CS 4013: Compiler Construction: Project 2

Nate Beckemeyer

December 2016

Introduction

For Project 2, I massaged the modified Pascal grammar into an $LL(1)$ grammar. According to that grammar, I implemented a recursive descent parser to construct the parse tree.

The compiler detects any lexical and syntax errors that occur, and reports them in the listing file.

1 Methodology

The recursive descent parser matches productions, starting with the "program" production. The productions are outlined in the grammars included below. The documents are in the following order:

1. The initial grammar without epsilon productions.
2. The new grammar having eliminated left recursion.
3. That grammar now left-factored to become an $LL(1)$ grammar.
4. The first and follows sets for each production.
5. The parse table.

1.1.	*program*	→	**program id** (*identifier_list*) **;** *declarations* *subprogram_declarations* *compound_statement* **.**
1.2.	*program*	→	**program id** (*identifier_list*) **;** *declarations* *compound_statement* **.**
1.3.	*program*	→	**program id** (*identifier_list*) **;** *subprogram_declarations* *compound_statement* **.**
1.4.	*program*	→	**program id** (*identifier_list*) **;** *compound_statement* **.**
2.1.	*identifier_list*	→	**id**
2.2.	*identifier_list*	→	*identifier_list* **, id**
3.1.	*declarations*	→	**var id :** *type* **;**
3.2.	*declarations*	→	*declarations* **var id :** *type* **;**
4.1.	*type*	→	*standard_type*
4.2.	*type*	→	**array** [**num .. num**] **of** *standard_type*
5.1.	*standard_type*	→	**integer**
5.2.	*standard_type*	→	**real**
6.1.	*subprogram_declarations*	→	*subprogram_declaration* **;**
6.2.	*subprogram_declarations*	→	*subprogram_declarations subprogram_declaration* **;**
7.1.	*subprogram_declaration*	→	*subprogram_head declarations* *subprogram_declarations compound_statement*
7.2.	*subprogram_declaration*	→	*subprogram_head* *subprogram_declarations compound_statement*
7.3.	*subprogram_declaration*	→	*subprogram_head declarations* *compound_statement*
7.4.	*subprogram_declaration*	→	*subprogram_head compound_statement*
8.	*subprogram_head*	→	**procedure id** *arguments* **;**
9.1.	*arguments*	→	(*parameter_list*)
9.2.	*arguments*	→	ϵ
10.1.	*parameter_list*	→	**id :** *type*
10.2.	*parameter_list*	→	*parameter_list* **; id :** *type*
11.	*compound_statement*	→	**begin** *optional_statements* **end**
12.1.	*optional_statements*	→	*statement_list*
12.2.	*optional_statements*	→	ϵ
13.1.	*statement_list*	→	*statement*
13.2.	*statement_list*	→	*statement_list* **;** *statement*
14.1.	*statement*	→	*variable* **assignop** *expression*
14.2.	*statement*	→	*procedure_statement*
14.3.	*statement*	→	*compound_statement*
14.4.	*statement*	→	**if** *expression* **then** *statement*
14.5.	*statement*	→	**if** *expression* **then** *statement* **else** *statement*
14.6.	*statement*	→	**while** *expression* **do** *statement*
15.1.	*variable*	→	**id**
15.2.	*variable*	→	**id** [*expression*]
16.1.	*procedure_statement*	→	**call id**
16.2.	*procedure_statement*	→	**call id** (*expression_list*)

17.1.	*expression_list*	→	*expression*
17.2.	*expression_list*	→	*expression_list* **,** *expression*
18.1.	*expression*	→	*simple_expression*
18.2.	*expression*	→	*simple_expression* **relop** *simple_expression*
19.1.	*simple_expression*	→	*term*
19.2.	*simple_expression*	→	*sign term*
19.3.	*simple_expression*	→	*simple_expression* **addop** *term*
20.1.	*term*	→	*factor*
20.2.	*term*	→	*term* **mulop** *factor*
21.1.	*factor*	→	**id**
21.2.	*factor*	→	**id** [*expression*]
21.3.	*factor*	→	**num**
21.4.	*factor*	→	(*expression*)
21.5.	*factor*	→	**not** *factor*
22.1.	*sign*	→	+
22.2.	*sign*	→	−

1.1.	*program*	→	**program id** (*identifier_list*) **;** *declarations* *subprogram_declarations* *compound_statement* **.**
1.2.	*program*	→	**program id** (*identifier_list*) **;** *declarations* *compound_statement* **.**
1.3.	*program*	→	**program id** (*identifier_list*) **;** *subprogram_declarations* *compound_statement* **.**
1.4.	*program*	→	**program id** (*identifier_list*) **;** *compound_statement* **.**
2.1.	*identifier_list*	→	**id** *identifier_list′*
2.2.1.	*identifier_list′*	→	**,** **id** *identifier_list′*
2.2.2.	*identifier_list′*	→	ϵ
3.1.	*declarations*	→	**var id :** *type* **;** *declarations′*
3.2.1.	*declarations′*	→	**var id :** *type* **;** *declarations′*
3.2.2.	*declarations′*	→	ϵ
4.1.	*type*	→	*standard_type*
4.2.	*type*	→	**array** [**num .. num**] **of** *standard_type*
5.1.	*standard_type*	→	**integer**
5.2.	*standard_type*	→	**real**
6.1.	*subprogram_declarations*	→	*subprogram_declaration* **;** *subprogram_declarations′*
6.2.1.	*subprogram_declarations′*	→	*subprogram_declaration* **;** *subprogram_declarations′*
6.2.2.	*subprogram_declarations′*	→	ϵ
7.1.	*subprogram_declaration*	→	*subprogram_head declarations* *subprogram_declarations compound_statement*
7.2.	*subprogram_declaration*	→	*subprogram_head declarations* *compound_statement*
7.3.	*subprogram_declaration*	→	*subprogram_head* *subprogram_declarations compound_statement*
7.4.	*subprogram_declaration*	→	*subprogram_head compound_statement*
8.	*subprogram_head*	→	**procedure id** *arguments* **;**
9.1.	*arguments*	→	(*parameter_list*)
9.2.	*arguments*	→	ϵ
10.1.	*parameter_list*	→	**id :** *type parameter_list′*
10.2.1.	*parameter_list′*	→	**; id :** *type parameter_list′*
10.2.2.	*parameter_list′*	→	ϵ
11.	*compound_statement*	→	**begin** *optional_statements* **end**
12.1.	*optional_statements*	→	*statement_list*
12.2.	*optional_statements*	→	ϵ
13.1.	*statement_list*	→	*statement statement_list′*
13.2.1.	*statement_list′*	→	**;** *statement statement_list′*
13.2.2.	*statement_list′*	→	ϵ
14.1.	*statement*	→	*variable* **assignop** *expression*
14.2.	*statement*	→	*procedure_statement*
14.3.	*statement*	→	*compound_statement*
14.4.	*statement*	→	**if** *expression* **then** *statement*
14.5.	*statement*	→	**if** *expression* **then** *statement* **else** *statement*

14.6.	*statement*	→	**while** *expression* **do** *statement*
15.1.	*variable*	→	**id**
15.2.	*variable*	→	**id** [*expression*]
16.1.	*procedure_statement*	→	**call id**
16.2.	*procedure_statement*	→	**call id** (*expression_list*)
17.1.	*expression_list*	→	*expression expression_list′*
17.2.1.	*expression_list′*	→	**,** *expression expression_list′*
17.2.2.	*expression_list′*	→	ϵ
18.1.	*expression*	→	*simple_expression*
18.2.	*expression*	→	*simple_expression* **relop** *simple_expression*
19.1.	*simple_expression*	→	*term simple_expression′*
19.2.	*simple_expression*	→	*sign term simple_expression′*
19.3.1.	*simple_expression′*	→	**addop** *term simple_expression′*
19.3.2.	*simple_expression′*	→	ϵ
20.1.	*term*	→	*factor term′*
20.2.1.	*term′*	→	**mulop** *factor term′*
20.2.2.	*term′*	→	ϵ
21.1.	*factor*	→	**id**
21.2.	*factor*	→	**id** [*expression*]
21.3.	*factor*	→	**num**
21.4.	*factor*	→	(*expression*)
21.5.	*factor*	→	**not** *factor*
22.1.	*sign*	→	+
22.2.	*sign*	→	−

1.	*program*	→	**program id** (*identifier_list*) **;** *declarations* *subprogram_declarations* *compound_statement* **.**
2.1.	*identifier_list*	→	**id** *identifier_list′*
2.2.1.	*identifier_list′*	→	**,** **id** *identifier_list′*
2.2.2.	*identifier_list′*	→	ϵ
3.1.	*declarations*	→	**var id :** *type* **;** *declarations*
3.2.	*declarations*	→	ϵ
4.1.	*type*	→	*standard_type*
4.2.	*type*	→	**array** [**num .. num**] **of** *standard_type*
5.1.	*standard_type*	→	**integer**
5.2.	*standard_type*	→	**real**
6.1.	*subprogram_declarations*	→	*subprogram_declaration* **;** *subprogram_declarations*
6.2.	*subprogram_declarations*	→	ϵ
7.	*subprogram_declaration*	→	*subprogram_head declarations* *subprogram_declarations compound_statement*
8.	*subprogram_head*	→	**procedure id** *arguments* **;**
9.1.	*arguments*	→	(*parameter_list*)
9.2.	*arguments*	→	ϵ
10.1.	*parameter_list*	→	**id :** *type parameter_list′*
10.2.1.	*parameter_list′*	→	**;** **id :** *type parameter_list′*
10.2.2.	*parameter_list′*	→	ϵ
11.	*compound_statement*	→	**begin** *optional_statements* **end**
12.1.	*optional_statements*	→	*statement_list*
12.2.	*optional_statements*	→	ϵ
13.1.	*statement_list*	→	*statement statement_list′*
13.2.1.	*statement_list′*	→	**;** *statement statement_list′*
13.2.2.	*statement_list′*	→	ϵ
14.1.	*statement*	→	*variable* **assignop** *expression*
14.2.	*statement*	→	*procedure_statement*
14.3.	*statement*	→	*compound_statement*
14.4.	*statement*	→	**while** *expression* **do** *statement*
14.5.	*statement*	→	**if** *expression* **then** *statement else′*
15.1.	*else′*	→	**else** *statement*
15.2.	*else′*	→	ϵ
16.	*variable*	→	**id** *array_access*
17.1.	*array_access*	→	[*expression*]
17.2.	*array_access*	→	ϵ
18.	*procedure_statement*	→	**call id** *optional_expressions*
19.1.	*optional_expressions*	→	(*expression_list*)
19.2.	*optional_expressions*	→	ϵ
20.1.	*expression_list*	→	*expression expression_list′*
20.2.1.	*expression_list′*	→	**,** *expression expression_list′*
20.2.2.	*expression_list′*	→	ϵ
21.	*expression*	→	*simple_expression related_expression*
22.1.	*related_expression*	→	**relop** *simple_expression*
22.2.	*related_expression*	→	ϵ
23.1.1.	*simple_expression*	→	*term simple_expression′*
23.1.2.	*simple_expression*	→	*sign term simple_expression′*
23.2.1.	*simple_expression′*	→	**addop** *term simple_expression′*
23.2.2.	*simple_expression′*	→	ϵ

24.1.	*term*	$\rightarrow$	*factor term′*
24.2.1.	*term′*	$\rightarrow$	**mulop** *factor term′*
24.2.2.	*term′*	$\rightarrow$	ϵ
25.1.1.	*factor*	$\rightarrow$	**id** *factor'*
25.1.2.	*factor*	$\rightarrow$	**num**
25.1.3.	*factor*	$\rightarrow$	(*expression*)
25.1.4.	*factor*	$\rightarrow$	**not** *factor*
25.2.1.	*factor'*	$\rightarrow$	[*expression*]
25.2.1.	*factor'*	$\rightarrow$	ϵ
26.1.	*sign*	$\rightarrow$	$+$
26.2.	*sign*	$\rightarrow$	$-$

ID	Name	First	Leads	Follows
1.	*program*	**program**		$
2.1.	*identifier_list*	**id**		)
2.2.1.	*identifier_list′*	**,**		)
2.2.2.	*identifier_list′*	ϵ	→	)
3.1.	*declarations*	**var**		**procedure begin**
3.2.	*declarations*	ϵ	→	**procedure begin**
4.1.	*type*	**integer real**		**;)**
4.2.	*type*	**array**		**;)**
5.1.	*standard_type*	**integer**		**;)**
5.2.	*standard_type*	**real**		**;)**
6.1.	*subprogram_declarations*	**procedure**		**begin**
6.2.	*subprogram_declarations*	ϵ	→	**begin**
7.	*subprogram_declaration*	**procedure**		**;**
8.	*subprogram_head*	**procedure**		**var procedure begin**
9.1.	*arguments*	**(**		**;**
9.2.	*arguments*	ϵ	→	**;**
10.1.	*parameter_list*	**id**		**)**
10.2.1.	*parameter_list′*	**;**		**)**
10.2.2.	*parameter_list′*	ϵ	→	**)**
11.	*compound_statement*	**begin**		**; . end else**
12.1.	*optional_statements*	**id call begin while if**		**end**
12.2.	*optional_statements*	ϵ	→	**end**
13.1.	*statement_list*	**id call begin while if**		**end**
13.2.1.	*statement_list′*	**;**		**end**
13.2.2.	*statement_list′*	ϵ	→	**end**
14.1.	*statement*	**id**		**; end else**
14.2.	*statement*	**call**		**; end else**
14.3.	*statement*	**begin**		**; end else**
14.4.	*statement*	**while**		**; end else**
14.5.	*statement*	**if**		**; end else**
15.1.	*else′*	**else**		**; end else**
15.2.	*else′*	ϵ	→	**; end else**
16.	*variable*	**id**		**assignop**
17.1.	*array_access*	**[**		**assignop**
17.2.	*array_access*	ϵ	→	**assignop**
18.	*procedure_statement*	**call**		**; end else**
19.1.	*optional_expressions*	**(**		**; end else**
19.2.	*optional_expressions*	ϵ	→	**; end else**
20.1.	*expression_list*	**id num (not + -**		**)**
20.2.1.	*expression_list′*	**,**		**)**
20.2.2.	*expression_list′*	ϵ	→	**)**
21.	*expression*	**id num (not + -**		**; end else do then]) ,**
22.1.	*related_expression*	**relop**		**; end else do then]) ,**
22.2.	*related_expression*	ϵ	→	**; end else do then]) ,**
23.1.1.	*simple_expression*	**id num (not**		**relop ; end else do then]) ,**
23.1.2.	*simple_expression*	**+ -**		**relop ; end else do then]) ,**
23.2.1.	*simple_expression′*	**addop**		**relop ; end else do then]) ,**
23.2.2.	*simple_expression′*	ϵ	→	**relop ; end else do then]) ,**
24.1.	*term*	**id num (not**		**addop relop ; end else do then]) ,**
24.2.1.	*term′*	**mulop**		**addop relop ; end else do then]) ,**
24.2.2.	*term′*	ϵ	→	**addop relop ; end else do then]) ,**
25.1.1.	*factor*	**id**		**mulop addop relop ; end else do then]) ,**
25.1.2.	*factor*	**num**		**mulop addop relop ; end else do then]) ,**
25.1.3.	*factor*	**(**		**mulop addop relop ; end else do then]) ,**
25.1.4.	*factor*	**not**		**mulop addop relop ; end else do then]) ,**
25.2.1.	*factor′*	**[**		**mulop addop relop ; end else do then]) ,**
25.2.1.	*factor′*	ϵ	→	**mulop addop relop ; end else do then]) ,**
26.1.	*sign*	**+**		**id num not (**
26.2.	*sign*	**-**		**id num not (**

Variable/**Terminal**	(	)	+	,	-	;	[	]	addop	array	assignop	begin	call	do	else	end	id	if	integer	mulop	not	num	procedure	program	real	relop	then	var	while
program																								1					
identifier_list																	2.1												
identifier_list′		2.2.2		2.2.1																									
declarations												3.2											3.2					3.1	
type										4.2									4.1						4.1				
standard_type																			5.1						5.2				
subprogram_declarations												6.2											6.1						
subprogram_declaration																							7						
subprogram_head																							8						
arguments	9.1					9.2																							
parameter_list																	10.1												
parameter_list′		10.2.2				10.2.1																							
compound_statement												11																	
optional_statements												12.1	12.1			12.2	12.1	12.1											12.1
statement_list												13.1	13.1				13.1	13.1											13.1
statement_list′						13.2.1										13.2.2													
statement												14.3	14.2				14.1	14.5											14.4
else′						15.2									15.1	15.2													
variable																	16												
array_access							17.1				17.2																		
procedure_statement													18																
optional_expressions	19.1					19.2									19.2	19.2													
expression_list	20.1		20.1		20.1												20.1				20.1	20.1							
expression_list′		20.2.2		20.2.1																									
expression	21		21		21									22.2	22.2	22.2	21				21	21							
related_expression		22.2		22.2		22.2		22.2									23.1.1				23.1.1	23.1.1				22.1	22.2		
simple_expression	23.1.1		23.1.2		23.1.2									23.2.2	23.2.2	23.2.2													
simple_expression′		23.2.2		23.2.2		23.2.2		23.2.2	23.2.1								24.1				24.1	24.1				23.2.2	23.2.2		
term	24.1																												
term′		24.2.2		24.2.2		24.2.2		24.2.2	24.2.2					24.2.2	24.2.2	24.2.2				24.2.1						24.2.2	24.2.2		
factor	25.1.3																25.1.1				25.1.4	25.1.2							
factor′		25.2.2		25.2.2		25.2.2	25.2.1	25.2.2	25.2.2					25.2.2	25.2.2	25.2.2				25.2.2						25.2.2	25.2.2		
sign			26.1		26.2																								

2 Implementation

Each production resides in its own file, where it's first and synch sets are specified. The "program" production is called when the compiler begins, and it requests the next tokens. From there, productions are called in accordance to the grammar rules and the tokens receieved from the lexical analyzer.

The "match" function matched the token and set a global variable named "current_tok" to the next token. If the match failed, an syntax error was printed to the listing file. Another operation called "require_synch" was included, in the event that no token could identify the current production in a grammar variable. *require_synch* would take in the firsts and synch sets of the grammar variables, print the appropriate error message ("recieved X; expected Y, Z"), then discard tokens until an item from the synch set were matched.

The productions mapped $1:1$ from terminal symbols to match and from grammar variables to productions. Which production to use in which grammar variable was determined by the value of *current_tok* (hence the $LL(1)$ property of our grammar).

If any errors were encountered while parsing, the error is added to the error queue. Then, the error is printed before the next token is collected.

3 Discussion & Conclusions

Implementing this project definitely taught me about the importance of an $LL(1)$ grammar, and how neat the recursive descent parser is. I also noticed that the *require_synch*() function would have been wonderful as a dynamically-scoped function.

