2  #include "parser.h"
3  #include "tokenizer.h"
4  #include "../prod/program.h"
5
6  int parse(struct parser *p)
7  {
8      next_token(p);
9      program(p);
10     return match(p, TOKEN_EOF);
11 }
12
13 int match(struct parser *p, int tok)
14 {
15     if (p->token.type == tok) {
16         if (tok == TOKEN_EOF) {
17             return 0;
18         } else {
19             printf("Matched token %s\n", token2str(
20                 tok));
21             return next_token(p);
22         }
23     } else {
24         expected_found(p, &tok, 1);
25         return next_token(p);
26     }
27 }
28 /* return -1; */
29
30 int parser_init(struct parser *p, const char*
31     symbol_file, const char* token_file, const
32     char* list_in, const char* list_out)
33 {
34     p->symbols = NULL;
35     if (idres_read(symbol_file, &p->symbols) ==
36         -1) {
37         return -1;
38     }
39     p->f_token = fopen(token_file, "r");
40     if (p->f_token == NULL) {
41         fprintf(stderr, "Could not open file \"%s
42             \"\n", token_file);
43         return -1;
44     }
45     p->buffer = malloc(LINELEN+1);
46     if (p->buffer == NULL) {
47         fprintf(stderr, "Could allocate memory\n")
48             ;
49         return -1;
50     }
51     p->lex_list = fopen(list_in, "r");
52     if (p->lex_list == NULL) {
53         fprintf(stderr, "Could not open file \"%s
54             \"\n", list_in);
55         return -1;
56     }
57     p->syn_list = fopen(list_out, "w");
58     if (p->syn_list == NULL) {
59         fprintf(stderr, "Could not open file \"%s
60             \"\n", list_out);
61         return -1;
62     }
63     p->tok_line = 0;
64     p->list_line = 0;
65     return 0;
66 }
67
68 void expected_found(struct parser *p, int*
69     expected, int size)
70 {
71     int i;
72     fprintf(p->syn_list, "SYNERR: Expected ");
73     if (size > 1) {
74         fprintf(p->syn_list, "one of ");
75     }
76     for (i = 0; i < size - 1; i++) {
77         fprintf(p->syn_list, "\"%s\"", token2str(
78             expected[i]));
79         if (size > 2) {
80             fputc(',', p->syn_list);
81         }
82         fputc(' ', p->syn_list);
83     }
84     if (size > 1) {
85         fprintf(p->syn_list, "or ");
86     }
87     fprintf(p->syn_list, "\"%s\" ", token2str(
88         expected[size-1]));
89     if (p->token.is_id) {
90         struct idres *ret = NULL;
91         idres_lookup(&p->symbols, p->token.val.ptr,
92             &ret);
93         if (ret == NULL) {
94             fprintf(stderr, "symbol not found\n");
95             exit(1);
96         }
97         fprintf(p->syn_list, "but found identifier
98             \"%s\"", ret->lexeme);
99     } else {
100         fprintf(p->syn_list, "but found \"%s\"",
101             token2str(p->token.type));
102     }
103 }
104
105 int sync(struct parser *p, int* sync_set, int
106     sync_size)
107 {
108     int i, found = 0;
109     while (!found) {
110         for (i = 0; i < sync_size; i++) {
111             if (p->token.type == sync_set[i]) {
112                 return 0;
113             }
114         }
115         if (p->token.type == TOKEN_EOF) {
116             return 0;
117         }
118     }
119 }

```

```

102     next_token(p);
103 }
104 return 0;
105 }
106
107 int parser_cleanup(struct parser *p)
108 {
109     int c;
110     while ((c = fgetc(p->lex_list)) != EOF) {
111         fputc(c, p->syn_list);
112     }
113     free(p->buffer);
114     fclose(p->f_token);
115     fclose(p->lex_list);
116     fclose(p->syn_list);
117     idres_clean(&p->symbols);
118     return 0;
119 }

```

---

Listing 40: parser/prod/arguments.h

---

```

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

```

---

Listing 41: parser/prod/arguments.c

---

