Sam Beckmann
Compiler Construction
Project Report

Introduction

My project is titled *Paper*, and its source code can be found at github.com/samvbeckmann/paper. It is a compiler for a subset of the Pascal language, written in C.

Input: The compiler takes in a list of paths to files to be compiled as command line arguments. For instance, paths to *test1.pas* and *test2.pas* may be used.

Output: For each compiled file, 3 output files are generated in the same directory as the file being compiled: The listing file, with the extension .listing (e.g. *test1.listing*) which contains a numbered copy of the source code as well as any errors generated by the compiler; a token file, with the extension .tokens (e.g. *test1.tokens*) Which contains the tokenized output of the input source; and a symbol file, with the extension .symbols (e.g. *test1.symbols*) which contains the full symbol table for the compiled file.

Project 1, the lexical analyzer, reads source Pascal files as input and outputs both a “listing” file and a “token” file for each input. The listing file contains the source code, with line numbers added, as well as a listing of all errors in each line. For example:

SRC

```
program test #@
```

LISTING FILE

```
1         program test #@
LEXERR:   Unrecognized Symbol:   #
LEXERR:   Unrecognized Symbol:   @
```

The token file contains all the tokens in the source program, each listed on it’s own line.

SRC

```
program test #@
```

TOKEN FILE

```
1            program      10                0
1            test         50                0x2840ab83d
1            #            99                1
1            @            99                1
```

Project 2, the syntax analyzer, constructs a parse tree of the input file using a recursive descent parser of an LL(1) grammar. In the event of improperly written source files, the analyzer prints the error to the listing file, and attempts to recover. For example:

SRC

```
program test(input, output;
```

LISTING FILE

```
1         program test(input, output;
SYNERR:   Expected ',' or ')', received ‘;'
SYNERR:   Expected 'var', 'procedure', or 'begin', received ‘EOF'
```

Project 3, the semantic analyzer, decorates the parse tree to perform type and scope checking. The production functions were given parameters for inherited attributes, and returned

decorations as synthesized attributes. Project 4 added memory addressing, maintaining consistent offset locations within each scope and assigning parameters and procedures their own offsets. In the case any of the semantic verification fails, semantic errors are printed to the listing file. For example:

SRC (SNIPPET)

```
a[3.3] := 2
```

LISTING FILE

```
1         a[3.3] := 2
SEMERR:   Incorrect array access.
```

After all decorations and parsing are complete, the compiler also prints the content of the symbol table to the symbol file.

SYMBOL FILE

```
* SCOPE: {id: test, type: PGM_NAME, num-params: 0}
|\
| * VAR: {id: input, type: PGM_PARAM, offset: 0}
| * VAR: {id: output, type: PGM_PARAM, offset: 0}
| * VAR: {id: a, type: INT, offset: 0}
| * VAR: {id: b, type: REAL, offset: 4}
| * VAR: {id: c, type: AINT, offset: 12}
| * SCOPE: {id: proc1, type: PROCEDURE, num-params: 1}
| |\
| | * VAR: {id: x, type: PP_INT, offset: 0}
| | * VAR: {id: d, type: INT, offset: 28}
```

Methodology

The lexical analyzer works as a NFAϵ, running the source file through 6 machines in series, until one returns a valid token. The last machine, catch_all, is guaranteed to return a token, so the compiler will be able to match all elements of the source file. The basic loop of the program is reading a line from the source program → Matching tokens in the line, advancing a pointer after each token matched → Getting a new line when the end of the line is reached by the pointer. This pointer is referred to as the “back” pointer, which points to the location in the file after which no token has been matched. Each machine uses a “forward” pointer, which begins equal to the back pointer and advances as the line is read. If the machine matches a token, the back pointer is set the forward pointer. If a machine fails to match a token, the forward pointer is reset to the back pointer for the next machine.

The implementation of the recursive descent parser required first converting the grammar to a LL(1) format. Since the grammar did not contain any ambiguity, the conversion was a three-step process: First all nullable productions were removed, in order to facilitate remove of left recursion. Next, all left-recursive productions were rewritten to not contain such recursion. Finally, the grammar was left-factored to ensure the viable prefix property. The original and final grammar, allow with the first and follows sets, are listed at the bottom of this section.

The semantic analysis was folded into the recursive descent parser as an L-attributed grammar. Decorations were made up of both type and width attributes. The type attribute was used in the parser to verify correct type assignments and references throughout the file. Type coercion is not supported. The width attribute was tracked when declaring parameters or variables, in order to calculate memory offsets for each address. The width of a production is equal to the total amount of space that production would take in memory. Scope checking was done through a Symbol table. Symbols are divided into two categories, scope-defining node and variable/parameter nodes. scope defining nodes references both the content of their scope and the next scope, while variable/parameter nodes reference only the next id within the same scope. The symbol table is also used to track the number of parameters to procedures, the types of IDs, and the offsets of variables and parameters.

Original Grammar

1 *program* → **program id (** *id_list* **);** *declarations sub_declarations compound_statement* **.**

2.1 *id_list* → **id**
2.2 *id_list* → *id_list* **, id**

3.1 *declarations* → *declarations* **var** *id* **:** *type* **;**
3.2 *declarations* → **ϵ**

4.1 *type* → *standard_type*
4.2 *type* → **array [num .. num] of** *standard_type*

5.1 *standard_type* → **integer**
5.2 *standard_type* → **real**

6.1 *subprogram_declarations* → *subprogram_declarations subprogram_declaration* **;**
6.2 *subprogram_declarations* → **ϵ**

7 *subprogram_declaration* → *subprogram_head declarations subprogram_declarations compound_statement*

8 *subprogram_head* → **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* **else** *statement*
14.5 *statement* → **while** *expression* **do** *statement*
14.6 *statement* → **if** *expression* **then** *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* → **-**

Massaged Grammar

1.1.1 *program* → **program id (** *id_list* **);** *program`*
1.2.1 *program`* → *declarations program``*
1.2.2 *program`* → *sub_declarations compound_statement* **.**
1.2.3 *program`* → *compound_statement* **.**
1.3.1 *program``* → *sub_declarations compound_statement* **.**
1.3.2 *program``* → *compound_statement* **.**

2.1.1 *id_list* → **id** *id_list`*
2.2.1 *id_list`* → **,** **id** *id_list`*
2.2.2 *id_list`* → **ϵ**

3.1.1 *declarations* → **var id :** *type* **;** *declarations`*
3.2.1 *declarations`* → **var id :** *type* **;** *declarations`*
3.2.2 *declarations`* → **ϵ**

4.1.1 *type* → *standard_type*
4.1.2 *type* → **array [num .. num] of** *standard_type*

5.1.1 *standard_type* → **integer**
5.1.2 *standard_type* → **real**

6.1.1 *sub_declarations* → *sub_declaration* **;** *sub_declarations`*
6.2.1 *sub_declarations`* → *sub_declaration* **;** *sub_declarations`*
6.2.2 *sub_declarations`* → **ϵ**

7.1.1 *sub_declaration* → *sub_head sub_declaration`*
7.2.1 *sub_declaration`* → *declarations sub_declaration``*
7.2.2 *sub_declaration`* → *sub_declarations compound_statement*
7.2.3 *sub_declaration`* → *compound_statement*
7.3.1 *sub_declaration``* → *sub_declarations compound_statement*
7.3.2 *sub_declaration``* → *compound_statement*

8.1.1 *sub_head* → **procedure id** *sub_head`*
8.2.1 *sub_head`* → *arguments* **;**
8.2.2 *sub_head`* → **;**

9.1.1 *arguments* → **(** *parameter_list* **)**

10.1.1 *parameter_list* → **id :** *type parameter_list`*
10.2.1 *parameter_list`* → **;** **id :** *type parameter_list`*
10.2.2 *parameter_list`* → **ϵ**

11.1.1 *compound_statement* → **begin** *compound_statement`*
11.2.1 *compound_statement`* → *optional_statements* **end**
11.2.2 *compound_statement`* → **end**

12.1.1 *optional_statements* → *statement_list*

13.1.1 *statement_list* → *statement statement_list`*

13.2.1 *statement_list`* → **;** *statement statement_list`*
13.2.2 *statement_list`* → **ϵ**

14.1.1 *statement* → *variable* **assignop** *expression*
14.1.2 *statement* → *procedure_statement*
14.1.3 *statement* → *compound_statement*
14.1.4 *statement* → **if** *expression* **then** *statement statement`*
14.1.5 *statement* → **while** *expression* **do** *statement*
14.2.1 *statement`* → **else** *statement*
14.2.2 *statement`* → **ϵ**

15.1.1 *variable* → **id** *variable`*
15.2.1 *variable`* → **[** *expression* **]**
15.2.2 *variable`* → **ϵ**

16.1.1 *procedure_statement* → **call id** *procedure_statement`*
16.2.1 *procedure_statement`* → **(** *expression_list* **)**
16.2.2 *procedure_statement`* → **ϵ**

17.1.1 *expression_list* → *expression expression_list`*
17.2.1 *expression_list`* → **,** *expression expression_list`*
17.2.2 *expression_list`* → **ϵ**

18.1.1 *expression* → *simple_expression expression`*
18.2.1 *expression`* → **relop** *simple_expression*
18.2.2 *expression`* → **ϵ**

19.1.1 *simple_expression* → *term simple_expression`*
19.1.2 *simple_expression* → *sign term simple_expression`*
19.2.1 *simple_expression`* → **addop** *term simple_expression`*
19.2.2 *simple_expression`* → **ϵ**cc a

20.1.1 *term* → *factor term`*
20.2.1 *term`* → **mulop** *factor term`*
20.2.2 *term`* → **ϵ**

21.1.1 *factor* → **id** *factor`*
21.1.2 *factor* → **num**
21.1.3 *factor* → **(** *expression* **)**
21.1.4 *factor* → **not** *factor*
21.2.1 *factor`* →**[** *expression* **]**
21.2.2 *factor`* → **ϵ**

22.1.1 *sign* → **+**
22.1.2 *sign* → **-**

After the grammar was converted to LL(1) form, first and follow sets, along with the parse table, could be constructed. These results are printed below.

#	Name	First	Follow	Solved Follow
1.1.1	**program**	program	**$**	**$**
1.2.1	**program`**	var	*f(program)*	**$**
1.2.2	**program`**	procedure		
1.2.3	**program`**	begin		
1.3.1	**program``**	procedure	*f(program)*	**$**
1.3.2	**program``**	begin		
2.1.1	**id_list**	id	**)**	**)**
2.2.1	**id_list`**	,	*f(id_list)* *f(id_list`)*	**)**
2.2.2	**id_list`**	**ϵ**		
3.1.1	**declarations**	var	**procedure begin**	**procedure begin**
3.2.1	**declarations`**	var	*f(declarations)* *f(declarations`)*	**procedure begin**
3.2.2	**declarations`**	**ϵ**		
4.1.1	**type**	integer I real	**;** *f(parameter_list)* *f(parameter_list`)*	**;)**
4.1.2	**type**	array		
5.1.1	**standard_type**	integer	*f(type)*	**;)**
5.1.2	**standard_type**	real		
6.1.1	**sub_declarations**	procedure	**begin**	**begin**
6.2.1	**sub_declarations`**	procedure	*f(sub_declarations)* *f(sub_declarations`)*	**begin**
6.2.2	**sub_declarations`**	**ϵ**		
7.1.1	**sub_declaration**	procedure	**;**	**;**
7.2.1	**sub_declaration`**	var	*f(sub_declaration)*	**;**
7.2.2	**sub_declaration`**	procedure		
7.2.3	**sub_declaration`**	begin		
7.3.1	**sub_declaration``**	procedure	*f(sub_declaration`)*	**;**
7.3.2	**sub_declaration``**	begin		
8.1.1	**sub_head**	procedure	**var procedure begin**	**var procedure begin**
8.2.1	**sub_head`**	(	*f(sub_head)*	**var procedure begin**
8.2.2	**sub_head`**	;		
9.1.1	**arguments**	(	**;**	**;**
10.1.1	**parameter_list**	id	**)**	**)**