I wrote this compiler in C, with no external code of any kind. It was compiled with clang on macOS Sierra.

Appendix 1: Sample Inputs and Outputs

3.1 Error-Filled Source File

Listing 1: Error-Full Source Code

```
1  program fib(input; output):
2    var a: int; var p: integer;
3    var numsArray : array [6..12] on integer;
4    var q: real;
5
6    procedure fib1(aReallyLongInt : integer; b : real, c
         : real);
7      begin
8          if a <= 1 then fib := c
9          else call fib (a - 1, c, b + c)
10     end;
11
12   procedure fib2(a : integer);
13     var b : real; var c : real; var sum : ;
14     var b : real;
15     procedure rawr3(b : real);
16       var q : real;
17       begin
18         q := b + 2.0;
19         call fib2(q).
20       end;
21
22     begin
23       a := a - 1;
24       b := call fib1(3);
25       sum := 1;
26       c := b;
27       while (a > 0 do
28         call 3;
29         begin
30           a := a - 1;
31           b := sum;
32           sum := c + sum;
33           c := b
34         end
35       fib2 := sum
36     end;
37
38   procedure init;
39     begin
```

```
40        n := 12;
41        if (1 and 2) or 3 then p := 12
42        else p := 14;
43        numsArray[3] := 15.56;
44        q := q[4];
45        q[4] := 12
46      end;
47
48      begin
49        call init;
50        call fib2;
51        call rawr3(34, 56);
52      end.
```

Listing 2: Error-Full Listing File

```
1         1 program fib(input; output):
2  SYNERR: Found ';'; expected ')', ',' instead.
3  SYNERR: Found ':'; expected ';' instead.
4         2   var a: int; var p: integer;
5  SYNERR: Found 'ID'; expected 'array', 'real', 'integer
       ' instead.
6  SYNERR: Found 'integer'; expected ';' instead.
7  SYNERR: Found ';'; expected 'begin', 'procedure', 'var
       ' instead.
8         3   var numsArray : array [6..12] on integer;
9         4   var q: real;
10        5
11        6   procedure fib1(aReallyLongInt : integer; b :
              real, c : real);
12 LEXERR:                  ID length exceeded 10
       characters:      aReallyLongInt
13 SYNERR: Found ','; expected ')', ';' instead.
14        7     begin
15        8         if a <= 1 then fib := c
16        9         else call fib (a - 1, c, b + c)
17       10     end;
18       11
19       12   procedure fib2(a : integer);
20       13     var b : real; var c : real; var sum : ;
21 SYNERR: Found ';'; expected 'array', 'real', 'integer'
       instead.
22       14     var b : real;
23 SYNERR: Found 'real'; expected ';' instead.
24 SYNERR: Found ';'; expected 'begin', 'procedure', 'var
       ' instead.
```

```
25       15     procedure rawr3(b : real);
26       16       var q : real;
27       17       begin
28       18         q := b + 2.0;
29       19         call fib2(q).
30  SYNERR: Found '.'; expected 'end', ';' instead.
31       20       end;
32       21
33       22     begin
34       23       a := a - 1;
35       24       b := call fib1(3);
36  SYNERR: Found 'call'; expected '-', '+', 'not', '(', '
       a number', 'ID' instead.
37  SYNERR: Found ')'; expected 'end', ';' instead.
38       25       sum := 1;
39       26       c := b;
40       27       while (a > 0 do
41       28         call 3;
42       29         begin
43       30           a := a - 1;
44       31           b := sum;
45       32           sum := c + sum;
46       33           c := b
47       34         end
48       35       fib2 := sum
49  SYNERR: Found 'ID'; expected ';' instead.
50  SYNERR: Found ':='; expected 'begin', 'procedure'
       instead.
51       36     end;
52       37
53       38   procedure init;
54       39     begin
55       40       n := 12;
56       41       if (1 and 2) or 3 then p := 12
57       42       else p := 14;
58       43       numsArray[3] := 15.56;
59       44       q := q[4];
60       45       q[4] := 12
61       46     end;
62  SYNERR: Found ';'; expected '.' instead.
63       47
64       48     begin
65  SYNERR: Found 'begin'; expected 'EOF' instead.
66       49       call init;
```

Listing 3: Error-Full Token File

```
1     FILEEND
2     ASSIGNOP
3     RELOP
4     ID
5     CONTROL
6     ADDOP
7     MULOP
8     WS
9     ARRAY
10    TYPE
11    VAR
12    NUM
13    PUNC
14    GROUP
15    INVERSE
16    LEXERR
17    SYNERR
18    SEMERR
          LineLexeme      Token Attribute    Token Type
    1       program      5     7
    1    fib      4       0x7ff40fc03860
    1      (     14      0
    1  input      4       0x7ff40fc03b70
    1      ;     13      1
    1      ;     17      0
    1output      4       0x7ff40fc03ff0
    1      )     14      1
    1      :     10      0
    1      :     17      0
    2    var     11      0
    2      a      4       0x7ff40fc04790
    2      :     10      0
    2    int      4       0x7ff40fc04a80
    2    int     17      0
    2      ;     13      1
    2    var     11      0
    2      p      4       0x7ff40fc05010
    2      :     10      0
    2       integer     10      1
    2       integer     17      0
    2      ;     13      1
    2      ;     17      0
    3    var     11      0
    3     numsArray      4       0x7ff40fc05c40
```

```
3     :    10     0
3 array     9     0
3     [    14     2
3     6    12     0
3    ..     9     1
3    12    12     0
3     ]    14     3
3    on     4      0x7ff40fc068a0
3      integer    10     1
3     ;    13     1
4   var    11     0
4     q     4      0x7ff40fc07090
4     :    10     0
4  real    10     2
4     ;    13     1
6    procedure     5     6
6  fib1     4      0x7ff40fc07ad0
6     (    14     0
6      aReallyLongInt     4      0x7ff40fc07ee0
6      aReallyLongInt    16     1
6     :    10     0
6      integer    10     1
6     ;    13     1
6     b     4      0x7ff40fc08670
6     :    10     0
6  real    10     2
6     ,    13     0
6     ,    17     0
6     c     4      0x7ff40fc08dd0
6     :    10     0
6  real    10     2
6     )    14     1
6     ;    13     1
7 begin     5     0
8    if     5     5
8     a     4      0x7ff40fc04790
8    <=     3     1
8     1    12     0
8  then     5     8
8   fib     4      0x7ff40fc03860
8    :=     2     0
8     c     4      0x7ff40fc08dd0
8   18     0
9  else     5     2
9  call     5    10
9   fib     4      0x7ff40fc03860
```

```
 9     (     14     0
 9   18      0
 9     a      4      0x7ff40fc04790
 9     -      6     1
 9     1     12     0
 9     ,     13     0
 9     c      4      0x7ff40fc08dd0
 9     ,     13     0
 9     b      4      0x7ff40fc08670
 9     +      6     0
 9     c      4      0x7ff40fc08dd0
 9     )     14     1
10   end      5     3
10     ;     13     1
12    procedure      5     6
12  fib2      4      0x7ff40fc0d120
12     (     14     0
12     a      4      0x7ff40fc04790
12     :     10     0
12      integer     10     1
12     )     14     1
12     ;     13     1
13   var     11     0
13     b      4      0x7ff40fc08670
13     :     10     0
13  real     10     2
13     ;     13     1
13   var     11     0
13     c      4      0x7ff40fc08dd0
13     :     10     0
13  real     10     2
13     ;     13     1
13   var     11     0
13   sum      4      0x7ff40fc0efa0
13     :     10     0
13     ;     13     1
13     ;     17     0
14   var     11     0
14     b      4      0x7ff40fc08670
14     :     10     0
14  real     10     2
14  real     17     0
14     ;     13     1
14     ;     17     0
15    procedure      5     6
15 rawr3      4      0x7ff40fc10730
```

```
137  15      (     14      0
138  15      b      4       0x7ff40fc08670
139  15      :     10      0
140  15   real     10      2
141  15      )     14      1
142  15      ;     13      1
143  16    var     11      0
144  16      q      4       0x7ff40fc07090
145  16      :     10      0
146  16   real     10      2
147  16      ;     13      1
148  17  begin      5      0
149  18      q      4       0x7ff40fc07090
150  18     :=      2      0
151  18      b      4       0x7ff40fc08670
152  18      +      6      0
153  18    2.0     12      1
154  18      ;     13      1
155  19   call      5     10
156  19   fib2      4       0x7ff40fc0d120
157  19      (     14      0
158  19      q      4       0x7ff40fc07090
159  19      )     14      1
160  19    18      0
161  19      .     13      2
162  19      .     17      0
163  20    end      5      3
164  20      ;     13      1
165  22  begin      5      0
166  23      a      4       0x7ff40fc04790
167  23     :=      2      0
168  23      a      4       0x7ff40fc04790
169  23      -      6      1
170  23      1     12      0
171  23      ;     13      1
172  24      b      4       0x7ff40fc08670
173  24     :=      2      0
174  24   call      5     10
175  24   call     17      0
176  24   fib1      4       0x7ff40fc07ad0
177  24      (     14      0
178  24      3     12      0
179  24      )     14      1
180  24      )     17      0
181  24      ;     13      1
182  25    sum      4       0x7ff40fc0efa0
```

```
183   25     :=      2      0
184   25      1     12      0
185   25      ;     13      1
186   26      c      4       0x7ff40fc08dd0
187   26     :=      2      0
188   26      b      4       0x7ff40fc08670
189   26      ;     13      1
190   27  while      5      9
191   27      (     14      0
192   27      a      4       0x7ff40fc04790
193   27      >      3      3
194   27      0     12      0
195   27     do      5      1
196   28   call      5     10
197   28      3     12      0
198   28      ;     13      1
199   29  begin      5      0
200   30      a      4       0x7ff40fc04790
201   30     :=      2      0
202   30      a      4       0x7ff40fc04790
203   30      -      6      1
204   30      1     12      0
205   30      ;     13      1
206   31      b      4       0x7ff40fc08670
207   31     :=      2      0
208   31    sum      4       0x7ff40fc0efa0
209   31      ;     13      1
210   32    sum      4       0x7ff40fc0efa0
211   32     :=      2      0
212   32      c      4       0x7ff40fc08dd0
213   32      +      6      0
214   32    sum      4       0x7ff40fc0efa0
215   32      ;     13      1
216   33      c      4       0x7ff40fc08dd0
217   33     :=      2      0
218   33      b      4       0x7ff40fc08670
219   34    end      5      3
220   35   fib2      4       0x7ff40fc0d120
221   35   fib2     17      0
222   35     :=      2      0
223   35     :=     17      0
224   35    sum      4       0x7ff40fc0efa0
225   36    end      5      3
226   36      ;     13      1
227   38    procedure      5      6
228   38   init      4       0x7ff40fc1e520
```

```
229  38      ;     13      1
230  39  begin      5      0
231  40      n      4       0x7ff40fc1ef80
232  40     :=      2      0
233  40     12     12      0
234  40    18      0
235  40      ;     13      1
236  41     if      5      5
237  41      (     14      0
238  41      1     12      0
239  41    and      7      2
240  41      2     12      0
241  41      )     14      1
242  41    18      0
243  41     or      6      2
244  41      3     12      0
245  41   then      5      8
246  41      p      4       0x7ff40fc05010
247  41     :=      2      0
248  41     12     12      0
249  41    18      0
250  42   else      5      2
251  42      p      4       0x7ff40fc05010
252  42     :=      2      0
253  42     14     12      0
254  42    18      0
255  42      ;     13      1
256  43     numsArray      4       0x7ff40fc05c40
257  43      [     14      2
258  43      3     12      0
259  43      ]     14      3
260  43     :=      2      0
261  43  15.56     12      1
262  43    18      0
263  43      ;     13      1
264  44      q      4       0x7ff40fc07090
265  44     :=      2      0
266  44      q      4       0x7ff40fc07090
267  44    18      0
268  44      [     14      2
269  44      4     12      0
270  44      ]     14      3
271  44      ;     13      1
272  45      q      4       0x7ff40fc07090
273  45      [     14      2
274  45      4     12      0
```

```
 45      ]     14      3
 45     :=      2      0
 45     12     12      0
 45    18      0
 46    end      5      3
 46      ;     13      1
 46      ;     17      0
 48  begin      5      0
 48  begin     17      0
 49   call      5     10
```

3.2 Error-Free Source File

Listing 4: Error-Free Source Code

```
program fib(input, output);
  var a: integer; var p: integer;
  var numsArray : array [6..12] of integer;
  var q: real;

  procedure fib1(a : integer; b : real; c : real);
    begin
        if a <= 1 then fib := c
        else call fib (a - 1, c, b + c)
    end;

  procedure fib2(a : integer);
    var b : real; var c : real; var sum : integer;
    var b : real;
    procedure rawr3(b : real);
      var q : real;
      begin
        q := b + 2.0;
        call fib2(q)
      end;

    begin
      a := a - 1;
      b := 0;
      sum := 1;
      c := b;
      while (a > 0) do
        begin
          a := a - 1;
          b := sum;
```

```
31            sum := c + sum;
32            c := b
33          end;
34        fib2 := sum
35      end;
36
37    procedure init;
38      begin
39        n := 12;
40        if (1 and 2) or 3 then p := 12
41        else p := 14;
42        numsArray[3] := 15.56;
43        q := q[4];
44        q[4] := 12
45      end;
46
47      begin
48        call init;
49        call fib2;
50        call rawr3(34, 56)
51      end.
```

Listing 5: Error-Free Listing File

```
1         1 program fib(input, output);
2         2   var a: integer; var p: integer;
3         3   var numsArray : array [6..12] of integer;
4         4   var q: real;
5         5
6         6   procedure fib1(a : integer; b : real; c :
              real);
7         7     begin
8         8         if a <= 1 then fib := c
9         9         else call fib (a - 1, c, b + c)
10       10     end;
11       11
12       12   procedure fib2(a : integer);
13       13     var b : real; var c : real; var sum :
              integer;
14       14     var b : real;
15       15     procedure rawr3(b : real);
16       16       var q : real;
17       17       begin
18       18         q := b + 2.0;
19       19         call fib2(q)
20       20       end;
```

```
21      21
22      22      begin
23      23        a := a - 1;
24      24        b := 0;
25      25        sum := 1;
26      26        c := b;
27      27        while (a > 0) do
28      28          begin
29      29            a := a - 1;
30      30            b := sum;
31      31            sum := c + sum;
32      32            c := b
33      33          end;
34      34        fib2 := sum
35      35      end;
36      36
37      37    procedure init;
38      38      begin
39      39        n := 12;
40      40        if (1 and 2) or 3 then p := 12
41      41        else p := 14;
42      42        numsArray[3] := 15.56;
43      43        q := q[4];
44      44        q[4] := 12
45      45      end;
46      46
47      47      begin
48      48        call init;
49      49        call fib2;
50      50        call rawr3(34, 56)
51      51      end.
```