```

1  /* -*- C -*-
2  *
3  * arguments.c
4  *
5  * Author: Benjamin T James
6  */
7
8  #include "arguments.h"
9  #include "parameter_list.h"
10
11  void arguments(struct parser *p)
12  {
13     int sync_set[] = {TOKEN_COLON};
14     int expected[] = {TOKEN_COLON, TOKEN_LPAREN};
15     const int sync_size = sizeof(sync_set)/sizeof
        (*sync_set);
16     const int expected_size = sizeof(expected)/
        sizeof(*expected);
17
18     switch (p->token.type) {
19     case TOKEN_LPAREN:
20         match(p, TOKEN_LPAREN);

```

```

21         parameter_list(p);
22         match(p, TOKEN_RPAREN);
23         break;
24     case TOKEN_COLON:
25         break;
26     default:
27         expected_found(p, expected, expected_size)
28         ;
29         sync(p, sync_set, sync_size);
30     }

```

---

Listing 42: parser/prod/compound\_statement.h

---

```

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

```

---

Listing 43: parser/prod/compound\_statement.c

---

```

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

```

```

26     sync(p, sync_set, sync_size);
27 }
28 }

```

Listing 44: parser/prod/declarations.h

```

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

```

Listing 45: parser/prod/declarations.c

```

1  /* -*- C -*-
2  *
3  * declarations.c
4  *
5  * Author: Benjamin T James
6  */
7
8  #include "declarations.h"
9  #include "type.h"
10 void declarations(struct parser *p)
11 {
12     int sync_set[] = {TOKEN_BEGIN, TOKEN_FUNCTION
13     };
14     int expected[] = {TOKEN_BEGIN, TOKEN_FUNCTION,
15     TOKEN_VAR};
16     const int sync_size = sizeof(sync_set)/sizeof
17     (*sync_set);
18     const int expected_size = sizeof(expected)/
19     sizeof(*expected);
20
21     switch (p->token.type) {
22     case TOKEN_VAR:
23         match(p, TOKEN_VAR);
24         match(p, TOKEN_ID);
25         match(p, TOKEN_COLON);
26         type(p);
27         match(p, TOKEN_SEMICOLON);
28         declarations(p);
29         break;
30     case TOKEN_BEGIN:
31     case TOKEN_FUNCTION:
32         break;
33     default:
34         expected_found(p, expected, expected_size)
35         ;
36         sync(p, sync_set, sync_size);
37     }
38 }

```

```

33 }

```

Listing 46: parser/prod/expression.h

```

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

```

Listing 47: parser/prod/expression.c

```

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

```

```

36     sync(p, sync_set, sync_size);
37 }
38 }

```

Listing 48: parser/prod/expression\_prime.h

```

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

```

Listing 49: parser/prod/expression\_prime.c

```

1  /* -*- C -*-
2  *
3  * expression_prime.c
4  *
5  * Author: Benjamin T James
6  */
7
8  #include "expression_prime.h"
9  #include "simple_expression.h"
10
11 void expression_prime(struct parser *p)
12 {
13     int sync_set[] = {TOKEN_DO, TOKEN_ELSE,
14                       TOKEN_END,
15                       TOKEN_THEN, TOKEN_RPAREN,
16                       TOKEN_RBRACKET,
17                       TOKEN_COMMA, TOKEN_SEMICOLON};
18     int expected[] = {TOKEN_RELOP, TOKEN_DO,
19                       TOKEN_ELSE,
20                       TOKEN_END, TOKEN_THEN, TOKEN_RPAREN,
21                       TOKEN_RBRACKET, TOKEN_COMMA,
22                       TOKEN_SEMICOLON};
23     const int sync_size = sizeof(sync_set)/sizeof
24     (*sync_set);
25     const int expected_size = sizeof(expected)/
26     sizeof(*expected);
27
28     switch (p->token.type) {
29     case TOKEN_RELOP:
30         match(p, TOKEN_RELOP);
31         simple_expression(p);
32         break;
33     case TOKEN_DO:
34     case TOKEN_ELSE:
35     case TOKEN_END:
36     case TOKEN_THEN:
37     case TOKEN_RPAREN:

```

