CS 4013: Compiler Construction: Projects 3 & 4

Nate Beckemeyer

December 2016

Introduction

For Projects 3 and 4, I decorated the $LL(1)$ grammar created in project 2 to the static semantics of our modified Pascal language. Then, using the synthesized and inherited attributes, I folded the decorations into the recursive descent parser.

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

1 Methodology

By paying attention to when certain events should happen in the productions, I managed to modify the RDP to enforce the static semantics, such as type-checking. See the included L-Attributed Definiton for more information.

Grammar with Decorations

Return type (s)

Global
Synthesized
Inherited (or other semantic action)

bool
- true on succeed
- false otherwise

checkAdd GreenNode (id: lex, PGNNAME)
offset = 0

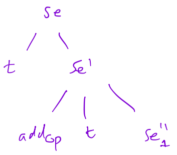
1.	void	*program*	→	**program id** (*identifier_list*) ; *declarations* *subprogram_declarations* *compound_statement* .	checkAdd GreenNode (id: lex, PGNNAME) offset = 0
2.1.	void	*identifier_list*	→	**id** *identifier_list′*	checkAdd BlueNode (id: lex, PGMNAME)
2.2.1.	void	*identifier_list′*	→	, **id** *identifier_list′*	checkAdd BlueNode (id: lex, PGMNAME)
2.2.2.		*identifier_list′*	→	**ϵ**	
3.1.	void	*declarations*	→	**var id** : *type* ; *declarations*	checkAdd BlueNode (id: lex, type.type)
3.2.		*declarations*	→	**ϵ**	
4.1.	TYPE	*type*	→	*standard_type*	type ← stdtype.type width ← st.width
4.2.		*type*	→	**array** [**num .. num**] **of** *standard_type*	type ← AINT if INT AREAL if REAL width ←(num−num +1)· st. width ERR if ERR ERR* otherwise offset +=width
5.1.	TYPE	*standard_type*	→	**integer**	type ← INT width ← 4
5.2.		*standard_type*	→	**real**	type ← REAL width ← 8
6.1.	void	*subprogram_declarations*	→	*subprogram_declaration* ; *subprogram_declarations*	
6.2.		*subprogram_declarations*	→	**ϵ**	
7.	void	*subprogram_declaration*	→	*subprogram_head declarations* *subprogram_declarations compound_statement*	pop From Green Stack
8.	void	*subprogram_head*	→	**procedure id** *arguments* ;	check AddGreenNode (id: lex, PROC) offset = 0
9.1.	void	*arguments*	→	(*parameter_list*)	
9.2.		*arguments*	→	**ϵ**	
10.1.	void	*parameter_list*	→	**id** : *type parameter_list′*	checkAddBlueNode(id:lex, "pp"++ type)
10.2.1.	void	*parameter_list′*	→	; **id** : *type parameter_list′*	
10.2.2.		*parameter_list′*	→	**ϵ**	

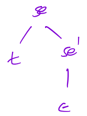
11.	void	*compound_statement*	→	**begin**
				optional_statements
				end	

12.1.	void	*optional_statements*	→	*statement_list*	
12.2.		*optional_statements*	→	**ϵ**	
13.1.	void	*statement_list*	→	*statement statement_list′*	
13.2.1.	void	*statement_list′*	→	; *statement statement_list′*	
13.2.2.		*statement_list′*	→	**ϵ**	
14.1.	void	*statement*	→	*variable* **assignop** *expression*	
14.2.		*statement*	→	*procedure_statement*	
14.3.		*statement*	→	*compound_statement*	
14.4.		*statement*	→	**while** *expression* **do** *statement*	Check type BOOL
14.5.		*statement*	→	**if** *expression* **then** *statement else′*	Check type BOOL
15.1.	void	*else′*	→	**else** *statement*	
15.2.		*else′*	→	**ϵ**	
16.	TYPE	*variable*	→	**id** *array_access*	Same as factor → id factor'
17.1.	TYPE	*array_access*	→	[*expression*]	Same as factor → id factor'
17.2.		*array_access*	→	**ϵ**	Same as factor → id factor'
18.	void	*procedure_statement*	→	**call id** *optional_expressions*	Must exist & be a procedure
19.1.	void	*optional_expressions*	→	(*expression_list*)	i ← pointer to the first item
19.2.		*optional_expressions*	→	**ϵ**	
20.1.	void	*expression_list*	→	*expression expression_list′*	i ← pointer to the right type - if match, continue ; else, fail
20.2.1.	void	*expression_list′*	→	, *expression expression_list′*	i ← pointer to the right type - if match, continue ; else, fail
20.2.2.		*expression_list′*	→	**ϵ**	
21.	TYPE	*expression*	→	*simple_expression related_expression*	
22.1.	TYPE	*related_expression*	→	**relop** *simple_expression*	
22.2.		*related_expression*	→	**ϵ**	
23.1.1.	TYPE	*simple_expression*	→	*term simple_expression′*	
23.1.2.		*simple_expression*	→	*sign term simple_expression′*	
23.2.1.	TYPE	*simple_expression′*	→	**addop** *term simple_expression′*	
23.2.2.		*simple_expression′*	→	**ϵ**	

s.t	v.t	e.t
ERR*	UNDECL	—
—	INT	INT
—	REAL	REAL
—	ERR	—
—	—	ERR
ERR*	Anything	Anything

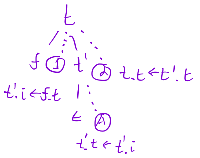
Looks exactly like the table for * and / for factor ; so do these:

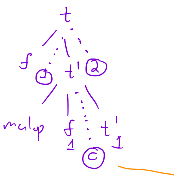




±

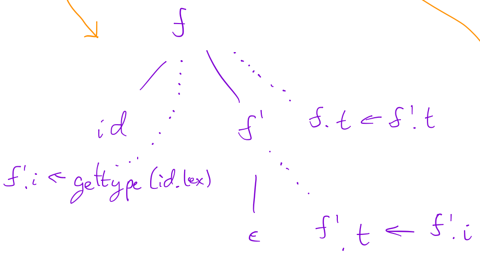
or





24.1.	TYPE	*term*	→	*factor term′*	
24.2.1.	TYPE	*term′*	→	**mulop** *factor term′*	
24.2.2.		*term′*	→	**ϵ**	
25.1.1.	TYPE	*factor*	→	**id** *factor'*	{type ← factor'.type}
25.1.2.		*factor*	→	**num**	{type ← num.type}
25.1.3.		*factor*	→	(*expression*)	{type ← exp.type}
25.1.4.		*factor*	→	**not** *factor*	
25.2.1.		*factor'*	→	[*expression*]	See table down below
25.2.1.		*factor'*	→	**ϵ**	{type ← gettype(id.lex)} (undeclared error, procedure, etc. - use a table)
26.1.	void	*sign*	→	+	
26.2.		*sign*	→	−	

f.type	f_1.type
BOOL	BOOL
ERR*	¬BOOL ^ ¬ERR
ERR	ERR



ⓒ

Mulops:

- *
- /
- div
- mod
- and

t.i	f.t	Mulop.op (* or /)	t'.i
INT	INT	*	INT
REAL	REAL	*	REAL
ERR	ERR	*	ERR
ERR*	INT	*	REAL

But there is no type coercion

Array:

f'.t	e.t	f'.i
INT	INT	AINT
REAL	INT	AREAL
ERR*	¬INT ¬ERR	
ERR*		¬ARRAY
ERR	ERR	
ERR		ERR

AINT :: Array of ints
AREAL :: Array of reals

Watch out for the possibility of array types !!

2 Implementation

I merely modified the productions to enforce the rules, according to the decorated grammar given above.

The declarations processing was interesting—I used a binary tree with left-child, right-sibling notation. I a pointer to the bottom of the tree at all times. Whenever anything was added to the tree, I updated the bottom pointer to point to the new data. Then, whenever a new scope was declared, I added a pointer to that tree node to the stack; whenever the new scope ended, I set the bottom pointer to the value popped from the stack, and set a flag to add to the right of the child.

If any errors were encountered while parsing, the error is added to the error queue. Then, the error is printed before the next token is collected. In this situation, multiple semantic errors could happen on the same token, so I had to create a separate message for each one and add it to a queue.

3 Discussion & Conclusions

Implementing this project definitely taught me about the importance of an $LL(1)$ grammar, and how neat the recursive descent parser is. Decorating the grammar is a really cool way of implementing the compiler.