#	Name	First	Follow	Solved Follow
10.2.1	**parameter_list`**	;	*f(parameter_list)* *f(parameter_list`)*	**)**
10.2.2	**parameter_list`**	**ϵ**		
11.1.1	**compound_statement**	begin	**.** *f(sub_declaration`)* *f(sub_declaration``)* *f(statement)*	**. ; else end**
11.2.1	**compound_statement`**	id l call l begin l if l while	*f(compound_statement)*	**. ; else end**
11.2.2	**compound_statement`**	end		
12.1.1	**optional_statements**	id l call l begin l if l while	**end**	**end**
13.1.1	**statement_list**	id l call l begin l if l while	*f(optional_statements)*	**end**
13.2.1	**statement_list`**	;	*f(statement_list)* *f(statement_list`)*	**end**
13.2.2	**statement_list`**	**ϵ**		
14.1.1	**statement**	id	**; else** *f(statement_list)* *f(statement_list`)* *f(statement)* *f(statement`)*	**; else end**
14.1.2	**statement**	call		
14.1.3	**statement**	begin		
14.1.4	**statement**	if		
14.1.5	**statement**	while		
14.2.1	**statement`**	else	*f(statement)*	**; else end**
14.2.2	**statement`**	**ϵ**		
15.1.1	**variable**	id	**assignop**	**assignop**
15.2.1	**variable`**	[	*f(variable)*	**assignop**
15.2.2	**variable`**	**ϵ**		
16.1.1	**procedure_statement**	call	*f(statement)*	**; else end**
16.2.1	**procedure_statement`**	(	*f(procedure_statement)*	**; else end**
16.2.2	**procedure_statement`**	**ϵ**		
17.1.1	**expression_list**	id l num l (l not l + l -	**)**	**)**
17.2.1	**expression_list`**	,	*f(expression_list)* *f(expression_list`)*	**)**
17.2.2	**expression_list`**	**ϵ**		

#	Name	First	Follow	Solved Follow
18.1.1	**expression**	id | num | (| not | + | -	**then do] ,)** *f(statement)* *f(expression_list)* *f(expression_list`)*	**then do] ,) ; else end**
18.2.1	**expression`**	relop	*f(expression)*	**then do] ,) ; else end**
18.2.2	**expression`**	**ϵ**		
19.1.1	**simple_expression**	id | num | (| not	**relop** *f(expression)* *f(expression`)*	**relop then do] ,) ; else end**
19.1.2	**simple_expression**	+ | -		
19.2.1	**simple_expression`**	addop	*f(simple_expression)* *f(simple_expression`)*	**relop then do] ,) ; else end**
19.2.2	**simple_expression`**	**ϵ**		
20.1.1	**term**	id | num | (| not	**addop** *f(simple_expression)* *f(simple_expression`)*	**addop relop then do] ,) ; else end**
20.2.1	**term`**	mulop	*f(term)* *f(term`)*	**addop relop then do] ,) ; else end**
20.2.2	**term`**	**ϵ**		
21.1.1	**factor**	id	**mulop** *f(term)* *f(term`)* *f(factor)*	**mulop addop relop then do] ,) ; else end**
21.1.2	**factor**	num		
21.1.3	**factor**	(		
21.1.4	**factor**	not		
21.2.1	**factor`**	[	*f(factor)*	**mulop addop relop then do] ,) ; else end**
21.2.2	**factor`**	**ϵ**		
22.1.1	**sign**	+	**id num (not**	**id num (not**
22.1.2	**sign**	-		

Implementation

The implementation of the analyzer is divided into three major parts, the analyzer framework, the machines, and the symbol table. The framework is responsible for reading the input files, calling each of the machines in order, and writing the returned tokens to the token and listing files. The machines file contains each of the seven machines, as well as the data structures associated with the tokens. The symbol table file manages the linked lists of ids and reserved words.

The seven machines are *whitespace*, *long_real*, *real*, *int*, *id_res*, *relop*, and *catchall*. The whitespace machine is called first, and is the only machine that does not return a token. The whitespace machine returns a new position for the forward pointer, advancing it past any whitespace. If there is no whitespace at the forward pointer, the return value matches the input. All other machines return an **optional_token**, which is either a token or null, with the exception of the catch_all machine, which is guaranteed to always return a token. If the catch_all machine does not recognize any valid token, it returns an “unrecognized symbol” lexerr token.

Reserved words are read in from a RESERVED_WORDS file in the same directory as the executable. Each line of the file contains a reserved word, a token type, and an attribute, space-separated. This file is read into a linked list that all IDs are compared against before they are added to the symbol table. the symbol table is another linked list, which contains all IDs found in the program.

The implementation of the syntax analyzer required some slight adjustments to the lexical analyzer. First, the method of parsing input had to be adjusted to read single tokens at a time, as opposed to entire lines, and the read token needed to be placed in global variable named **tok**. This could then be used by all functions in the recursive descent parser. Secondly, since the parser would handle getting new tokens, the previous method to read to file was stripped out.

Te parser has three major components: The **parse** function, the **match** function, and a series of functions specific to the productions of the grammar. The parse function kickstarts the parsing processing, reading the first token, calling the starting production function, and finally matching an EOF token to conclude parsing. The match function takes in a terminal symbol in the language, and checks if the current token is of the same type as the literal token. If not, it prints a syntax error and reads in the next token.

The 40 production functions each have mimic the form of one or more of the productions in the grammar. The calls begin by checking the token against all possible viable prefixes for that production. If a viable prefix is found, the appropriate match and function calls are made for that production. If no viable prefix is found, a syntax error is thrown, and the program continues reading in new tokens until a synch token is found for that production.

Keeping track of the various productions with done with a **Derivation** enum, instances of which were passed to a synch function that could match tokens against all of the synch sets. Syntax error handling was abstracted out to function call which only required expected and received tokens to allow for more robust error output if so desired.

Semantic analysis of the grammar was directly folded into the parser. Functions that required attributes were modified to accept a **Decoration** struct as inherited attributes, and return a Decoration struct as synthesized attributes. Decorations consisted of types and widths. Type checking was performed within the production methods, with error-handling built into the parser. Scope checking was done through the data-structure implementation of the symbol table. Two methods, *check_add_green_node* and *check_add_blue_node*, were made to handle insertion into the table and conflicting names, while a *get_type* method handled verifying variables were declared before they were referenced within the proper scope. Parameter checking was also performed by referencing the contents of a scope-defining node in the symbol table.

Discussion and Conclusions

Although the compiler does not output proper intermediate code, it performs all analysis required by our grammar, and would be fairly simple to adapt to target MIPS, if assembly code was needed.

The lexical and syntax parsers work together to provide the basic program loop of the compiler: A token is identified and read in from the source file, and passed to the parser, which matches the token with the proper construct. This process is then repeated until the EOF token is read. As our grammar is L-attributed, all needed decorations and parsing can be done in a single, left to right depth-first pass. It would require heavy rewriting to implement non LAD features (such as a proper one-down rule implementation) but decorations that do not required a different parsing technique, such as type coercion from ints to reals, could be an easy extension to the project.

The output of the symbol table accurately and easily shows what is visible at each scope, and any errors are written onto their own lines in the listing file. A clean listing file indicates the source file was parsed with no errors, and could be compiled into intermediate code.

Paper is 2773 lines and uses no external C libraries. For testing, this project was complied using gcc on a machine running macOS Sierra. I promise no support for non-Unix compliant systems. The project is licensed under the MIT license. I, Sam Beckmann, am the sole contributor to this project.

Appendix 1: Sample Inputs and Outputs

Standard Test File

SRC

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

  procedure proc1(x:integer; y:real;
                  z:array [1..2] of integer; q: real);
    var d: integer;
    begin
      a:= 2;
      z[a] := 4;
      c[3] := 3
     end;

   procedure proc2(x: integer; y: integer);
     var e: real;

     procedure proc3(n: integer; z: real);
       var e: integer;

       procedure proc4(a: integer; z: array [1..3] of real);
         var x: integer;
         begin
           a:= e
         end;

       begin
         a:= e;
         e:= c[e]
       end;

     begin
       call proc1(x, e, c, b);
       call proc3(c[1], e);
       e := e + 4.44;
       a:= (a mod y) div x;
       while ((a >= 4) and ((b <= e)
                     or (not (a = c[a])))) do
         begin
           a:= c[a] + 1
         end
     end;

begin
  call proc2(c[4], c[5]);
  call proc2(c[4],2);
  if (a < 2) then a:= 1 else a := a + 2;
  if (b > 4.2) then a := c[a]
end.
```

LISTING FILE

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

```
6
7           procedure proc1(x:integer; y:real;
8                           z:array [1..2] of integer; q: real);
9             var d: integer;
10            begin
11              a:= 2;
12              z[a] := 4;
13              c[3] := 3
14            end;
15
16           procedure proc2(x: integer; y: integer);
17             var e: real;
18
19             procedure proc3(n: integer; z: real);
20               var e: integer;
21
22               procedure proc4(a: integer; z: array [1..3] of real);
23                 var x: integer;
24                 begin
25                   a:= e
26                 end;
27
28               begin
29                 a:= e;
30                 e:= c[e]
31               end;
32
33             begin
34               call proc1(x, e, c, b);
35               call proc3(c[1], e);
36               e := e + 4.44;
37               a:= (a mod y) div x;
38               while ((a >= 4) and ((b <= e)
39                               or (not (a = c[a])))) do
40                 begin
41                   a:= c[a] + 1
42                 end
43             end;
44
45         begin
46           call proc2(c[4], c[5]);
47           call proc2(c[4],2);
48           if (a < 2) then a:= 1 else a := a + 2;
49           if (b > 4.2) then a := c[a]
50         end.
51
```

SYMBOL FILE

```
* SCOPE: {id: test, type: PGM_NAME, num-params: 0}
|\
| * VAR: {id: input, type: PGM_PARAM, offset: 0}
| * VAR: {id: output, type: PGM_PARAM, offset: 0}
| * VAR: {id: a, type: INT, offset: 0}
| * VAR: {id: b, type: REAL, offset: 4}
| * VAR: {id: c, type: AINT, offset: 12}
| * SCOPE: {id: proc1, type: PROCEDURE, num-params: 4}
| |\
| | * VAR: {id: x, type: PP_INT, offset: 0}
| | * VAR: {id: y, type: PP_REAL, offset: 4}
| | * VAR: {id: z, type: PP_AINT, offset: 12}
| | * VAR: {id: q, type: PP_REAL, offset: 20}
| | * VAR: {id: d, type: INT, offset: 28}
| * SCOPE: {id: proc2, type: PROCEDURE, num-params: 2}
| |\
```

```
| | * VAR: {id: x, type: PP_INT, offset: 0}
| | * VAR: {id: y, type: PP_INT, offset: 4}
| | * VAR: {id: e, type: REAL, offset: 8}
| | * SCOPE: {id: proc3, type: PROCEDURE, num-params: 2}
| | |\
| | | * VAR: {id: n, type: PP_INT, offset: 0}
| | | * VAR: {id: z, type: PP_REAL, offset: 4}
| | | * VAR: {id: e, type: INT, offset: 12}
| | | * SCOPE: {id: proc4, type: PROCEDURE, num-params: 2}
| | | |\
| | | | * VAR: {id: a, type: PP_INT, offset: 0}
| | | | * VAR: {id: z, type: PP_AREAL, offset: 4}
| | | | * VAR: {id: x, type: INT, offset: 28}
```

Error Test File

SRC

```
program errortest(input, output;
var : integer; var num : ;

procedure errortest(f, s: integer; t: array [3..1] of integer);
begin
    if f then s := f;
    errortest();
    t[2.3] := 5.2328723423;
end
```

LISTING FILE

```
1         program errortest(input, output;
SYNERR:   Expected ',' or ')', received ';'
2         var : integer; var num : ;
3
4         procedure errortest(f, s: integer; t: array [3..1] of integer);
5         begin
6             if f then s := f;
SEMERR:   Use of undeclared identifier: 'f'
SEMERR:   Use of undeclared identifier: 's'
SEMERR:   Use of undeclared identifier: 'f'
7             errortest();
SEMERR:   Use of undeclared identifier: 'errortest'
SYNERR:   Expected '[' or '=', received '('
8             t[2.3] := 5.2328723423;
LEXERR:   Extra Long Real:    5.2328723423
SYNERR:   Expected 'id', 'num', '(', 'not', '+', or '-', received
'5.2328723423'
9         end
SYNERR:   Expected 'id', 'call', 'begin', 'if', or 'while', received ‘end'
```

SYMBOL FILE