Listing 6: Error-Free Token File

```
1   1     FILEEND
2   2     ASSIGNOP
3   3     RELOP
4   4     ID
5   5     CONTROL
6   6     ADDOP
7   7     MULOP
8   8     WS
9   9     ARRAY
10  10    TYPE
11  11    VAR
12  12    NUM
```

```
13  13   PUNC
14  14   GROUP
15  15   INVERSE
16  16   LEXERR
17  17   SYNERR
18  18   SEMERR
19         LineLexeme      Token Attribute   Token Type
20    1       program      5     7
21    1    fib      4      0x7fab12d03700
22    1      (     14     0
23    1 input      4      0x7fab12d03a10
24    1      ,     13     0
25    1output      4      0x7fab12d03d90
26    1      )     14     1
27    1      ;     13     1
28    2    var     11     0
29    2      a      4      0x7fab12d04460
30    2      :     10     0
31    2       integer     10     1
32    2      ;     13     1
33    2    var     11     0
34    2      p      4      0x7fab12d04c60
35    2      :     10     0
36    2       integer     10     1
37    2      ;     13     1
38    3    var     11     0
39    3     numsArray      4      0x7fab12d05680
40    3      :     10     0
41    3  array      9      0
42    3      [     14     2
43    3      6     12     0
44    3     ..      9     1
45    3     12     12     0
46    3      ]     14     3
47    3     of      9     2
48    3       integer     10     1
49    3      ;     13     1
50    4    var     11     0
51    4      q      4      0x7fab12d06ae0
52    4      :     10     0
53    4   real     10     2
54    4      ;     13     1
55    6     procedure      5     6
56    6   fib1      4      0x7fab12d07550
57    6      (     14     0
58    6      a      4      0x7fab12d04460
```

```
 6     :    10     0
 6      integer    10     1
 6     ;    13     1
 6     b     4      0x7fab12d07e60
 6     :    10     0
 6  real    10     2
 6     ;    13     1
 6     c     4      0x7fab12d084c0
 6     :    10     0
 6  real    10     2
 6     )    14     1
 6     ;    13     1
 7 begin     5     0
 8    if     5     5
 8     a     4      0x7fab12d04460
 8    <=     3     1
 8     1    12     0
 8  then     5     8
 8   fib     4      0x7fab12d03700
 8    :=     2     0
 8     c     4      0x7fab12d084c0
 8   18     0
 9  else     5     2
 9  call     5    10
 9   fib     4      0x7fab12d03700
 9     (    14     0
 9   18     0
 9     a     4      0x7fab12d04460
 9     -     6     1
 9     1    12     0
 9     ,    13     0
 9     c     4      0x7fab12d084c0
 9     ,    13     0
 9     b     4      0x7fab12d07e60
 9     +     6     0
 9     c     4      0x7fab12d084c0
 9     )    14     1
10   end     5     3
10     ;    13     1
12    procedure     5     6
12  fib2     4      0x7fab12d0c840
12     (    14     0
12     a     4      0x7fab12d04460
12     :    10     0
12      integer    10     1
12     )    14     1
```

```
105  12      ;     13      1
106  13    var     11      0
107  13      b      4       0x7fab12d07e60
108  13      :     10      0
109  13   real     10      2
110  13      ;     13      1
111  13    var     11      0
112  13      c      4       0x7fab12d084c0
113  13      :     10      0
114  13   real     10      2
115  13      ;     13      1
116  13    var     11      0
117  13    sum      4       0x7fab12d0e6c0
118  13      :     10      0
119  13       integer     10      1
120  13      ;     13      1
121  14    var     11      0
122  14      b      4       0x7fab12d07e60
123  14      :     10      0
124  14   real     10      2
125  14      ;     13      1
126  14    18      0
127  15     procedure      5      6
128  15 rawr3      4       0x7fab12d0fe30
129  15      (     14      0
130  15      b      4       0x7fab12d07e60
131  15      :     10      0
132  15   real     10      2
133  15      )     14      1
134  15      ;     13      1
135  16    var     11      0
136  16      q      4       0x7fab12d06ae0
137  16      :     10      0
138  16   real     10      2
139  16      ;     13      1
140  17 begin      5      0
141  18      q      4       0x7fab12d06ae0
142  18     :=      2      0
143  18      b      4       0x7fab12d07e60
144  18      +      6      0
145  18    2.0     12      1
146  18      ;     13      1
147  19   call      5     10
148  19   fib2      4       0x7fab12d0c840
149  19      (     14      0
150  19      q      4       0x7fab12d06ae0
```

```
151   19      )     14      1
152   19   18       0
153   20   end       5      3
154   20     ;      13      1
155   22 begin       5      0
156   23     a       4       0x7fab12d04460
157   23    :=       2      0
158   23     a       4       0x7fab12d04460
159   23     -       6      1
160   23     1      12      0
161   23     ;      13      1
162   24     b       4       0x7fab12d07e60
163   24    :=       2      0
164   24     0      12      0
165   24     ;      13      1
166   24   18       0
167   25   sum       4       0x7fab12d0e6c0
168   25    :=       2      0
169   25     1      12      0
170   25     ;      13      1
171   26     c       4       0x7fab12d084c0
172   26    :=       2      0
173   26     b       4       0x7fab12d07e60
174   26     ;      13      1
175   27 while       5      9
176   27     (      14      0
177   27     a       4       0x7fab12d04460
178   27     >       3      3
179   27     0      12      0
180   27     )      14      1
181   27    do       5      1
182   28 begin       5      0
183   29     a       4       0x7fab12d04460
184   29    :=       2      0
185   29     a       4       0x7fab12d04460
186   29     -       6      1
187   29     1      12      0
188   29     ;      13      1
189   30     b       4       0x7fab12d07e60
190   30    :=       2      0
191   30   sum       4       0x7fab12d0e6c0
192   30     ;      13      1
193   30   18       0
194   31   sum       4       0x7fab12d0e6c0
195   31    :=       2      0
196   31     c       4       0x7fab12d084c0
```

```
31      +       6       0
31    sum       4        0x7fab12d0e6c0
31      ;     13       1
31    18      0
32      c       4        0x7fab12d084c0
32     :=       2       0
32      b       4        0x7fab12d07e60
33    end       5       3
33      ;     13       1
34   fib2       4        0x7fab12d0c840
34     :=       2       0
34    sum       4        0x7fab12d0e6c0
34    18      0
35    end       5       3
35      ;     13       1
37     procedure       5       6
37   init       4        0x7fab12d1ce40
37      ;     13       1
38  begin       5       0
39      n       4        0x7fab12d1d8f0
39     :=       2       0
39     12     12       0
39    18      0
39      ;     13       1
40     if       5       5
40      (     14       0
40      1     12       0
40    and       7       2
40      2     12       0
40      )     14       1
40    18      0
40     or       6       2
40      3     12       0
40   then       5       8
40      p       4        0x7fab12d04c60
40     :=       2       0
40     12     12       0
41   else       5       2
41      p       4        0x7fab12d04c60
41     :=       2       0
41     14     12       0
41      ;     13       1
42     numsArray       4       0x7fab12d05680
42      [     14       2
42      3     12       0
42      ]     14       3
```

```
243  42     :=      2      0
244  42  15.56     12      1
245  42      ;     13      1
246  42    18      0
247  43      q      4       0x7fab12d06ae0
248  43     :=      2      0
249  43      q      4       0x7fab12d06ae0
250  43      [     14      2
251  43      4     12      0
252  43      ]     14      3
253  43      ;     13      1
254  43    18      0
255  44      q      4       0x7fab12d06ae0
256  44      [     14      2
257  44      4     12      0
258  44      ]     14      3
259  44     :=      2      0
260  44    18      0
261  44     12     12      0
262  45    end      5      3
263  45      ;     13      1
264  47  begin      5      0
265  48   call      5     10
266  48   init      4       0x7fab12d1ce40
267  48      ;     13      1
268  49   call      5     10
269  49   fib2      4       0x7fab12d0c840
270  49      ;     13      1
271  49    18      0
272  50   call      5     10
273  50  rawr3      4       0x7fab12d0fe30
274  50      (     14      0
275  50    18      0
276  50     34     12      0
277  50      ,     13      0
278  50     56     12      0
279  50      )     14      1
280  51    end      5      3
281  51      .     13      2
282  52    EOF      1      0
```

Appendix 2: Program Listings

Listing 7: compiler.c

```
#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>

#include "dataStructures/linkedList/linkedlist.h"
#include "errorHandler/errorHandler.h"
#include "globals/globals.h"
#include "handler/handler.h"
#include "parser/parser.h"

// Global file constants
static const char TOKEN_PATH[] = "out/tokens.dat";
static const char LISTING_PATH[] = "out/listing.txt";
static const char MEM_PATH[] = "out/mem.txt";
static const char RESWORD_PATH[] = "compiler/data/reswords.dat";

// Returns 1 on failure, 0 on success.
int init(char* sourcePath) {
return initializeGlobals() && initializeErrorHandler() &&
initializeHandler(sourcePath, RESWORD_PATH, LISTING_PATH, TOKEN_PATH, MEM_PATH)
? 0 : 1;
}

int run()
{
generateParseTree();

return 0;
}

int main(int argc, char *argv[]) {
if (argc != 2) {
fprintf(stderr, "%s\n", "Expected exactly one file to compile!");
} else {
if (init(argv[1]) == 0) {
if (run() != 0)
fprintf(stderr, "%s\n", "Run failed. Could not terminate properly.");
} else {
fprintf(stderr, "%s\n", "Initialization process failed in tokenizer.");
}
}
return 0;
}
```

Listing 8: declarationsTree.h

```
#ifndef DECLARATIONS_TREE_H
#define DECLARATIONS_TREE_H

#include <stdbool.h>

#include "../../tokenizer/tokens.h"

typedef struct tree_node {
char* lex; // The lexeme
LangType type; // The type
union {
bool param; // True if param
bool add_right; // True if add right to green node
};

struct tree_node* left;
struct tree_node* right;
struct tree_node* parent;
} tree_node;

```

```
typedef struct LinkedTree {
struct node* head;
} DeclarationsTree;

// Green nodes designate scopes, and blue nodes designate variables
bool check_add_node(Token* decl);
tree_node* get_last_green_node();
tree_node* start_param_matching(Token* id);
void reached_end_of_scope();
LangType get_type(Token* id);

#endif // DECLARATIONS_TREE_H
```

Listing 9: linkedlist.h

```
#ifndef LINKED_H_
#define LINKED_H_

// Behaves like a stack
struct node {
void* data;
struct node* next;
};

typedef struct LinkedNodes {
struct node* head;
int size;
} LinkedList;

// Add an item to the front of the linked list
int add(LinkedList* list, void* data, int size);

// Pop an item from the front of the linked list
void* pop(LinkedList* list);

#endif // LINKED_H_
```

Listing 10: errorHandler.h

```
#ifndef ERROR_HANDLER_H
#define ERROR_HANDLER_H
#include "../tokenizer/tokens.h"

extern const char* lexErrs[];
char* synErr;
char* semErr;

void throw_sem_error(char* msg);
void throw_syn_error(Token* received, const Token** expected, int exp_size);
void throw_lex_error(enum TokenType attribute, int aspect, int start, int length);
int initializeErrorHandler();

Token* getNextErrorToken();

#endif // ERROR_HANDLER_H
```

Listing 11: globals.h

```
#ifndef GLOBALS_H
#define GLOBALS_H

extern int START;
extern int LINE;
extern char* BUFFER;

void updateLine(char* line);
int initializeGlobals();

#endif // GLOBALS_H
```

Listing 12: handler.h

```
#ifndef HANDLER_H
#define HANDLER_H

#include<stdbool.h>
```

```
#include "../tokenizer/tokens.h"

int initializeHandler(const char* sourcePath, const char* resPath,
const char* listingPath, const char* tokenPath,
const char* memPath);
bool handleToken(Token* token);
void outputWidth(char* lex, int width);

#endif // HANDLER_H
```

Listing 13: parser.h

```
#ifndef PARSER_H
#define PARSER_H
#include<stdbool.h>

int generateParseTree();
Token* match(const Token* source, bool strict);
void require_sync(const Token* sync_set[], int size,
const Token* first_set[], int first_size);

#endif // PARSER_H
```

Listing 14: productions.h

```
#ifndef voidS_H
#define voidS_H
#include <stdio.h>
#include <stdlib.h>

#include "../../globals/globals.h"
#include "../../tokenizer/tokens.h"
#include "../../dataStructures/declarationsTree/declarationsTree.h"
#include "../../errorHandler/errorHandler.h"

extern Token* current_tok;

// All of these must have their follows added to the sync set
void program();
void id_list();
void id_list_tail();
void declarations();
LangType type();
LangType standard_type();
void subprogram_declarations();
void subprogram_declaration();
bool subprogram_head();
void arguments();
void parameter_list();
void parameter_list_tail();
void compound_statement();
void optional_statements();
void statement_list();
void statement_list_tail();
void statement();
void else_tail();
LangType variable();
LangType array_access(LangType id_type);
void procedure_statement();
void optional_expressions(tree_node* to_match, bool should_error);
void expression_list(tree_node* to_match, bool should_error);
void expression_list_tail(tree_node* to_match, bool should_error);
LangType expression();
LangType related_expression();
LangType simple_expression();
LangType simple_expression_tail();
LangType term();
```

```
LangType term_tail();
LangType factor();
LangType factor_tail();
void sign();

#endif // voidS_H
```

Listing 15: symbolTable.h

```
#ifndef SYMBOL_TABLE_H
#define SYMBOL_TABLE_H

int initSymbolTable();
char* checkSymbolTable(char*
    name);
char* pushToSymbolTable(char*
    name, size_t length);

#endif // SYMBOL_TABLE_H
```

Listing 16: machines.h

```
#ifndef MACHINES_H
#define MACHINES_H
#include <stdio.h>
#include "../tokens.h"

typedef int (*machine)(Token*,
     char*, int);

int intMachine(Token* storage,
     char* str, int start);
int realMachine(Token* storage
    , char* str, int start);
int longRealMachine(Token*
    storage, char* str, int
    start);
int grouping(Token* storage,
    char* str, int start);
int catchall(Token* storage,
    char* str, int start);
int mulop(Token* storage, char
    * str, int start);
int addop(Token* storage, char
    * str, int start);
int whitespace(Token* storage,
     char* str, int start);
int relop(Token* storage, char
    * str, int start);

int idres(Token* storage, char
    * str, int start);
int initIDResMachine(FILE*
    resFile);

extern const machine machines
    [];
#endif // MACHINES_H
```

Listing 17: tokenizer.h

```
#ifndef PROCESSOR_H_
#define PROCESSOR_H_
#include<stdio.h>
#include "tokens.h"

Token* getNextToken();
int initializeTokens(FILE*
    resFile);

#endif // PROCESSOR_H_
```

Listing 18: tokens.h

```
#ifndef TOKENS_H
#define TOKENS_H

#include<stdbool.h>

#include "../dataStructures/
    linkedList/linkedList.h"

// Must have a boolean
    indicating whether it is a
    parameter or not
typedef enum LangType {ERR,
    REAL, INT, BOOL, PGNAME,
    PPNAME,
 PROC, AINT, AREAL} LangType;

enum TokenType {NOOP, FILEEND,
     ASSIGNOP, RELOP, ID,
CONTROL, ADDOP, MULOP, WS,
    ARRAY, TYPE,
```

```
14 VAR, NUM, PUNC, GROUP, INVERSE
       ,
15 LEXERR, SYNERR, SEMERR};
16
17 // The token data type
18 typedef struct T_Type {
19 enum TokenType attribute; //
       Attribute
20
21 union { // Aspect or character
        pointer
22 int aspect;
23 char* id;
24 };
25
26 int start; // Start in the
       line
27 int length; // Length of the
       lexeme
28
29 union { // Value of the number
       , or length of the array
30 int int_val;
31 double real_val;
32 int array_length;
33 };
34
35 LangType type; // The type of
       the token
36 bool param; // Whether the
       token is a parameter or not
37
38 } Token;
39
40 extern const Token eof_tok;
41 extern const Token lparen_tok;
42 extern const Token rparen_tok;
43 extern const Token plus_tok;
44 extern const Token comma_tok;
45 extern const Token minus_tok;
46 extern const Token semic_tok;
47 extern const Token colon_tok;
48 extern const Token period_tok;
49 extern const Token dotdot_tok;
50 extern const Token lbrac_tok;
51 extern const Token rbrac_tok;
52 extern const Token addop_tok;
53 extern const Token array_tok;
54 extern const Token
       assignop_tok;
55 extern const Token begin_tok;
56 extern const Token call_tok;
57 extern const Token do_tok;
58 extern const Token else_tok;
59 extern const Token end_tok;
60 extern const Token id_tok;
61 extern const Token if_tok;
62 extern const Token integer_tok
       ;
63 extern const Token
       integer_val_tok;
64 extern const Token of_tok;
65 extern const Token
       real_val_tok;
66 extern const Token mulop_tok;
67 extern const Token not_tok;
68 extern const Token num_tok;
69 extern const Token
       procedure_tok;
70 extern const Token program_tok
       ;
71 extern const Token real_tok;
72 extern const Token relop_tok;
73 extern const Token then_tok;
74 extern const Token var_tok;
75 extern const Token while_tok;
76
77 extern const char* catNames
       [19];
78 extern const char* typeNames
       [9];
79 const Token* getTokenFromLex(
       char* lex);
80 const char* getLexFromToken(
       Token* token, bool strict);
81
82 // The type; else null if
       impossible
83 LangType convert_to_array(
       LangType type);
84 LangType convert_from_array(
       LangType type);
```

```
85
86  // Returns the type produced
        by the operation
87  LangType type_lookup(LangType
        first, LangType second,
        Token* op);
88
89  // Returns true if the tokens
        are equivalent, false
        otherwise
90  bool tokens_equal(const Token*
         p1, Token* p2, bool strict
        );
91
92
93  #endif // TOKENS_H
```

Listing 19: declarationsTree.c

```
1   #include <stdlib.h>
2   #include <stdio.h>
3
4   #include "../../handler/
        handler.h"
5   #include "../../errorHandler/
        errorHandler.h"
6   #include "../../globals/
        globals.h"
7   #include "declarationsTree.h"
8
9   static int offset = 0;
10
11  static DeclarationsTree*
        d_tree = NULL;
12  static tree_node* bottom_node
        = NULL;
13
14  static LinkedList*
        green_node_stack = NULL;
15
16  static void initialize_d_tree
        () {
17  d_tree = malloc(sizeof(*d_tree
        ));
18  green_node_stack = malloc(
        sizeof(*green_node_stack));
19  }
20
21  static int get_width(Token*
        val) {
22  switch (val -> type) {
23  case INT: return 4;
24  case REAL: return 8;
25
26  case AINT: return 4*(val ->
        array_length);
27  case AREAL: return 8*(val ->
        array_length);
28
29  default: return 1000000;
30  }
31  }
32
33  static bool check_node(char*
        id, bool green) {
34  tree_node* current_node =
        bottom_node;
35  while (current_node != NULL) {
36  // Already exists
37  if (id == current_node -> lex)
38  return true;
39
40  if (!green && (current_node ->
         type == PROC ||
        current_node -> type ==
        PGNAME))
41  return false;
42
43  // We've passed the most
        recent green node, and this
         is a blue one
44  if (!green && (current_node ->
         type == PROC ||
        current_node -> type ==
        PGNAME))
45  break;
46
47  current_node = current_node ->
         parent;
48  }
49
50  return false;
```

```
}

static bool check_add_green_node(Token* decl) {
if (d_tree == NULL)
initialize_d_tree();

if (bottom_node == NULL)
{
tree_node* addition = malloc(sizeof(*addition));
addition -> lex = decl -> id;
addition -> type = decl -> type;
addition -> add_right = false;

// Add it to the top of the stack
add(green_node_stack, &addition, sizeof(&addition));

addition -> left = NULL;
addition -> right = NULL;
addition -> parent = NULL;
bottom_node = addition;
return true;
}

// Check if it's been declared at all
if (check_node(decl -> id, true))
return false;

offset = 0;

// It hasn't been declared; create it
tree_node* addition = malloc(sizeof(*addition));
addition -> lex = decl -> id;
addition -> type = decl -> type;
addition -> add_right = false;

// Add it to the top of the stack
add(green_node_stack, &addition, sizeof(&addition));

addition -> left = NULL;
addition -> right = NULL;
addition -> parent = bottom_node;

// Make it the new bottom node
if (bottom_node -> add_right == true)
bottom_node -> right = addition;
else
bottom_node -> left = addition;

bottom_node = addition;

return true;
}


static bool check_add_blue_node(Token* decl) {
// If there's no scope, that's an error!
if (d_tree == NULL)
return false;

// It's been declared in the scope already
if (check_node(decl -> id, false))
return false;

// It hasn't been declared; create it
tree_node* addition = malloc(sizeof(*addition));
addition -> lex = decl -> id;
```

```
117 //printf("%s\n", addition ->
        lex);
118 addition -> type = decl ->
        type;
119 addition -> param = decl ->
        param;
120
121 if (!addition -> param)
122 {
123 outputWidth(addition -> lex,
        offset);
124 offset += get_width(decl);
125 }
126
127 addition -> left = NULL;
128 addition -> right = NULL;
129 addition -> parent =
        bottom_node;
130
131 bottom_node -> left = addition
        ;
132 bottom_node = addition;
133
134 //printf("(%s, %s)\n",
        bottom_node -> lex,
        bottom_node -> parent ->
        lex);
135 return true;
136 }
137
138 tree_node*
        start_param_matching(Token*
         id) {
139 tree_node* current_node =
        bottom_node;
140 while (current_node != NULL)
141 {
142 if (current_node -> type ==
        PROC && current_node -> lex
         == id -> id)
143 return current_node;
144 current_node = current_node ->
         parent;
145 }
146
147 return NULL;
148 }
149
150 bool check_add_node(Token*
        decl) {
151 char* errorMessage ;
152 switch (decl -> type) {
153 case PGNAME:
154 case PROC: if (!
        check_add_green_node(decl))
         {
155  errorMessage = calloc(100,
         sizeof(*errorMessage));
156  sprintf(errorMessage,
157 "A program or procedure named
        '%.*s' is already defined
        in this scope!",
158 decl -> length, &BUFFER[decl
        -> start]);
159 throw_sem_error(errorMessage);
160 return false;
161  }
162  return true;
163
164 default: if (!
        check_add_blue_node(decl))
        {
165  errorMessage = calloc(100,
         sizeof(*errorMessage));
166  sprintf(errorMessage,
167 "A variable named '%.*s' is
        already defined in the
        local scope!",
168 decl -> length, &BUFFER[decl
        -> start]);
169 throw_sem_error(errorMessage);
170 return false;
171  }
172  return true;
173 }
174 }
175
176 void reached_end_of_scope() {
177 bottom_node = (*(tree_node**)
        pop(green_node_stack));
178 bottom_node -> add_right =
        true;
```

```
}


LangType get_type(Token* id) {
if (id == NULL)
return ERR;

tree_node* current_node =
    bottom_node;
while (current_node != NULL)
{
if (current_node -> lex == id
    -> id)
return current_node -> type;

current_node = current_node ->
     parent;
}

return NULL;
}
```