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

Appendix 1: Sample Inputs and Outputs

3.1 Error-Filled

Listing 1: Error-Full Source Code

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

  procedure fib1(aReallyLongInt : integer; b : real , c : real);
    begin
        if a <= 1.20 then fib := c
        else call fib (a - 01, c, b + c)
    end;

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

    begin
      a := a - 1;
      fib1(3.00);
      sum := 1;
      c := b;
      while a > 0) do
        call 3;
        begin
          a := a - 1;
          b := sum;
          sum := c + sum;
          c := b
        end;
      fib2 := sum
    end;

  procedure init;
    begin
```

```
      n := 12;
      if (1 and 2) or 3 then p := 12
      else p := 14;
      numsArray[3] := 15.560;
      q := q[4];
      q[4] := 12
    end;

    begin*
      call init;
      call fib2;
      call rawr3(34, 56);
    end.

a
```

Listing 2: Error-Full Listing File

```
      1 program fib(input; output):
SYNERR: Found ';'; expected ')', ',' instead.
SYNERR: Found ':'; expected ';' instead.
      2   var a: int; var p: integer;
SYNERR: Found 'ID'; expected 'array', 'real', 'integer' instead.
SYNERR: Found 'integer'; expected ';' instead.
SYNERR: Found ';'; expected 'begin', 'procedure', 'var' instead.
      3   var numsArray : array [6..12] on integer;
      4   var q: real;
      5
      6   procedure fib1(aReallyLongInt : integer; b : real, c : real);
LEXERR:                 ID length exceeded 10 characters:      aReallyLongInt
SYNERR: Found ','; expected ')', ';' instead.
      7     begin
      8         if a <= 1.20 then fib := c
SEMERR: No variable 'a' is defined in the local scope!
LEXERR:                               Trailing 0 in real:                1.20
SEMERR: Cannot assign to ID 'fib' of type 'PROGRAM'!
      9         else call fib (a - 01, c, b + c)
SEMERR: No variable ' ' is defined in the local scope!
SEMERR: Procedure 'fib' not in scope!
SEMERR: No variable 'a' is defined in the local scope!
LEXERR:                                 Leading 0 in
```

```
   int:                  01
SEMERR: No variable 'c' is defined in the local scope!
SEMERR: No variable 'c' is defined in the local scope!
     10     end;
     11
     12   procedure fib2(a : integer);
     13     var b : real; var c : real; var sum : ;
SYNERR: Found ';'; expected 'array', 'real', 'integer'
    instead.
     14     var b : real;
SYNERR: Found 'real'; expected ';' instead.
SYNERR: Found ';'; expected 'begin', 'procedure', 'var
   ' instead.
     15     procedure rawr3(b : real);
     16       var q : real;
     17       begin
     18         q := b + 2.0;
     19         call fib2(q).
SEMERR: Expected type INT, not REAL!
SYNERR: Found '.'; expected 'end', ';' instead.
     20       end;
     21
     22     begin
     23       a := a - 1;
     24       fib1(3.00);
SYNERR: Found '('; expected '[', ':=' instead.
LEXERR:                               Trailing 0 in
   real:                3.00
     25       sum := 1;
     26       c := b;
     27       while a > 0) do
SYNERR: Found ')'; expected 'do' instead.
SYNERR: Found 'do'; expected 'if', 'while', 'begin', '
   call', 'ID' instead.
     28         call 3;
     29         begin
     30           a := a - 1;
     31           b := sum;
SEMERR: No variable 'sum' is defined in the local
   scope!
     32           sum := c + sum;
SEMERR: ID 'sum' not in scope!
SEMERR: No variable 'sum' is defined in the local
   scope!
     33           c := b
     34         end;
     35        fib2 := sum
SEMERR: Cannot assign to ID 'fib2' of type 'PROCEDURE'!
     36      end;
SEMERR: No variable 'sum' is defined in the local scope!
     37
     38    procedure init;
     39      begin
     40        n := 12;
SEMERR: ID 'n' not in scope!
     41        if (1 and 2) or 3 then p := 12
SEMERR: Expected BOOL and BOOL for use with 'and', received INT and INT!
SEMERR: ID 'p' not in scope!
     42        else p := 14;
SEMERR: ID 'p' not in scope!
     43        numsArray[3] := 15.560;
LEXERR:                               Trailing 0 in real:              15.560
SEMERR: ID 'numsArray' not in scope!
     44        q := q[4];
SEMERR: ID 'q' not in scope!
SEMERR: No variable 'q' is defined in the local scope!
     45        q[4] := 12
SEMERR: ID 'q' not in scope!
     46      end;
     47
     48      begin*
SYNERR: Found '*'; expected 'array', 'end', 'if', 'while', 'begin', 'call', 'ID' instead.
     49        call init;
     50        call fib2;
     51        call rawr3(34, 56);
     52      end.
     53
     54 a
SYNERR: Found 'ID'; expected 'EOF' instead.
```

Listing 3: Error-Full Semantic Mem File

```
        ID        Memory Offset
         a                    0
         b                    0
         c                    8
       sum                   16
```

```
         q                   0
```

Listing 4: Error-Full Token File

```
1    FILEEND
2    ASSIGNOP
3    RELOP
4    ID
5    CONTROL
6    ADDOP
7    MULOP
8    WS
9    ARRAY
10   TYPE
11   VAR
12   NUM
13   PUNC
14   GROUP
15   INVERSE
16   LEXERR
17   SYNERR
18   SEMERR
      LineLexeme     Token Attribute   Token Type
  1      program     5     7
  1   fib     4      0x7ff6d6600fd0
  1     (    14      0
  1 input     4      0x7ff6d66012e0
  1     ;    13      1
  1     ;    17      0
  1output     4      0x7ff6d6601760
  1     )    14      1
  1     :    10      0
  1     :    17      0
  2   var    11      0
  2     a     4      0x7ff6d6601f00
  2     :    10      0
  2   int     4      0x7ff6d66021f0
  2   int    17      0
  2     ;    13      1
  2   var    11      0
  2     p     4      0x7ff6d6602780
  2     :    10      0
  2      integer    10     1
  2      integer    17     0
  2     ;    13      1
  2     ;    17      0
```

```
3   var    11     0
3    numsArray     4      0x7ff6d6700a60
3     :    10     0
3 array     9     0
3     [    14     2
3     6    12     0
3    ..     9     1
3    12    12     0
3     ]    14     3
3    on     4      0x7ff6d67016c0
3      integer    10     1
3     ;    13     1
4   var    11     0
4     q     4      0x7ff6d6701eb0
4     :    10     0
4  real    10     2
4     ;    13     1
6    procedure     5     6
6  fib1     4      0x7ff6d67028f0
6     (    14     0
6      aReallyLongInt     4      0x7ff6d6702d00
6      aReallyLongInt    16     1
6     :    10     0
6      integer    10     1
6     ;    13     1
6     b     4      0x7ff6d6703490
6     :    10     0
6  real    10     2
6     ,    13     0
6     ,    17     0
6     c     4      0x7ff6d6703bf0
6     :    10     0
6  real    10     2
6     )    14     1
6     ;    13     1
7 begin     5     0
8    if     5     5
8     a     4      0x7ff6d6601f00
8    <=     3     1
8   18     0
8  1.20    12     0
8  1.20    16     9
8  then     5     8
8   fib     4      0x7ff6d6600fd0
8    :=     2     0
8     c     4      0x7ff6d6703bf0
```

```
 8   18     0
 9  else     5     2
 9   18     0
 9  call     5    10
 9   fib     4      0x7ff6d6600fd0
 9     (    14     0
 9   18     0
 9     a     4      0x7ff6d6601f00
 9     -     6     1
 9   18     0
 9    01    12     0
 9    01    16     7
 9     ,    13     0
 9     c     4      0x7ff6d6703bf0
 9     ,    13     0
 9   18     0
 9     b     4      0x7ff6d6703490
 9     +     6     0
 9     c     4      0x7ff6d6703bf0
 9     )    14     1
 9   18     0
10   end     5     3
10     ;    13     1
12    procedure      5     6
12  fib2     4      0x7ff6d67088f0
12     (    14     0
12     a     4      0x7ff6d6601f00
12     :    10     0
12      integer     10     1
12     )    14     1
12     ;    13     1
13   var    11     0
13     b     4      0x7ff6d6703490
13     :    10     0
13  real    10     2
13     ;    13     1
13   var    11     0
13     c     4      0x7ff6d6703bf0
13     :    10     0
13  real    10     2
13     ;    13     1
13   var    11     0
13   sum     4      0x7ff6d670a770
13     :    10     0
13     ;    13     1
13     ;    17     0
```