```
* SCOPE: {id: errortest, type: PGM_NAME, num-params: 0}
|\
| * VAR: {id: input, type: PGM_PARAM, offset: 0}
| * VAR: {id: output, type: PGM_PARAM, offset: 0}
```

Appendix 2: Code Listing

ANALYZER.C

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "machines.h"
#include "analyzer.h"
#include "symbols.h"
#include "word_defs.h"
#include "parser.h"

// Class variables
int line;
char *forward;
FILE *sfp;
FILE *lfp;
FILE *tfp;
FILE *stp;
struct Token tok;

int main(int argc, char *argv[])
{
        for(int i = 1; i < argc; i++) {
                compile_file(argv[i]);
        }
}

/*
 * Constant array of error code strings. Used for
reporting error in a human
 * readable format in the listing file.
 */
const char * const error_codes[] = {
                "Unrecognized Sym:",
                "Extra Long ID:",
                "Extra Long Integer:",
                "Extra Long Real:",
                "Leading Zeroes:" };

/*
 * Compiles the given Pascal file.
 * Creates two files in the directory of the given
file:
 *      - .listing file which displays the source
with line numbers and errors.
 *      - .tokens file which has a line for each
token in the source.
 *
 * Arguments: src -> path to source file.
 */
static void compile_file(char src[])
{
        global_sym_table = malloc(sizeof(struct
Symbol));
        global_sym_table -> next = NULL;

        forward_eye = malloc(sizeof(struct Symbol));
        eye = NULL;

        FILE *rfp;

        char noext[40];
        strcpy(noext, src);
        *(strrchr(noext, '.') + 1) = '\0';

        char lfname[50];
        strcpy(lfname, noext);
        strcat(lfname, "listing");

        char tkname[50];
        strcpy(tkname, noext);
        strcat(tkname, "tokens");

        char stname[50];
        strcpy(stname, noext);
        strcat(stname, "symbols");

        sfp = fopen(src, "r");
        lfp = fopen(lfname, "w");
        tfp = fopen(tkname, "w");
        rfp = fopen("RESERVED_WORDS", "r");
        stp = fopen(stname, "w");


        if (sfp == NULL) {
                fprintf(stderr, "Source file \"%s\"
does not exist.\n", src);
                return;
        } else if (rfp == NULL) {
                fprintf(stderr, "RESERVED_WORDS file
not found.\n");
                return;
        }

        initialize_reserved_words(rfp);

        line = 0;

        forward = get_next_line();

        parse();

        print_symbol_table(stp);

        fclose(sfp);
        fclose(lfp);
        fclose(tfp);
        fclose(rfp);
        fclose(stp);
}

static char* get_next_line()
{
        static char buff[72];
        fgets(buff, 72, (FILE*) sfp);
        if (feof(sfp)) {
                buff[0] = EOF;
                line++;
        } else {
                fprintf(lfp, "%-10d", ++line);
                fputs(buff, lfp);
        }
        return buff;
}

static void parse()
{
        tok = get_token();
        program_call();
        match(EOF_TYPE);
}

void match(int token_type)
{
        if (tok.token_type == EOF_TYPE) {
            return;
        } else if (tok.token_type == token_type) {
                tok = get_token();
        } else {
                synerr(type_str(token_type),
tok.lexeme);
                tok = get_token();
        }
}

/**
 * Gets the next token from the file.
 *
 * Returns:
 */
struct Token get_token()
{
        struct Token token = match_token();
```

```
        forward = token.forward;

        if (token.is_id) {
                fprintf(tfp, "%4d\t%-20s\t%-2d\t%-
p\n",
                                    line,
                                    token.lexeme,
                                    token.token_type,

token.attribute.ptr);
        } else {
                fprintf(tfp, "%4d\t%-20s\t%-2d\t%-
d\n",
                                    line,
                                    token.lexeme,
                                    token.token_type,

token.attribute.attribute);
        }

        if (token.token_type == 99) {
                fprintf(lfp, "LEXERR:   %-20s%s\n",

error_codes[token.attribute.attribute- 1],
                        token.lexeme);
        }

        return token;
}

/*
 * Runs a buffer through all of the machines to
match a token.
 *
 * Arguments: forward -> Pointer to memory location
to begin reading from.
 *
 * Returns: Token that was matched from one of the
machines. Some token will
 *          always be matched by the catch-all
machine, so this is garunteed.
 */
static struct Token match_token()
{
        forward = ws_machine(forward);

        while (*forward == '\n') {
                forward = get_next_line();
                forward = ws_machine(forward);
        }

        if (*forward == EOF) {
                return make_token("EOF", EOF_TYPE,
0, NULL);
        }

        union Optional_Token result;

        result = longreal_machine(forward);
        if (result.nil != NULL)
                return result.token;

        result = real_machine(forward);
        if (result.nil != NULL)
                return result.token;

        result = int_machine(forward);
        if (result.nil != NULL)
                return result.token;

        result = id_res_machine(forward);
        if (result.nil != NULL)
                return result.token;

        result = relop_machine(forward);
        if (result.nil != NULL)
                return result.token;

        return catchall_machine(forward);
}
```

```
/**
 * Prints a syntax error to the list file.
 *
 * Arguments: expc -> String of expected values.
 *            rec -> String of received value.
 */
void synerr(char* expc, char* rec)
{
        fprintf(lfp, "SYNERR:   Expected %s,
received '%s'\n", expc, rec);
}

/**
 * Gets the string associated with each token type.
 *
 * Arguments: tokenType -> token type to get string
from
 */
static char * type_str(int tokenType) {
        switch (tokenType) {
        case PROGRAM:
                return "'program'";
        case FUNCTION:
                return "'function'";
        case PROCEDURE:
                return "'procedure'";
        case BEGIN:
                return "'begin'";
        case END:
                return "'end'";
        case IF:
                return "'if'";
        case THEN:
                return "'then'";
        case ELSE:
                return "'else'";
        case WHILE:
                return "'while'";
        case DO:
                return "'do'";
        case NOT:
                return "'not'";
        case ARRAY:
                return "'array'";
        case OF:
                return "'of'";
        case VAR:
                return "'var'";
        case EOF_TYPE:
                return "'EOF'";
        case CALL:
                return "'call'";
        case SEMI:
                return "';'";
        case COMMA:
                return "','";
        case PAREN_OPEN:
                return "'('";
        case PAREN_CLOSE:
                return "')'";
        case BR_OPEN:
                return "'['";
        case BR_CLOSE:
                return "']'";
        case COLON:
                return "':'";
        case ASSIGN:
                return "'='";
        case DOT:
                return "'.'";
        case TWO_DOT:
                return "'..'";
        case NUM:
                return "a number";
        case ID:
                return "an id";
        case MULOP:
                return "'*', '/', or 'and'";
        case ADDOP:
                return "'+', '-', or 'or'";
```

```
        case RELOP:
                return "'>', '<', '>=', '<=', '<>'";
        case STANDARD_TYPE:
                return "'integer' or 'real'";
        default:
                return "";
        }
}
```

ANALYZER.H

```
#ifndef ANALYZER_H
#define ANALYZER_H

#include "machines.h"
#include <stdio.h>

extern struct Token tok;
extern FILE *lfp;

/*
 * Compiles the given Pascal file.
 * Creates two files in the directory of the given
file:
 *      - .listing file which displays the source
with line numbers and errors.
 *      - .tokens file which has a line for each
token in the source.
 *
 * Arguments: src -> path to source file.
 */
static void compile_file(char src[]);

static char* get_next_line();
static char* type_str(int token_type);
static void parse();
void match(int token_type);
void update_tok(struct Token token);
struct Token get_token();
void synerr(char* expc, char* rec);

/*
 * Runs a buffer through all of the machines to
match a token.
 *
 * Arguments: forward -> Pointer to memory location
to begin reading from.
 *
 * Returns: Token that was matched from one of the
machines. Some token will
 *          always be matched by the catch-all
machine, so this is garunteed.
 */
static struct Token match_token();

struct Token get_token();


#endif
```