```

33     case TOKEN_RBRACKET:
34     case TOKEN_COMMA:
35     case TOKEN_SEMICOLON:
36         break;
37     default:
38         expected_found(p, expected, expected_size)
39         ;
40         sync(p, sync_set, sync_size);
41     }
42 }

```

Listing 50: parser/prod/expression\_list.h

```

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

```

Listing 51: parser/prod/expression\_list.c

```

1  /* -*- C -*-
2  *
3  * expression_list.c
4  *
5  * Author: Benjamin T James
6  */
7
8  #include "expression_list.h"
9  #include "expression_list_prime.h"
10 #include "expression.h"
11 void expression_list(struct parser *p)
12 {
13     int sync_set[] = {TOKEN_RPAREN};
14     int expected[] = {TOKEN_ID, TOKEN_NUM_INTEGER,
15                       TOKEN_NUM_REAL, TOKEN_LPAREN,
16                       TOKEN_NOT, TOKEN_SIGN};
17     const int sync_size = sizeof(sync_set)/sizeof
18     (*sync_set);
19     const int expected_size = sizeof(expected)/
20     sizeof(*expected);
21
22     switch (p->token.type) {
23     case TOKEN_ID:
24     case TOKEN_NUM_INTEGER:
25     case TOKEN_NUM_REAL:
26     case TOKEN_LPAREN:
27     case TOKEN_NOT:
28     case TOKEN_SIGN:
29         expression(p);

```



```

28     expression_list_prime(p);
29     break;
30 default:
31     expected_found(p, expected, expected_size)
32         ;
33     sync(p, sync_set, sync_size);
34 }

```

Listing 52: parser/prod/expression\_list\_prime.h

```

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

```

Listing 53: parser/prod/expression\_list\_prime.c

```

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

```

```

30
31 }

```

Listing 54: parser/prod/factor.h

```

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

```

Listing 55: parser/prod/factor.c

```

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

```

<pre> 36     case TOKEN_LPAREN: 37         match(p, TOKEN_LPAREN); 38         expression(p); 39         match(p, TOKEN_RPAREN); 40         break; 41     case TOKEN_NOT: 42         match(p, TOKEN_NOT); 43         factor(p); 44         break; 45     default: 46         expected_found(p, expected, expected_size) 47         ; 48         sync(p, sync_set, sync_size); 49     } 50 }</pre>	<pre> 23         TOKEN_RBRACKET, TOKEN_COMMA, 24         TOKEN_SEMICOLON}; 25     const int sync_size = sizeof(sync_set)/sizeof 26         (*sync_set); 27     const int expected_size = sizeof(expected)/ 28         sizeof(*expected); 29 30     switch (p-&gt;token.type) { 31     case TOKEN_LBRACKET: 32         match(p, TOKEN_LBRACKET); 33         expression(p); 34         match(p, TOKEN_RBRACKET); 35         break; 36     case TOKEN_LPAREN: 37         match(p, TOKEN_LPAREN); 38         expression_list(p); 39         match(p, TOKEN_RPAREN); 40         break; 41     case TOKEN_MULOP: 42     case TOKEN_SIGN: /* used as addition/ 43         subtraction here */ 44     case TOKEN_ADDOP: 45     case TOKEN_RELOP: 46     case TOKEN_DO: 47     case TOKEN_ELSE: 48     case TOKEN_END: 49     case TOKEN_THEN: 50     case TOKEN_RPAREN: 51     case TOKEN_RBRACKET: 52     case TOKEN_COMMA: 53     case TOKEN_SEMICOLON: 54         break; 55     default: 56         expected_found(p, expected, expected_size) 57         ; 58         sync(p, sync_set, sync_size); 59     }</pre>
--	--

Listing 56: parser/prod/factor\_prime.h