Listing 20: linkedlist.c

```
#include<stdlib.h>
#include "linkedlist.h"


int add(LinkedList* list, void
     *data, int size)
{
struct node* addition = malloc
    (sizeof(*addition));
addition -> data = malloc(size
    );
addition -> next = (list ->
    head);
// Do a byte-by-byte copy of
    the data
for (int i = 0; i < size; i++)
*(char *) (addition -> data +
    i) = *(char *) (data + i);
list -> size++;

list -> head = addition;

return list -> size;
}

void* pop(LinkedList* list)
{
struct node* head = list ->
    head;
struct node* next = head ->
    next;

void* data = head -> data;
list -> head = next;
list -> size--;

//free(head); // TODO this is
    necessary; should fix
return data;
}
```

Listing 21: errorHandler.c

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


#include "errorHandler.h"

static LinkedList* errorList;

const char* lexErrs[] = {"
    Unrecognized symbol:",
 "ID length exceeded 10
     characters:",
 "Int length exceeded 10
     characters:",
 "Integer part of real exceeded
      5 characters:",
 "Fractional part of real
     exceeded 5 characters:",
 "Exponent part of long real
     exceeded 2 characters:",
 "Missing exponent part of long
      real:",
 "Leading 0 in int:",
 "Excessive leading 0 in real:"
     ,
```

```
 "Trailing 0 in real:",
 "Leading 0 in exponent:",
 "Attempt to use real exponent: "};

char* synErr;
char* semErr;



int initializeErrorHandler()
{
errorList = malloc(sizeof(*errorList));
return errorList != NULL;
}

void throw_syn_error(Token* received, const Token** expected, int exp_size)
{
// Generate token
Token* errToken = malloc(sizeof(*errToken));
errToken -> attribute = SYNERR;
errToken -> aspect = 0;
errToken -> start = received -> start;
errToken -> length = received -> length;

add(errorList, errToken, sizeof(*errToken));

// Generate error message
// Calculate space needed
int size = strlen("Found ’’; expected ");
size += strlen(getLexFromToken(received, true));
for (int i = exp_size - 1; i >= 0; i--) {
size += strlen("’’");
size += strlen(getLexFromToken(expected[i], expected[i] -> start));
if (i > 0)
size += strlen(", ");
}
size += strlen(" instead.");
size += 1; // Null terminator

synErr = malloc(sizeof(*synErr) * size);
synErr[size - 1] = ’\0’;
strcpy(synErr, "Found ’");
int current = 7;
int len = strlen(getLexFromToken(received, true));
strcpy(&synErr[current], getLexFromToken(received, true));
current += len;
strcpy(&synErr[current], "’ expected ");
current += 12;
for (int i = exp_size - 1; i >= 0; i--) {
strcpy(&synErr[current], "’");
current += 1;
len = strlen(getLexFromToken(expected[i], expected[i] -> start));
strcpy(&synErr[current], getLexFromToken(expected[i], expected[i] -> start));
current += len;
strcpy(&synErr[current], "’");
current += 1;
if (i > 0) {
strcpy(&synErr[current], ", ");
current += 2;
}
}
strcpy(&synErr[current], " instead.");
}

void throw_sem_error(char* msg
```

```
    ) {
// Generate error token
Token* errToken = malloc(
    sizeof(*errToken));
errToken -> attribute = SEMERR
    ;
errToken -> aspect = 0;
errToken -> start = 0;
errToken -> length = 0;

add(errorList, errToken,
    sizeof(*errToken));

// Set the msg
semErr = msg;
}

void throw_lex_error(enum
    TokenType attribute, int
    aspect, int start, int
    length)
{
Token* errToken = malloc(
    sizeof(*errToken));
errToken -> attribute =
    attribute;
errToken -> aspect = aspect;
errToken -> start = start;
errToken -> length = length;

add(errorList, errToken,
    sizeof(*errToken));
}

Token* getNextErrorToken()
{
if (errorList -> size > 0)
return (Token *) pop(errorList
    );

return NULL;
}
```

Listing 22: globals.c

```
#include<string.h>
#include<stdlib.h>
#include<stdbool.h>
#include<stdio.h>
#include "globals.h"

char* BUFFER;
int LINE = 0;
int START = 0;

int initializeGlobals()
{
BUFFER = malloc(sizeof(char*)
    *73);
return (BUFFER != NULL);
}

void updateLine(char* line)
{
START = 0;
LINE++;
strcpy(BUFFER, line);
}
```

Listing 23: handler.c

```
#include<stdio.h>

#include "handler.h"
#include "../globals/globals.h
    "
#include "../tokenizer/
    tokenizer.h"
#include "../errorHandler/
    errorHandler.h"

static FILE* listingFile;
static FILE* tokenFile;
static FILE* sourceFile;
static FILE* memFile;

static const int
    TokenLineSpace = 10;
static const int
    TokenTypeSpace = 20;
static const int
    TokenAttrSpace = 20;
```

```
16 static const intTokenLexSpace
       = 20;
17
18 static const int
       ListingLineSpace = 7;
19 static const int
       ListingErrSpace = 50;
20 static const int
       ListingLexSpace = 20;
21
22 static const int MemNameSpace
       = 10;
23 static const int MemValSpace =
        20;
24
25 void writeEOFToken()
26 {
27 fprintf(tokenFile, "%*d%*.*s%*
       d%*d\n", TokenLineSpace,
       LINE, TokenLexSpace,
28 3, "EOF", TokenTypeSpace,
       FILEEND, TokenAttrSpace, 0)
       ;
29 }
30
31 int initializeHandler(const
       char* sourcePath, const
       char* resPath,
32 const char* listingPath, const
        char* tokenPath,
33 const char* memPath)
34 {
35 if ((sourceFile = fopen(
       sourcePath, "r")) == NULL)
36 {
37 fprintf(stderr, "%s\n", "
       Source was null?");
38 return 0;
39 }
40
41 FILE* resFile = fopen(resPath,
        "r");
42 initializeTokens(resFile);
43 fclose(resFile);
44
45 if ((listingFile = fopen(
       listingPath, "w+")) == NULL
        ||
46 (tokenFile = fopen(tokenPath,
       "w+")) == NULL ||
47 (memFile = fopen(memPath, "w+"
       )) == NULL)
48 return 0;
49
50 for (size_t i = FILEEND; i <=
       SEMERR; i++) {
51 fprintf(tokenFile, "%-5zu%s\n"
       , i, catNames[i]);
52 }
53
54
55 char line[72];
56 if (fgets(line, sizeof(line),
       sourceFile) != NULL)
57 {
58 updateLine(line);
59 fprintf(listingFile, "%*d\t%s"
       , ListingLineSpace, LINE,
       line);
60 } else {
61 writeEOFToken();
62 }
63
64 fprintf(tokenFile, "%*s%*s%*s
       %*s\n", TokenLineSpace, "
       Line",
65 TokenLexSpace, "Lexeme",
66 TokenAttrSpace, "Token
       Attribute",
67 TokenTypeSpace, "Token Type");
68
69 fprintf(memFile, "%*s%*s\n",
       MemNameSpace, "ID",
70 MemValSpace, "Memory Offset");
71 return 1;
72 }
73
74 void outputWidth(char* lex,
       int width) {
75 fprintf(memFile, "%*s%*d\n",
       MemNameSpace, lex,
       MemValSpace, width);
```

```
}

void writeError(Token* description)
{
fprintf(tokenFile, "%*d%*.*s%*d%*d\n", TokenLineSpace, LINE,
TokenLexSpace, description -> length, &BUFFER[description -> start],
TokenTypeSpace, description -> attribute, TokenAttrSpace,
description -> aspect);
if (description -> attribute == LEXERR)
fprintf(listingFile, "%*s:%*s%*.*s\n", ListingLineSpace - 1,
catNames[description -> attribute], ListingErrSpace,
lexErrs[description -> aspect], ListingLexSpace,
description -> length, &BUFFER[description -> start]);
else if (description -> attribute == SYNERR)
fprintf(listingFile, "%*s: %s\n", ListingLineSpace - 1,
catNames[description -> attribute], synErr);
else if (description -> attribute == SEMERR)
fprintf(listingFile, "%*s: %s\n", ListingLineSpace - 1,
catNames[description -> attribute], semErr);
}

void writeToken(Token* token)
{
// Don't bother including in the output file.
if (token -> attribute == WS || token -> attribute == NOOP)
return;

if (token -> attribute >= LEXERR)
{
writeError(token);
return;
}


fprintf(tokenFile, "%*d%*.*s%*d", TokenLineSpace, LINE, TokenLexSpace,
token -> length, &BUFFER[token -> start], TokenTypeSpace,
token -> attribute);
switch (token -> attribute) {
case ID:
fprintf(tokenFile, "%*p", TokenAttrSpace, token -> id);
break;

default:
fprintf(tokenFile, "%*d", TokenAttrSpace, token -> aspect);
break;
}
fprintf(tokenFile, "\n");
}

bool handleToken(Token* token)
{
writeToken(token);
if (token -> attribute == WS && token -> aspect == 1) // A newline
{
char line[72];
if (fgets(line, sizeof(line), sourceFile) != NULL)
{
updateLine(line);
fprintf(listingFile, "%*d\t%s"
```

```
    , ListingLineSpace, LINE,
    line);
} else { // Error or end of
    file (assume the latter)
LINE++;
writeEOFToken();
return false; // Terminate
}
}
return true; // Continue
}
```

Listing 24: parser.c

```
#include<stdlib.h>
#include<stdbool.h>

#include "../tokenizer/tokens.
    h"
#include "productions/
    productions.h"
#include "../tokenizer/
    tokenizer.h"
#include "../handler/handler.h
    "
#include "../errorHandler/
    errorHandler.h"

Token* current_tok = NULL;

static bool sequence_running =
     true;

Token* get_next_relevant_token
    ()
{
const Token* next = malloc(
    sizeof(*next));
if (sequence_running)
{
do {
next = getNextToken();
if (!handleToken(next))
{
sequence_running = false;
next = &eof_tok;
break;
}
} while (next -> attribute ==
    WS || next -> attribute ==
    NOOP
 || next -> attribute >= LEXERR
     );
} else {
next = &eof_tok;
}

return next;
}

void require_sync(const Token*
     sync_set[], int size,
const Token* first_set[], int
    first_size)
{
throw_syn_error(current_tok,
    first_set, first_size);

while (true) {
for (int i = 0; i < size; i++)
if (tokens_equal(sync_set[i],
    current_tok, sync_set[i] ->
     start))
 return;

current_tok =
    get_next_relevant_token();
}
}

// Attempts to match the
    source token with the
    current token;
// if it is found, it returns
    the matched token (for use
    in the RDP).
// If it is not found, then
    match returns null.
Token* match(const Token*
    source, bool strict)
{
if (tokens_equal(source,
```

```
    current_tok, strict))
{
Token* prev_tok = current_tok;
current_tok =
    get_next_relevant_token();
return prev_tok;
}
else
{
throw_syn_error(current_tok, &
    source, 1);
current_tok =
    get_next_relevant_token();
return NULL;
}
}

bool generateParseTree()
{
current_tok = malloc(sizeof(*
    current_tok));
current_tok =
    get_next_relevant_token();
program();
return match(&eof_tok, false);
}
```

Listing 25: arguments.c

```
#include<stdbool.h>
#include<stdlib.h>

#include "productions.h"
#include "../parser.h"
#include "../../tokenizer/
    tokens.h"

static const Token* first_set
    [] = {&lparen_tok, &
    semic_tok};
static const int first_size =
    sizeof(first_set)/sizeof(
    first_set[0]);

static const Token* sync_set[]
     = {&eof_tok, &semic_tok};
static const int sync_size =
    sizeof(sync_set)/sizeof(
    sync_set[0]);

static void synch()
{
require_sync(sync_set,
    sync_size, first_set,
    first_size);
}

// Needs implementing: None
void arguments()
{
// Production 9.1
if (tokens_equal(&lparen_tok,
    current_tok, true))
{
match(&lparen_tok, true);
parameter_list();
match(&rparen_tok, true);
return;

// Production 9.2
} else if (tokens_equal(&
    semic_tok, current_tok,
    true))
return; // Epsilon

synch();
}
```

Listing 26: array$_access.c$

```
#include<stdbool.h>
#include<stdlib.h>

#include "productions.h"
#include "../parser.h"
#include "../../tokenizer/
    tokens.h"

static const Token* first_set
    [] = {&assignop_tok, &
    lbrac_tok};
static const int first_size =
```

```
    sizeof(first_set)/sizeof(
    first_set[0]);

static const Token* sync_set[]
     = {&eof_tok, &assignop_tok
    };
static const int sync_size =
    sizeof(sync_set)/sizeof(
    sync_set[0]);

static void synch()
{
require_sync(sync_set,
    sync_size, first_set,
    first_size);
}

static LangType array_compare(
    LangType a_vals, LangType
    e_type) {
if ((a_vals == INT || a_vals
    == REAL) && e_type == INT)
return a_vals;
if (a_vals != ERR)
{
char* errorMessage = calloc
    (100, sizeof(*errorMessage)
    );
sprintf(errorMessage, "Attempt
     to index variable of type
    %s!", typeNames[a_vals]);
throw_sem_error(errorMessage);
}

return ERR;
}

// Needs implementing: None
LangType array_access(LangType
     id_type)
{
// Production 17.1
if (tokens_equal(&lbrac_tok,
    current_tok, true))
{
match(&lbrac_tok, true);
LangType e_type = expression()
    ;
match(&rbrac_tok, true);
LangType n_type =
    convert_from_array(id_type)
    ;
return array_compare(n_type,
    e_type);

// Production 17.2
} else if (tokens_equal(&
    assignop_tok, current_tok,
    true))
return id_type; // epsilon

synch();
return ERR;
}
```

Listing 27: compound$_s$*tatement.c*

```
#include<stdbool.h>
#include<stdlib.h>

#include "productions.h"
#include "../parser.h"
#include "../../tokenizer/
    tokens.h"

static const Token* first_set
    [] = {&begin_tok};
static const int first_size =
    sizeof(first_set)/sizeof(
    first_set[0]);

static const Token* sync_set[]
     = {&eof_tok, &semic_tok, &
    period_tok,
&end_tok, &else_tok};
static const int sync_size =
    sizeof(sync_set)/sizeof(
    sync_set[0]);

static void synch()
{
require_sync(sync_set,
```

```
    sync_size, first_set,
    first_size);
}


// Needs implementing: None
void compound_statement()
{
// Production 11
if (tokens_equal(&begin_tok,
    current_tok, true))
{
match(&begin_tok, true);
optional_statements();
match(&end_tok, true);
return;
}

synch();
}
```

Listing 28: declarations.c

```
#include<stdbool.h>
#include<stdlib.h>

#include "productions.h"
#include "../parser.h"
#include "../../tokenizer/
    tokens.h"

static const Token* first_set
    [] = {&var_tok, &
    procedure_tok, &begin_tok};
static const int first_size =
    sizeof(first_set)/sizeof(
    first_set[0]);

static const Token* sync_set[]
     = {&eof_tok, &
    procedure_tok, &begin_tok};
static const int sync_size =
    sizeof(sync_set)/sizeof(
    sync_set[0]);

static void synch()
{
require_sync(sync_set,
    sync_size, first_set,
    first_size);
}

// Needs implementing: None
void declarations()
{
// Production 3.1
if (tokens_equal(&var_tok,
    current_tok, true))
{
match(&var_tok, true);
Token* id_ref = match(&id_tok,
     false);
match(&colon_tok, true);
if (id_ref != NULL) {
id_ref -> type = type(id_ref);
id_ref -> param = false;
check_add_node(id_ref);
} else {
type(NULL);
}
match(&semic_tok, true);
declarations();
return;

// Production 3.2
} else if (tokens_equal(&
    begin_tok, current_tok,
    true)
 || tokens_equal(&procedure_tok
     , current_tok, true))
 return; // epsilon

synch();
}
```

Listing 29: else$_t$$ail.c$

```
#include<stdbool.h>
#include<stdlib.h>

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

```
#include "../../tokenizer/
    tokens.h"

static const Token* first_set
    [] = {&else_tok, &semic_tok
    , &end_tok, &else_tok};
static const int first_size =
    sizeof(first_set)/sizeof(
    first_set[0]);

static const Token* sync_set[]
     = {&eof_tok, &semic_tok, &
    end_tok, &else_tok};
static const int sync_size =
    sizeof(sync_set)/sizeof(
    sync_set[0]);

static void synch()
{
require_sync(sync_set,
    sync_size, first_set,
    first_size);
}

// Needs implementing: None
void else_tail()
{
// Production 15.1
if (tokens_equal(&else_tok,
    current_tok, true)) // else
{
match(&else_tok, true);
statement();
return;

// Production 15.2
} else if (tokens_equal(&
    end_tok, current_tok, true)
     // end
 || tokens_equal(&semic_tok,
     current_tok, true)) // ;
return; // epsilon

synch();
}
```

Listing 30: expression.c

```
#include<stdbool.h>
#include<stdlib.h>

#include "productions.h"
#include "../parser.h"
#include "../../tokenizer/
    tokens.h"

static const Token* first_set
    [] = {&id_tok, &num_tok, &
    lparen_tok, &not_tok,
 &plus_tok, &minus_tok};
static const int first_size =
    sizeof(first_set)/sizeof(
    first_set[0]);

static const Token* sync_set[]
     = {&eof_tok, &semic_tok, &
    end_tok, &else_tok,
&do_tok, &then_tok, &rbrac_tok
    , &rparen_tok,
&comma_tok};
static const int sync_size =
    sizeof(sync_set)/sizeof(
    sync_set[0]);

static void synch()
{
require_sync(sync_set,
    sync_size, first_set,
    first_size);
}

// Needs implementing: None
LangType expression()
{
// Production 21
if (tokens_equal(&lparen_tok,
    current_tok, true)
|| tokens_equal(&addop_tok,
    current_tok, false)
|| tokens_equal(&id_tok,
    current_tok, false)
|| tokens_equal(&not_tok,
```

```
    current_tok, true)
|| tokens_equal(&num_tok,
    current_tok, false))
{
LangType s_type =
    simple_expression();
return related_expression(
    s_type);
}

synch();
return ERR;
}
```