MACHINES.C

```
#include "machines.h"
#include "word_defs.h"
#include "symbols.h"
#include <string.h>
#include <ctype.h>
#include <stdbool.h>
#include <stdlib.h>

/*
 * Factory for Optional_Tokens.
 * Takes in needed parameters for a token, and makes
an Optional_Token with
 * those parameters. Abstracts the creation of
Optional_Token structs.
 *
 * Arguments: lexeme -> Literal of matched lexeme.
 *            type -> Integer representation of
token's type.
 *            attr -> Integer representation of
token's attribute.
 *            forward -> Pointer to the char after
this lexeme ended in buffer.
 *
 * Returns: An Optional_Token with the given
parameters. Not a null optional.
 */
union Optional_Token make_optional(
                        char lexeme[],
                        int type,
                        int attr,
                        char *forward) {
        return wrap_token(make_token(lexeme, type,
attr, forward));
}

/*
 * Factory for Tokens.
 * Takes in needed parameters for a token, and makes
an Optional_Token with
 * those paratmers. Abstracts the creation of Token
structs.
 *
 * Arguments: lexeme -> Literal of matched lexeme.
 *            type -> Integer representation of
token's type.
 *            attr -> Integer representation of
token's attribute.
 *            forward -> Pointer to the char after
this lexeme ended in buffer.
 *
 * Returns: A Token with the given parameters. This
does not create an id token.
 */
struct Token make_token(char lexeme[], int type, int
attr, char *forward) {
        struct Token token;
        strcpy(token.lexeme, lexeme);
        token.token_type = type;
        token.is_id = 0;
        token.attribute.attribute = attr;
        token.forward = forward;
        return token;
}

/*
 * Creates an Optional_Token which is nil.
 * Used as standard factory of nil Optional_Token
structs.
 *
 * Returns: Optional_Token with "nil" as the token.
 */
union Optional_Token null_optional() {
        union Optional_Token op_token;
        op_token.nil = NULL;
        return op_token;
}

/*
 * Wraps a token as an Optional_Token, so that it
can be returned as such.
 *
 * Arguments: token -> Token that is to be wrapped.
 *
 * Returns: Optional_Token that contains the
paramter "token"
 */
union Optional_Token wrap_token(struct Token token)
{
        union Optional_Token op_token;
        op_token.token = token;
        return op_token;
}

/*
 * Reads a series of digits until a non-digit
character is read, returning a
 * buffer of read digits.
```

```
 *
 * Arguments: forward -> Pointer to where begin
reading.
 *
 * Returns: char pointer to buffer or read digits.
 */
static char * read_digits(char *forward) {
        char * buff = malloc(30);
        int i = 0;
        char value = *forward++;
        while (isdigit(value)) {
                buff[i] = value;
                value = *forward++;
                i++;
        }
        buff[i] = '\0';
        return buff;
}

/*
 * Machine that matches whitespace.
 *
 * Arguments: forward -> Pointer to memory location
to begin reading from.

 * Returns: Pointer to first non-whitespace
character matched.
 */
char * ws_machine(char *forward)
{
        char value;
        do {
                value = *forward++;
        } while (value == ' ' || value == '\t');
        forward--;

        return forward;
}

/*
 * Machine that reads real numbers containing an
exponent, or "Long Reals".
 *
 * A long real consists of 1-5 digits, a decimal
point, 1-5 digits, "E",
 * an optional sign (+|-), and 1-2 digits.
 *
 * Arguments: forward -> Pointer to memory location
to begin reading from.
 *
 * Returns: an Optional_Token representing the
matched long real, or a nil
 *          Optional_Token if no long real is
matched.
 */
union Optional_Token longreal_machine(char *forward)
{
        char real_lit[30];
        bool extra_long = false;
        bool lead_zeros = false;

        char * first_part = read_digits(forward);
        int len = strlen(first_part);
        forward += len;
        strcpy(real_lit, first_part);

        if (len == 0)
                return null_optional();
        else if (len > 5)
                extra_long = true;
        else if (first_part[0] == '0' && len != 1)
                lead_zeros = true;

        char value = *forward++;
        if (value != '.')
                return null_optional();
        strncat(real_lit, &value, 1);

        char *second_part = read_digits(forward);
        len = strlen(second_part);
        forward += len;
        strcat(real_lit, second_part);

        if (len == 0)
                return null_optional();
        else if (len > 5)
                extra_long = true;
        else if (second_part[0] == '0' && len != 1)
                lead_zeros = true;

        value = *forward++;
        if (value != 'E')
                return null_optional();
        strncat(real_lit, &value, 1);

        value = *forward++;
        if (value == '-' || value == '+')
                strncat(real_lit, &value, 1);
        else
                forward--;

        char *exponent = read_digits(forward);
        len = strlen(exponent);
        forward += len;
        strcat(real_lit, exponent);

        if (len == 0)
                return null_optional();
        else if (len > 2)
                extra_long = true;
        else if (exponent[0] == '0')
                lead_zeros = true;

        if (extra_long)
                return make_optional(real_lit,
LEXERR, EXTRA_LONG_REAL, forward);
        else if (lead_zeros)
                return make_optional(real_lit,
LEXERR, LEADING_ZEROES, forward);
        else
                return make_optional(real_lit, NUM,
LONG_REAL, forward);

        return null_optional();
}

/*
 * Machine that reads real numbers.
 *
 * A real number consists of 1-5 digits, a decimal
point, and 1-5 digits.
 *
 * Arguments: forward -> Pointer to memory location
to begin reading from.
 *
 * Returns: An Optional_Token representing the
matched real, or a nil
 *          Optional_Token if no real number is
matched.
 */
union Optional_Token real_machine(char *forward)
{
        char real_lit[30];
        bool extra_long = false;
        bool lead_zeros = false;

        char * first_part = read_digits(forward);
        int len = strlen(first_part);
        forward += len;
        strcpy(real_lit, first_part);

        if (len == 0)
                return null_optional();
        else if (len > 5)
                extra_long = true;
        else if (first_part[0] == '0' && len != 1)
                lead_zeros = true;

        char value = *forward++;
        if (value != '.')
                return null_optional();
        strncat(real_lit, &value, 1);
```

```
        char *second_part = read_digits(forward);
        len = strlen(second_part);
        forward += len;
        strcat(real_lit, second_part);

        if (len == 0)
                return null_optional();
        else if (len > 5)
                extra_long = true;
        else if (second_part[0] == '0' && len != 1)
                lead_zeros = true;

        if (extra_long)
                return make_optional(real_lit,
LEXERR, EXTRA_LONG_REAL, forward);
        else if (lead_zeros)
                return make_optional(real_lit,
LEXERR, LEADING_ZEROES, forward);
        else
                return make_optional(real_lit, NUM,
REAL, forward);
}

/*
 * Machine that reads integers.
 *
 * An integer consists of 1-10 digits with no
leading zeroes.
 *
 * Arguments: forward -> Pointer to memory location
to begin reading from.
 *
 * Returns: An Optional_Token representing the
matched integer, or a nil
 *          Optional_Token if no integer is matched.
 */
union Optional_Token int_machine(char *forward)
{
        char *digits = read_digits(forward);
        int len = strlen(digits);
        forward += len;

        if (len == 0)
                return null_optional();
        else if (digits[0] == '0' && len != 1)
                return make_optional(digits, LEXERR,
LEADING_ZEROES, forward);
        else if (len > 10)
                return make_optional(digits, LEXERR,
EXTRA_LONG_INT, forward);
        else
                return make_optional(digits, NUM,
INTEGER, forward);
}

/*
 * Machine that matches ids and reserved words.
 *
 * An ID consists of a letter, followed by 0-9
digits or letters.
 * If the matched string is equivalent to a reserved
word, returns the token
 * that represents the reserved word.
 * Otherwise, adds the ID to the symbol table if it
is not already there,
 * and returns an Optional_Token containg the
matched ID and a reference
 * to it in the symbol table.
 *
 * Arguments: forward -> Pointer to memory location
to begin reading from.
 *
 * Returns: A LEXERR Optional_Token if an error is
encountered, or a a nil
 *          Optional_Token if no id or reserved word
is matched.
 */
union Optional_Token id_res_machine(char *forward)
{
        char word[30];
        int i = 0;
        char value = *forward++;
        while (isalnum(value)) {
                word[i] = value;
                value = *forward++;
                i++;
        }
        forward--;
        word[i] = '\0';

        if (i == 0)
                return null_optional();
        else if (i > 10)
                return make_optional(word, LEXERR,
EXTRA_LONG_ID, forward);

        union Optional_Token res =
check_reserved_words(word);
        if (res.nil != NULL) {
                res.token.forward = forward;
                return res;
        } else {
                // REVIEW: Removed use of symbol
table here for now.
                // struct Symbol *sym_ptr =
add_symbol(word);
                struct Token token;
                strcpy(token.lexeme, word);
                token.token_type = ID;
                token.is_id = 1;
                // token.attribute.ptr = sym_ptr;
                token.forward = forward;
                return wrap_token(token);
        }
}

/*
 * Machine that matches relational operators, or
"Relops".
 *
 * Valid relops: '<', '>', '==', '<=', '>=', '<>'.
 *
 * Arguments: forward -> Pointer to memory location
to begin reading from.
 *
 * Returns: An Optional_Token representing the
matched relop, or a nil
 *          Optional_Token if no relop is matched.
 */
union Optional_Token relop_machine(char *forward)
{
        char value = *forward++;
        switch (value) {
        case '<':
                value = *forward++;
                switch (value) {
                case '>':
                        return make_optional("<>",
RELOP, NEQ, forward);
                case '=':
                        return make_optional("<=",
RELOP, LT_EQ, forward);
                default:
                        forward--;
                        return make_optional("<",
RELOP, LT, forward);
                }
        case '>':
                value = *forward++;
                if (value == '=') {
                        return make_optional(">=",
RELOP, GT_EQ, forward);
                } else {
                        forward--;
                        return make_optional(">",
RELOP, GT, forward);
                }
        case '=':
                return make_optional("=", RELOP, EQ,
forward);
        default:
```

```
                return null_optional();
        }
}

/*
 * Machine that caches all other tokens not matched
by a previous machine.
 *
 * If no valid token is matched by this machine, it
returns a LEXERR for an
 * unrecognized symbol. This garuntees this machine
will always return a token.
 *
 * Arguments: forward -> Pointer to memory location
to begin reading from.
 *
 * Returns: Token either containing a valid token,
attribute pair, or a LEXERR
 *          token if no valid token is matched.
 */
struct Token catchall_machine(char *forward)
{
        char value = *forward++;
        char lexeme[2];

        switch (value) {
        case '+':
                return make_token("+", ADDOP, ADD,
forward);
        case '-':
                return make_token("-", ADDOP, SUB,
forward);
        case '*':
                return make_token("*", MULOP, MULT,
forward);
        case '/':
                return make_token("/", MULOP,
DIVIDE, forward);
        case ';':
                return make_token(";", SEMI, 0,
forward);
        case ',':
                return make_token(",", COMMA, 0,
forward);
        case '(':
                return make_token("(", PAREN_OPEN,
0, forward);
        case ')':
                return make_token(")", PAREN_CLOSE,
0, forward);
        case '[':
                return make_token("[", BR_OPEN, 0,
forward);
        case ']':
                return make_token("]", BR_CLOSE, 0,
forward);
        case ':':
                value = *forward++;
                if (value == '=') {
                        return make_token(":=",
ASSIGN, 0, forward);
                } else {
                        forward--;
                        return make_token(":",
COLON, 0, forward);
                }
        case '.':
                value = *forward++;
                if (value == '.') {
                        return make_token("..",
TWO_DOT, 0, forward);
                } else {
                        forward--;
                        return make_token(".", DOT,
0, forward);
                }
        default:
                lexeme[0] = value;
                lexeme[1] = '\0';
                return make_token(lexeme, LEXERR,
UNRECOG_SYM, forward);
        }
}
```

MACHINES.H

```
#ifndef MACHINES_H
#define MACHINES_H

/*
 * A token is the basic unit the Pascal
interpretation.
 *
 * Fields: lexeme -> The literal from source that is
this token.
 *         is_id -> 1 if this token represents an
id, otherwise 0.
 *         token_type -> integer that represents
this token's type.
 *         Attribute.attribute -> Integer that
represents the type's attribute.
 *         Attribute.ptr -> Pointer to a symbol in
the symbol table.
 *                          Used if this token is an
id.
 *         forward -> Pointer to next position in
buffer after lexeme.
 *                    Used to update the forward
pointer, then discarded.
 */
struct Token {
        char lexeme[20];
        int is_id;
        int token_type;
        union Attribute {
                int attribute;
                struct Symbol *ptr;
        } attribute;
        char *forward;
};

/*
 * An Optional_Token is either a token or null.
 * Used as a return type for machines that may not
match a token.
 *
 * Fields: nil -> Void pointer if the Optional_Token
is nil.
 *         token -> Token if Optional_Token is not
null.
 */
union Optional_Token {
        void *nil;
        struct Token token;
};

/*
 * Factory for Optional_Tokens.
 * Takes in needed parameters for a token, and makes
an Optional_Token with
 * those parameters. Abstracts the creation of
Optional_Token structs.
 *
 * Arguments: lexeme -> Literal of matched lexeme.
 *            type -> Integer representation of
token's type.
 *            attr -> Integer representation of
token's attribute.
 *            forward -> Pointer to the char after
this lexeme ended in buffer.
 *
 * Returns: An Optional_Token with the given
parameters. Not a null optional.
 */
union Optional_Token make_optional(char lexeme[],
int type, int attr, char *forward);

/*
 * Factory for Tokens.
 * Takes in needed parameters for a token, and makes
an Optional_Token with
```

```
 * those paratmers. Abstracts the creation of Token
structs.
 *
 * Arguments: lexeme -> Literal of matched lexeme.
 *            type -> Integer representation of
token's type.
 *            attr -> Integer representation of
token's attribute.
 *            forward -> Pointer to the char after
this lexeme ended in buffer.
 *
 * Returns: A Token with the given parameters. This
does not create an id token.
 */
struct Token make_token(char lexeme[], int type, int
attr, char *forward);

/*
 * Creates an Optional_Token which is nil.
 * Used as standard factory of nil Optional_Token
structs.
 *
 * Returns: Optional_Token with "nil" as the token.
 */
union Optional_Token null_optional();

/*
 * Wraps a token as an Optional_Token, so that it
can be returned as such.
 *
 * Arguments: token -> Token that is to be wrapped.
 *
 * Returns: Optional_Token that contains the
paramter "token"
 */
union Optional_Token wrap_token(struct Token token);

/*
 * Machine that matches whitespace.
 *
 * Arguments: forward -> Pointer to memory location
to begin reading from.

 * Returns: Pointer to first non-whitespace
character matched.
 */
char * ws_machine(char *forward);

/*
 * Machine that reads real numbers containing an
exponent, or "Long Reals".
 *
 * A long real consists of 1-5 digits, a decimal
point, 1-5 digits, "E",
 * an optional sign (+|-), and 1-2 digits.
 *
 * Arguments: forward -> Pointer to memory location
to begin reading from.
 *
 * Returns: an Optional_Token representing the
matched long real, or a nil
 *          Optional_Token if no long real is
matched.
 */
union Optional_Token longreal_machine(char
*forward);

/*
 * Machine that reads real numbers.
 *
 * A real number consists of 1-5 digits, a decimal
point, and 1-5 digits.
 *
 * Arguments: forward -> Pointer to memory location
to begin reading from.
 *
 * Returns: An Optional_Token representing the
matched real, or a nil
 *          Optional_Token if no real number is
matched.
 */
union Optional_Token real_machine(char *forward);

/*
 * Machine that reads integers.
 *
 * An integer consists of 1-10 digits with no
leading zeroes.
 *
 * Arguments: forward -> Pointer to memory location
to begin reading from.
 *
 * Returns: An Optional_Token representing the
matched integer, or a nil
 *          Optional_Token if no integer is matched.
 */
union Optional_Token int_machine(char *forward);

/*
 * Machine that matches ids and reserved words.
 *
 * An ID consists of a letter, followed by 0-9
digits or letters.
 * If the matched string is equivalent to a reserved
word, returns the token
 * that represents the reserved word.
 * Otherwise, adds the ID to the symbol table if it
is not already there,
 * and returns an Optional_Token containg the
matched ID and a reference
 * to it in the symbol table.
 *
 * Arguments: forward -> Pointer to memory location
to begin reading from.
 *
 * Returns: A LEXERR Optional_Token if an error is
encountered, or a a nil
 *          Optional_Token if no id or reserved word
is matched.
 */
union Optional_Token id_res_machine(char *forward);

/*
 * Machine that matches relational operators, or
"Relops".
 *
 * Valid relops: '<', '>', '==', '<=', '>=', '<>'.
 *
 * Arguments: forward -> Pointer to memory location
to begin reading from.
 *
 * Returns: An Optional_Token representing the
matched relop, or a nil
 *          Optional_Token if no relop is matched.
 */
union Optional_Token relop_machine(char *forward);

/*
 * Machine that caches all other tokens not matched
by a previous machine.
 *
 * If no valid token is matched by this machine, it
returns a LEXERR for an
 * unrecognized symbol. This garuntees this machine
will always return a token.
 *
 * Arguments: forward -> Pointer to memory location
to begin reading from.
 *
 * Returns: Token either containing a valid token,
attribute pair, or a LEXERR
 *          token if no valid token is matched.
 */
struct Token catchall_machine(char *forward);

#endif
```