```

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

Listing 57: parser/prod/factor\_prime.c

```

1  /* -*- C -*-
2  *
3  * factor_prime.c
4  *
5  * Author: Benjamin T James
6  */
7
8  #include "factor_prime.h"
9  #include "expression.h"
10 #include "expression_list.h"
11
12 void factor_prime(struct parser *p)
13 {
14     int sync_set[] = {TOKEN_MULOP, TOKEN_ADDOP,
15         TOKEN_SIGN, TOKEN_RELOP,
16         TOKEN_DO, TOKEN_ELSE, TOKEN_END,
17         TOKEN_THEN, TOKEN_RPAREN,
18         TOKEN_RBRACKET, TOKEN_COMMA,
19         TOKEN_SEMICOLON};
20     int expected[] = {TOKEN_LBRACKET, TOKEN_LPAREN
21         , TOKEN_MULOP,
22         TOKEN_ADDOP, TOKEN_SIGN, TOKEN_RELOP,
23         TOKEN_DO, TOKEN_ELSE, TOKEN_END,
24         TOKEN_THEN, TOKEN_RPAREN,
```

Listing 58: parser/prod/identifier\_list.h

```

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

Listing 59: parser/prod/identifier\_list.c

```

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

```

3  * identifier_list.c
4  *
5  * Author: Benjamin T James
6  */
7
8  #include "identifier_list.h"
9  #include "identifier_list_prime.h"
10 void identifier_list(struct parser *p)
11 {
12     int sync_set[] = {TOKEN_RPAREN};
13     int expected[] = {TOKEN_ID};
14     const int sync_size = sizeof(sync_set)/sizeof
15         (*sync_set);
16     const int expected_size = sizeof(expected)/
17         sizeof(*expected);
18
19     switch (p->token.type) {
20     case TOKEN_ID:
21         match(p, TOKEN_ID);
22         identifier_list_prime(p);
23         break;
24     default:
25         expected_found(p, expected, expected_size)
26         ;
27         sync(p, sync_set, sync_size);
28     }
29 }

```

Listing 60: parser/prod/identifier\_list\_prime.h

```

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

```

Listing 61: parser/prod/identifier\_list\_prime.c

```

1  /* -*- C -*-
2  *
3  * identifier_list_prime.c
4  *
5  * Author: Benjamin T James
6  */
7
8  #include "identifier_list_prime.h"
9
10 void identifier_list_prime(struct parser *p)
11 {
12     int sync_set[] = {TOKEN_RPAREN};
13     int expected[] = {TOKEN_COMMA, TOKEN_RPAREN};

```

```

14     const int sync_size = sizeof(sync_set)/sizeof
15         (*sync_set);
16     const int expected_size = sizeof(expected)/
17         sizeof(*expected);
18
19     switch (p->token.type) {
20     case TOKEN_COMMA:
21         match(p, TOKEN_COMMA);
22         match(p, TOKEN_ID);
23         identifier_list_prime(p);
24         break;
25     case TOKEN_RPAREN:
26         break;
27     default:
28         expected_found(p, expected, expected_size)
29         ;
30         sync(p, sync_set, sync_size);
31     }
32 }

```

Listing 62: parser/prod/optional\_statements.h

```

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

```

Listing 63: parser/prod/optional\_statements.c

```

1  /* -*- C -*-
2  *
3  * optional_statements.c
4  *
5  * Author: Benjamin T James
6  */
7
8  #include "optional_statements.h"
9  #include "statement_list.h"
10
11 void optional_statements(struct parser *p)
12 {
13     int sync_set[] = {TOKEN_END};
14     int expected[] = {TOKEN_BEGIN, TOKEN_ID,
15         TOKEN_IF, TOKEN_WHILE};
16     const int sync_size = sizeof(sync_set)/sizeof
17         (*sync_set);
18     const int expected_size = sizeof(expected)/
19         sizeof(*expected);
20
21     switch (p->token.type) {

```

```

19     case TOKEN_BEGIN:
20     case TOKEN_ID:
21     case TOKEN_IF:
22     case TOKEN_WHILE:
23         statement_list(p);
24         break;
25     case TOKEN_END:
26         break;
27     default:
28         expected_found(p, expected, expected_size)
29         ;
30         sync(p, sync_set, sync_size);
31 }

```

---

Listing 64: parser/prod/parameter\_list.h

```

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