Listing 31: expression_list.c

```
#include<stdbool.h>
#include<stdlib.h>

#include "productions.h"
#include "../parser.h"
#include "../../tokenizer/
    tokens.h"

static const Token* first_set
    [] = {&id_tok, &num_tok, &
    lparen_tok, &not_tok,
 &plus_tok, &minus_tok};
static const int first_size =
    sizeof(first_set)/sizeof(
    first_set[0]);

static const Token* sync_set[]
     = {&eof_tok, &rparen_tok};
static const int sync_size =
    sizeof(sync_set)/sizeof(
    sync_set[0]);

static void synch()
{
require_sync(sync_set,
    sync_size, first_set,
    first_size);
}

// Needs implementing: None
void expression_list(tree_node
    * to_match, bool
    should_error)
{
// Production 20.1
if (tokens_equal(&lparen_tok,
    current_tok, true)
|| tokens_equal(&addop_tok,
    current_tok, false) // + OR
     -
|| tokens_equal(&id_tok,
    current_tok, false) // ID
|| tokens_equal(&not_tok,
    current_tok, true)
|| tokens_equal(&num_tok,
    current_tok, false)) // num
{
char* errorMessage;
if (to_match == NULL &&
    should_error)
{
errorMessage= calloc(100,
    sizeof(*errorMessage));
sprintf(errorMessage, "Attempt
     to pass extraneous
    parameter!");
throw_sem_error(errorMessage);
}
LangType e_type = expression()
    ;
if (should_error && to_match
    != NULL && e_type != ERR &&
     e_type != to_match -> type
    ) {
errorMessage= calloc(100,
    sizeof(*errorMessage));
sprintf(errorMessage, "
    Expected type %s, not %s!",
typeNames[to_match -> type],
    typeNames[e_type]);
throw_sem_error(errorMessage);
}
expression_list_tail(to_match
    == NULL || !to_match ->
    param ? NULL : to_match ->
    left, e_type != ERR &&
```

```
    should_error);
return;
}

synch();
}
```

Listing 32: expression$_{l}ist_{t}ail.c$

```
#include<stdbool.h>
#include<stdlib.h>

#include "productions.h"
#include "../parser.h"
#include "../../tokenizer/
    tokens.h"

static const Token* first_set
    [] = {&comma_tok, &
    rparen_tok};
static const int first_size =
    sizeof(first_set)/sizeof(
    first_set[0]);

static const Token* sync_set[]
     = {&eof_tok, &rparen_tok};
static const int sync_size =
    sizeof(sync_set)/sizeof(
    sync_set[0]);

static void synch()
{
require_sync(sync_set,
    sync_size, first_set,
    first_size);
}

// Needs implementing: None
void expression_list_tail(
    tree_node* to_match, bool
    should_error)
{
char* errorMessage;
// Production 20.2.1
if (tokens_equal(&comma_tok,
    current_tok, true))
{
match(&comma_tok, true);
if (to_match == NULL &&
    should_error)
{
errorMessage= calloc(100,
    sizeof(*errorMessage));
sprintf(errorMessage, "Attempt
     to pass extraneous
    parameters!");
throw_sem_error(errorMessage);
}
LangType e_type = expression()
    ;
if (should_error && to_match
    != NULL && e_type !=
    to_match -> type) {
errorMessage= calloc(100,
    sizeof(*errorMessage));
sprintf(errorMessage, "
    Expected type %s, not %s!",
typeNames[to_match -> type],
    typeNames[e_type]);
throw_sem_error(errorMessage);
}
expression_list_tail(to_match
    == NULL ? NULL : to_match
    -> left, should_error);
return;

// Production 20.2.2
} else if (tokens_equal(&
    rparen_tok, current_tok,
    true))
{
if (to_match != NULL &&
    to_match -> param &&
    should_error) {
errorMessage= calloc(100,
    sizeof(*errorMessage));
sprintf(errorMessage, "
    Expected %s, not the end of
     the parameters!",
typeNames[to_match -> type]);
throw_sem_error(errorMessage);
}
```

```
52 return; // epsilon
53 }
54
55 synch();
56 }
```

Listing 33: factor.c

```
1  #include<stdbool.h>
2  #include<stdlib.h>
3
4  #include "productions.h"
5  #include "../parser.h"
6  #include "../../tokenizer/
       tokens.h"
7
8  static const Token* first_set
       [] = {&id_tok, &num_tok, &
       lparen_tok, &not_tok};
9  static const int first_size =
       sizeof(first_set)/sizeof(
       first_set[0]);
10
11 static const Token* sync_set[]
        = {&eof_tok, &mulop_tok, &
       addop_tok, &relop_tok,
12 &semic_tok, &end_tok, &
       else_tok, &do_tok,
13 &then_tok, &rbrac_tok, &
       rparen_tok,
14 &comma_tok};
15 static const int sync_size =
       sizeof(sync_set)/sizeof(
       sync_set[0]);
16
17 static void synch()
18 {
19 require_sync(sync_set,
       sync_size, first_set,
       first_size);
20 }
21
22 // Needs implementing: 25.1.2
23 LangType factor()
24 {
25 // Production 25.1.1
26 if (tokens_equal(&id_tok,
       current_tok, false)) { //
       id
27 Token* id_ref;
28 id_ref = match(&id_tok, false)
       ; // id
29 LangType id_type = get_type(
       id_ref);
30 return factor_tail(id_type);
31
32 // Production 25.1.2
33 } else if (tokens_equal(&
       num_tok, current_tok, false
       )) { // num
34 Token* num_type;
35 num_type = match(&num_tok,
       false);
36 return num_type -> aspect == 0
        ? INT : REAL;
37
38 // Production 25.1.3
39 } else if (tokens_equal(&
       lparen_tok, current_tok,
       true)) { // (
40 match(&lparen_tok, true);
41 LangType e_type = expression()
       ;
42 match(&rparen_tok, true); // )
43 return e_type;
44
45 // Production 25.1.4
46 } else if (tokens_equal(&
       not_tok, current_tok, true)
       ) { // not
47 Token* not_op;
48 not_op = match(&not_tok, true)
       ;
49 LangType f_type = factor();
50 return type_lookup(f_type, ERR
       , not_op);
51 }
52
53
54 synch();
55 return ERR;
56 }
```

Listing 34: factor$_t$*ail.c*

```
#include<stdbool.h>

#include "productions.h"
#include "../parser.h"
#include "../../tokenizer/tokens.h"

static const Token* first_set[] = {&lbrac_tok, &mulop_tok, &addop_tok, &relop_tok,
 &semic_tok, &end_tok, &else_tok, &do_tok,
 &then_tok, &rbrac_tok, &rparen_tok,
 &comma_tok};
static const int first_size = sizeof(first_set)/sizeof(first_set[0]);

static const Token* sync_set[] = {&eof_tok, &mulop_tok, &addop_tok, &relop_tok,
&semic_tok, &end_tok, &else_tok, &do_tok,
&then_tok, &rbrac_tok, &rparen_tok,
&comma_tok};
static const int sync_size = sizeof(sync_set)/sizeof(sync_set[0]);

static void synch()
{
require_sync(sync_set, sync_size, first_set, first_size);
}

static LangType array_compare(LangType a_vals, LangType e_type) {
if ((a_vals == INT || a_vals == REAL) && e_type == INT)
return a_vals;
if (a_vals != ERR)
{
char* errorMessage = calloc(100, sizeof(*errorMessage));
sprintf(errorMessage, "Attempt to index variable of type %s!", typeNames[a_vals]);
throw_sem_error(errorMessage);
}

return ERR;
}

// Needs implementing: None
LangType factor_tail(id_type)
{
// Production 25.2.1
if (tokens_equal(&lbrac_tok, current_tok, true)) {
match(&lbrac_tok, true);
LangType e_type = expression();
match(&rbrac_tok, true);
LangType n_type = convert_from_array(id_type);
return array_compare(n_type, e_type);

// Production 25.2.2
} else if (tokens_equal(&rparen_tok, current_tok, true)
|| tokens_equal(&comma_tok, current_tok, true)
|| tokens_equal(&semic_tok, current_tok, true)
|| tokens_equal(&rbrac_tok, current_tok, true)
|| tokens_equal(&addop_tok, current_tok, false)
|| tokens_equal(&do_tok,
```

```
    current_tok, true)
|| tokens_equal(&else_tok,
    current_tok, true)
|| tokens_equal(&end_tok,
    current_tok, true)
|| tokens_equal(&mulop_tok,
    current_tok, false)
|| tokens_equal(&relop_tok,
    current_tok, false)
|| tokens_equal(&then_tok,
    current_tok, true))
return id_type; // epsilon

synch();
return ERR;
}
```

Listing 35: id$_list$.c

```
#include<stdbool.h>
#include<stdlib.h>

#include "productions.h"
#include "../parser.h"
#include "../../tokenizer/
    tokens.h"

static const Token* first_set
    [] = {&id_tok};
static const int first_size =
    sizeof(first_set)/sizeof(
    first_set[0]);

static const Token* sync_set[]
     = {&eof_tok, &rparen_tok};
static const int sync_size =
    sizeof(sync_set)/sizeof(
    sync_set[0]);

static void synch()
{
require_sync(sync_set,
    sync_size, first_set,
    first_size);
}

// Needs implementing: None
void id_list()
{
Token* id_ref;
// Production 2.1
if (tokens_equal(&id_tok,
    current_tok, false)) {
id_ref = match(&id_tok, false)
    ;
if (id_ref != NULL) {
id_ref -> type = PPNAME;
id_ref -> param = true;
check_add_node(id_ref);
}
id_list_tail();
return;
}

synch();
}
```

Listing 36: id$_list_tail$.c

```
#include<stdbool.h>
#include<stdlib.h>

#include "productions.h"
#include "../parser.h"
#include "../../tokenizer/
    tokens.h"

static const Token* first_set
    [] = {&comma_tok, &
    rparen_tok};
static const int first_size =
    sizeof(first_set)/sizeof(
    first_set[0]);

static const Token* sync_set[]
     = {&eof_tok, &rparen_tok};
static const int sync_size =
    sizeof(sync_set)/sizeof(
    sync_set[0]);

static void synch()
{
```

```
16 require_sync(sync_set,
       sync_size, first_set,
       first_size);
17 }
18
19 // Needs implementing: None
20 void id_list_tail()
21 {
22 // Production 2.2.1
23 if (tokens_equal(&comma_tok,
       current_tok, true))
24 {
25 match(&comma_tok, true);
26 Token* id_ref;
27
28 id_ref = match(&id_tok, false)
       ;
29 if (id_ref != NULL) {
30 id_ref -> type = PPNAME;
31 id_ref -> param = true;
32 check_add_node(id_ref);
33 }
34
35 id_list_tail();
36 return;
37
38 // Production 2.2.2
39 } else if (tokens_equal(&
       rparen_tok, current_tok,
       true))
40 return; // Epsilon
41
42 synch();
43 }
```

Listing 37: optional$_e$*xpressions.c*

```
1  #include<stdbool.h>
2  #include<stdlib.h>
3
4  #include "productions.h"
5  #include "../parser.h"
6  #include "../../tokenizer/
       tokens.h"
7
8  static const Token* first_set
       [] = {&lparen_tok, &
       semic_tok, &end_tok,
9   &else_tok};
10 static const int first_size =
       sizeof(first_set)/sizeof(
       first_set[0]);
11
12 static const Token* sync_set[]
        = {&eof_tok, &semic_tok, &
       end_tok, &else_tok};
13 static const int sync_size =
       sizeof(sync_set)/sizeof(
       sync_set[0]);
14
15 static void synch()
16 {
17 require_sync(sync_set,
       sync_size, first_set,
       first_size);
18 }
19
20 // Needs implementing: None
21 void optional_expressions(
       tree_node* to_match, bool
       should_error)
22 {
23 char* errorMessage;
24 // Production 19.1
25 if (tokens_equal(&lparen_tok,
       current_tok, true))
26 {
27 match(&lparen_tok, true);
28 expression_list(to_match,
       should_error);
29 match(&rparen_tok, true);
30 return;
31
32 // Production 19.2
33 } else if (tokens_equal(&
       semic_tok, current_tok,
       true)
34 || tokens_equal(&else_tok,
       current_tok, true)
35 || tokens_equal(&end_tok,
       current_tok, true))
36 {
```

```
37 if (to_match != NULL &&
       should_error) {
38 errorMessage= calloc(100,
       sizeof(*errorMessage));
39 sprintf(errorMessage, "
       Expected another argument
       of type %s!",
40 typeNames[to_match -> type]);
41 throw_sem_error(errorMessage);
42 }
43
44 return; // epsilon
45 }
46
47 synch();
48 }
```

Listing 38: optional$_s$*tatements.c*

```
1  #include<stdbool.h>
2  #include<stdlib.h>
3
4  #include "productions.h"
5  #include "../parser.h"
6  #include "../../tokenizer/
       tokens.h"
7
8  static const Token* first_set
       [] = {&id_tok, &call_tok, &
       begin_tok, &while_tok,
9   &if_tok, &end_tok, &array_tok
        };
10 static const int first_size =
       sizeof(first_set)/sizeof(
       first_set[0]);
11
12 static const Token* sync_set[]
        = {&eof_tok, &end_tok};
13 static const int sync_size =
       sizeof(sync_set)/sizeof(
       sync_set[0]);
14
15 static void synch()
16 {
17 require_sync(sync_set,
       sync_size, first_set,
       first_size);
18 }
19
20 // Needs implementing: None
21 void optional_statements()
22 {
23 // Production 12.1
24 if (tokens_equal(&begin_tok,
       current_tok, true) // begin
25 || tokens_equal(&call_tok,
       current_tok, true) // call
26 || tokens_equal(&id_tok,
       current_tok, false) // ID
27 || tokens_equal(&if_tok,
       current_tok, true) // if
28 || tokens_equal(&while_tok,
       current_tok, true)) //
       while
29 {
30 statement_list();
31 return;
32
33 // Production 12.2
34 } else if (tokens_equal(&
       end_tok, current_tok, true)
       ) // end
35 return; // epsilon
36
37 synch();
38 }
```

Listing 39: parameter$_l$*ist.c*

```
1  #include<stdbool.h>
2  #include<stdlib.h>
3
4  #include "productions.h"
5  #include "../parser.h"
6  #include "../../tokenizer/
       tokens.h"
7
8  static const Token* first_set
       [] = {&id_tok};
9  static const int first_size =
       sizeof(first_set)/sizeof(
       first_set[0]);
```

```

static const Token* sync_set[]
     = {&eof_tok, &rparen_tok};
static const int sync_size =
    sizeof(sync_set)/sizeof(
    sync_set[0]);

static void synch()
{
require_sync(sync_set,
    sync_size, first_set,
    first_size);
}

// Needs implementing: None
void parameter_list()
{
// Production 10.1
if (tokens_equal(&id_tok,
    current_tok, false)) {
Token* id_ref;
id_ref = match(&id_tok, false)
    ; // ID
match(&colon_tok, true);
if (id_ref != NULL) {
id_ref -> param = true;
id_ref -> type = type(id_ref);
check_add_node(id_ref);
} else {
type(NULL);
}
parameter_list_tail();
return;
}

synch();
}
```

Listing 40: parameter$_l ist_t ail.c$

```
#include<stdbool.h>
#include<stdlib.h>

#include "productions.h"
#include "../parser.h"
#include "../../tokenizer/
    tokens.h"

static const Token* first_set
    [] = {&semic_tok, &
    rparen_tok};
static const int first_size =
    sizeof(first_set)/sizeof(
    first_set[0]);

static const Token* sync_set[]
     = {&eof_tok, &rparen_tok};
static const int sync_size =
    sizeof(sync_set)/sizeof(
    sync_set[0]);

static void synch()
{
require_sync(sync_set,
    sync_size, first_set,
    first_size);
}

// Needs implementing: None
void parameter_list_tail()
{
// Production 10.2.1
if (tokens_equal(&semic_tok,
    current_tok, true)) // ;
{
match(&semic_tok, true); // ;
Token* id_ref = match(&id_tok,
     false); // ID
match(&colon_tok, true); // :
if (id_ref != NULL) {
id_ref -> param = true;
id_ref -> type = type(id_ref);
check_add_node(id_ref);
} else {
type(NULL);
}
parameter_list_tail();
return;

// Production 10.2.2
} else if (tokens_equal(&
    rparen_tok, current_tok,
```

```
    true)) // )
return; // epsilon

synch();
}
```

Listing 41: procedure$_s$*tatement.c*

```
#include<stdbool.h>
#include<stdlib.h>

#include "productions.h"
#include "../parser.h"
#include "../../tokenizer/
    tokens.h"

static const Token* first_set
    [] = {&call_tok};
static const int first_size =
    sizeof(first_set)/sizeof(
    first_set[0]);

static const Token* sync_set[]
     = {&eof_tok, &semic_tok, &
    end_tok, &else_tok};
static const int sync_size =
    sizeof(sync_set)/sizeof(
    sync_set[0]);

static void synch()
{
require_sync(sync_set,
    sync_size, first_set,
    first_size);
}

// Needs implementing: None
void procedure_statement()
{
char* errorMessage;
// Production 18
if (tokens_equal(&call_tok,
    current_tok, true)) // call
{
Token* id_ref;
match(&call_tok, true); //
    call
id_ref = match(&id_tok, false)
    ;
if (id_ref != NULL) {
tree_node* addition =
    start_param_matching(id_ref
    );
if (addition == NULL) {
errorMessage= calloc(100,
    sizeof(*errorMessage));
sprintf(errorMessage, "
    Procedure '%s' not in scope
    !", id_ref -> id);
throw_sem_error(errorMessage);

optional_expressions(NULL,
    false);
} else
 optional_expressions(addition
     -> left == NULL ? NULL :
     addition -> left -> param
     ? addition -> left : NULL,
      true);
} else {
optional_expressions(NULL,
    false);
}
return;
}

synch();
}
```

Listing 42: program.c

```
#include<stdbool.h>
#include<stdlib.h>
#include<stdio.h>

#include "productions.h"
#include "../parser.h"
#include "../../tokenizer/
    tokens.h"

static const Token* first_set
    [] = {&program_tok};
```

```
10 static const int first_size =
       sizeof(first_set)/sizeof(
       first_set[0]);
11
12 static const Token* sync_set[]
        = {&eof_tok};
13 static const int sync_size =
       sizeof(sync_set)/sizeof(
       sync_set[0]);
14
15 static void synch()
16 {
17 require_sync(sync_set,
       sync_size, first_set,
       first_size);
18 }
19
20 // Needs implementing: None
21 void program()
22 {
23 Token* id_ref;
24 // Production 1
25 if (tokens_equal(&program_tok,
        current_tok, true)) {
26 match(&program_tok, true); //
       program
27 id_ref = match(&id_tok, false)
       ; // id
28 if (id_ref != NULL) {
29 id_ref -> type = PGNAME;
30 id_ref -> param = false;
31 check_add_node(id_ref);
32 }
33 match(&lparen_tok, true); // (
34 id_list();
35 match(&rparen_tok, true); // )
36 match(&semic_tok, true); // ;
37 declarations();
38 subprogram_declarations();
39 compound_statement();
40 match(&period_tok, true); // .
41 return;
42 }
43
44 synch();
45 }
```