```
14    var     11      0
14      b      4       0x7ff6d6703490
14      :     10      0
14   real     10      2
14   real     17      0
14      ;     13      1
14      ;     17      0
15    procedure      5      6
15  rawr3      4       0x7ff6d670bf00
15      (     14      0
15      b      4       0x7ff6d6703490
15      :     10      0
15   real     10      2
15      )     14      1
15      ;     13      1
16    var     11      0
16      q      4       0x7ff6d6701eb0
16      :     10      0
16   real     10      2
16      ;     13      1
17  begin      5      0
18      q      4       0x7ff6d6701eb0
18     :=      2      0
18      b      4       0x7ff6d6703490
18      +      6      0
18    2.0     12      1
18      ;     13      1
19   call      5     10
19   fib2      4       0x7ff6d67088f0
19      (     14      0
19      q      4       0x7ff6d6701eb0
19      )     14      1
19    18      0
19      .     13      2
19      .     17      0
20    end      5      3
20      ;     13      1
22  begin      5      0
23      a      4       0x7ff6d6601f00
23     :=      2      0
23      a      4       0x7ff6d6601f00
23      -      6      1
23      1     12      0
23      ;     13      1
24   fib1      4       0x7ff6d67028f0
24      (     14      0
24     (    17     0
24  3.00    12     0
24  3.00    16     9
24     )    14     1
24     ;    13     1
25   sum     4      0x7ff6d670a770
25    :=     2     0
25     1    12     0
25     ;    13     1
26     c     4      0x7ff6d6703bf0
26    :=     2     0
26     b     4      0x7ff6d6703490
26     ;    13     1
27 while     5     9
27     a     4      0x7ff6d6601f00
27     >     3     3
27     0    12     0
27     )    14     1
27     )    17     0
27    do     5     1
27    do    17     0
28  call     5    10
28     3    12     0
28     ;    13     1
29 begin     5     0
30     a     4      0x7ff6d6601f00
30    :=     2     0
30     a     4      0x7ff6d6601f00
30     -     6     1
30     1    12     0
30     ;    13     1
31     b     4      0x7ff6d6703490
31    :=     2     0
31   sum     4      0x7ff6d670a770
31     ;    13     1
31   18     0
32   sum     4      0x7ff6d670a770
32    :=     2     0
32     c     4      0x7ff6d6703bf0
32   18     0
32     +     6     0
32   sum     4      0x7ff6d670a770
32     ;    13     1
32   18     0
33     c     4      0x7ff6d6703bf0
33    :=     2     0
```

```
33     b     4      0x7ff6d6703490
34   end     5     3
34     ;    13     1
35  fib2     4      0x7ff6d67088f0
35    :=     2     0
35   sum     4      0x7ff6d670a770
35   18      0
36   end     5     3
36   18      0
36     ;    13     1
38    procedure     5     6
38  init     4      0x7ff6d671a030
38     ;    13     1
39 begin     5     0
40     n     4      0x7ff6d671aae0
40    :=     2     0
40    12    12     0
40   18      0
40     ;    13     1
41    if     5     5
41     (    14     0
41     1    12     0
41   and     7     2
41     2    12     0
41     )    14     1
41   18      0
41    or     6     2
41     3    12     0
41  then     5     8
41     p     4      0x7ff6d6602780
41    :=     2     0
41    12    12     0
41   18      0
42  else     5     2
42     p     4      0x7ff6d6602780
42    :=     2     0
42    14    12     0
42   18      0
42     ;    13     1
43    numsArray     4      0x7ff6d6700a60
43     [    14     2
43     3    12     0
43     ]    14     3
43    :=     2     0
4315.560    12     0
4315.560    16     9
```

```
43   18     0
43     ;    13     1
44     q     4      0x7ff6d6701eb0
44    :=     2     0
44     q     4      0x7ff6d6701eb0
44   18     0
44     [    14     2
44   18     0
44     4    12     0
44     ]    14     3
44     ;    13     1
45     q     4      0x7ff6d6701eb0
45     [    14     2
45     4    12     0
45     ]    14     3
45    :=     2     0
45    12    12     0
45   18     0
46   end     5     3
46     ;    13     1
48 begin     5     0
48     *     7     0
48     *    17     0
49  call     5    10
49  init     4      0x7ff6d671a030
49     ;    13     1
50  call     5    10
50  fib2     4      0x7ff6d67088f0
50     ;    13     1
51  call     5    10
51 rawr3     4      0x7ff6d670bf00
51     (    14     0
51    34    12     0
51     ,    13     0
51    56    12     0
51     )    14     1
51     ;    13     1
52   end     5     3
52     .    13     2
54     a     4      0x7ff6d6601f00
54     a    17     0
55   EOF     1     0
```

3.2 Just Semantic Errors

Listing 5: Just Semantic Source Code

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

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

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

    begin
      q := not 3;
      q := (3 < 4) and (3.6 < p);
      a := a - 1;
      call fib1(3.2, 1);
      sum := 1;
      c := b;
      while not (a > 0) do
        call rawr3(b * 4);
      begin
        a := a - 1;
        b := sum;
        sum := c + sum;
        c := b
      end;
      fib2 := sum
    end;

  procedure init;
    begin
```

```
      n := 12;
      if not (1 < 74) or 3 then p:=(1 > 2) and(3 < n)
        else p:=not 2;
      numsArray[3] := 15.56;
      q := q[4];
      q[4] := 12
    end;

  begin
    call init;
    call fib2;
    call rawr3(34, 56)
  end.
```

Listing 6: Just Semantic Listing File

```
      1 program fib(input, output);
      2   var a: integer; var p: integer;
      3   var numsArray : array [6..12] of integer;
      4   var q: real;
      5
      6   procedure fib1(aLongInt : integer; b : real;
      7                 c : integer; d: integer);
      8     begin
      9         if a <= 1 then fib := c
SEMERR: Cannot assign to ID ’fib’ of type ’PROGRAM’!
     10         else call fib (a - 1, c, b + c)
SEMERR: Procedure ’fib’ not in scope!
SEMERR: Attempt to add incompatible types REAL and INT!
     11     end;
     12
     13   procedure fib2(a : integer);
     14     var b : real; var c : real; var sum : integer;
     15     var b : real;
SEMERR: A variable named ’b’ is already defined in the local scope!
     16     procedure rawr3(b : real);
     17       var q : real;
     18       begin
     19         q := b + sum;
SEMERR: Attempt to add incompatible types REAL and INT!
     20         call fib2(q)
SEMERR: Expected type INT, not REAL!
```

```
     21       end;
     22
     23     begin
     24       q := not 3;
SEMERR: Expected BOOL use with ’not’, received INT!
     25       q := (3 < 4) and (3.6 < p);
SEMERR: Attempt to compare incompatible types REAL and INT!
     26       a := a - 1;
     27       call fib1(3.2, 1);
SEMERR: Expected type INT, not REAL!
SEMERR: Expected type REAL, not INT!
SEMERR: Expected INT, not the end of the parameters!
     28       sum := 1;
     29       c := b;
     30       while not (a > 0) do
     31         call rawr3(b * 4);
SEMERR: Attempt to multiply or divide incompatible types REAL and INT!
     32       begin
     33         a := a - 1;
     34         b := sum;
SEMERR: Attempt to convert type REAL into type INT in assignment!
     35         sum := c + sum;
SEMERR: Attempt to add incompatible types REAL and INT!
     36         c := b
     37       end;
     38       fib2 := sum
SEMERR: Cannot assign to ID ’fib2’ of type ’PROCEDURE’!
     39     end;
     40
     41   procedure init;
     42     begin
     43       n := 12;
SEMERR: ID ’n’ not in scope!
     44       if not (1 < 74) or 3 then p:=(1 > 2) and (3 < n)
SEMERR: Expected BOOL and BOOL for use with ’or’, received BOOL and INT!
SEMERR: No variable ’n’ is defined in the local scope!
     45         else p:=not 2;
SEMERR: Expected BOOL use with ’not’, received INT!
     46       numsArray[3] := 15.56;
```

```
SEMERR: Attempt to convert type INT into type REAL in assignment!
     47       q := q[4];
SEMERR: Attempt to index variable of type REAL!
     48       q[4] := 12
SEMERR: Attempt to index variable of type REAL!
     49     end;
     50
     51   begin
     52     call init;
     53     call fib2;
SEMERR: Expected an argument of type INT!
     54     call rawr3(34, 56)
SEMERR: Procedure 'rawr3' not in scope!
     55   end.
```

Listing 7: Just Semantic Mem File

```
        ID       Memory Offset
         a                   0
         p                   4
 numsArray                   8
         q                  36
         b                   0
         c                   8
       sum                  16
         q                   0
```