```

---

Listing 65: parser/prod/parameter\_list.c

```

1  /* -*- C -*-
2  *
3  * parameter_list.c
4  *
5  * Author: Benjamin T James
6  */
7
8 #include "parameter_list.h"
9 #include "type.h"
10 #include "parameter_list_prime.h"
11 void parameter_list(struct parser *p)
12 {
13     int sync_set[] = {TOKEN_RPAREN};
14     int expected[] = {TOKEN_ID};
15     const int sync_size = sizeof(sync_set)/sizeof
16         (*sync_set);
17     const int expected_size = sizeof(expected)/
18         sizeof(*expected);
19
20     switch (p->token.type) {
21     case TOKEN_ID:
22         match(p, TOKEN_ID);
23         match(p, TOKEN_COLON);
24         type(p);
25         parameter_list_prime(p);
26         break;

```

```

25     default:
26         expected_found(p, expected, expected_size)
27         ;
28         sync(p, sync_set, sync_size);
29     }
30 }

```

---

Listing 66: parser/prod/parameter\_list\_prime.h

```

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

```

---

Listing 67: parser/prod/parameter\_list\_prime.c

```

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

```

```

31     }
32 }

```

Listing 68: parser/prod/program.h

```

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

```

Listing 69: parser/prod/program.c

```

1  /* -*- C -*-
2  *
3  * program.c
4  *
5  * Author: Benjamin T James
6  */
7
8  #include "program.h"
9  #include "identifier_list.h"
10 #include "declarations.h"
11 #include "subprogram_declarations.h"
12 #include "compound_statement.h"
13 void program(struct parser *p)
14 {
15     int expected = TOKEN_PROGRAM;
16
17     switch (p->token.type) {
18     case TOKEN_PROGRAM:
19         match(p, TOKEN_PROGRAM);
20         match(p, TOKEN_ID);
21         match(p, TOKEN_LPAREN);
22         identifier_list(p);
23         match(p, TOKEN_RPAREN);
24         match(p, TOKEN_SEMICOLON);
25         declarations(p);
26         subprogram_declarations(p);
27         compound_statement(p);
28         match(p, TOKEN_PERIOD);
29         break;
30     default:
31         expected_found(p, &expected, 1);
32         sync(p, NULL, 0);
33     }
34 }

```

Listing 70: parser/prod/sign.h

```

1  /* -*- C -*-

```

```

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

```

Listing 71: parser/prod/sign.c

```

1  /* -*- C -*-
2  *
3  * sign.c
4  *
5  * Author: Benjamin T James
6  */
7
8  #include "sign.h"
9
10 void sign(struct parser *p)
11 {
12     int sync_set[] = {TOKEN_ID, TOKEN_NUM_INTEGER,
13                       TOKEN_NUM_REAL, TOKEN_LPAREN,
14                       TOKEN_NOT};
15     int expected[] = {TOKEN_SIGN};
16     const int sync_size = sizeof(sync_set)/sizeof
17         (*sync_set);
18     const int expected_size = sizeof(expected)/
19         sizeof(*expected);
20
21     switch (p->token.type) {
22     case TOKEN_SIGN:
23         match(p, TOKEN_SIGN);
24         break;
25     default:
26         expected_found(p, expected, expected_size);
27         sync(p, sync_set, sync_size);
28     }
29 }

```

Listing 72: parser/prod/simple\_expression.h

```

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

```

```

12 void simple_expression(struct parser *p);
13
14
15 #endif

```

---

Listing 73: parser/prod/simple\_expression.c

---

```

1  /* -*- C -*-
2  *
3  * simple_expression.c
4  *
5  * Author: Benjamin T James
6  */
7
8  #include "simple_expression.h"
9  #include "term.h"
10 #include "simple_expression_prime.h"
11 #include "sign.h"
12
13 void simple_expression(struct parser *p)
14 {
15     int sync_set[] = {TOKEN_RELOP, TOKEN_DO,
16                       TOKEN_ELSE,
17                       TOKEN_END, TOKEN_THEN, TOKEN_RPAREN,
18                       TOKEN_RBRACKET, TOKEN_COMMA,
19                       TOKEN_SEMICOLON};
20     int expected[] = {TOKEN_ID, TOKEN_NUM_INTEGER,
21                      TOKEN_NUM_REAL, TOKEN_LPAREN,
22                      TOKEN_NOT, TOKEN_SIGN};
23     const int sync_size = sizeof(sync_set)/sizeof
24         (*sync_set);
25     const int expected_size = sizeof(expected)/
26         sizeof(*expected);
27
28     switch (p->token.type) {
29     case TOKEN_SIGN:
30         sign(p);
31         term(p);
32         simple_expression_prime(p);
33         break;
34     case TOKEN_ID:
35     case TOKEN_NUM_INTEGER:
36     case TOKEN_NUM_REAL:
37     case TOKEN_LPAREN:
38     case TOKEN_NOT:
39         term(p);
40         simple_expression_prime(p);
41         break;
42     default:
43         expected_found(p, expected, expected_size)
44         ;
45         sync(p, sync_set, sync_size);
46     }
47 }