Listing 43: related$_e$*xpression.c*

```
1  #include<stdbool.h>
2  #include<stdlib.h>
3
4  #include "productions.h"
5  #include "../parser.h"
6  #include "../../tokenizer/
       tokens.h"
7
8  static const Token* first_set
       [] = {&relop_tok,
9   &semic_tok, &end_tok, &
        else_tok, &do_tok,
10  &then_tok, &rbrac_tok, &
        rparen_tok,
11  &comma_tok};
12 static const int first_size =
       sizeof(first_set)/sizeof(
       first_set[0]);
13
14 static const Token* sync_set[]
        = {&eof_tok, &semic_tok, &
       end_tok,
15 &else_tok, &do_tok, &then_tok,
        &rbrac_tok,
16 &rparen_tok, &comma_tok};
17 static const int sync_size =
       sizeof(sync_set)/sizeof(
       sync_set[0]);
18
19 static void synch()
20 {
21 require_sync(sync_set,
       sync_size, first_set,
       first_size);
22 }
23
24 // Needs implementing: None
25 LangType related_expression(
       LangType s_type)
26 {
27 // Production 22.1
28 if (tokens_equal(&relop_tok,
```

```
    current_tok, false)) {
Token* relop_op;
relop_op = match(&relop_tok,
    false);
LangType s1_type =
    simple_expression();
return type_lookup(s_type,
    s1_type, relop_op);

// Production 22.2
} else if (tokens_equal(&
    rparen_tok, current_tok,
    true)
|| tokens_equal(&comma_tok,
    current_tok, true)
|| tokens_equal(&semic_tok,
    current_tok, true)
|| tokens_equal(&rbrac_tok,
    current_tok, true)
|| tokens_equal(&do_tok,
    current_tok, true)
|| tokens_equal(&else_tok,
    current_tok, true)
|| tokens_equal(&end_tok,
    current_tok, true)
|| tokens_equal(&then_tok,
    current_tok, true))
return s_type; // epsilon

synch();
return ERR;
}
```

Listing 44: sign.c

```
#include<stdbool.h>
#include<stdlib.h>

#include "productions.h"
#include "../parser.h"
#include "../../tokenizer/
    tokens.h"

static const Token* first_set
    [] = {&plus_tok, &minus_tok
    };
static const int first_size =
    sizeof(first_set)/sizeof(
    first_set[0]);

static const Token* sync_set[]
     = {&eof_tok, &id_tok, &
    num_tok,
&not_tok, &rparen_tok};
static const int sync_size =
    sizeof(sync_set)/sizeof(
    sync_set[0]);

static void synch()
{
require_sync(sync_set,
    sync_size, first_set,
    first_size);
}

// Needs implementing: None
void sign()
{
// Production 24.2.1
if (tokens_equal(&plus_tok,
    current_tok, true)) {
match(&plus_tok, true);
return;

// Production 24.2.2
} else if (tokens_equal(&
    minus_tok, current_tok,
    true)) {
match(&minus_tok, true);
return; // epsilon
}
synch();
}
```

Listing 45: simple$_e$*xpression.c*

```
#include<stdbool.h>
#include<stdlib.h>

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

```
    tokens.h"
7
8   static const Token* first_set
        [] = {&id_tok, &num_tok, &
        lparen_tok, &not_tok,
9    &plus_tok, &minus_tok};
10  static const int first_size =
        sizeof(first_set)/sizeof(
        first_set[0]);
11
12  static const Token* sync_set[]
         = {&eof_tok, &relop_tok, &
        semic_tok,
13  &end_tok, &else_tok, &do_tok,
        &then_tok,
14  &rbrac_tok, &rparen_tok, &
        comma_tok};
15  static const int sync_size =
        sizeof(sync_set)/sizeof(
        sync_set[0]);
16
17  static void synch()
18  {
19  require_sync(sync_set,
        sync_size, first_set,
        first_size);
20  }
21
22  // Needs implementing: None
23  LangType simple_expression()
24  {
25  char* errorMessage;
26  // Production 23.1.1
27  if (tokens_equal(&lparen_tok,
        current_tok, true)
28  || tokens_equal(&id_tok,
        current_tok, false)
29  || tokens_equal(&not_tok,
        current_tok, true)
30  || tokens_equal(&num_tok,
        current_tok, false))
31  {
32  LangType t_type = term();
33  return simple_expression_tail(
        t_type);
34
35  // Production 23.1.2
36  } else if (tokens_equal(&
        plus_tok, current_tok, true
        )
37   || tokens_equal(&minus_tok,
         current_tok, true)) {
38  sign();
39  LangType t_type = term();
40  if (t_type != INT && t_type !=
         REAL && t_type != ERR)
41  {
42  errorMessage= calloc(100,
        sizeof(*errorMessage));
43  sprintf(errorMessage, "
        Expected number for use
        with sign, not %s!",
44  typeNames[t_type]);
45  throw_sem_error(errorMessage);
46  }
47  return simple_expression_tail(
        t_type);
48  }
49
50  synch();
51  return ERR;
52  }
```

Listing 46: simple$_e$*xpression*$_t$*ail.c*

```
1   #include<stdbool.h>
2   #include<stdlib.h>
3
4   #include "productions.h"
5   #include "../parser.h"
6   #include "../../tokenizer/
        tokens.h"
7
8   static const Token* first_set
        [] = {&addop_tok, &
        relop_tok,
9    &semic_tok, &end_tok, &
         else_tok, &do_tok,
10   &then_tok, &rbrac_tok, &
         rparen_tok,
11   &comma_tok};
12  static const int first_size =
```

```
    sizeof(first_set)/sizeof(
    first_set[0]);

static const Token* sync_set[]
     = {&eof_tok, &relop_tok, &
    semic_tok,
&end_tok, &else_tok, &do_tok,
    &then_tok,
&rbrac_tok, &rparen_tok, &
    comma_tok};
static const int sync_size =
    sizeof(sync_set)/sizeof(
    sync_set[0]);

static void synch()
{
require_sync(sync_set,
    sync_size, first_set,
    first_size);
}

// Needs implementing: None
LangType
    simple_expression_tail(
    LangType t_type)
{
// Production 23.2.1
if (tokens_equal(&addop_tok,
    current_tok, false)) {
Token* addop_op;
addop_op = match(&addop_tok,
    false);
LangType t_type2 = term();
return simple_expression_tail(
    type_lookup(t_type, t_type2
    , addop_op));


// Production 23.2.2
} else if (tokens_equal(&
    rparen_tok, current_tok,
    true)
|| tokens_equal(&comma_tok,
    current_tok, true)
|| tokens_equal(&semic_tok,
    current_tok, true)
|| tokens_equal(&rbrac_tok,
    current_tok, true)
|| tokens_equal(&do_tok,
    current_tok, true)
|| tokens_equal(&else_tok,
    current_tok, true)
|| tokens_equal(&end_tok,
    current_tok, true)
|| tokens_equal(&relop_tok,
    current_tok, false)
|| tokens_equal(&then_tok,
    current_tok, true))
return t_type; // epsilon

synch();
return ERR;
}
```

Listing 47: standard$_t$*ype.c*

```
#include<stdbool.h>
#include<stdlib.h>

#include "productions.h"
#include "../parser.h"
#include "../../tokenizer/
    tokens.h"

static const Token* first_set
    [] = {&integer_tok, &
    real_tok};
static const int first_size =
    sizeof(first_set)/sizeof(
    first_set[0]);

static const Token* sync_set[]
     = {&eof_tok, &semic_tok, &
    rparen_tok};
static const int sync_size =
    sizeof(sync_set)/sizeof(
    sync_set[0]);

static void synch()
{
require_sync(sync_set,
    sync_size, first_set,
```

```
    first_size);
}

// Needs implementing: None
LangType standard_type()
{
// Production 5.1
if (tokens_equal(&integer_tok,
     current_tok, true)) //
    integer
{
match(&integer_tok, true);
return INT;

// Production 5.2
} else if (tokens_equal(&
    real_tok, current_tok, true
    )) { // real
match(&real_tok, true);
return REAL;
}

synch();
return ERR;
}
```

Listing 48: statement.c

```
#include<stdbool.h>
#include<stdlib.h>

#include "productions.h"
#include "../parser.h"
#include "../../tokenizer/
    tokens.h"

static const Token* first_set
    [] = {&id_tok, &call_tok, &
    begin_tok, &while_tok,
 &if_tok};
static const int first_size =
    sizeof(first_set)/sizeof(
    first_set[0]);

static const Token* sync_set[]
     = {&eof_tok, &semic_tok, &
    end_tok, &else_tok};
static const int sync_size =
    sizeof(sync_set)/sizeof(
    sync_set[0]);

static void synch()
{
require_sync(sync_set,
    sync_size, first_set,
    first_size);
}

// Needs implementing: None
void statement()
{
char* errorMessage;
// Production 14.1
if (tokens_equal(&id_tok,
    current_tok, false)) { //
    id
Token* id_ref = current_tok;
LangType v_type = variable();
match(&assignop_tok, true);

if (get_type(id_ref) == ERR)
// The only way for this to
    error is an undeclared
    variable
{
errorMessage = calloc(100,
    sizeof(*errorMessage));
sprintf(errorMessage, "ID '%s'
     not in scope!",
id_ref -> id);
throw_sem_error(errorMessage);
expression();
} else if (v_type != ERR &&
    v_type != INT && v_type !=
    REAL)
{
errorMessage = calloc(100,
    sizeof(*errorMessage));
sprintf(errorMessage, "Cannot
    assign to ID '%s' of type
    '%s'!",
id_ref -> id, typeNames[v_type
```

```
    ]);
throw_sem_error(errorMessage);
expression();
} else {
LangType e_type = expression()
    ;
type_lookup(v_type, e_type, &
    assignop_tok);
}
return;

// Production 14.2
} else if (tokens_equal(&
    call_tok, current_tok, true
    )) { // call
procedure_statement();
return;

// Production 14.3
} else if (tokens_equal(&
    begin_tok, current_tok,
    true)) { // begin
compound_statement();
return;

// Production 14.4
} else if (tokens_equal(&
    while_tok, current_tok,
    true)) { // while
match(&while_tok, true); //
    while
LangType e_type = expression()
    ;
if (e_type != BOOL && e_type
    != ERR)
{
errorMessage= calloc(100,
    sizeof(*errorMessage));
sprintf(errorMessage, "
    Expression in while must be
     boolean, not %s!",
typeNames[e_type]);
throw_sem_error(errorMessage);
}
match(&do_tok, true);
statement();
return;

// Production 14.5
} else if (tokens_equal(&
    if_tok, current_tok, true))
     { // if
match(&if_tok, true); // if
LangType e_type = expression()
    ;
if (e_type != BOOL && e_type
    != ERR)
{
errorMessage= calloc(100,
    sizeof(*errorMessage));
sprintf(errorMessage, "If
    clause must be a boolean
    expression, not %s!",
typeNames[e_type]);
throw_sem_error(errorMessage);
}
match(&then_tok, true); //
    then
statement();
else_tail();
return;
}


synch();
}
```

Listing 49: statement$_list.c$

```
#include<stdbool.h>
#include<stdlib.h>

#include "productions.h"
#include "../parser.h"
#include "../../tokenizer/
    tokens.h"
static const Token* first_set
    [] = {&id_tok, &call_tok, &
    begin_tok, &while_tok,
 &if_tok};
static const int first_size =
    sizeof(first_set)/sizeof(
```

```
    first_set[0]);

static const Token* sync_set[]
     = {&eof_tok, &end_tok};
static const int sync_size =
    sizeof(sync_set)/sizeof(
    sync_set[0]);

static void synch()
{
require_sync(sync_set,
    sync_size, first_set,
    first_size);
}

// Needs implementing: None
void statement_list()
{
// Production 13.1
if (tokens_equal(&begin_tok,
    current_tok, true)
|| tokens_equal(&call_tok,
    current_tok, true)
|| tokens_equal(&id_tok,
    current_tok, false)
|| tokens_equal(&if_tok,
    current_tok, true)
|| tokens_equal(&while_tok,
    current_tok, true))
{
statement();
statement_list_tail();
return;
}

synch();
}
```

Listing 50: statement$_l$ist$_t$ail.c

```
#include<stdbool.h>
#include<stdlib.h>

#include "productions.h"
#include "../parser.h"
#include "../../tokenizer/
    tokens.h"

static const Token* first_set
    [] = {&semic_tok, &end_tok
    };
static const int first_size =
    sizeof(first_set)/sizeof(
    first_set[0]);

static const Token* sync_set[]
     = {&eof_tok, &end_tok};
static const int sync_size =
    sizeof(sync_set)/sizeof(
    sync_set[0]);

static void synch()
{
require_sync(sync_set,
    sync_size, first_set,
    first_size);
}

// Needs implementing: None
void statement_list_tail()
{
// Production 13.2.1
if (tokens_equal(&semic_tok,
    current_tok, true))
{
match(&semic_tok, true);
statement();
statement_list_tail();
return;


// Production 13.2.2
} else if (tokens_equal(&
    end_tok, current_tok, true)
    ) // end
return; // epsilon

synch();
}
```

Listing 51: subprogram$_d$eclaration.c

```c
#include<stdbool.h>
#include<stdlib.h>

#include "productions.h"
#include "../parser.h"
#include "../../tokenizer/
    tokens.h"

static const Token* first_set
    [] = {&procedure_tok};
static const int first_size =
    sizeof(first_set)/sizeof(
    first_set[0]);

static const Token* sync_set[]
     = {&procedure_tok};
static const int sync_size =
    sizeof(sync_set)/sizeof(
    sync_set[0]);

static void synch()
{
require_sync(sync_set,
    sync_size, first_set,
    first_size);
}

// Needs implementing: None
void subprogram_declaration()
{
// Production 7
if (tokens_equal(&
    procedure_tok, current_tok,
     true)) // procedure
{
bool declared =
    subprogram_head();
declarations();
subprogram_declarations();
compound_statement();

if (declared)
reached_end_of_scope(); // pop
     from stack
return;
}

synch();
}
```

Listing 52: subprogram$_d$*eclarations.c*

```c
#include<stdbool.h>
#include<stdlib.h>

#include "productions.h"
#include "../parser.h"
#include "../../tokenizer/
    tokens.h"

static const Token* first_set
    [] = {&procedure_tok, &
    begin_tok};
static const int first_size =
    sizeof(first_set)/sizeof(
    first_set[0]);

static const Token* sync_set[]
     = {&eof_tok, &begin_tok};
static const int sync_size =
    sizeof(sync_set)/sizeof(
    sync_set[0]);

static void synch()
{
require_sync(sync_set,
    sync_size, first_set,
    first_size);
}

// Needs implementing: None
void subprogram_declarations()
{
// Production 6.1
if (tokens_equal(&
    procedure_tok, current_tok,
     true)) // procedure
{
subprogram_declaration();
match(&semic_tok, true); // ;
subprogram_declarations();
```

```
28  return;
29
30  // Production 6.2
31  } else if (tokens_equal(&
        begin_tok, current_tok,
        true)) // begin
32  return; // Epsilon
33
34  synch();
35  }
```

Listing 53: subprogram$_h$*ead.c*

```
1   #include<stdbool.h>
2   #include<stdlib.h>
3
4   #include "productions.h"
5   #include "../parser.h"
6   #include "../../tokenizer/
        tokens.h"
7
8   static const Token* first_set
        [] = {&procedure_tok};
9   static const int first_size =
        sizeof(first_set)/sizeof(
        first_set[0]);
10
11  static const Token* sync_set[]
         = {&eof_tok, &var_tok, &
        procedure_tok,
12  &begin_tok};
13  static const int sync_size =
        sizeof(sync_set)/sizeof(
        sync_set[0]);
14
15  static void synch()
16  {
17  require_sync(sync_set,
        sync_size, first_set,
        first_size);
18  }
19
20  // Needs implementing: None
21  bool subprogram_head()
22  {
23  bool result = false;
24  // Production 8
25  if (tokens_equal(&
        procedure_tok, current_tok,
         true)) // procedure
26  {
27  Token* id_ref;
28  match(&procedure_tok, true);
        // procedure
29  id_ref = match(&id_tok, false)
        ;
30  if (id_ref != NULL) {
31  id_ref -> type = PROC;
32  id_ref -> param = false;
33  result = check_add_node(id_ref
        );
34  }
35  arguments();
36  match(&semic_tok, true); // ;
37  return result;
38  }
39
40  synch();
41  return result;
42  }
```

Listing 54: term.c

```
1   #include<stdbool.h>
2   #include<stdlib.h>
3
4   #include "productions.h"
5   #include "../parser.h"
6   #include "../../tokenizer/
        tokens.h"
7
8   static const Token* first_set
        [] = {&id_tok, &num_tok, &
        lparen_tok, &not_tok};
9   static const int first_size =
        sizeof(first_set)/sizeof(
        first_set[0]);
10
11  static const Token* sync_set[]
         = {&eof_tok, &addop_tok, &
        relop_tok, &semic_tok,
12  &end_tok, &else_tok, &do_tok,
```

```
    &then_tok,
&rbrac_tok, &rparen_tok, &
    comma_tok};
static const int sync_size =
    sizeof(sync_set)/sizeof(
    sync_set[0]);

static void synch()
{
require_sync(sync_set,
    sync_size, first_set,
    first_size);
}

// Needs implementing: None
LangType term()
{
// Production 24.1
if (tokens_equal(&lparen_tok,
    current_tok, true) // (
|| tokens_equal(&id_tok,
    current_tok, false) // ID
|| tokens_equal(&not_tok,
    current_tok, true) // not
|| tokens_equal(&num_tok,
    current_tok, false)) { //
    num
LangType f_type = factor();
return term_tail(f_type);
}

synch();
return ERR;
}
```

Listing 55: term$_t$*ail.c*

```
#include<stdbool.h>
#include<stdlib.h>

#include "productions.h"
#include "../parser.h"
#include "../../tokenizer/
    tokens.h"

static const Token* first_set
    [] = {&mulop_tok, &
    addop_tok, &relop_tok,
 &semic_tok, &end_tok, &
     else_tok, &do_tok,
 &then_tok, &rbrac_tok, &
     rparen_tok,
 &comma_tok};
static const int first_size =
    sizeof(first_set)/sizeof(
    first_set[0]);

static const Token* sync_set[]
     = {&eof_tok, &addop_tok, &
    relop_tok, &semic_tok,
&end_tok, &else_tok, &do_tok,
    &then_tok,
&rbrac_tok, &rparen_tok, &
    comma_tok};
static const int sync_size =
    sizeof(sync_set)/sizeof(
    sync_set[0]);

static void synch()
{
require_sync(sync_set,
    sync_size, first_set,
    first_size);
}