PARSER.C

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

```
#include "word_defs.h"
#include "analyzer.h"
#include "synch_set.h"
#include "parser.h"
#include "types.h"
#include "symbols.h"

static void program_tail_call();
static void program_tail_tail_call();
static void id_list_call();
static void id_list_tail_call();
static void declarations_call();
static void declarations_tail_call();
static struct Decoration type_call();
static struct Decoration standard_type_call();
static void sub_declarations_call();
static void sub_declarations_tail_call();
static void sub_declaration_call();
static void sub_declaration_tail_call();
static void sub_declaration_tail_tail_call();
static void sub_head_call();
static void sub_head_tail_call();
static void arguments_call();
static void parameter_list_call();
static void parameter_list_tail_call();
static void compound_statement_call();
static void compound_statement_tail_call();
static void optional_statements_call();
static void statement_list_call();
static void statement_list_tail_call();
static void statement_call();
static void statement_tail_call();
static struct Decoration variable_call();
static struct Decoration variable_tail_call(struct
Decoration inherited);
static void procedure_statement_call();
static void procedure_statement_tail_call();
static int expression_list_call(int num_parms,
struct Symbol *param);
static int expression_list_tail_call(int num_parms,
struct Symbol *param);
static struct Decoration expression_call();
static struct Decoration expression_tail_call(struct
Decoration inherited);
static struct Decoration simple_expression_call();
static struct Decoration
simple_expression_tail_call(struct Decoration
inherited);
static struct Decoration term_call();
static struct Decoration term_tail_call(struct
Decoration inherited);
static struct Decoration factor_call();
static struct Decoration factor_tail_call(struct
Decoration inherited);
static void sign_call();

static int offset;
static int counter;

/**
 * Creates a proper Decoration struct from a given
type.
 * @param  in_type Type to assign the type field of
the Decoration struct
 * @return A new Decoration that contains the type
of the input
 */
static struct Decoration make_type_decoration(enum
Type in_type)
{
        struct Decoration *dec =
malloc(sizeof(struct Decoration));
        dec -> type = in_type;
        return *dec;
}

static struct Decoration make_decoration(enum Type
in_type, int in_width)
{
        struct Decoration *dec =
malloc(sizeof(struct Decoration));
        dec -> type = in_type;
        dec -> width = in_width;
        return *dec;
}

/**
 * Initializes the recursive decent parser.
 * Precondition: The first token of the source file
is loaded into "tok"
 */
void program_call()
{
        if (tok.token_type == PROGRAM) {
                match(PROGRAM);
                struct Token id_tok = tok;
                match(ID);
                check_add_green_node(id_tok.lexeme,
PG_NAME);
                match(PAREN_OPEN);
                id_list_call();
                match(PAREN_CLOSE);
                match(SEMI);
                program_tail_call();
        } else {
                synerr("'program'", tok.lexeme);
                enum Derivation dir = program;
                while (!synch(dir, tok.token_type))
                        tok = get_token();
        }
}

static void program_tail_call()
{
        offset = 0;
        counter = 0;
        switch (tok.token_type) {
        case VAR:
                declarations_call();
                program_tail_tail_call();
                break;
        case PROCEDURE:
                sub_declarations_call();
                compound_statement_call();
                match(DOT);
                break;
        case BEGIN:
                compound_statement_call();
                match(DOT);
                break;
        default:
                synerr("'var', 'procedure', or
'begin'", tok.lexeme);
                enum Derivation dir = program_tail;
                while (!synch(dir, tok.token_type))
                        tok = get_token();
        }
}

static void program_tail_tail_call()
{
        switch (tok.token_type) {
        case PROCEDURE:
                sub_declarations_call();
                compound_statement_call();
                match(DOT);
                break;
        case BEGIN:
                compound_statement_call();
                match(DOT);
                break;
        default:
                synerr("'procedure' or 'begin'",
tok.lexeme);
                enum Derivation dir =
program_tail_tail;
                while (!synch(dir, tok.token_type))
                        tok = get_token();
        }
}

static void id_list_call()
```

```
{
        if (tok.token_type == ID) {
                struct Token id_tok = tok;
                match(ID);
                check_add_blue_node(id_tok.lexeme,
PG_PARM, 0);
                id_list_tail_call();
        } else {
                synerr("'id'", tok.lexeme);
                enum Derivation dir = id_list;
                while (!synch(dir, tok.token_type))
                        tok = get_token();
        }
}

static void id_list_tail_call()
{
        switch (tok.token_type) {
        case COMMA:
                match(COMMA);
                struct Token id_tok = tok;
                match(ID);
                check_add_blue_node(id_tok.lexeme,
PG_PARM, 0);
                id_list_tail_call();
                break;
        case PAREN_CLOSE:
                break;
        default:
                synerr("',' or ')'", tok.lexeme);
                enum Derivation dir = id_list_tail;
                while (!synch(dir, tok.token_type))
                        tok = get_token();
        }
}

static void declarations_call()
{
        if (tok.token_type == VAR) {
                match(VAR);
                struct Token id_tok = tok;
                match(ID);
                match(COLON);
                struct Decoration type_dec =
type_call();
                check_add_blue_node(id_tok.lexeme,
type_dec.type, offset);
                offset += type_dec.width;
                match(SEMI);
                declarations_tail_call();
        } else {
                synerr("'var'", tok.lexeme);
                enum Derivation dir = declarations;
                while (!synch(dir, tok.token_type))
                        tok = get_token();
        }
}

static void declarations_tail_call()
{
        switch(tok.token_type) {
        case VAR:
                match(VAR);
                struct Token id_tok = tok;
                match(ID);
                match(COLON);
                struct Decoration type_dec =
type_call();
                check_add_blue_node(id_tok.lexeme,
type_dec.type, offset);
                offset += type_dec.width;
                match(SEMI);
                declarations_tail_call();
                break;
        case PROCEDURE:
        case BEGIN:
                break;
        default:
                synerr("'var', 'procedure' or
'begin'", tok.lexeme);
                enum Derivation dir =
declarations_tail;
                while (!synch(dir, tok.token_type))
                        tok = get_token();
        }
}

static struct Decoration type_call()
{
        int arrayLen;
        int ok = 0;
        switch(tok.token_type) {
        case STANDARD_TYPE:
                return standard_type_call();
        case ARRAY: // REVIEW: Not sure about the
logic of array type processing
                match(ARRAY);
                match(BR_OPEN);
                struct Token num1 = tok;
                match(NUM);
                match(TWO_DOT);
                struct Token num2 = tok;
                match(NUM);
                match(BR_CLOSE);
                if (num1.token_type == NUM &&
num2.token_type == NUM) {
                        if (num1.attribute.attribute
== REAL || num2.attribute.attribute == REAL) {
                                fprintf(lfp,
"SEMERR:   Atempt to use real number for array
length.\n");
                        } else if
(num1.attribute.attribute == INTEGER &&
num2.attribute.attribute == INTEGER) {
                                arrayLen =
atoi(num2.lexeme) - atoi(num1.lexeme) + 1;
                                ok = 1;
                        } else {
                                fprintf(lfp,
"SEMERR:   Unrecognized input for array length.\n");
                        }
                } else if (num1.token_type != LEXERR
&& num2.token_type != LEXERR) {
                        fprintf(lfp, "SEMERR:
Attempt to use non-number for array length.\n");
                }
                match(OF);
                struct Decoration std_type =
standard_type_call();
                if (ok) {
                        int width = arrayLen *
std_type.width;
                        if (std_type.type == INT)
                                return
make_decoration(AINT, width);
                        else if (std_type.type ==
REAL_TYPE)
                                return
make_decoration(AREAL, width);
                        else {
                                fprintf(lfp,
"SEMERR:   Unexpected type for array.\n");
                                return
make_type_decoration(ERR);
                        }
                } else {
                        return
make_type_decoration(ERR);
                }
        default:
                synerr("'integer', 'real' or
'array'", tok.lexeme);
                enum Derivation dir = type;
                while (!synch(dir, tok.token_type))
                        tok = get_token();
                return make_type_decoration(ERR);
        }
}

static struct Decoration standard_type_call()
{
```

```
        int attribute;
        switch(tok.token_type) {
        case STANDARD_TYPE:
                attribute = tok.attribute.attribute;
                match(STANDARD_TYPE);
                if (attribute == 1) {
                        return make_decoration(INT,
4);
                } else {
                        return
make_decoration(REAL_TYPE, 8);
                }
        default:
                synerr("'integer' or 'real'",
tok.lexeme);
                enum Derivation dir = standard_type;
                while (!synch(dir, tok.token_type))
                        tok = get_token();
                return make_type_decoration(ERR);
        }
}

static void sub_declarations_call()
{
        if (tok.token_type == PROCEDURE) {
                sub_declaration_call();
                pop_scope_stack();
                match(SEMI);
                sub_declarations_tail_call();
        } else {
                synerr("'procedure'", tok.lexeme);
                enum Derivation dir =
sub_declarations;
                while (!synch(dir, tok.token_type))
                        tok = get_token();
        }
}

static void sub_declarations_tail_call()
{
        switch(tok.token_type) {
        case PROCEDURE:
                sub_declaration_call();
                pop_scope_stack();
                match(SEMI);
                sub_declarations_tail_call();
                break;
        case BEGIN:
                break;
        default:
                synerr("'procedure' or 'begin'",
tok.lexeme);
                enum Derivation dir =
sub_declarations_tail;
                while (!synch(dir, tok.token_type))
                        tok = get_token();
        }
}

static void sub_declaration_call()
{
        if (tok.token_type == PROCEDURE) {
                sub_head_call();
                enter_num_params(counter);
                sub_declaration_tail_call();
        } else {
                synerr("'procedure'", tok.lexeme);
                enum Derivation dir =
sub_declaration;
                while (!synch(dir, tok.token_type))
                        tok = get_token();
        }
}

static void sub_declaration_tail_call()
{
        switch(tok.token_type) {
        case VAR:
                declarations_call();
                sub_declaration_tail_tail_call();
                break;
        case PROCEDURE:
                sub_declarations_call();
                compound_statement_call();
                break;
        case BEGIN:
                compound_statement_call();
                break;
        default:
                synerr("'var', 'procedure', or
'begin'", tok.lexeme);
                enum Derivation dir =
sub_declaration_tail;
                while (!synch(dir, tok.token_type))
                        tok = get_token();
        }
}

static void sub_declaration_tail_tail_call()
{
        switch(tok.token_type) {
        case PROCEDURE:
                sub_declarations_call();
                compound_statement_call();
                break;
        case BEGIN:
                compound_statement_call();
                break;
        default:
                synerr("'procedure' or 'begin'",
tok.lexeme);
                enum Derivation dir =
sub_declaration_tail_tail;
                while (!synch(dir, tok.token_type))
                        tok = get_token();
        }
}

static void sub_head_call()
{
        if (tok.token_type == PROCEDURE) {
                offset = 0;
                counter = 0;
                match(PROCEDURE);
                struct Token id_tok = tok;
                match(ID);
                check_add_green_node(id_tok.lexeme,
PROC);
                sub_head_tail_call();
        } else {
                synerr("'procedure'", tok.lexeme);
                enum Derivation dir = sub_head;
                while (!synch(dir, tok.token_type))
                        tok = get_token();
        }
}

static void sub_head_tail_call()
{
        switch(tok.token_type) {
        case PAREN_OPEN:
                arguments_call();
                match(SEMI);
                break;
        case SEMI:
                match(SEMI);
                break;
        default:
                synerr("'(' or ';'", tok.lexeme);
                enum Derivation dir = sub_head_tail;
                while (!synch(dir, tok.token_type))
                        tok = get_token();
        }
}

static void arguments_call()
{
        if (tok.token_type == PAREN_OPEN) {
                match(PAREN_OPEN);
                parameter_list_call();
                match(PAREN_CLOSE);
        } else {
```

```
                synerr("'procedure'", tok.lexeme);
                enum Derivation dir = sub_head;
                while (!synch(dir, tok.token_type))
                        tok = get_token();
        }
}

static void parameter_list_call()
{
        if (tok.token_type == ID) {
                struct Token id_tok = tok;
                match(ID);
                match(COLON);
                struct Decoration type =
type_call();
                check_add_blue_node(id_tok.lexeme,
make_param(type.type), offset);
                offset = offset + type.width;
                counter++;
                parameter_list_tail_call();
        } else {
                synerr("'id'", tok.lexeme);
                enum Derivation dir =
parameter_list;
                while (!synch(dir, tok.token_type))
                        tok = get_token();
        }
}

static void parameter_list_tail_call()
{
        switch(tok.token_type) {
        case SEMI:
                match(SEMI);
                struct Token id_tok = tok;
                match(ID);
                match(COLON);
                struct Decoration type =
type_call();
                check_add_blue_node(id_tok.lexeme,
make_param(type.type), offset);
                offset = offset + type.width;
                counter++;
                parameter_list_tail_call();
                break;
        case PAREN_CLOSE:
                break;
        default:
                synerr("';' or ')'", tok.lexeme);
                enum Derivation dir =
parameter_list_tail;
                while (!synch(dir, tok.token_type))
                        tok = get_token();
        }
}

static void compound_statement_call()
{
        if (tok.token_type == BEGIN) {
                match(BEGIN);
                compound_statement_tail_call();
        } else {
                synerr("'begin'", tok.lexeme);
                enum Derivation dir =
compound_statement;
                while (!synch(dir, tok.token_type))
                        tok = get_token();
        }
}

static void compound_statement_tail_call()
{
        switch(tok.token_type) {
        case ID:
        case CALL:
        case BEGIN:
        case IF:
        case WHILE:
                optional_statements_call();
                match(END);
                break;
        case END:
                match(END);
                break;
        default:
                synerr("'id', 'call', 'begin', 'if',
'while', or 'end'", tok.lexeme);
                enum Derivation dir =
compound_statement_tail;
                while (!synch(dir, tok.token_type))
                        tok = get_token();
        }
}

static void optional_statements_call()
{
        switch(tok.token_type) {
        case ID:
        case CALL:
        case BEGIN:
        case IF:
        case WHILE:
                statement_list_call();
                break;
        default:
                synerr("'id', 'call', 'begin', 'if',
or 'while'", tok.lexeme);
                enum Derivation dir =
optional_statements;
                while (!synch(dir, tok.token_type))
                        tok = get_token();
        }
}

static void statement_list_call()
{
        switch(tok.token_type) {
        case ID:
        case CALL:
        case BEGIN:
        case IF:
        case WHILE:
                statement_call();
                statement_list_tail_call();
                break;
        default:
                synerr("'id', 'call', 'begin', 'if',
or 'while'", tok.lexeme);
                enum Derivation dir =
statement_list;
                while (!synch(dir, tok.token_type))
                        tok = get_token();
        }
}

static void statement_list_tail_call()
{
        switch(tok.token_type) {
        case SEMI:
                match(SEMI);
                statement_call();
                statement_list_tail_call();
                break;
        case END:
                break;
        default:
                synerr("';' or 'end'", tok.lexeme);
                enum Derivation dir =
statement_list_tail;
                while (!synch(dir, tok.token_type))
                        tok = get_token();
        }
}

static void statement_call()
{
        struct Decoration exp_dec;
        switch(tok.token_type) {
        case ID:
                variable_call();
                match(ASSIGN);
                expression_call();
```

```
                break;
        case CALL:
                procedure_statement_call();
                break;
        case BEGIN:
                compound_statement_call();
                break;
        case IF:
                match(IF);
                exp_dec = expression_call();
                if (exp_dec.type != BOOL &&
exp_dec.type != ERR) {
                        fprintf(lfp, "SEMERR:
Attempt to use non-boolean expression in if
statement.\n");
                }
                match(THEN);
                statement_call();
                statement_tail_call();
                break;
        case WHILE:
                match(WHILE);
                exp_dec = expression_call();
                if (exp_dec.type != BOOL &&
exp_dec.type != ERR) {
                        fprintf(lfp, "SEMERR:
Attempt to use non-boolean expression in while loop.
\n");
                }
                match(DO);
                statement_call();
                break;