```

Listing 74: parser/prod/simple\_expression\_prime.h

```

1  /* -*- C -*-
2  *
3  * simple_expression_prime.h
4  *

```

```

5  * Author: Benjamin T James
6  */
7
8  #ifndef SIMPLE_EXPRESSION_PRIME_H
9  #define SIMPLE_EXPRESSION_PRIME_H
10
11 #include "../parser.h"
12
13 void simple_expression_prime(struct parser *p);
14
15 #endif

```

Listing 75: parser/prod/simple\_expression\_prime.c

```

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

```

```

43     case TOKEN_COMMA:
44     case TOKEN_SEMICOLON:
45         break;
46     default:
47         expected_found(p, expected, expected_size)
48             ;
49         sync(p, sync_set, sync_size);
50     }

```

Listing 76: parser/prod/standard\_type.h

```

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

```

```

8  #ifndef STANDARD_TYPE_H
9  #define STANDARD_TYPE_H

```

```

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

```

Listing 77: parser/prod/standard\_type.c

```

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

```

```

28 }

```

Listing 78: parser/prod/statement.h

```

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

```

```

8  #ifndef STATEMENT_H
9  #define STATEMENT_H
10
11 #include "../parser.h"
12
13 void statement(struct parser *p);
14
15 #endif

```

Listing 79: parser/prod/statement.c

```

1  /* -*- C -*-
2  *
3  * statement.c
4  *
5  * Author: Benjamin T James
6  */
7
8  #include "statement.h"
9  #include "compound_statement.h"
10 #include "variable.h"
11 #include "expression.h"
12 #include "statement_prime.h"
13
14 void statement(struct parser *p)
15 {
16     int sync_set[] = {TOKEN_END, TOKEN_ELSE,
17                       TOKEN_SEMICOLON};
18     int expected[] = {TOKEN_BEGIN, TOKEN_ID,
19                       TOKEN_IF, TOKEN_WHILE};
20     const int sync_size = sizeof(sync_set)/sizeof
21         (*sync_set);
22     const int expected_size = sizeof(expected)/
23         sizeof(*expected);
24
25     switch (p->token.type) {
26     case TOKEN_BEGIN:
27         compound_statement(p);
28         break;
29     case TOKEN_ID:
30         variable(p);
31         match(p, TOKEN_ASSIGN);
32         expression(p);
33         break;
34     case TOKEN_IF:
35         match(p, TOKEN_IF);
36         expression(p);
37         match(p, TOKEN_THEN);
38         statement(p);
39         statement_prime(p);

```

```

36         break;
37     case TOKEN_WHILE:
38         match(p, TOKEN_WHILE);
39         expression(p);
40         match(p, TOKEN_DO);
41         statement(p);
42         break;
43     default:
44         expected_found(p, expected, expected_size)
45         ;
46         sync(p, sync_set, sync_size);
47 }

```

---

Listing 80: parser/prod/statement\_prime.h

---

```

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

```

---

Listing 81: parser/prod/statement\_prime.c

---

```

1  /* -*- C -*-
2  *
3  * statement_prime.c
4  *
5  * Author: Benjamin T James
6  */
7
8 #include "statement_prime.h"
9 #include "statement.h"
10
11 void statement_prime(struct parser *p)
12 {
13     int sync_set[] = {TOKEN_ELSE, TOKEN_END,
14                       TOKEN_SEMICOLON};
15     int expected[] = {TOKEN_ELSE, TOKEN_END,
16                       TOKEN_SEMICOLON};
17     const int sync_size = sizeof(sync_set)/sizeof
18     (*sync_set);
19     const int expected_size = sizeof(expected)/
20     sizeof(*expected);
21
22     switch (p->token.type) {
23     case TOKEN_ELSE:
24         match(p, TOKEN_ELSE);
25         statement(p);
26     case TOKEN_SEMICOLON:
27     case TOKEN_END:

```

```

24         break;
25     default:
26         expected_found(p, expected, expected_size)
27         ;
28         sync(p, sync_set, sync_size);
29 }

```

---

Listing 82: parser/prod/statement\_list.h

---

```

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

```

---

Listing 83: parser/prod/statement\_list.c

---

```

1  /* -*- C -*-
2  *
3  * statement_list.c
4  *
5  * Author: Benjamin T James
6  */
7
8 #include "statement_list.h"
9 #include "statement_list_prime.h"
10 #include "statement.h"
11 void statement_list(struct parser *p)
12 {
13     int sync_set[] = {TOKEN_END};
14     int expected[] = {TOKEN_BEGIN, TOKEN_ID,
15                       TOKEN_IF, TOKEN_WHILE};
16     const int sync_size = sizeof(sync_set)/sizeof
17     (*sync_set);
18     const int expected_size = sizeof(expected)/
19     sizeof(*expected);
20
21     switch (p->token.type) {
22     case TOKEN_BEGIN:
23     case TOKEN_ID:
24     case TOKEN_IF:
25     case TOKEN_WHILE:
26         statement(p);
27         statement_list_prime(p);
28         break;
29     case TOKEN_END:
30         break;
31     default:
32         expected_found(p, expected, expected_size)
33         ;

```



```

30     sync(p, sync_set, sync_size);
31 }
32 }

```

Listing 84: parser/prod/statement\_list\_prime.h

```

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

```

Listing 85: parser/prod/statement\_list\_prime.c

```

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

```

Listing 86: parser/prod/subprogram\_declaration.h

```

1  /* -*- C -*-

```

```

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

```

Listing 87: parser/prod/subprogram\_declaration.c

```

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

```

Listing 88: parser/prod/subprogram\_declarations.h

```

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

```

```

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

```

Listing 89: parser/prod/subprogram\_declarations.c

```

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

```

Listing 90: parser/prod/subprogram\_head.h

```

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

```

```

14
15 #endif

```

Listing 91: parser/prod/subprogram\_head.c

```

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

```

Listing 92: parser/prod/term.h

```

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

```

Listing 93: parser/prod/term.c

```

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

```

Listing 94: parser/prod/term\_prime.h

```

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

```

#endif

Listing 95: parser/prod/term\_prime.c

```

1  /* -*- C -*-
2  *
3  * term_prime.c
4  *
5  * Author: Benjamin T James
6  */
7
8  #include "term_prime.h"
9  #include "factor.h"
10 void term_prime(struct parser *p)
11 {
12     int sync_set[] = {TOKEN_SIGN, TOKEN_ADDOP,
13                       TOKEN_RELOP,
14                       TOKEN_DO, TOKEN_ELSE, TOKEN_END,
15                       TOKEN_THEN, TOKEN_LPAREN,
16                       TOKEN_RBRACKET, TOKEN_COMMA,
17                       TOKEN_SEMICOLON};
18     int expected[] = {TOKEN_MULOP, TOKEN_SIGN,
19                      TOKEN_ADDOP, TOKEN_RELOP,
20                      TOKEN_DO, TOKEN_ELSE, TOKEN_END,
21                      TOKEN_THEN, TOKEN_LPAREN,
22                      TOKEN_RBRACKET, TOKEN_COMMA,
23                      TOKEN_SEMICOLON};
24     const int sync_size = sizeof(sync_set)/sizeof
25         (*sync_set);
26     const int expected_size = sizeof(expected)/
27         sizeof(*expected);
28
29     switch (p->token.type) {
30     case TOKEN_MULOP:
31         match(p, TOKEN_MULOP);
32         factor(p);
33         term_prime(p);
34         break;
35     case TOKEN_SIGN: /* used as addition/
36                      subtraction here */
37     case TOKEN_ADDOP:
38     case TOKEN_RELOP:
39     case TOKEN_DO:
40     case TOKEN_ELSE:
41     case TOKEN_END:
42     case TOKEN_THEN:
43     case TOKEN_LPAREN:
44     case TOKEN_RBRACKET:
45     case TOKEN_COMMA:
46     case TOKEN_SEMICOLON:
47         break;
48     default:
49         expected_found(p, expected, expected_size)
50             ;
51         sync(p, sync_set, sync_size);
52     }
53 }