Listing 8: Just Semantic Token File

```
1    FILEEND
2    ASSIGNOP
3    RELOP
4    ID
5    CONTROL
6    ADDOP
7    MULOP
8    WS
9    ARRAY
10   TYPE
11   VAR
12   NUM
13   PUNC
14   GROUP
15   INVERSE
16   LEXERR
```

```
17   SYNERR
18   SEMERR
      LineLexeme      Token Attribute   Token Type
  1      program      5     7
  1   fib     4       0x7fba52500fd0
  1     (    14      0
  1 input     4       0x7fba525012e0
  1     ,    13      0
  1output     4       0x7fba52501660
  1     )    14      1
  1     ;    13      1
  2   var    11      0
  2     a     4       0x7fba52501d30
  2     :    10      0
  2      integer     10     1
  2     ;    13      1
  2   var    11      0
  2     p     4       0x7fba52502530
  2     :    10      0
  2      integer     10     1
  2     ;    13      1
  3   var    11      0
  3    numsArray      4      0x7fba52502f50
  3     :    10      0
  3 array     9      0
  3     [    14      2
  3     6    12      0
  3    ..     9      1
  3    12    12      0
  3     ]    14      3
  3    of     9      2
  3      integer     10     1
  3     ;    13      1
  4   var    11      0
  4     q     4       0x7fba525043b0
  4     :    10      0
  4  real    10      2
  4     ;    13      1
  6    procedure      5     6
  6  fib1     4       0x7fba52504e20
  6     (    14      0
  6     aLongInt      4      0x7fba52505170
  6     :    10      0
  6      integer     10     1
  6     ;    13      1
  6     b     4       0x7fba52505830
```

```
 6     :    10     0
 6  real    10     2
 6     ;    13     1
 7     c     4      0x7fba52506790
 7     :    10     0
 7      integer    10      1
 7     ;    13     1
 7     d     4      0x7fba52506e50
 7     :    10     0
 7      integer    10      1
 7     )    14     1
 7     ;    13     1
 8 begin     5     0
 9    if     5     5
 9     a     4      0x7fba52501d30
 9    <=     3     1
 9     1    12     0
 9  then     5     8
 9   fib     4      0x7fba52500fd0
 9    :=     2     0
 9     c     4      0x7fba52506790
 9   18      0
10  else     5     2
10  call     5    10
10   fib     4      0x7fba52500fd0
10     (    14     0
10   18      0
10     a     4      0x7fba52501d30
10     -     6     1
10     1    12     0
10     ,    13     0
10     c     4      0x7fba52506790
10     ,    13     0
10     b     4      0x7fba52505830
10     +     6     0
10     c     4      0x7fba52506790
10     )    14     1
10   18      0
11   end     5     3
11     ;    13     1
13    procedure     5      6
13  fib2     4      0x7fba5250b340
13     (    14     0
13     a     4      0x7fba52501d30
13     :    10     0
13      integer    10      1
```

```
13      )     14      1
13      ;     13      1
14    var     11      0
14      b      4      0x7fba52505830
14      :     10      0
14   real     10      2
14      ;     13      1
14    var     11      0
14      c      4      0x7fba52506790
14      :     10      0
14   real     10      2
14      ;     13      1
14    var     11      0
14    sum      4      0x7fba5250d1c0
14      :     10      0
14       integer    10      1
14      ;     13      1
15    var     11      0
15      b      4      0x7fba52505830
15      :     10      0
15   real     10      2
15      ;     13      1
15    18      0
16     procedure      5      6
16  rawr3      4      0x7fba5250e950
16      (     14      0
16      b      4      0x7fba52505830
16      :     10      0
16   real     10      2
16      )     14      1
16      ;     13      1
17    var     11      0
17      q      4      0x7fba525043b0
17      :     10      0
17   real     10      2
17      ;     13      1
18  begin      5      0
19      q      4      0x7fba525043b0
19     :=      2      0
19      b      4      0x7fba52505830
19      +      6      0
19    sum      4      0x7fba5250d1c0
19      ;     13      1
19    18      0
20   call      5     10
20   fib2      4      0x7fba5250b340
```

```
20     (    14     0
20     q     4      0x7fba525043b0
20     )    14     1
20   18     0
21   end     5     3
21     ;    13     1
23 begin     5     0
24     q     4      0x7fba525043b0
24    :=     2     0
24   not    15     0
24     3    12     0
24     ;    13     1
24   18     0
25     q     4      0x7fba525043b0
25    :=     2     0
25     (    14     0
25     3    12     0
25     <     3     0
25     4    12     0
25     )    14     1
25   and     7     2
25     (    14     0
25   3.6    12     1
25     <     3     0
25     p     4      0x7fba52502530
25     )    14     1
25   18     0
25     ;    13     1
26     a     4      0x7fba52501d30
26    :=     2     0
26     a     4      0x7fba52501d30
26     -     6     1
26     1    12     0
26     ;    13     1
27  call     5    10
27  fib1     4      0x7fba52504e20
27     (    14     0
27   3.2    12     1
27     ,    13     0
27   18     0
27     1    12     0
27     )    14     1
27   18     0
27   18     0
27     ;    13     1
28   sum     4      0x7fba5250d1c0
```

```
28    :=      2     0
28     1     12     0
28     ;     13     1
29     c      4      0x7fba52506790
29    :=      2     0
29     b      4      0x7fba52505830
29     ;     13     1
30 while      5     9
30   not     15     0
30     (     14     0
30     a      4      0x7fba52501d30
30     >      3     3
30     0     12     0
30     )     14     1
30    do      5     1
31  call      5    10
31 rawr3      4      0x7fba5250e950
31     (     14     0
31     b      4      0x7fba52505830
31     *      7     0
31     4     12     0
31     )     14     1
31   18      0
31     ;     13     1
32 begin      5     0
33     a      4      0x7fba52501d30
33    :=      2     0
33     a      4      0x7fba52501d30
33     -      6     1
33     1     12     0
33     ;     13     1
34     b      4      0x7fba52505830
34    :=      2     0
34   sum      4      0x7fba5250d1c0
34     ;     13     1
34   18      0
35   sum      4      0x7fba5250d1c0
35    :=      2     0
35     c      4      0x7fba52506790
35     +      6     0
35   sum      4      0x7fba5250d1c0
35     ;     13     1
35   18      0
36     c      4      0x7fba52506790
36    :=      2     0
36     b      4      0x7fba52505830
```

```
37   end      5      3
37     ;     13      1
38  fib2      4       0x7fba5250b340
38    :=      2      0
38   sum      4       0x7fba5250d1c0
38   18      0
39   end      5      3
39     ;     13      1
41     procedure      5      6
41  init      4       0x7fba5260f370
41     ;     13      1
42 begin      5      0
43     n      4       0x7fba5260fe20
43    :=      2      0
43    12     12      0
43   18      0
43     ;     13      1
44    if      5      5
44   not     15      0
44     (     14      0
44     1     12      0
44     <      3      0
44    74     12      0
44     )     14      1
44    or      6      2
44     3     12      0
44  then      5      8
44   18      0
44     p      4       0x7fba52502530
44    :=      2      0
44     (     14      0
44     1     12      0
44     >      3      3
44     2     12      0
44     )     14      1
44   and      7      2
44     (     14      0
44     3     12      0
44     <      3      0
44     n      4       0x7fba5260fe20
44     )     14      1
44   18      0
45  else      5      2
45     p      4       0x7fba52502530
45    :=      2      0
45   not     15      0
```

```
45     2    12     0
45     ;    13     1
45   18     0
46    numsArray     4      0x7fba52502f50
46     [    14     2
46     3    12     0
46     ]    14     3
46    :=     2     0
46 15.56    12     1
46     ;    13     1
46   18     0
47     q     4      0x7fba525043b0
47    :=     2     0
47     q     4      0x7fba525043b0
47     [    14     2
47     4    12     0
47     ]    14     3
47     ;    13     1
47   18     0
48     q     4      0x7fba525043b0
48     [    14     2
48     4    12     0
48     ]    14     3
48    :=     2     0
48   18     0
48    12    12     0
49   end     5     3
49     ;    13     1
51 begin     5     0
52  call     5    10
52  init     4      0x7fba5260f370
52     ;    13     1
53  call     5    10
53  fib2     4      0x7fba5250b340
53     ;    13     1
53   18     0
54  call     5    10
54 rawr3     4      0x7fba5250e950
54     (    14     0
54   18     0
54    34    12     0
54     ,    13     0
54    56    12     0
54     )    14     1
55   end     5     3
55     .    13     2
```

56	EOF	1	0

3.3 Error-Free

Listing 9: Error-Free Source Code

```

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 10: Error-Free 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.
```

Listing 11: Error-Free Mem File

```
         ID        Memory Offset
          a                    0
          b                    4
          c                   12
          d                    0
          e                    0
          e                    0
          x                    0
```