        default:
                synerr("'id', 'call', 'begin', 'if',
or 'while'", tok.lexeme);
                enum Derivation dir = statement;
                while (!synch(dir, tok.token_type))
                        tok = get_token();
        }
}

static void statement_tail_call()
{
        switch(tok.token_type) {
        case ELSE:
                match(ELSE);
                statement_call();
                break;
        case SEMI:
        case END:
                break;
        default:
                synerr("'else', ';', or 'end'",
tok.lexeme);
                enum Derivation dir =
statement_tail;
                while (!synch(dir, tok.token_type))
                        tok = get_token();
        }
}

static struct Decoration variable_call()
{
        if (tok.token_type == ID) {
                struct Token id_tok = tok;
                match(ID);
                enum Type type =
get_type(id_tok.lexeme);
                return
variable_tail_call(make_type_decoration(type));
        } else {
                synerr("'id'", tok.lexeme);
                enum Derivation dir = variable;
                while (!synch(dir, tok.token_type))
                        tok = get_token();
                return make_type_decoration(ERR);
        }
}

static struct Decoration variable_tail_call(struct
Decoration inherited)
{
        switch(tok.token_type) {
        case BR_OPEN:
                match(BR_OPEN);
                struct Decoration exp_dec =
expression_call();
                enum Type exp_type = exp_dec.type;
                match(BR_CLOSE);
                if (exp_type == INT &&
(inherited.type == AINT || inherited.type ==
PP_AINT)) {
                        return
make_type_decoration(INT);
                } else if (exp_type == INT &&
(inherited.type == AREAL || inherited.type ==
PP_AREAL)) {
                        return
make_type_decoration(REAL_TYPE);
                } else if (exp_dec.type == ERR ||
inherited.type == ERR) {
                        return
make_type_decoration(ERR);
                } else {
                        fprintf(lfp, "SEMERR:
Incorrect array access.\n");
                        return
make_type_decoration(ERR);
                }
        case ASSIGN:
                return inherited;
        default:
                synerr("'[' or '='", tok.lexeme);
                enum Derivation dir = variable_tail;
                while (!synch(dir, tok.token_type))
                        tok = get_token();
                return make_type_decoration(ERR);
        }
}

static void procedure_statement_call()
{
        if (tok.token_type == CALL) {
                match(CALL);
                struct Token id_tok = tok;
                match(ID);
                enum Type id_type =
get_type(id_tok.lexeme);
                if (id_type == PROC) {

procedure_statement_tail_call(get_proc_pointer(id_to
k.lexeme));
                } else if (id_type == ERR) {

procedure_statement_tail_call(NULL);
                } else {
                        fprintf(lfp, "SEMERR:   '%s'
is not a procedure.\n", id_tok.lexeme);

procedure_statement_tail_call(NULL);
                }
        } else {
                synerr("'call'", tok.lexeme);
                enum Derivation dir =
procedure_statement;
                while (!synch(dir, tok.token_type))
                        tok = get_token();
        }
}

static void procedure_statement_tail_call(struct
Symbol *proc)
{
        switch(tok.token_type) {
        case PAREN_OPEN:
                match(PAREN_OPEN);
                if (proc == NULL) {
                        expression_list_call(0,
NULL);
                } else {
```

```
                        int net_params =
expression_list_call(proc -> num_parms, proc ->
content);
                        if (net_params != 0) {
                                fprintf(lfp,
"SEMERR:   Incorrect number of parameters in
procedure call.\n");
                        }
                }
                match(PAREN_CLOSE);
                break;
        case SEMI:
        case ELSE:
        case END:
                if (proc == NULL) {
                        return;
                } else if (proc -> num_parms != 0) {
                        fprintf(lfp, "SEMERR:
Incorrect number of parameters in procedure call.
\n");
                }
                break;
        default:
                synerr("'(', ';', 'else', or 'end'",
tok.lexeme);
                enum Derivation dir =
procedure_statement_tail;
                while (!synch(dir, tok.token_type))
                        tok = get_token();
        }
}

static int expression_list_call(int num_parms,
struct Symbol *param)
{
        struct Decoration exp_dec;
        switch(tok.token_type) {
        case ID:
        case NUM:
        case PAREN_OPEN:
        case NOT:
        case ADDOP:
                exp_dec = expression_call();
                if (param == NULL || num_parms <= 0)
{
                        return
expression_list_tail_call(num_parms - 1, param);
                } else if (!
verify_param(exp_dec.type, param -> type)) {
                        fprintf(lfp, "SEMERR:
Incorrect type for parameter.\n");
                }
                return
expression_list_tail_call(num_parms - 1, param ->
next);
        default:
                synerr("'id', 'num', '(', 'not',
'+', or '-'", tok.lexeme);
                enum Derivation dir =
expression_list;
                while (!synch(dir, tok.token_type))
                        tok = get_token();
                return 0;
        }
}

static int expression_list_tail_call(int num_parms,
struct Symbol *param)
{
        switch(tok.token_type) {
        case COMMA:
                match(COMMA);
                struct Decoration exp_dec =
expression_call();
                if (param == NULL || num_parms <= 0)
{
                        return
expression_list_tail_call(num_parms - 1, param);
                } else if (!
verify_param(exp_dec.type, param -> type)) {
                        fprintf(lfp, "SEMERR:
Incorrect type for parameter.\n");
                }
                return
expression_list_tail_call(num_parms - 1, param ->
next);
        case PAREN_CLOSE:
                return num_parms;
        default:
                synerr("',' or ')'", tok.lexeme);
                enum Derivation dir =
expression_list_tail;
                while (!synch(dir, tok.token_type))
                        tok = get_token();
                return 0;
        }
}

static struct Decoration expression_call()
{
        struct Decoration exp_type;

        switch(tok.token_type) {
        case ID:
        case NUM:
        case PAREN_OPEN:
        case NOT:
        case ADDOP:
                exp_type = simple_expression_call();
                return
expression_tail_call(exp_type);
        default:
                synerr("'id', 'num', '(', 'not',
'+', or '-'", tok.lexeme);
                enum Derivation dir = expression;
                while (!synch(dir, tok.token_type))
                        tok = get_token();
                return make_type_decoration(ERR);
        }
}

static struct Decoration expression_tail_call(struct
Decoration inherited)
{
        switch(tok.token_type) {
        case RELOP:
                match(RELOP);
                struct Decoration exp_type =
simple_expression_call();
                if
(num_type_agreement(exp_type.type, inherited.type))
{
                        return
make_type_decoration(BOOL);
                } else if (exp_type.type == ERR ||
inherited.type == ERR) {
                        return
make_type_decoration(ERR);
                } else {
                        fprintf(lfp, "SEMERR:
Incompatible types for relop operation.\n");
                        return
make_type_decoration(ERR);
                }
        case THEN:
        case DO:
        case BR_CLOSE:
        case COMMA:
        case PAREN_CLOSE:
        case SEMI:
        case ELSE:
        case END:
                return inherited;
        default:
                synerr("'>', '<', '<=' '>=', '<>',
'=', 'then', 'do', ']', ',', ')', ';', 'else', or
'end'", tok.lexeme);
                enum Derivation dir =
expression_tail;
                while (!synch(dir, tok.token_type))
                        tok = get_token();
```

```
                return make_type_decoration(ERR);
        }
}

static struct Decoration simple_expression_call()
{
        struct Decoration t_type;
        switch(tok.token_type) {
        case ID:
        case NUM:
        case PAREN_OPEN:
        case NOT:
                t_type = term_call();
                return
simple_expression_tail_call(t_type);
        case ADDOP:
                sign_call();
                t_type = term_call();
                if (t_type.type != INT &&
t_type.type != REAL_TYPE) {
                        fprintf(lfp, "SEMERR:
Attempt to add sign to unsigned type.\n");
                        t_type =
make_type_decoration(ERR);
                }
                return
simple_expression_tail_call(t_type);
        default:
                synerr("'id', 'num', '(' 'not', '+',
or '-'", tok.lexeme);
                enum Derivation dir =
simple_expression;
                while (!synch(dir, tok.token_type))
                        tok = get_token();
                return make_type_decoration(ERR);
        }
}

static struct Decoration
simple_expression_tail_call(struct Decoration
inherited)
{
        int op;
        switch(tok.token_type) {
        case ADDOP:
                op = tok.attribute.attribute;
                match(ADDOP);
                struct Decoration t_type =
term_call();
                switch (op) {
                case ADD:
                case SUB:
                        if
(integer_agreement(t_type.type, inherited.type)) {
                                return
make_decoration(INT, 4);
                        } else if
(real_agreement(t_type.type, inherited.type)) {
                                return
make_decoration(REAL, 8);
                        } else if (t_type.type ==
ERR || inherited.type == ERR) {
                                return
make_type_decoration(ERR);
                        } else {
                                fprintf(lfp,
"SEMERR:   Incompatible types for addop operation.
\n");
                                return
make_type_decoration(ERR);
                        }
                case OR:
                        if (t_type.type == BOOL &&
inherited.type == BOOL) {
                                return t_type;
                        } else if (t_type.type ==
ERR || inherited.type == ERR) {
                                return
make_type_decoration(ERR);
                        } else {
                                fprintf(lfp,
"SEMERR:  Incompatible types for or operation.\n");
                                return
make_type_decoration(ERR);
                        }
                default:
                        fprintf(lfp, "SEMERR:
Unrecognized addop.\n");
                        return
make_type_decoration(ERR);
                }
        case RELOP:
        case THEN:
        case DO:
        case BR_CLOSE:
        case COMMA:
        case PAREN_CLOSE:
        case SEMI:
        case ELSE:
        case END:
                return inherited;
        default:
                synerr("'+', '-', 'or', '>', '<',
'<=' '>=', '<>', '=', 'then', 'do', ']', ',', ')',
';', 'else', or 'end'", tok.lexeme);
                enum Derivation dir =
simple_expression_tail;
                while (!synch(dir, tok.token_type))
                        tok = get_token();
                return make_type_decoration(ERR);
        }
}

static struct Decoration term_call()
{
        struct Decoration fac_type;
        switch(tok.token_type) {
        case ID:
        case NUM:
        case PAREN_OPEN:
        case NOT:
                fac_type = factor_call();
                return term_tail_call(fac_type);
        default:
                synerr("'id', 'num' '(', or 'not'",
tok.lexeme);
                enum Derivation dir = term;
                while (!synch(dir, tok.token_type))
                        tok = get_token();
                return make_type_decoration(ERR);
        }
}

static struct Decoration term_tail_call(struct
Decoration inherited)
{
        int op;
        switch(tok.token_type) {
        case MULOP:
                op = tok.attribute.attribute;
                match(MULOP);
                struct Decoration fac_type =
factor_call();
                switch (op) {
                case MULT:
                        if
(integer_agreement(fac_type.type, inherited.type)) {
                                return
make_decoration(INT, 4);
                        } else if
(real_agreement(fac_type.type, inherited.type)) {
                                return
make_decoration(REAL, 8);
                        } else if (fac_type.type ==
ERR || inherited.type == ERR) {
                                return
make_type_decoration(ERR);
                        } else {
                                fprintf(lfp,
"SEMERR:   Incompatible types for mult operation.
\n");
```

```
                                return
make_type_decoration(ERR);
                        }
                case DIVIDE:
                case DIV:
                        if
(integer_agreement(fac_type.type, inherited.type)) {
                                return
make_decoration(INT, 4);
                        } else if
(real_agreement(fac_type.type, inherited.type)) {
                                return
make_decoration(REAL, 8);
                        } else if (fac_type.type ==
ERR || inherited.type == ERR) {
                                return
make_type_decoration(ERR);
                        } else {
                                fprintf(lfp,
"SEMERR:   Incompatible types for div operation.
\n");
                                return
make_type_decoration(ERR);
                        }
                case MOD:
                        if
(integer_agreement(fac_type.type, inherited.type)) {
                                return fac_type;
                        } else if (fac_type.type ==
ERR || inherited.type == ERR) {
                                return
make_type_decoration(ERR);
                        } else {
                                fprintf(lfp,
"SEMERR:   Incompatible types for mod operation.
\n");
                                return
make_type_decoration(ERR);
                        }
                case AND:
                        if ((fac_type.type == BOOL
&& inherited.type == BOOL)) {
                                return fac_type;
                        } else if (fac_type.type ==
ERR || inherited.type == ERR) {
                                return
make_type_decoration(ERR);
                        } else {
                                fprintf(lfp,
"SEMERR:   Incompatible types for and operation.
\n");
                                return
make_type_decoration(ERR);
                        }
                default:
                        fprintf(lfp, "SEMERR:
Unrecognized mulop.\n");
                        return
make_type_decoration(ERR);
                }
        case ADDOP:
        case RELOP:
        case THEN:
        case DO:
        case BR_CLOSE:
        case COMMA:
        case PAREN_CLOSE:
        case SEMI:
        case ELSE:
        case END:
                return inherited;
        default:
                synerr("'*', '/', 'and', '+', '-',
'or', '>', '<', '<=' '>=', '<>', '=', 'then', 'do',
']', ',', ')', ';', 'else', or 'end'", tok.lexeme);
                enum Derivation dir = term_tail;
                while (!synch(dir, tok.token_type))
                        tok = get_token();
                return make_type_decoration(ERR);
        }
}
```

```
static struct Decoration factor_call()
{
        struct Token id_tok;
        struct Decoration num_type;
        switch(tok.token_type) {
        case ID:
                id_tok = tok;
                match(ID);
                enum Type lex_type =
get_type(id_tok.lexeme);
                return
factor_tail_call(make_type_decoration(lex_type));
        case NUM:
                if (tok.attribute.attribute == 1)
                        num_type =
make_type_decoration(INT);
                else {
                        num_type =
make_type_decoration(REAL_TYPE);
                }
                match(NUM);
                return num_type;
        case PAREN_OPEN:
                match(PAREN_OPEN);
                struct Decoration exp_type =
expression_call();
                match(PAREN_CLOSE);
                return exp_type;
        case NOT:
                match(NOT);
                struct Decoration fac_type =
factor_call();
                if (fac_type.type == BOOL) {
                        return fac_type;
                } else if (fac_type.type == ERR) {
                        return fac_type;
                } else {
                        fprintf(lfp, "SEMERR:
'not' used with non-boolean expression.\n");
                        return
make_type_decoration(ERR);
                }
        default:
                synerr("'id', 'num' '(', or 'not'",
tok.lexeme);
                enum Derivation dir = factor;
                while (!synch(dir, tok.token_type))
                        tok = get_token();
                return make_type_decoration(ERR);
        }
}

static struct Decoration factor_tail_call(struct
Decoration inherited)
{
        switch(tok.token_type) {
        case BR_OPEN:
                match(BR_OPEN);
                struct Decoration exp_dec =
expression_call();
                enum Type exp_type = exp_dec.type;
                match(BR_CLOSE);
                if (exp_type == INT &&
inherited.type == AINT) {
                        return
make_type_decoration(INT);
                } else if (exp_type == INT &&
inherited.type == AREAL) {
                        return
make_type_decoration(REAL_TYPE);
                } else if (inherited.type != AINT
                                && inherited.type !=
AREAL
                                && inherited.type !=
ERR) {
                        fprintf(lfp, "SEMERR:
Array access of non-array object\n");
                        return
make_type_decoration(ERR);
```

```
                } else if (exp_type != INT &&
exp_type != ERR) {
                        fprintf(lfp, "SEMERR:
Array reference is not an integer.\n");
                        return
make_type_decoration(ERR);
                } else {
                        return
make_type_decoration(ERR);
                }
        case MULOP:
        case ADDOP:
        case RELOP:
        case THEN:
        case DO:
        case BR_CLOSE:
        case COMMA:
        case PAREN_CLOSE:
        case SEMI:
        case ELSE:
        case END:
                return inherited;
        default:
                synerr("'[', '*', '/', 'and', '+',
'-', 'or', '>', '<', '<=' '>=', '<>', '=', 'then',
'do', ']', ',', ')', ';', 'else', or 'end'",
tok.lexeme);
                enum Derivation dir = factor_tail;
                while (!synch(dir, tok.token_type))
                        tok = get_token();
                return make_type_decoration(ERR);
        }
}

static void sign_call()
{
        if (tok.token_type == ADDOP &&
                        (tok.attribute.attribute ==
ADD || tok.attribute.attribute == SUB)) {
                match(ADDOP);
        } else {
                synerr("'+' or '-'", tok.lexeme);
                enum Derivation dir = sign;
                while (!synch(dir, tok.token_type))
                        tok = get_token();
        }
}
```