```

Listing 96: parser/prod/type.h

```

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

```

Listing 97: parser/prod/type.c

```

1  /* -*- C -*-
2  *
3  * type.c
4  *
5  * Author: Benjamin T James
6  */
7
8  #include "type.h"
9  #include "standard_type.h"
10 void type(struct parser *p)
11 {
12     int sync_set[] = {TOKEN_SEMICOLON,
13                       TOKEN_RPAREN};
14     int expected[] = {TOKEN_INTEGER, TOKEN_REAL,
15                       TOKEN_ARRAY};
16     const int sync_size = sizeof(sync_set)/sizeof
17     (*sync_set);
18     const int expected_size = sizeof(expected)/
19     sizeof(*expected);
20
21     switch (p->token.type) {
22     case TOKEN_INTEGER:
23     case TOKEN_REAL:
24         standard_type(p);
25         break;
26     case TOKEN_ARRAY:
27         match(p, TOKEN_ARRAY);
28         match(p, TOKEN_LBRACKET);
29         match(p, TOKEN_NUM_INTEGER);
30         match(p, TOKEN_ELLIPSIS);
31         match(p, TOKEN_NUM_INTEGER);
32         match(p, TOKEN_RBRACKET);
33         match(p, TOKEN_OF);
34         standard_type(p);
35         break;
36     default:
37         expected_found(p, expected, expected_size)
38         ;
39         sync(p, sync_set, sync_size);
40     }
41 }

```

Listing 98: parser/prod/variable.h

```

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

```

Listing 99: parser/prod/variable.c

```

1  /* -*- C -*-
2  *
3  * variable.c
4  *
5  * Author: Benjamin T James
6  */
7
8  #include "variable.h"
9  #include "variable_prime.h"
10
11 void variable(struct parser *p)
12 {
13     int sync_set[] = {TOKEN_ASSIGN};
14     int expected[] = {TOKEN_ID};
15     const int sync_size = sizeof(sync_set)/sizeof
16     (*sync_set);
17     const int expected_size = sizeof(expected)/
18     sizeof(*expected);
19
20     switch (p->token.type) {
21     case TOKEN_ID:
22         match(p, TOKEN_ID);
23         variable_prime(p);
24         break;
25     default:
26         expected_found(p, expected, expected_size)
27         ;
28         sync(p, sync_set, sync_size);
29     }
30 }

```

Listing 100: parser/prod/variable\_prime.h

```

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

```

10			
11	#include "../parser.h"	14	};
12			const int sync_size = sizeof(sync_set)/sizeof
13	void variable_prime(struct parser *p);	15	(*sync_set);
14			const int expected_size = sizeof(expected)/
15	#endif	16	sizeof(*expected);
		17	switch (p->token.type) {
		18	case TOKEN_LBRACKET:
		19	match(p, TOKEN_LBRACKET);
		20	expression(p);
		21	match(p, TOKEN_RBRACKET);
		22	break;
		23	case TOKEN_ASSIGN:
		24	break;
		25	default:
		26	expected_found(p, expected, expected_size)
		27	;
		28	sync(p, sync_set, sync_size);
		29	}
		30	}

---

Listing 101: parser/prod/variable\_prime.c

```

1  /* -*- C -*-
2  *
3  * variable_prime.c
4  *
5  * Author: Benjamin T James
6  */
7
8  #include "variable_prime.h"
9  #include "expression.h"
10 void variable_prime(struct parser *p)
11 {
12     int sync_set[] = {TOKEN_ASSIGN};
13     int expected[] = {TOKEN_LBRACKET, TOKEN_ASSIGN

```