Listing 12: Error-Free Token File

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

```
14   GROUP
15   INVERSE
16   LEXERR
17   SYNERR
18   SEMERR
      LineLexeme     Token Attribute    Token Type
  2      program     5     7
  2  test     4      0x7f8012600dc0
  2     (    14     0
  2 input     4      0x7f8012601160
  2     ,    13     0
  2output     4      0x7f80126014e0
  2     )    14     1
  2     ;    13     1
  3   var    11     0
  3     a     4      0x7f8012601bb0
  3     :    10     0
  3      integer    10      1
  3     ;    13     1
  4   var    11     0
  4     b     4      0x7f8012602560
  4     :    10     0
  4  real    10     2
  4     ;    13     1
  5   var    11     0
  5     c     4      0x7f8012602eb0
  5     :    10     0
  5 array     9     0
  5     [    14     2
  5     1    12     0
  5    ..     9     1
  5     2    12     0
  5     ]    14     3
  5    of     9     2
  5      integer    10      1
  5     ;    13     1
  7    procedure     5     6
  7 proc1     4      0x7f8012604480
  7     (    14     0
  7     x     4      0x7f80126046f0
  7     :    10     0
  7      integer    10      1
  7     ;    13     1
  7     y     4      0x7f8012604c90
  7     :    10     0
  7  real    10     2
```

```
 7     ;    13     1
 8     z     4      0x7f8012605bf0
 8     :    10     0
 8 array     9     0
 8     [    14     2
 8     1    12     0
 8    ..     9     1
 8     2    12     0
 8     ]    14     3
 8    of     9     2
 8      integer    10     1
 8     ;    13     1
 8     q     4      0x7f8012606be0
 8     :    10     0
 8  real    10     2
 8     )    14     1
 8     ;    13     1
 9   var    11     0
 9     d     4      0x7f80126076c0
 9     :    10     0
 9      integer    10     1
 9     ;    13     1
10 begin     5     0
11     a     4      0x7f8012601bb0
11    :=     2     0
11     2    12     0
11     ;    13     1
12     z     4      0x7f8012605bf0
12     [    14     2
12     a     4      0x7f8012601bb0
12     ]    14     3
12    :=     2     0
12     4    12     0
12     ;    13     1
13     c     4      0x7f8012602eb0
13     [    14     2
13     3    12     0
13     ]    14     3
13    :=     2     0
13     3    12     0
14   end     5     3
14     ;    13     1
16    procedure     5     6
16 proc2     4      0x7f801260ad50
16     (    14     0
16     x     4      0x7f80126046f0
```

```
16      :     10      0
16       integer     10      1
16      ;     13      1
16      y      4       0x7f8012604c90
16      :     10      0
16       integer     10      1
16      )     14      1
16      ;     13      1
17    var     11      0
17      e      4       0x7f801260c180
17      :     10      0
17   real     10      2
17      ;     13      1
19     procedure      5      6
19  proc3      4       0x7f801260cdc0
19      (     14      0
19      n      4       0x7f801260d030
19      :     10      0
19       integer     10      1
19      ;     13      1
19      z      4       0x7f8012605bf0
19      :     10      0
19   real     10      2
19      )     14      1
19      ;     13      1
20    var     11      0
20      e      4       0x7f801260c180
20      :     10      0
20       integer     10      1
20      ;     13      1
22     procedure      5      6
22  proc4      4       0x7f801260f070
22      (     14      0
22      a      4       0x7f8012601bb0
22      :     10      0
22       integer     10      1
22      ;     13      1
22      z      4       0x7f8012605bf0
22      :     10      0
22  array      9      0
22      [     14      2
22      1     12      0
22     ..      9      1
22      3     12      0
22      ]     14      3
22     of      9      2
```

```
22   real     10     2
22      )     14     1
22      ;     13     1
23    var     11     0
23      x      4      0x7f80126046f0
23      :     10     0
23        integer     10     1
23      ;     13     1
24  begin      5     0
25      a      4      0x7f8012601bb0
25     :=      2     0
25      e      4      0x7f801260c180
26    end      5     3
26      ;     13     1
28  begin      5     0
29      a      4      0x7f8012601bb0
29     :=      2     0
29      e      4      0x7f801260c180
29      ;     13     1
30      e      4      0x7f801260c180
30     :=      2     0
30      c      4      0x7f8012602eb0
30      [     14     2
30      e      4      0x7f801260c180
30      ]     14     3
31    end      5     3
31      ;     13     1
33  begin      5     0
34   call      5    10
34  proc1      4      0x7f8012604480
34      (     14     0
34      x      4      0x7f80126046f0
34      ,     13     0
34      e      4      0x7f801260c180
34      ,     13     0
34      c      4      0x7f8012602eb0
34      ,     13     0
34      b      4      0x7f8012602560
34      )     14     1
34      ;     13     1
35   call      5    10
35  proc3      4      0x7f801260cdc0
35      (     14     0
35      c      4      0x7f8012602eb0
35      [     14     2
35      1     12     0
```

```
35      ]   14    3
35      ,   13    0
35      e    4     0x7f801260c180
35      )   14    1
35      ;   13    1
36      e    4     0x7f801260c180
36     :=    2    0
36      e    4     0x7f801260c180
36      +    6    0
36   4.44   12    1
36      ;   13    1
37      a    4     0x7f8012601bb0
37     :=    2    0
37      (   14    0
37      a    4     0x7f8012601bb0
37    mod    7    4
37      y    4     0x7f8012604c90
37      )   14    1
37    div    7    3
37      x    4     0x7f80126046f0
37      ;   13    1
38  while    5    9
38      (   14    0
38      (   14    0
38      a    4     0x7f8012601bb0
38     >=    3    4
38      4   12    0
38      )   14    1
38    and    7    2
38      (   14    0
38      (   14    0
38      b    4     0x7f8012602560
38     <=    3    1
38      e    4     0x7f801260c180
38      )   14    1
39     or    6    2
39      (   14    0
39    not   15    0
39      (   14    0
39      a    4     0x7f8012601bb0
39      =    3    2
39      c    4     0x7f8012602eb0
39      [   14    2
39      a    4     0x7f8012601bb0
39      ]   14    3
39      )   14    1
39      )   14     1
39      )   14     1
39      )   14     1
39     do    5     1
40  begin    5     0
41      a    4      0x7f8012601bb0
41     :=    2     0
41      c    4      0x7f8012602eb0
41      [   14     2
41      a    4      0x7f8012601bb0
41      ]   14     3
41      +    6     0
41      1   12     0
42    end    5     3
43    end    5     3
43      ;   13     1
45  begin    5     0
46   call    5    10
46  proc2    4      0x7f801260ad50
46      (   14     0
46      c    4      0x7f8012602eb0
46      [   14     2
46      4   12     0
46      ]   14     3
46      ,   13     0
46      c    4      0x7f8012602eb0
46      [   14     2
46      5   12     0
46      ]   14     3
46      )   14     1
46      ;   13     1
47   call    5    10
47  proc2    4      0x7f801260ad50
47      (   14     0
47      c    4      0x7f8012602eb0
47      [   14     2
47      4   12     0
47      ]   14     3
47      ,   13     0
47      2   12     0
47      )   14     1
47      ;   13     1
48     if    5     5
48      (   14     0
48      a    4      0x7f8012601bb0
48      <    3     0
```

```
48      2   12    0
48      )   14    1
48   then    5    8
48      a    4     0x7f8012601bb0
48     :=    2    0
48      1   12    0
48   else    5    2
48      a    4     0x7f8012601bb0
48     :=    2    0
48      a    4     0x7f8012601bb0
48      +    6    0
48      2   12    0
48      ;   13    1
49     if    5    5
49      (   14    0
49      b    4     0x7f8012602560
49      >    3    3
49    4.2   12    1
49      )   14    1
49   then    5    8
49      a    4     0x7f8012601bb0
49     :=    2    0
49      c    4     0x7f8012602eb0
49      [   14    2
49      a    4     0x7f8012601bb0
49      ]   14    3
50    end    5    3
50      .   13    2
51    EOF    1    0
```

Appendix 2: Program Listings

Listing 13: compiler.c

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

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

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

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

int run()
{
generateParseTree();

return 0;
}

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

Listing 14: declarationsTree.h

```c
#ifndef DECLARATIONS_TREE_H
#define DECLARATIONS_TREE_H

#include <stdbool.h>

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

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

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

```

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

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

#endif // DECLARATIONS_TREE_H
```

Listing 15: linkedlist.h

```
#ifndef LINKED_H_
#define LINKED_H_

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

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

// Add an item to the front of the linked list
int add(LinkedList* list, void * data, int size);
// For use as a queue; slow, do not use
int addLast(LinkedList* list, void *data, int size);


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

#endif // LINKED_H_
```

Listing 16: errorHandler.h

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

extern const char* lexErrs[];
char* synErr;
LinkedList* semErrs;

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

Token* getNextErrorToken();

#endif // ERROR_HANDLER_H
```

Listing 17: globals.h

```
#ifndef GLOBALS_H
#define GLOBALS_H

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

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

#endif // GLOBALS_H
```

Listing 18: handler.h

```c
#ifndef HANDLER_H
#define HANDLER_H

#include<stdbool.h>
#include "../tokenizer/tokens.h"

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

#endif // HANDLER_H
```

Listing 19: parser.h

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

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

#endif // PARSER_H
```

Listing 20: productions.h

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

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

extern Token* current_tok;

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

#endif // voidS_H
```

Listing 21: symbolTable.h

```
#ifndef SYMBOL_TABLE_H
#define SYMBOL_TABLE_H

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

#endif // SYMBOL_TABLE_H
```