PARSER.H

```
#ifndef PARSER_H
#define PARSER_H

#include "types.h"

struct Decoration {
        enum Type type;
        int width;
};

void program_call();

#endif
```

SYMBOLS.C

```
#include "symbols.h"
#include "analyzer.h"
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

struct Symbol *global_sym_table; // REVIEW: Remove
variable
struct Reserved_Word *reserved_word_table;
struct Symbol *eye;
struct Symbol *forward_eye;
static struct SymbolStack *scope_stack;
static int is_green_node(struct Symbol node);
static void print_symbol_line(FILE *out, int
num_levels, struct Symbol *current);
static void print_bars(FILE *out, int num);
static void print_temp_line(FILE *out, int num);

/**
 * REVIEW: Remove this function.
 * Adds a symbol to the symbol table if it is not
already present. If the symbol
 * is already present, returns a pointer to that
Symbol.
 *
 * Arguments: word -> literal symbol to be added to
the table.
 *
 * Returns: A pointer to the symbol in the table.
 */
struct Symbol * add_symbol(char word[])
{
        struct Symbol *current = global_sym_table;

        while (current -> next != NULL) {
                if (strcmp(current -> word, word) ==
0)
                        return current;
                current = current -> next;
        }

        current -> next = malloc(sizeof(struct
Symbol));
        strcpy(current -> word, word);
        current -> next -> next = NULL;

        return current;
}

struct Symbol * check_add_green_node(char lex[],
enum Type type)
{
        struct Symbol *current = eye;
        if (current != NULL) {
                while (current -> previous != NULL)
{
                        if (is_green_node(*current))
{
                                if (strcmp(lex,
current -> word) == 0) {
                                        fprintf(lfp,
"SEMERR:   Reuse of scope id '%s'\n", lex);
                                        return NULL;
                                }
                        }
                        current = current ->
previous;
                }
        }


        // No name conflicts
        strcpy(forward_eye -> word, lex);
        forward_eye -> type = type;
        forward_eye -> offset = 0;
        forward_eye -> previous = eye;
        forward_eye -> next = malloc(sizeof(struct
Symbol));
        forward_eye -> content =
malloc(sizeof(struct Symbol));

        eye = forward_eye;
        forward_eye = eye -> content;

        // Add scope to stack
        struct SymbolStack *push =
malloc(sizeof(struct SymbolStack));
        push -> symbol = eye;
        push -> previous = scope_stack;
        scope_stack = push;

        return eye;
}
```

```
void check_add_blue_node(char lex[], enum Type type,
int offset)
{
        struct Symbol *current = eye;
        while(!is_green_node(*current)) {
                if (strcmp(lex, current -> word) ==
0) {
                        fprintf(lfp, "SEMERR:
Reuse of id '%s'\n", lex);
                        return;
                } else {
                        current = current ->
previous;
                }
        }

        strcpy(forward_eye -> word, lex);
        forward_eye -> type = type;
        forward_eye -> offset = offset;
        forward_eye -> previous = eye;
        forward_eye -> next = malloc(sizeof(struct
Symbol));

        eye = forward_eye;
        forward_eye = eye -> next;
}

void pop_scope_stack()
{
        eye = scope_stack -> symbol;
        forward_eye = eye -> next;
        scope_stack = scope_stack -> previous;
}

void enter_num_params(int counter)
{
        scope_stack -> symbol -> num_parms =
counter;
}

static int is_green_node(struct Symbol node)
{
        return node.type == PROC || node.type ==
PG_NAME;
}

enum Type get_type(char lex[])
{
        struct Symbol *current = eye;
        while(current -> previous != NULL) {
                if (strcmp(current -> word, lex) ==
0)
                        return current -> type;
                else
                        current = current ->
previous;
        }
        fprintf(lfp, "SEMERR:   Use of undeclared
identifier: '%s'\n", lex);
        return ERR;
}

struct Symbol * get_proc_pointer(char lexeme[])
{

        struct Symbol *current = eye;

        while (current -> previous != NULL) {
                if (is_green_node(*current) &&
strcmp(current -> word, lexeme) == 0)
                        return current;
                current = current -> previous;
        }

        fprintf(lfp, "SEMERR:   Did not find pointer
in stack.\n");
        return NULL;
}

void print_symbol_table(FILE *out)
{
        struct Symbol *current = eye;
        while(current -> previous != NULL) {
                current = current -> previous;
        }
        print_symbol_line(out, 0, current);
}

static void print_symbol_line(FILE *out, int
num_levels, struct Symbol *current)
{

        if (is_green_node(*current)) {
                print_bars(out, num_levels);
                fprintf(out, "* SCOPE: {id: %s,
type: %s, num-params: %d}\n", current -> word,
get_type_name(current -> type), current ->
num_parms);
                print_temp_line(out, num_levels +
1);
                print_symbol_line(out, num_levels +
1, current -> content);
                print_symbol_line(out, num_levels,
current -> next);
        } else if (current -> next != NULL) {
                print_bars(out, num_levels);
                fprintf(out, "* VAR: {id: %s, type:
%s, offset: %d}\n", current -> word,
get_type_name(current -> type), current -> offset);
                print_symbol_line(out, num_levels,
current -> next);
        }
}

static void print_bars(FILE *out, int num)
{
        for (int i = 0; i < num; i++)
                fprintf(out, "| ");
}

static void print_temp_line(FILE *out, int num)
{
        if (num > 0)
                fprintf(out, "|");

        for (int i = 1; i < num; i++)
                fprintf(out, " |");

        fprintf(out, "\\\n");
}

/*
 * Adds a reserved word to the reserved word table.
 *
 * Arguments: word -> Literal of the word to be
added.
 *            type -> Token type associated with the
reserved word.
 *            attr -> Token attribute associated
with the reserved word.
 *
 * Returns: A pointer to the reserved word added to
the table.
 */
struct Reserved_Word * add_reserved_word(char
word[], int type, int attr)
{
        struct Reserved_Word *current =
reserved_word_table;

        while (current -> next != NULL) {
                current = current -> next;
        }

        current -> next = malloc(sizeof(struct
Reserved_Word));
        strcpy(current -> word, word);
        current -> token_type = type;
        current -> attribute = attr;
        current -> next -> next = NULL;
```

```
        return current -> next;
}

/*
 * Checks if a given word is in the reserved word
table.
 *
 * Arguments: word -> Literal of the word to be
checked.
 *
 * Returns: The token associated with the reserved
word. If no reserved word is
 *          found, returns a null Optional_Token.
 */
union Optional_Token check_reserved_words(char
word[])
{
        struct Reserved_Word *current =
reserved_word_table;

        do {
                if (strcmp(current -> word, word) ==
0) {
                        return make_optional(word,
                                        current ->
token_type,
                                        current ->
attribute,
                                        NULL);
                }
                current = current -> next;
        } while (current -> next != NULL);

        return null_optional();
}

/*
 * Initializes the reserved word table from the
RESERVED_WORDS file.
 *
 * Arguments: rfp -> Pointer to the reserved word
file.
 *
 * Returns: A pointer to the reserved word table.
 */
struct Reserved_Word *
initialize_reserved_words(FILE *rfp)
{
        reserved_word_table = malloc(sizeof(struct
Reserved_Word));
        reserved_word_table -> next = NULL;
        char buff[80];
        fgets(buff, 80, rfp);

        while(!feof(rfp)) {
                if (buff[0] != '\n') {
                        char word[11];
                        int type;
                        int attr;

                        sscanf(buff, "%s %d %d",
word, &type, &attr);
                        add_reserved_word(word,
type, attr);
                }
                fgets(buff, 80, rfp);
        }
        return reserved_word_table;
}
```