// Needs implementing: None
LangType term_tail(LangType
    f_type)
{
// Production 24.2.1
if (tokens_equal(&mulop_tok,
    current_tok, false)) { //
    MULOP
Token* mulop_op = match(&
    mulop_tok, false);
LangType f2_type = factor();
return term_tail(type_lookup(
    f_type, f2_type, mulop_op))
    ;

// Production 24.2.2
} else if (tokens_equal(&
```

```
        rparen_tok, current_tok,
        true)
35  || tokens_equal(&comma_tok,
        current_tok, true)
36  || tokens_equal(&semic_tok,
        current_tok, true)
37  || tokens_equal(&rbrac_tok,
        current_tok, true)
38  || tokens_equal(&addop_tok,
        current_tok, false)
39  || tokens_equal(&do_tok,
        current_tok, true)
40  || tokens_equal(&else_tok,
        current_tok, true)
41  || tokens_equal(&end_tok,
        current_tok, true)
42  || tokens_equal(&relop_tok,
        current_tok, false)
43  || tokens_equal(&then_tok,
        current_tok, true))
44  return f_type; // epsilon
45
46  synch();
47  return ERR;
48  }
```

Listing 56: type.c

```
1   #include<stdbool.h>
2   #include<stdlib.h>
3
4   #include "productions.h"
5   #include "../parser.h"
6   #include "../../tokenizer/
        tokens.h"
7
8   static const Token* first_set
        [] = {&integer_tok, &
        real_tok, &array_tok};
9   static const int first_size =
        sizeof(first_set)/sizeof(
        first_set[0]);
10
11  static const Token* sync_set[]
         = {&array_tok, &
        integer_tok, &real_tok};
12  static const int sync_size =
        sizeof(sync_set)/sizeof(
        sync_set[0]);
13
14  static void synch()
15  {
16  require_sync(sync_set,
        sync_size, first_set,
        first_size);
17  }
18
19  // Needs implementing: None
20  LangType type(Token* id_ref)
21  {
22  // Production 4.2
23  if (tokens_equal(&array_tok,
        current_tok, true))
24  {
25  char* errorMessage;
26  Token* numI;
27  Token* numF;
28  match(&array_tok, true); //
        array
29  match(&lbrac_tok, true); // [
30  numI = match(&num_tok, false);
         // num
31  match(&dotdot_tok, true); //
        ..
32  numF = match(&num_tok, false);
         // num
33  match(&rbrac_tok, true); // ]
34  match(&of_tok, true); // of
35  if (numI != NULL && numF !=
        NULL && id_ref != NULL)
36  if (type_lookup(numI -> aspect
         == 0 ? INT : REAL, numF ->
         aspect == 0 ? INT : REAL,
        &dotdot_tok) != ERR) {
37  if (numI -> int_val >= numF ->
         int_val) {
38  errorMessage= calloc(100,
        sizeof(*errorMessage));
39  sprintf(errorMessage, "
        Expected array end index %d
         to be strictly greater
        than start %d", numF ->
```

```
    int_val, numI -> int_val);
throw_sem_error(errorMessage);
}
id_ref -> array_length = numF
    -> int_val - numI ->
    int_val + 1;
}
return convert_to_array(
    standard_type());

// Production 4.1
} else if (tokens_equal(&
    integer_tok, current_tok,
    true) // int
 || tokens_equal(&real_tok,
     current_tok, true)) //
     real
{
return standard_type();
}

synch();
return ERR;
}
```

Listing 57: variable.c

```
#include<stdbool.h>
#include<stdlib.h>

#include "productions.h"
#include "../parser.h"
#include "../../tokenizer/
    tokens.h"

static const Token* first_set
    [] = {&id_tok};
static const int first_size =
    sizeof(first_set)/sizeof(
    first_set[0]);

static const Token* sync_set[]
     = {&eof_tok, &assignop_tok
    };
static const int sync_size =
    sizeof(sync_set)/sizeof(
    sync_set[0]);

static void synch()
{
require_sync(sync_set,
    sync_size, first_set,
    first_size);
}

// Needs implementing: None
LangType variable()
{
// Production 16
if (tokens_equal(&id_tok,
    current_tok, false)) // id
{
Token* id_ref;
id_ref = match(&id_tok, false)
    ;
return array_access(get_type(
    id_ref));
}

synch();
return ERR;
}
```

Listing 58: symbolTable.c

```
#include<stdlib.h>
#include<string.h>
#include<stdio.h> // TODO
    Remove

#include "../dataStructures/
    linkedList/linkedList.h"
#include "symbolTable.h"

LinkedList* symbolTable;

int initSymbolTable()
{
symbolTable = malloc(sizeof(*
    symbolTable));
symbolTable -> head = 0;
return 0;
```

```
}

char* pushToSymbolTable(char*
    name, size_t length)
{
add(symbolTable, name, sizeof(
    char)*length);
return (char *)(symbolTable ->
     head -> data);
}

char* checkSymbolTable(char*
    word)
{
// Then check the symbol table
struct node* node =
    symbolTable -> head;
while (node)
{
if (strcmp((char *) node ->
    data, word) == 0) // Match
return (char *)(node -> data);
node = node -> next;
}

return NULL;
}
```

Listing 59: addop.c

```
#include "../tokens.h"
#include "machines.h"

int addop(Token* storage, char
    * str, int start)
{
storage -> attribute = ADDOP;
switch (str[start])
{
case '+':
storage -> aspect = 0;
start++;
return start;

case '-':
storage -> aspect = 1;
start++;
return start;

default: break;
}

return start;
}
```

Listing 60: catchall.c

```
#include<string.h>

#include "../tokens.h"
#include "machines.h"

int catchall(Token* storage,
    char* str, int start)
{
if (strncmp(&str[start], ":=",
     2) == 0)
{
storage -> attribute =
    ASSIGNOP;
storage -> aspect = 0;
start += 2;
} else if (strncmp(&str[start
    ], "..", 2) == 0)
{
storage -> attribute = ARRAY;
storage -> aspect = 1;
start += 2;
} else if (str[start] == ':'){
storage -> attribute = TYPE;
storage -> aspect = 0;
start++;
} else if (str[start] == ',')
{
storage -> attribute = PUNC;
storage -> aspect = 0;
start++;
} else if (str[start] == ';')
{
storage -> attribute = PUNC;
storage -> aspect = 1;
start++;
```

```
} else if (str[start] == '.')
{
storage -> attribute = PUNC;
storage -> aspect = 2;
start++;
}

return start;
}
```

Listing 61: grouping.c

```
#include "../tokens.h"
#include "machines.h"

int grouping(Token* storage,
    char* str, int start)
{
storage -> attribute = GROUP;
switch (str[start])
{
case '(':
storage -> aspect = 0;
start++;
break;

case ')':
storage -> aspect = 1;
start++;
break;

case '[':
storage -> aspect = 2;
start++;
break;

case ']':
storage -> aspect = 3;
start++;
break;

default:
break;
}

return start;
}
```

Listing 62: idres.c

```
#include<string.h>
#include<stdlib.h>
#include<ctype.h>
#include<stdio.h>
#include<stdbool.h>

#include "machines.h"
#include "../../errorHandler/
    errorHandler.h"
#include "../../symbolTable/
    symbolTable.h"
#include "../../dataStructures
    /linkedList/linkedList.h"
#include "../tokens.h"

static char** reservedWords;
static int numReserved;
static enum TokenType*
    categories;
static int* attributes;

static int getIndex(const char
    ** array, size_t arr_size,
    char* item)
{
while (arr_size > 0)
{
if (strcmp(array[arr_size -
    1], item) == 0)
return arr_size - 1;
arr_size--;
}
return -1;
}

static int initResWords(FILE*
    resFile)
{
static const int length = 11;
LinkedList* resWords = malloc(
    sizeof(*resWords));
LinkedList* cats = malloc(
```

```
    sizeof(*cats));
LinkedList* attrs = malloc(
    sizeof(*attrs));

char word[length] = {0};
char attribute[length] = {0};
int attr = 0;
//while (fgets(word, length,
    resFile))
while (true)
{
fscanf(resFile, "%s", word);
if (feof(resFile))
break;
fscanf(resFile, "%s",
    attribute); // The actual
    name.
fscanf(resFile, "%d", &attr);
numReserved = add(resWords, &
    word, length*sizeof(char));
add(cats, &attribute, length*
    sizeof(char));
add(attrs, &attr, sizeof(int))
    ;
}

// Initialize the lexeme table
reservedWords = malloc(
    numReserved*sizeof(char*));
struct node* node = resWords
    -> head;

for (size_t i = 0; i <
    numReserved; i++) {
reservedWords[i] = (char *)
    node -> data;
node = node -> next;
}

// Initialize the attribute
    table
categories = malloc(
    numReserved*sizeof(enum
    TokenType));
node = cats -> head;

for (size_t i = 0; i <
    numReserved; i++) {
categories[i] = (enum
    TokenType) getIndex(
    catNames,
 sizeof(catNames)/sizeof(char*)
     ,
 (char *) node -> data);
node = node -> next;
}

// Initialize the attribute
    table
attributes = malloc(
    numReserved*sizeof(int));
node = attrs -> head;

for (size_t i = 0; i <
    numReserved; i++) {
attributes[i] = *(int *) node
    -> data;
node = node -> next;
}


return 0;
}

int initIDResMachine(FILE*
    resFile)
{
if (initSymbolTable() == 0 &&
    initResWords(resFile) == 0)
return 0;
else
return 1;
}

static int isReserved(char*
    word)
{
// Check the reserved words
    table for a match first
for (size_t i = 0; i <
    numReserved; i++) {
if (!reservedWords[i] ||
```

```
        strcmp(reservedWords[i],
        word) == 0) // Match
98  return i;
99  }
100
101 return -1;
102 }
103
104 int idres(Token* storage, char
        * str, int start)
105 {
106 int initial = start;
107 LinkedList* id = malloc(sizeof
        (*id));
108 storage -> attribute = ID;
109 storage -> aspect = 0;
110 char next = str[start];
111 if (isalpha(next)) // Can
        actually be an id/reserved
112 {
113 size_t wordSize = 0;
114 do
115 {
116 wordSize = add(id, &next,
        sizeof(char*));
117 start++;
118 next = str[start];
119 } while(isalpha(next) ||
        isdigit(next)); // Match ID
120
121 // The string of the id name
122 char* name = malloc((wordSize
        + 1)*sizeof(char));
123 name[wordSize] = '\0';
124 struct node* node = id -> head
        ;
125 for (size_t i = 0; i <
        wordSize; i++) {
126 name[wordSize - i - 1] = *(
        char *)(node -> data);
127 node = node -> next;
128 }
129
130 int index = -1;
131 char* address = 0;
132 if ((index = isReserved(name))
        >= 0)
133 { // It's a reserved word!
134 storage -> attribute =
        categories[index];
135 storage -> aspect = attributes
        [index];
136 }
137 else if ((address =
        checkSymbolTable(name)))
138 storage -> id = address;
139 else
140 storage -> id =
        pushToSymbolTable(name,
        wordSize);
141
142 }
143 if (start - initial > 10) //
        ID Too long err
144 {
145 //storage -> attribute = NOOP;
         TODO investigate
146 throw_lex_error(LEXERR, 1,
        initial, start - initial);
147 }
148 return start;
149 }
```

Listing 63: mulop.c

```
1  #include "../tokens.h"
2  #include "machines.h"
3
4  int mulop(Token* storage, char
       * str, int start)
5  {
6  storage -> attribute = MULOP;
7  if (str[start] == '*')
8  {
9  storage -> aspect = 0;
10 start++;
11 } else if (str[start] == '/')
12 {
13 storage -> aspect = 1;
14 start++;
15 }
16
```

```
17 return start;
18 }
```

Listing 64: numbers.c

```
1  #include<stdbool.h>
2  #include<stdlib.h>
3  #include<ctype.h>
4
5  #include "../tokens.h"
6  #include "machines.h"
7  #include "../../errorHandler/
       errorHandler.h"
8
9  // Assumes that "str" is valid
       as an integer.
10 char* parseNum(LinkedList*
       chars, bool real)
11 {
12 char* num = malloc((chars ->
       size + 1) * sizeof(char));
13 size_t count = chars -> size;
14 num[count--] = 0;
15 struct node* node = chars ->
       head;
16 while (node)
17 {
18 num[count--] = *(char *)node
       -> data;
19 node = node -> next;
20 }
21
22 return num;
23 }
24
25 double parseReal(LinkedList*
       digits)
26 {
27 char* array = parseNum(digits,
       true);
28 double val = strtod(array,
       NULL);
29 free(array);
30 return val;
31 }
32
33 int parseInt(LinkedList*
       digits)
34 {
35 char* array = parseNum(digits,
       false);
36 int val = (int) strtol(array,
       NULL, 10);
37 free(array);
38 return val;
39 }
40
41 int intMachine(Token* storage,
       char* str, int start)
42 {
43 storage -> attribute = NUM;
44
45 bool errored = false;
46 int initial = start;
47
48 LinkedList* digits = malloc(
       sizeof(*digits));
49 while (isdigit(str[start]))
50 add(digits, &str[start++],
       sizeof(char*));
51
52 if (start - initial > 10)
53 {
54 errored = true;
55 throw_lex_error(LEXERR, 2,
       initial, start - initial);
56 }
57 if (start > initial + 1 && str
       [initial] == '0')
58 {
59 errored = true;
60 throw_lex_error(LEXERR, 7,
       initial, start - initial);
61 }
62 // TODO investigate (all of
       these machines)
63 /*if (errored)
64         storage -> attribute =
              NOOP;
65     else*/ if (start > initial)
           // It's a proper
           integer!
```

```
{
storage -> aspect = 0;
storage -> int_val = parseInt(digits);
}

return start;
}

// NOTE: Pay attention to memory stuff here (the linked list takes up space).
int realMachine(Token* storage, char* str, int start)
{
storage -> attribute = NUM;

int initial = start;
bool errored = false;

int intPart = 0;
int fracPart = 0;

LinkedList* digits = malloc(sizeof(*digits));
while (isdigit(str[start]))
add(digits, &str[start++], sizeof(char*));

intPart = start - initial;
if (intPart == 0) // Not a real. Must start with a digit.
return initial;

if (str[start] == '.')
add(digits, &str[start++], sizeof(char*));
else // Not a real
return initial;


while (isdigit(str[start]))
add(digits, &str[start++], sizeof(char*));

fracPart = start - (initial + intPart + 1);

if (fracPart == 0) // Not a real
return initial;

// Now, we check for errors.
if (intPart > 5)
{
throw_lex_error(LEXERR, 3, initial, start - initial);
errored = true;
}
if (fracPart > 5)
{
throw_lex_error(LEXERR, 4, initial, start - initial);
errored = true;
}
if (str[initial] == '0' && intPart > 1) // Leading zero!
{
throw_lex_error(LEXERR, 8, initial, start - initial);
errored = true;
}
if (str[start - 1] == '0' && fracPart > 1) // Trailing zero!
{
throw_lex_error(LEXERR, 9, initial, start - initial);
errored = true;
}
/*
    if (errored)
        storage -> attribute = NOOP;*/
else
{
storage -> aspect = 1;
storage -> real_val = parseReal(digits);
```

```
135 }
136
137 return start;
138 }
139
140 int longRealMachine(Token*
        storage, char* str, int
        start)
141 {
142 storage -> attribute = NUM;
143
144 int initial = start;
145 bool errored = false;
146
147 int intPart = 0;
148 int fracPart = 0;
149 int expPart = 0;
150
151 LinkedList* digits = malloc(
        sizeof(*digits));
152 while (isdigit(str[start]))
153 add(digits, &str[start++],
        sizeof(char*));
154
155 intPart = start - initial;
156 if (intPart == 0) // Not a
        real. Must start with a
        digit.
157 return initial;
158
159 // REAL part
160 if (str[start] == '.')
161 add(digits, &str[start++],
        sizeof(char*));
162 else // Not a real
163 return initial;
164
165
166 while (isdigit(str[start]))
167 add(digits, &str[start++],
        sizeof(char*));
168
169 fracPart = start - (initial +
        intPart + 1);
170
171 if (fracPart == 0) // Not a
        real
172 return initial;
173
174
175 // LONG REAL part
176 int signum = 0;
177
178 if (str[start] == 'E')
179 add(digits, &str[start++],
        sizeof(char*));
180 else // Not a long real
181 return initial;
182
183 if (str[start] == '+' || str[
        start] == '-')
184 {
185 signum++;
186 add(digits, &str[start++],
        sizeof(char*));
187 }
188
189 while (isdigit(str[start]))
190 add(digits, &str[start++],
        sizeof(char*));
191
192 expPart = start - (initial +
        fracPart + intPart + signum
         + 2);
193
194 if (expPart == 0) // Not a
        long real
195 return initial;
196
197
198 // Now, we check for errors.
199 if (intPart > 5)
200 {
201 throw_lex_error(LEXERR, 3,
        initial, start - initial);
202 errored = true;
203 }
204 if (fracPart > 5)
205 {
206 throw_lex_error(LEXERR, 4,
        initial, start - initial);
207 errored = true;
```

```
208 }
209 if (str[initial] == '0' &&
        intPart > 1) // Leading
        zero!
210 {
211 throw_lex_error(LEXERR, 8,
        initial, start - initial);
212 errored = true;
213 }
214 if (str[start - expPart - 2]
        == '0' && fracPart > 1) //
        Trailing zero in real!
215 {
216 throw_lex_error(LEXERR, 9,
        initial, start - initial);
217 errored = true;
218 }
219 if (expPart > 2) // Exponent
        too long!
220 {
221 throw_lex_error(LEXERR, 5,
        initial, start - initial);
222 errored = true;
223 }
224 if (str[start - expPart] == '0
        ') // Leading zero in
        exponent!
225 {
226 throw_lex_error(LEXERR, 10,
        initial, start - initial);
227 errored = true;
228 }
229 /*
230     if (errored)
231         storage -> attribute =
              NOOP;
232     else*/
233 {
234 storage -> aspect = 1;
235 storage -> real_val =
        parseReal(digits);
236 }
237
238 return start;
239 }
```

Listing 65: relop.c