Listing 22: machines.h

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

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

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

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

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

Listing 23: tokenizer.h

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

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

#endif // PROCESSOR_H_
```

Listing 24: tokens.h

```
#ifndef TOKENS_H
#define TOKENS_H

#include<stdbool.h>

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

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

enum TokenType {NOOP, FILEEND, ASSIGNOP, RELOP, ID,
CONTROL, ADDOP, MULOP, WS, ARRAY, TYPE,
VAR, NUM, PUNC, GROUP, INVERSE,
LEXERR, SYNERR, SEMERR};

// The token data type
typedef struct T_Type {
enum TokenType attribute; // Attribute

union { // Aspect or character pointer
int aspect;
char* id;
};

int start; // Start in the line
int length; // Length of the lexeme

union { // Value of the number, or length of the array
int int_val;
double real_val;
int array_length;
};

LangType type; // The type of the token
bool param; // Whether the token is a parameter or not

} Token;

extern const Token eof_tok;
extern const Token lparen_tok;
extern const Token rparen_tok;
extern const Token plus_tok;
extern const Token comma_tok;
extern const Token minus_tok;
extern const Token semic_tok;
extern const Token colon_tok;
extern const Token period_tok;
extern const Token dotdot_tok;
extern const Token lbrac_tok;
extern const Token rbrac_tok;
extern const Token addop_tok;
extern const Token array_tok;
extern const Token assignop_tok;
extern const Token begin_tok;
extern const Token call_tok;
extern const Token do_tok;
extern const Token else_tok;
extern const Token end_tok;
extern const Token id_tok;
extern const Token if_tok;
extern const Token integer_tok;
extern const Token integer_val_tok;
extern const Token of_tok;
extern const Token real_val_tok;
extern const Token mulop_tok;
extern const Token not_tok;
extern const Token num_tok;
extern const Token procedure_tok;
extern const Token program_tok;
extern const Token real_tok;
extern const Token relop_tok;
extern const Token then_tok;
extern const Token var_tok;
extern const Token while_tok;

extern const char* catNames[19];
extern const char* typeNames[9];
const Token* getTokenFromLex(char* lex);
const char* getLexFromToken(Token* token, bool strict);

```

```c
// The type; else null if impossible
LangType convert_to_array(LangType type);
LangType convert_from_array(LangType type);

// Returns the type produced by the operation
LangType type_lookup(LangType first, LangType second, Token* op);

// Returns true if the tokens are equivalent, false otherwise
bool tokens_equal(const Token* p1, Token* p2, bool strict);


#endif // TOKENS_H
```

Listing 25: declarationsTree.c

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

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

static int offset = 0;

static DeclarationsTree* d_tree = NULL;
static tree_node* bottom_node = NULL;

static LinkedList* green_node_stack = NULL;

static void initialize_d_tree() {
d_tree = malloc(sizeof(*d_tree));
green_node_stack = malloc(sizeof(*green_node_stack));
}

static int get_width(Token* val) {
switch (val -> type) {
case INT: return 4;
case REAL: return 8;

case AINT: return 4*(val -> array_length);
case AREAL: return 8*(val -> array_length);

default: return 1000000;
}
}

static bool check_node(char* id, bool green) {
tree_node* current_node = bottom_node;
while (current_node != NULL) {
// Already exists
if (id == current_node -> lex)
return true;

if (!green && (current_node -> type == PROC || current_node -> type == PGNAME))
return false;

// We've passed the most recent green node, and this is a blue one
if (!green && (current_node -> type == PROC || current_node -> type == PGNAME))
break;

current_node = current_node -> parent;
}

return false;
}

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

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

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

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

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

offset = 0;

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

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

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

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

bottom_node = addition;

return true;
}


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

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

```

// It hasn’t been declared; create it
tree_node* addition = malloc(sizeof(*addition));
addition -> lex = decl -> id;
//printf("%s\n", addition -> lex);
addition -> type = decl -> type;
addition -> param = decl -> param;

if (!addition -> param)
{
outputWidth(addition -> lex, offset);
offset += get_width(decl);
}

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

bottom_node -> left = addition;
bottom_node = addition;

//printf("(%s, %s)\n", bottom_node -> lex, bottom_node -> parent -> lex);
return true;
}

tree_node* start_param_matching(Token* id) {
tree_node* current_node = bottom_node;
while (current_node != NULL)
{
if (current_node -> type == PROC && current_node -> lex == id -> id)
return current_node;
current_node = current_node -> parent;
}

return NULL;
}

bool check_add_node(Token* decl) {
char* errorMessage ;
switch (decl -> type) {
case PGNAME:
case PROC: if (!check_add_green_node(decl)) {
 errorMessage = calloc(100, sizeof(*errorMessage));
 sprintf(errorMessage,
"A program or procedure named ’%.*s’ is already defined in this scope!",
decl -> length, &BUFFER[decl -> start]);
throw_sem_error(errorMessage);
return false;
 }
 return true;

default: if (!check_add_blue_node(decl)) {
 errorMessage = calloc(100, sizeof(*errorMessage));
 sprintf(errorMessage,
"A variable named ’%.*s’ is already defined in the local scope!",
decl -> length, &BUFFER[decl -> start]);
throw_sem_error(errorMessage);
return false;
 }
 return true;
}
}
```

```

void reached_end_of_scope() {
bottom_node = (*(tree_node**) pop(green_node_stack));
bottom_node -> add_right = true;
}


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

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

current_node = current_node -> parent;
}

return NULL;
}
```

Listing 26: linkedlist.c

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

#include "linkedlist.h"


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

list -> head = addition;

return list -> size;
}

// For use as a queue; slow, do not use
int addLast(LinkedList* list, void *data, int size)
{
struct node* addition = malloc(sizeof(*addition));
addition -> data = malloc(size);
addition -> next = NULL;

for (int i = 0; i < size; i++)
*(char *) (addition -> data + i) = *(char *) (data + i);

struct node* current = list -> head;

if (list -> size == 0)
list -> head = addition;
else {
while (current -> next != NULL)
current = current -> next;

current -> next = addition;
}
list -> size++;

}

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

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

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

Listing 27: errorHandler.c

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


#include "errorHandler.h"

static LinkedList* errorList;

const char* lexErrs[] = {"Unrecognized symbol:",
 "ID length exceeded 10 characters:",
 "Int length exceeded 10 characters:",
 "Integer part of real exceeded 5 characters:",
 "Fractional part of real exceeded 5 characters:",
 "Exponent part of long real exceeded 2 characters:",
 "Missing exponent part of long real:",
 "Leading 0 in int:",
 "Excessive leading 0 in real:",
 "Trailing 0 in real:",
 "Leading 0 in exponent:",
 "Attempt to use real exponent:"};
char* synErr;
LinkedList* semErrs;


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

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

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

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

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

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

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

// Set the msg
addLast(semErrs, &msg, sizeof(&msg));
}

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

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

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

return NULL;
}
```

Listing 28: globals.c

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

```

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

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

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

Listing 29: handler.c

```
#include<stdio.h>

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

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

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

static const int
    ListingLineSpace = 7;
static const int
    ListingErrSpace = 50;
static const int
    ListingLexSpace = 20;

static const int MemNameSpace
    = 10;
static const int MemValSpace =
     20;

void writeEOFToken()
{
fprintf(tokenFile, "%*d%*.*s%*
    d%*d\n", TokenLineSpace,
    LINE, TokenLexSpace,
3, "EOF", TokenTypeSpace,
    FILEEND, TokenAttrSpace, 0)
    ;
}

int initializeHandler(const
    char* sourcePath, const
    char* resPath,
const char* listingPath, const
     char* tokenPath,
const char* memPath)
{
if ((sourceFile = fopen(
    sourcePath, "r")) == NULL)
{
fprintf(stderr, "%s\n", "
    Source was null?");
return 0;
}

FILE* resFile = fopen(resPath,
     "r");
initializeTokens(resFile);
fclose(resFile);

if ((listingFile = fopen(
    listingPath, "w+")) == NULL
     ||
(tokenFile = fopen(tokenPath,
    "w+")) == NULL ||
(memFile = fopen(memPath, "w+"
    )) == NULL)
return 0;

for (size_t i = FILEEND; i <=
    SEMERR; i++) {
fprintf(tokenFile, "%-5zu%s\n"
    , i, catNames[i]);
}


char line[72];
if (fgets(line, sizeof(line),
    sourceFile) != NULL)
{
updateLine(line);
fprintf(listingFile, "%*d\t%s"
    , ListingLineSpace, LINE,
    line);
} else {
writeEOFToken();
}

fprintf(tokenFile, "%*s%*s%*s
    %*s\n", TokenLineSpace, "
    Line",
TokenLexSpace, "Lexeme",
TokenAttrSpace, "Token
    Attribute",
TokenTypeSpace, "Token Type");

fprintf(memFile, "%*s%*s\n",
    MemNameSpace, "ID",
MemValSpace, "Memory Offset");
return 1;
}

void outputWidth(char* lex,
    int width) {
fprintf(memFile, "%*s%*d\n",
    MemNameSpace, lex,
    MemValSpace, width);
}