SYMBOLS.H

```
#ifndef SYMBOLS_H
#define SYMBOLS_H

#include "machines.h"
#include "types.h"
#include <stdio.h>

/**
 * A Symbol for an ID in the symbol table.
 * REVIEW: Needs Documentaton
 * Fields: word -> Literal of the lexeme symbol.
 *         ptr -> Pointer to the next symbol in the
table.
 */
struct Symbol {
        char word[11];
        enum Type type;
        int offset;
        int num_parms;
        struct Symbol *content;
        struct Symbol *previous;
        struct Symbol *next;
};

/*
 * Contains a reserved word from the reserved word
table.
 *
 * Fields: word -> Literal of the reserved word.
 *         token_type -> Integer of token type
assoicated with the word.
 *         attribute -> Integer of attribute
associated with the word.
 *         next -> Pointer to the next reserved word
in the table.
 */
struct Reserved_Word {
        char word[11];
        int token_type;
        int attribute;
        struct Reserved_Word *next;
};

struct SymbolStack {
        struct Symbol *symbol;
        struct SymbolStack *previous;
};

/*
 * Global symbol table. Pointer to first item in the
linked list.
 */
extern struct Symbol *global_sym_table;

extern struct Symbol *forward_eye;

extern struct Symbol *eye;

/*
 * Rerved word table. Pointer fo first item in the
linked list.
 */
extern struct Reserved_Word *reserved_word_table;

/*
 * Adds a symbol to the symbol table if it is not
already present. If the symbol
 * is already present, returns a pointer to that
Symbol.
 *
 * Arguments: word -> literal symbol to be added to
the table.
 *
 * Returns: A pointer to the symbol in the table.
 */
struct Symbol * add_symbol(char word[]);

struct Symbol * check_add_green_node(char lex[],
enum Type type);

void check_add_blue_node(char lex[], enum Type type,
int offset);

void pop_scope_stack();

void enter_num_params(int counter);

enum Type get_type(char lex[]);
```

```
struct Symbol * get_proc_pointer(char lexeme[]);

void print_symbol_table(FILE *out);

/*
 * Checks if a given word is in the reserved word
table.
 *
 * Arguments: word -> Literal of the word to be
checked.
 *
 * Returns: The token associated with the reserved
word. If no reserved word is
 *          found, returns a null Optional_Token.
 */
union Optional_Token check_reserved_words(char
word[]);

/*
 * Initializes the reserved word table from the
RESERVED_WORDS file.
 *
 * Arguments: rfp -> Pointer to the reserved word
file.
 *
 * Returns: A pointer to the reserved word table.
 */
struct Reserved_Word *
initialize_reserved_words(FILE *rfp);

#endif
```

SYNCH_SET.C

```
#include "synch_set.h"

#include "word_defs.h"

int synch(enum Derivation dir, int token_type)
{
        if (token_type == EOF_TYPE)
                return 1;

        switch(dir) {
        case id_list:
        case id_list_tail:
        case parameter_list:
        case parameter_list_tail:
        case expression_list:
        case expression_list_tail:
                return token_type == PAREN_CLOSE;
        case declarations:
        case declarations_tail:
                return token_type == PROCEDURE ||
token_type == BEGIN;
        case type:
        case standard_type:
                return token_type == SEMI ||
token_type == PAREN_CLOSE;
        case sub_declarations:
        case sub_declarations_tail:
                return token_type == BEGIN;
        case sub_declaration:
        case sub_declaration_tail:
        case sub_declaration_tail_tail:
        case arguments:
                return token_type == SEMI;
        case sub_head:
        case sub_head_tail:
                return token_type == VAR
                        || token_type == PROCEDURE
                        || token_type == BEGIN;
        case compound_statement:
        case compound_statement_tail:
                return token_type == DOT
                        || token_type == SEMI
                        || token_type == ELSE
                        || token_type == END;
        case optional_statements:
        case statement_list:
                return token_type == END;
        case statement_list_tail:
                return token_type == END;
        case statement:
        case statement_tail:
        case procedure_statement:
        case procedure_statement_tail:
                return token_type == SEMI
                        || token_type == ELSE
                        || token_type == END;
        case variable:
        case variable_tail:
                return token_type == ASSIGN;
        case expression:
        case expression_tail:
                return token_type == THEN
                        || token_type == DO
                        || token_type == BR_CLOSE
                        || token_type == COMMA
                        || token_type == PAREN_CLOSE
                        || token_type == SEMI
                        || token_type == ELSE
                        || token_type == END;
        case simple_expression:
        case simple_expression_tail:
                return token_type == RELOP
                        || token_type == THEN
                        || token_type == DO
                        || token_type == BR_CLOSE
                        || token_type == COMMA
                        || token_type == PAREN_CLOSE
                        || token_type == SEMI
                        || token_type == ELSE
                        || token_type == END;
        case term:
        case term_tail:
                return token_type == ADDOP
                        || token_type == RELOP
                        || token_type == THEN
                        || token_type == DO
                        || token_type == BR_CLOSE
                        || token_type == COMMA
                        || token_type == PAREN_CLOSE
                        || token_type == SEMI
                        || token_type == ELSE
                        || token_type == END;
        case factor:
        case factor_tail:
                return token_type == MULOP
                        || token_type == ADDOP
                        || token_type == RELOP
                        || token_type == THEN
                        || token_type == DO
                        || token_type == BR_CLOSE
                        || token_type == COMMA
                        || token_type == PAREN_CLOSE
                        || token_type == SEMI
                        || token_type == ELSE
                        || token_type == END;
        case sign:
                return token_type == ID
                        || token_type == NUM
                        || token_type == PAREN_OPEN
                        || token_type == NOT;
        default:
                return 0;
        }
}
```

SYNCH_SET.H

```
#ifndef SYNCH_SET_H
#define SYNCH_SET_H

enum Derivation
{
        program,
        program_tail,
        program_tail_tail,
        id_list,
        id_list_tail,
```

```
        declarations,
        declarations_tail,
        type,
        standard_type,
        sub_declarations,
        sub_declarations_tail,
        sub_declaration,
        sub_declaration_tail,
        sub_declaration_tail_tail,
        sub_head,
        sub_head_tail,
        arguments,
        parameter_list,
        parameter_list_tail,
        compound_statement,
        compound_statement_tail,
        optional_statements,
        statement_list,
        statement_list_tail,
        statement,
        statement_tail,
        variable,
        variable_tail,
        procedure_statement,
        procedure_statement_tail,
        expression_list,
        expression_list_tail,
        expression,
        expression_tail,
        simple_expression,
        simple_expression_tail,
        term,
        term_tail,
        factor,
        factor_tail,
        sign
};

int synch(enum Derivation dir, int token_type);

#endif
```

TYPES.C

```
#include "types.h"
#include "analyzer.h"
#include <stdio.h>

enum Type make_param(enum Type input)
{
        switch(input) {
        case INT:
        case PP_INT:
                return PP_INT;
        case REAL_TYPE:
        case PP_REAL:
                return PP_REAL;
        case AINT:
        case PP_AINT:
                return PP_AINT;
        case AREAL:
        case PP_AREAL:
                return PP_AREAL;
        default:
                fprintf(lfp, "SEMERR:   Unsuitable
type for parameter.\n");
                return input;
        }
}

const char* get_type_name(enum Type type)
{
   switch (type)
   {
      case INT: return "INT";
      case REAL_TYPE: return "REAL";
      case AINT: return "AINT";
      case AREAL: return "AREAL";
      case BOOL: return "BOOL";
      case PG_NAME: return "PGM_NAME";
      case PG_PARM: return "PGM_PARAM";
      case PROC: return "PROCEDURE";
      case PP_INT: return "PP_INT";
      case PP_REAL: return "PP_REAL";
      case PP_AINT: return "PP_AINT";
      case PP_AREAL: return "PP_AREAL";
      default: return "UNKNOWN TYPE";
   }
}

int num_type_agreement(enum Type first, enum Type
second)
{
        return integer_agreement(first, second) ||
real_agreement(first, second);
}

int integer_agreement(enum Type first, enum Type
second)
{
        return (first == INT || first == PP_INT)
                && (second == INT || second ==
PP_INT);
}

int real_agreement(enum Type first, enum Type
second)
{
        return (first == REAL_TYPE || first ==
PP_REAL)
                && (second == REAL_TYPE || second ==
PP_REAL);
}

int verify_param(enum Type input, enum Type
expected)
{
        return make_param(input) == expected;
}
```

TYPES.H

```
#ifndef TYPES_H
#define TYPES_H

enum Type
{
        INT,
        REAL_TYPE,
        AINT,
        AREAL,
        BOOL,
        PG_NAME,
        PG_PARM,
        PROC,
        ERR,
        PP_INT,
        PP_REAL,
        PP_AINT,
        PP_AREAL
};

enum Type make_param(enum Type input);

const char* get_type_name(enum Type type);

int num_type_agreement(enum Type first, enum Type
second);

int integer_agreement(enum Type first, enum Type
second);

int real_agreement(enum Type first, enum Type
second);

int verify_param(enum Type input, enum Type
expected);

#endif
```

WORD_DEFS.H

```
#ifndef WORD_DEFS_H
#define WORD_DEFS_H

// token types
#define PROGRAM 10
#define FUNCTION 11
#define PROCEDURE 12
#define BEGIN 13
#define END 14
#define IF 15
#define THEN 16
#define ELSE 17
#define WHILE 18
#define DO 19
#define NOT 20
#define ARRAY 21
#define OF 22
#define VAR 23
#define EOF_TYPE 24
#define CALL 25

#define SEMI 30
#define COMMA 31
#define PAREN_OPEN 32
#define PAREN_CLOSE 33
#define BR_OPEN 34
#define BR_CLOSE 35
#define COLON 36
#define ASSIGN 37
#define DOT 38
#define TWO_DOT 39

#define NUM 40
#define ID 50
#define MULOP 60
#define ADDOP 70
#define RELOP 80
#define STANDARD_TYPE 90
#define LEXERR 99

// Addops
#define ADD 1
#define SUB 2
#define OR 3

// Mulops
#define MULT 1
#define DIVIDE 2
#define DIV 3
#define MOD 4
#define AND 5

// Relops
#define LT 1
#define GT 2
#define LT_EQ 3
#define GT_EQ 4
#define EQ 5
#define NEQ 6

// Standard types
#define INTEGER 1
#define REAL 2
#define LONG_REAL 3

// Error Codes
#define UNRECOG_SYM 1
#define EXTRA_LONG_ID 2
#define EXTRA_LONG_INT 3
#define EXTRA_LONG_REAL 4
#define LEADING_ZEROES 5

#endif
```

RESERVED_WORDS

```
and 60 5
array 21 0
begin 13 0
div 60 3
do 19 0
else 17 0
end 14 0
function 11 0
if 15 0
integer 90 1
mod 60 4
not 20 0
of 22 0
or 70 3
procedure 12 0
program 10 0
real 90 2
then 16 0
var 23 0
while 18 0
call 25 0
```