```
1  #include "../tokens.h"
2  #include "machines.h"
3
4  int relop(Token* storage, char
       * str, int start)
5  {
6  storage -> attribute = RELOP;
7  char next = str[start];
8  switch (next) {
9  case '<':
10 start++;
11 if (str[start] == '=')
12 {
13 storage -> aspect = 1;
14 start++;
15 } else if (str[start] == '>')
16 {
17 storage -> aspect = 5;
18 start++;
19 } else {
20 storage -> aspect = 0;
21 }
22 break;
23
24 case '=':
25 start++;
26 storage -> aspect = 2;
27 break;
28
29 case '>':
30 start++;
31 if (str[start] == '=')
32 {
33 storage -> aspect = 4;
34 start++;
35 } else {
36 storage -> aspect = 3;
37 }
38 break;
39
40 default: break; // Do not
       increment; continue on to
       the next machine.
41 }
```

```

return start;
}
```

Listing 66: whitespace.c

```
#include<stdlib.h>
#include<ctype.h>

#include "../tokens.h"
#include "machines.h"

int whitespace(Token* storage,
     char* str, int start)
{
storage -> attribute = WS;
if (isspace(str[start]))
{
storage -> aspect = 0;
if (str[start] == '\n')
storage -> aspect = 1;
start++;
}
return start;
}
```

Listing 67: tokenizer.c

```
#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>
#include<string.h>

#include "tokenizer.h"
#include "../dataStructures/
    linkedList/linkedlist.h"
#include "machines/machines.h"
#include "../errorHandler/
    errorHandler.h"
#include "../globals/globals.h
    "

const machine machines[] = {
    whitespace, idres,
    longRealMachine,
    realMachine, intMachine,
    grouping, catchall, relop,
    addop, mulop};

// Initialization stuff
static bool initialized =
    false;

int initializeTokens(FILE*
    resFile)
{
if (resFile) {
initIDResMachine(resFile);
initialized = true;
} else {
fprintf(stderr, "%s\n", "
    Reserved words file for
    tokenizer null!");
}
return 1;
}

static Token*
    generateNextToken()
{
if (initialized) {
Token* current = malloc(sizeof
    (*current)); // TODO
    necessary allocation?
if ((current =
    getNextErrorToken()) !=
    NULL)
return current;
else
 current = malloc(sizeof(*
     current));

int end;
current -> start = START;
for (int i = 0; i < sizeof(
    machines)/sizeof(machine);
    i++)
{
current -> aspect = 0;
end = (*machines[i])(current,
    BUFFER, START);
```

```
44 if (end > START) {
45 current -> length = end -
       START;
46 START = end;
47 return current;
48 }
49 }
50
51 // Unrecognized symbol error.
       This error is manual
       because it takes
52 // the place of a lexeme,
       rather than being processed
        during one.
53 throw_lex_error(LEXERR, 0,
       START, 1);
54 //current -> attribute = NOOP;
55 START++;
56 return current;
57 } else {
58 fprintf(stderr, "%s\n", "
       Tokenizer not initialized.
       Aborting.");
59 return NULL;
60 }
61 }
62
63
64 Token* getNextToken()
65 {
66 Token* next = malloc(sizeof(*
       next));
67 do {
68 next = generateNextToken();
69 } while (next -> attribute ==
       NOOP);
70
71 return next;
72 }
```

Listing 68: tokens.c

```
1 #include<string.h>
2 #include<stdlib.h>
3 #include<stdio.h>
4
5 #include "../errorHandler/
      errorHandler.h"
6 #include "tokens.h"
7
8 const char* catNames[] = {"
      NOOP", "FILEEND", "ASSIGNOP
      ", "RELOP", "ID",
9  "CONTROL", "ADDOP", "MULOP", "
       WS", "ARRAY", "TYPE",
10  "VAR", "NUM", "PUNC", "GROUP",
       "INVERSE",
11  "LEXERR", "SYNERR", "SEMERR"};
12
13 const char* typeNames[] = {"
      ERR", "REAL", "INT", "BOOL"
      , "PROGRAM",
14  "PROGRAM_PARAMETER", "
       PROCEDURE",
15  "INT ARRAY", "REAL ARRAY"};
16
17 const Token eof_tok = {FILEEND
      , 0, false, 0, 0};
18 const Token lparen_tok = {
      GROUP, 0, true, 0, 0};
19 const Token rparen_tok = {
      GROUP, 1, true, 0, 0};
20 const Token plus_tok = {ADDOP,
       0, true, 0, 0};
21 const Token comma_tok = {PUNC,
       0, true, 0, 0};
22 const Token minus_tok = {ADDOP
      , 1, true, 0, 0};
23 const Token semic_tok = {PUNC,
       1, true, 0, 0};
24 const Token colon_tok = {TYPE,
       0, true, 0, 0};
25 const Token dotdot_tok = {
      ARRAY, 1, true, 0, 0};
26 const Token period_tok = {PUNC
      , 2, true, 0, 0};
27 const Token lbrac_tok = {GROUP
      , 2, true, 0, 0};
28 const Token rbrac_tok = {GROUP
      , 3, true, 0, 0};
29 const Token addop_tok = {ADDOP
      , 0, false, 0, 0};
```

```
30 const Token array_tok = {ARRAY
       , 0, true, 0, 0};
31 const Token assignop_tok = {
       ASSIGNOP, 0, true, 0, 0};
32 const Token begin_tok = {
       CONTROL, 0, true, 0, 0};
33 const Token call_tok = {
       CONTROL, 10, true, 0, 0};
34 const Token do_tok = {CONTROL,
        1, true, 0, 0};
35 const Token else_tok = {
       CONTROL, 2, true, 0, 0};
36 const Token end_tok = {CONTROL
       , 3, true, 0, 0};
37 const Token id_tok = {ID, 0,
       false, 0, 0};
38 const Token if_tok = {CONTROL,
        5, true, 0, 0};
39 const Token integer_tok = {
       TYPE, 1, true, 0, 0};
40 const Token integer_val_tok =
       {NUM, 0, true, 0, 0};
41 const Token of_tok = {ARRAY,
       2, true, 0, 0};
42 const Token real_val_tok = {
       NUM, 1, true, 0, 0};
43 const Token mulop_tok = {MULOP
       , 0, false, 0, 0};
44 const Token not_tok = {INVERSE
       , 0, true, 0, 0};
45 const Token num_tok = {NUM, 0,
        false, 0, 0};
46 const Token procedure_tok = {
       CONTROL, 6, true, 0, 0};
47 const Token program_tok = {
       CONTROL, 7, true, 0, 0};
48 const Token real_tok = {TYPE,
       2, true, 0, 0};
49 const Token relop_tok = {RELOP
       , 0, false, 0, 0};
50 const Token then_tok = {
       CONTROL, 8, true, 0, 0};
51 const Token var_tok = {VAR, 0,
        true, 0, 0};
52 const Token while_tok = {
       CONTROL, 9, true, 0, 0};
53
54 static const char* lexes[] = {
       "(", ")", "+", ",", "-", ";
       ", ":", "[", "]", "addop",
55 "array", "assignop", "begin",
       "call", "do", "else",
56 "end", "ID", "if", "integer",
       "mulop", "not",
57 "num", "procedure", "program",
        "real", "relop",
58 "then", "var", "while", "EOF",
        "..", ":", ".",
59  "int value", "of", "real value
        "};
60
61 static const Token* tokens[] =
        {&lparen_tok, &rparen_tok,
        &plus_tok, &comma_tok, &
       minus_tok, &semic_tok,
62 &colon_tok, &lbrac_tok, &
       rbrac_tok, &addop_tok, &
       array_tok, &assignop_tok,
63 &begin_tok, &call_tok, &do_tok
       , &else_tok, &end_tok, &
       id_tok,
64 &if_tok, &integer_tok, &
       mulop_tok, &not_tok, &
       num_tok,
65 &procedure_tok, &program_tok,
       &real_tok, &relop_tok, &
       then_tok,
66 &var_tok, &while_tok, &eof_tok
       , &dotdot_tok,
67 &colon_tok, &period_tok, &
       integer_val_tok,
68 &of_tok, &real_val_tok};
69
70 const Token* getTokenFromLex(
       char* lex) {
71 for (int i = 0;i < sizeof(
       lexes); i++) {
72 if (strcmp(lexes[i], lex) ==
       0)
73 return tokens[i];
74 }
75
```

```
76  return NULL;
77  }
78
79  const char* getLexFromToken(
        Token* token, bool strict)
        {
80  switch (token -> attribute) {
81  case FILEEND: return "EOF";
82  case ASSIGNOP: return ":=";
83
84  case RELOP: if (strict)
85  switch (token -> aspect) {
86  case 0: return "<";
87  case 1: return "<=";
88  case 2: return "=";
89  case 3: return ">";
90  case 4: return ">=";
91  case 5: return "<>";
92  }
93  else return "RELOP";
94
95  case ID: return "ID";
96
97  case CONTROL: if (!strict)
        return "CONTROL"; else
98  switch (token -> aspect) {
99  case 0: return "begin";
100 case 1: return "do";
101 case 2: return "else";
102 case 3: return "end";
103 case 4: return "function";
104 case 5: return "if";
105 case 6: return "procedure";
106 case 7: return "program";
107 case 8: return "then";
108 case 9: return "while";
109 case 10: return "call";
110 }
111
112 case ADDOP: if (!strict)
        return "ADDOP"; else
113 switch (token -> aspect) {
114 case 0: return "+";
115 case 1: return "-";
116 }
117
118 case MULOP: if (!strict)
        return "MULOP"; else
119 switch (token -> aspect) {
120 case 0: return "*";
121 case 1: return "/";
122 }
123
124 case ARRAY: if (!strict)
        return "ARRAY"; else
125 switch (token -> aspect) {
126 case 0: return "array";
127 case 1: return "..";
128 case 2: return "of";
129 }
130
131 case TYPE: switch (token ->
        aspect) {
132 case 0: return ":";
133 case 1: return "integer";
134 case 2: return "real";
135 }
136
137 case VAR: switch (token ->
        aspect) {
138 case 0: return "var";
139 }
140
141 case NUM: if (!strict) return
        "a number"; else
142 switch (token -> aspect) {
143 case 0: return "integer value"
        ;
144 case 1: return "real value";
145 }
146
147 case PUNC: switch (token ->
        aspect) {
148 case 0: return ",";
149 case 1: return ";";
150 case 2: return ".";
151 }
152
153 case GROUP: switch (token ->
        aspect) {
154 case 0: return "(";
155 case 1: return ")";
```

```
156 case 2: return "[";
157 case 3: return "]";
158 }
159
160 case INVERSE: switch (token ->
        aspect) {
161 case 0: return "not";
162 }
163
164 case NOOP:
165 case WS:
166 case LEXERR:
167 case SYNERR:
168 case SEMERR: return "An error
        in the compiler has
        occurred.";
169 }
170 }
171
172 // Returns true if the tokens
        are equivalent, false
        otherwise
173 bool tokens_equal(const Token*
        p1, Token* p2, bool strict
        ) {
174 return p1 -> attribute == p2
        -> attribute &&
175 (!strict || p1 -> aspect == p2
        -> aspect);
176 }
177
178 LangType convert_to_array(
        LangType type) {
179 char* errorMessage;
180 switch (type) {
181 case INT: return AINT;
182 case REAL: return AREAL;
183
184 // Type mismatch!!
185 default:errorMessage= calloc
        (150, sizeof(*errorMessage)
        );
186 sprintf(errorMessage, "Attempt
        to create array using type
        %s; must use integer or
        real instead!", typeNames[
        type]);
187 throw_sem_error(errorMessage);
188
189 case ERR: return ERR;
190 }
191 }
192
193 LangType convert_from_array(
        LangType type) {
194 char* errorMessage;
195 switch (type) {
196 case AINT: return INT;
197 case AREAL: return REAL;
198
199
200 default: errorMessage = calloc
        (100, sizeof(*errorMessage)
        );
201  sprintf(errorMessage, "Attempt
        to index variable of type
        %s!", typeNames[type]);
202  throw_sem_error(errorMessage);
203 case ERR: return ERR;
204 }
205 }
206
207 static LangType
        assignop_lookup(LangType
        first, LangType second) {
208 char* errorMessage;
209 if (first == ERR || second ==
        ERR) // just an err
210 return ERR;
211 else if (first != INT && first
        != REAL) {
212 errorMessage= calloc(100,
        sizeof(*errorMessage));
213 sprintf(errorMessage, "Cannot
        assign values to variables
        of type %s!", typeNames[
        first]);
214 throw_sem_error(errorMessage);
215 return ERR;
216 }
217 else if (second != INT &&
        second != REAL) {
```

```
218 errorMessage= calloc(100,
        sizeof(*errorMessage));
219 sprintf(errorMessage, "Attempt
         to assign value %s; only
        reals and integers can be
        assigned!", typeNames[
        second]);
220 throw_sem_error(errorMessage);
221 return ERR;
222 }
223 else if (first != second) {
224 errorMessage= calloc(100,
        sizeof(*errorMessage));
225 sprintf(errorMessage, "Attempt
         to convert type %s into
        type %s in assignment!",
        typeNames[first], typeNames
        [second]);
226 throw_sem_error(errorMessage);
227 return ERR;
228 }
229
230 return NULL;
231 }
232
233 static LangType relop_lookup(
        LangType first, LangType
        second) {
234 char* errorMessage;
235 if (first == second && (first
        == INT || first == REAL))
236 return BOOL;
237 else if (first != ERR &&
        second != ERR) {
238 errorMessage= calloc(100,
        sizeof(*errorMessage));
239 sprintf(errorMessage, "Attempt
         to compare incompatible
        types %s and %s!",
        typeNames[first], typeNames
        [second]);
240 throw_sem_error(errorMessage);
241 }
242
243 return ERR;
244 }
245
246 static LangType addop_lookup(
        LangType first, LangType
        second, int opcode) {
247 char* errorMessage;
248 switch (opcode) {
249 case 0:
250 case 1: if (first == second &&
         (first == INT || first ==
        REAL))
251 return first;
252 else if (first != ERR &&
        second != ERR) {
253 errorMessage= calloc(100,
        sizeof(*errorMessage));
254 sprintf(errorMessage, "Attempt
         to add incompatible types
        %s and %s!", typeNames[
        first], typeNames[second]);
255 throw_sem_error(errorMessage);
256 return ERR;
257 }
258
259 return ERR;
260
261
262 case 2: if (first == second &&
         first == BOOL)
263 return BOOL;
264 else if (first != ERR &&
        second != ERR) {
265 errorMessage= calloc(100,
        sizeof(*errorMessage));
266 sprintf(errorMessage, "
        Expected BOOL and BOOL for
        use with 'or', received %s
        and %s!", typeNames[first],
         typeNames[second]);
267 throw_sem_error(errorMessage);
268 }
269
270 return ERR;
271
272 default: return NULL;
273 }
274 }
```

```
275
276 static LangType mulop_lookup(
        LangType first, LangType
        second, int opcode) {
277 char* errorMessage;
278
279 switch (opcode) {
280 case 0:
281 case 1: if (first == second &&
         (first == INT || first ==
        REAL))
282 return first;
283 else if ((first == REAL &&
        second == INT)
284  || (first == INT && second ==
         REAL)) {
285  errorMessage= calloc(100,
         sizeof(*errorMessage));
286  sprintf(errorMessage, "Attempt
         to multiply or divide
         incompatible types %s and
         %s!", typeNames[first],
         typeNames[second]);
287  throw_sem_error(errorMessage);
288 }
289 else if (first != ERR &&
        second != ERR) {
290 errorMessage= calloc(100,
        sizeof(*errorMessage));
291 sprintf(errorMessage, "
        Expceted ints or reals for
        multiplication, received %s
         and %s!", typeNames[first
        ], typeNames[second]);
292 throw_sem_error(errorMessage);
293 }
294
295 return ERR;
296
297
298 case 2: if (first == second &&
         first == BOOL) // and
299 return BOOL;
300 else if (first != ERR &&
        second != ERR)
301 {
302 errorMessage= calloc(100,
        sizeof(*errorMessage));
303 sprintf(errorMessage, "
        Expected BOOL and BOOL for
        use with 'and', received %s
         and %s!", typeNames[first
        ], typeNames[second]);
304 throw_sem_error(errorMessage);
305 }
306
307 return ERR;
308
309 case 3: // div; mod
310 case 4: if (first == second &&
         first == INT)
311 return INT;
312 else if (first != ERR &&
        second != ERR) {
313 errorMessage= calloc(100,
        sizeof(*errorMessage));
314 sprintf(errorMessage,
315 "Integers required with %s,
        received %s and %s!",
316 opcode == 3 ? "div" : "mod",
        typeNames[first],
317 typeNames[second]);
318 throw_sem_error(errorMessage);
319 }
320
321 return ERR;
322
323 default: return NULL;
324 }
325 }
326
327 static LangType not_lookup(
        LangType first, LangType
        second) {
328 char* errorMessage;
329
330 if (first == BOOL) // and
331 return BOOL;
332 else if (first != ERR)
333 {
334 errorMessage= calloc(100,
        sizeof(*errorMessage));
```

```
335 sprintf(errorMessage, "
        Expected BOOL use with 'not
        ', received %s!", typeNames
        [first]);
336 throw_sem_error(errorMessage);
337 }
338
339 return ERR;
340 }
341
342 static LangType array_lookup(
        LangType first, LangType
        second) {
343 if (first == second && first
        == INT)
344 return INT;
345 else if (first != ERR)
346 {
347 char* errorMessage = calloc
        (100, sizeof(*errorMessage)
        );
348 sprintf(errorMessage, "Attempt
         to index variable of type
        %s!", typeNames[first]);
349 throw_sem_error(errorMessage);
350 } else if (second != ERR){
351 char* errorMessage = calloc
        (100, sizeof(*errorMessage)
        );
352 sprintf(errorMessage, "Attempt
         to use variable of type %s
         to index array!",
        typeNames[second]);
353 throw_sem_error(errorMessage);
354 }
355
356 return ERR;
357 }
358
359 LangType type_lookup(LangType
        first, LangType second,
        Token* op) {
360 if (first == ERR || second ==
        ERR || op == NULL)
361 return ERR;
362
363 switch (op -> attribute) {
364 // Operations which are
        meaninngless
365 case NOOP:
366 case LEXERR:
367 case SYNERR:
368 case SEMERR:
369 case GROUP:
370 case PUNC:
371 case FILEEND:
372 case ID:
373 case CONTROL:
374 case WS:
375 case TYPE:
376 case VAR:
377 case NUM: return NULL;
378
379 case ASSIGNOP: return
        assignop_lookup(first,
        second);
380 case RELOP: return
        relop_lookup(first, second)
        ;
381 case ADDOP: return
        addop_lookup(first, second,
         op -> aspect);
382 case ARRAY: return
        array_lookup(first, second)
        ;
383 case MULOP: return
        mulop_lookup(first, second,
         op -> aspect);
384 case INVERSE: return
        not_lookup(first, second);
385
386 }
387 }
```