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

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

```

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


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

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

bool handleToken(Token* token)
{
writeToken(token);
if (token -> attribute == WS && token -> aspect == 1) // A newline
{
char line[72];
if (fgets(line, sizeof(line), sourceFile) != NULL)
{
updateLine(line);
fprintf(listingFile, "%*d\t%s", ListingLineSpace, LINE, line);
} else { // Error or end of file (assume the latter)
LINE++;
writeEOFToken();
return false; // Terminate
}
}
return true; // Continue
}
```

Listing 30: parser.c

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

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

Token* current_tok = NULL;

static bool sequence_running = true;

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

return next;
}

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

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

current_tok = get_next_relevant_token();
}
}

// Attempts to match the source token with the current token;
// if it is found, it returns the matched token (for use in the RDP).
// If it is not found, then match returns null.
Token* match(const Token* source, bool strict)
{
if (tokens_equal(source, current_tok, strict))
{
Token* prev_tok = current_tok;
current_tok = get_next_relevant_token();
return prev_tok;
}
else
{
throw_syn_error(current_tok, &source, 1);
current_tok = get_next_relevant_token();
return NULL;
}
}

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

Listing 31: arguments.c

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

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

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

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

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

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

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

synch();
}
```

Listing 32: array$_{a}ccess.c$

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

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

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

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

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

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

return ERR;
}

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

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

synch();
return ERR;
}
```

Listing 33: compound$_s$*tatement.c*

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

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

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

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

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


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

synch();
}
```

Listing 34: declarations.c

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

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

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

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

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

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

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

synch();
}
```

Listing 35: else$_{t}ail.c$

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

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

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

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

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

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

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

synch();
}
```

Listing 36: expression.c

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

```
#include<stdlib.h>

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

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

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

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

// Needs implementing: None
LangType expression()
{
// Production 21
if (tokens_equal(&lparen_tok, current_tok, true)
|| tokens_equal(&addop_tok, current_tok, false)
|| tokens_equal(&id_tok, current_tok, false)
|| tokens_equal(&not_tok, current_tok, true)
|| tokens_equal(&num_tok, current_tok, false))
{
LangType s_type = simple_expression();
return related_expression(s_type);
}

synch();
return ERR;
}
```

Listing 37: expression$_{l}ist.c$

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

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

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

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

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

// Needs implementing: None
void expression_list(tree_node* to_match, bool should_error)
```

```
{
// Production 20.1
if (tokens_equal(&lparen_tok, current_tok, true)
|| tokens_equal(&addop_tok, current_tok, false) // + OR -
|| tokens_equal(&id_tok, current_tok, false) // ID
|| tokens_equal(&not_tok, current_tok, true)
|| tokens_equal(&num_tok, current_tok, false)) // num
{
char* errorMessage;
if (to_match == NULL && should_error)
{
errorMessage= calloc(100, sizeof(*errorMessage));
sprintf(errorMessage, "Attempt to pass extraneous parameter!");
throw_sem_error(errorMessage);
}
LangType e_type = expression();
if (should_error && to_match != NULL && to_match -> param && e_type != ERR && e_type != to_match -> type) {
errorMessage= calloc(100, sizeof(*errorMessage));
sprintf(errorMessage, "Expected type %s, not %s!",
typeNames[to_match -> type], typeNames[e_type]);
throw_sem_error(errorMessage);
}
expression_list_tail(to_match == NULL || !to_match -> param ? NULL :
to_match -> left, e_type != ERR && should_error);
return;
}

synch();
}
```

Listing 38: expression$_{l}ist_{t}ail.c$

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

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

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

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

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

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

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

synch();
}
```

Listing 39: factor.c

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

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

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

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

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

// Needs implementing: 25.1.2
LangType factor()
{
```

```
// Production 25.1.1
if (tokens_equal(&id_tok, current_tok, false)) { // id
char* errorMessage;
Token* id_ref = match(&id_tok, false); // id
LangType id_type = get_type(id_ref);
if (id_type == NULL) {
errorMessage = calloc(100, sizeof(*errorMessage));
sprintf(errorMessage,
 "No variable ’%.*s’ is defined in the local scope!",
 id_ref -> length, &BUFFER[id_ref -> start]);
 throw_sem_error(errorMessage);
 id_type = ERR;
}
return factor_tail(id_type);

// Production 25.1.2
} else if (tokens_equal(&num_tok, current_tok, false)) { // num
Token* num_type;
num_type = match(&num_tok, false);
return num_type -> aspect == 0 ? INT : REAL;

// Production 25.1.3
} else if (tokens_equal(&lparen_tok, current_tok, true)) { // (
match(&lparen_tok, true);
LangType e_type = expression();
match(&rparen_tok, true); // )
return e_type;

// Production 25.1.4
} else if (tokens_equal(&not_tok, current_tok, true)) { // not
match(&not_tok, true);
LangType f_type = factor();
return type_lookup(f_type, INT, &not_tok);
}


synch();
return ERR;
}
```

Listing 40: factor$_{t}ail.c$

```
#include<stdbool.h>

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

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

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

static void synch()
```

```
{
require_sync(sync_set, sync_size, first_set, first_size);
}

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

return ERR;
}

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

// Production 25.2.2
} else if (tokens_equal(&rparen_tok, current_tok, true)
|| tokens_equal(&comma_tok, current_tok, true)
|| tokens_equal(&semic_tok, current_tok, true)
|| tokens_equal(&rbrac_tok, current_tok, true)
|| tokens_equal(&addop_tok, current_tok, false)
|| tokens_equal(&do_tok, current_tok, true)
|| tokens_equal(&else_tok, current_tok, true)
|| tokens_equal(&end_tok, current_tok, true)
|| tokens_equal(&mulop_tok, current_tok, false)
|| tokens_equal(&relop_tok, current_tok, false)
|| tokens_equal(&then_tok, current_tok, true))
return id_type; // epsilon

synch();
return ERR;
}
```

Listing 41: id$_{l}$*ist.c*

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

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

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

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

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

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

synch();
}
```

Listing 42: id$_list_tail.c$

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

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

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

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

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

// Needs implementing: None
void id_list_tail()
{
// Production 2.2.1
if (tokens_equal(&comma_tok,
    current_tok, true))
{
match(&comma_tok, true);
Token* id_ref;

id_ref = match(&id_tok, false)
    ;
if (id_ref != NULL) {
id_ref -> type = PPNAME;
id_ref -> param = true;
check_add_node(id_ref);
}

id_list_tail();
return;

// Production 2.2.2
} else if (tokens_equal(&
    rparen_tok, current_tok,
    true))
return; // Epsilon

synch();
}
```

Listing 43: optional$_expressions.c$

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

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

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

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

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

// Needs implementing: None
void optional_expressions(tree_node* to_match, bool should_error)
{
char* errorMessage;
// Production 19.1
if (tokens_equal(&lparen_tok, current_tok, true))
{
match(&lparen_tok, true);
expression_list(to_match, should_error);
match(&rparen_tok, true);
return;

// Production 19.2
} else if (tokens_equal(&semic_tok, current_tok, true)
|| tokens_equal(&else_tok, current_tok, true)
|| tokens_equal(&end_tok, current_tok, true))
{
if (to_match != NULL && should_error) {
errorMessage= calloc(100, sizeof(*errorMessage));
sprintf(errorMessage, "Expected an argument of type %s!",
typeNames[to_match -> type]);
throw_sem_error(errorMessage);
}

return; // epsilon
}

synch();
}
```

Listing 44: optional$_s$*tatements.c*

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

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

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

static const Token* sync_set[]
```

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

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

// Needs implementing: None
void optional_statements()
{
// Production 12.1
if (tokens_equal(&begin_tok, current_tok, true) // begin
|| tokens_equal(&call_tok, current_tok, true) // call
|| tokens_equal(&id_tok, current_tok, false) // ID
|| tokens_equal(&if_tok, current_tok, true) // if
|| tokens_equal(&while_tok, current_tok, true)) // while
{
statement_list();
return;

// Production 12.2
} else if (tokens_equal(&end_tok, current_tok, true)) // end
return; // epsilon

synch();
}
```

Listing 45: parameter$_{l}$*ist.c*

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

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

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

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

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

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

synch();
}
```

Listing 46: parameter$_{l}ist_{t}ail.c$

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

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

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

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

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

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

// Production 10.2.2
} else if (tokens_equal(&rparen_tok, current_tok, true)) // )
return; // epsilon

synch();
}
```

Listing 47: procedure$_{s}tatement.c$

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

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

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

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

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

```

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

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

synch();
}
```

Listing 48: program.c

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

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

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

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

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

// Needs implementing: None
void program()
{
Token* id_ref;
// Production 1
if (tokens_equal(&program_tok, current_tok, true)) {
match(&program_tok, true); // program
id_ref = match(&id_tok, false); // id
if (id_ref != NULL) {
id_ref -> type = PGNAME;
id_ref -> param = false;
check_add_node(id_ref);
}
match(&lparen_tok, true); // (
id_list();
match(&rparen_tok, true); // )
match(&semic_tok, true); // ;
declarations();
subprogram_declarations();
compound_statement();
match(&period_tok, true); // .
return;
}

synch();
}
```

Listing 49: related$_{e}xpression.c$

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

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

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

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

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

// Needs implementing: None
LangType related_expression(
    LangType s_type)
{
// Production 22.1
if (tokens_equal(&relop_tok,
    current_tok, false)) {
Token* relop_op;
relop_op = match(&relop_tok,
    false);
LangType s1_type =
    simple_expression();
return type_lookup(s_type,
    s1_type, relop_op);

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

synch();
return ERR;
}
```

Listing 50: sign.c

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

```

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

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

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

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

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

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

Listing 51: simple$_{e}xpression.c$

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

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

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

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

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

// Needs implementing: None
LangType simple_expression()
{
char* errorMessage;
// Production 23.1.1
if (tokens_equal(&lparen_tok,
    current_tok, true)
|| tokens_equal(&id_tok,
    current_tok, false)
|| tokens_equal(&not_tok,
    current_tok, true)
|| tokens_equal(&num_tok,
    current_tok, false))
{
LangType t_type = term();
return simple_expression_tail(
    t_type);

// Production 23.1.2
} else if (tokens_equal(&
    plus_tok, current_tok, true
    )
 || tokens_equal(&minus_tok,
     current_tok, true)) {
sign();
LangType t_type = term();
if (t_type != INT && t_type !=
     REAL && t_type != ERR)
{
errorMessage= calloc(100,
    sizeof(*errorMessage));
sprintf(errorMessage, "
    Expected number for use
    with sign, not %s!",
typeNames[t_type]);
throw_sem_error(errorMessage);
}
return simple_expression_tail(
    t_type);
}

synch();
return ERR;
}
```

Listing 52: simple$_{e}xpression_{t}ail.c$

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

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

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

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

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

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


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

synch();
return ERR;
}
```

Listing 53: standard$_{t}ype.c$

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

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

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

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

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

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

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

synch();
return ERR;
}
```

Listing 54: statement.c

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

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

```

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

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

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

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

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

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

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

// Production 14.4
} else if (tokens_equal(&while_tok, current_tok, true)) { // while
match(&while_tok, true); // while
LangType e_type = expression();
if (e_type != BOOL && e_type != ERR)
{
```

```
errorMessage= calloc(100, sizeof(*errorMessage));
sprintf(errorMessage, "Expression in while must be boolean, not %s!",
typeNames[e_type]);
throw_sem_error(errorMessage);
}
match(&do_tok, true);
statement();
return;

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


synch();
}
```

Listing 55: statement$_l$*ist.c*

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

#include "productions.h"
#include "../parser.h"
#include "../../tokenizer/tokens.h"
static const Token* first_set[] = {&id_tok, &call_tok, &begin_tok, &while_tok,
 &if_tok};
static const int first_size = sizeof(first_set)/sizeof(first_set[0]);

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

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

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

synch();
}
```

Listing 56: statement$_{l}ist_{t}ail.c$

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

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

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

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

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

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


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

synch();
}
```

Listing 57: subprogram$_{d}eclaration.c$

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

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

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

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

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

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

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

synch();
}
```

Listing 58: subprogram$_{d}eclarations.c$

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

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

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

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

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

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

// Production 6.2
} else if (tokens_equal(&begin_tok, current_tok, true)) // begin
return; // Epsilon

synch();
}
```

Listing 59: subprogram$_{h}ead.c$

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

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

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

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

static void synch()
```

```
{
require_sync(sync_set,
    sync_size, first_set,
    first_size);
}

// Needs implementing: None
bool subprogram_head()
{
bool result = false;
// Production 8
if (tokens_equal(&
    procedure_tok, current_tok,
     true)) // procedure
{
Token* id_ref;
match(&procedure_tok, true);
    // procedure
id_ref = match(&id_tok, false)
    ;
if (id_ref != NULL) {
id_ref -> type = PROC;
id_ref -> param = false;
result = check_add_node(id_ref
    );
}
arguments();
match(&semic_tok, true); // ;
return result;
}

synch();
return result;
}
```

Listing 60: term.c

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

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

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

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

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

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

synch();
return ERR;
}
```

Listing 61: term$_{t}ail.c$

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

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

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

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

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

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

// Production 24.2.2
} else if (tokens_equal(&rparen_tok, current_tok, true)
|| tokens_equal(&comma_tok, current_tok, true)
|| tokens_equal(&semic_tok, current_tok, true)
|| tokens_equal(&rbrac_tok, current_tok, true)
|| tokens_equal(&addop_tok, current_tok, false)
|| tokens_equal(&do_tok, current_tok, true)
|| tokens_equal(&else_tok, current_tok, true)
|| tokens_equal(&end_tok, current_tok, true)
|| tokens_equal(&relop_tok, current_tok, false)
|| tokens_equal(&then_tok, current_tok, true))
return f_type; // epsilon

synch();
return ERR;
}
```

Listing 62: type.c

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

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

```

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

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

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

// Needs implementing: None
LangType type(Token* id_ref)
{
// Production 4.2
if (tokens_equal(&array_tok,
    current_tok, true))
{
char* errorMessage;
Token* numI;
Token* numF;
match(&array_tok, true); //
    array
match(&lbrac_tok, true); // [
numI = match(&num_tok, false);
     // num
match(&dotdot_tok, true); //
    ..
numF = match(&num_tok, false);
     // num
match(&rbrac_tok, true); // ]
match(&of_tok, true); // of
if (numI != NULL && numF !=
    NULL && id_ref != NULL)
if (type_lookup(numI -> aspect
     == 0 ? INT : REAL, numF ->
     aspect == 0 ? INT : REAL,
    &dotdot_tok) != ERR) {
if (numI -> int_val >= numF ->
     int_val) {
errorMessage= calloc(100,
    sizeof(*errorMessage));
sprintf(errorMessage, "
    Expected array end index %d
     to be strictly greater
    than start %d", numF ->
    int_val, numI -> int_val);
throw_sem_error(errorMessage);
}
id_ref -> array_length = numF
    -> int_val - numI ->
    int_val + 1;
}
return convert_to_array(
    standard_type());

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

synch();
return ERR;
}
```

Listing 63: variable.c

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

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

```

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

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

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

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

synch();
return ERR;
}
```

Listing 64: symbolTable.c

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

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

LinkedList* symbolTable;

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

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

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

return NULL;
}
```

Listing 65: addop.c

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

int addop(Token* storage, char* str, int start)
```

```
{
storage -> attribute = ADDOP;
switch (str[start])
{
case ’+’:
storage -> aspect = 0;
start++;
return start;

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

default: break;
}

return start;
}
```

Listing 66: catchall.c

```
#include<string.h>

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

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

return start;
}
```

Listing 67: grouping.c

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

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

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

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

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

default:
break;
}

return start;
}
```

Listing 68: idres.c

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

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

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

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

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

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

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

for (size_t i = 0; i < numReserved; i++) {
reservedWords[i] = (char *)
```

```
    node -> data;
node = node -> next;
}

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

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

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

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


return 0;
}

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

```
return 1;
}

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

return -1;
}

int idres(Token* storage, char
    * str, int start)
{
int initial = start;
LinkedList* id = malloc(sizeof
    (*id));
storage -> attribute = ID;
storage -> aspect = 0;
char next = str[start];
if (isalpha(next)) // Can
    actually be an id/reserved
{
size_t wordSize = 0;
do
{
wordSize = add(id, &next,
    sizeof(char*));
start++;
next = str[start];
} while(isalpha(next) ||
    isdigit(next)); // Match ID

// The string of the id name
char* name = malloc((wordSize
    + 1)*sizeof(char));
name[wordSize] = '\0';
struct node* node = id -> head
```

```
    ;
for (size_t i = 0; i <
    wordSize; i++) {
name[wordSize - i - 1] = *(
    char *)(node -> data);
node = node -> next;
}

int index = -1;
char* address = 0;
if ((index = isReserved(name))
     >= 0)
{ // It's a reserved word!
storage -> attribute =
    categories[index];
storage -> aspect = attributes
    [index];
}
else if ((address =
    checkSymbolTable(name)))
storage -> id = address;
else
storage -> id =
    pushToSymbolTable(name,
    wordSize);

}
if (start - initial > 10) //
    ID Too long err
{
//storage -> attribute = NOOP;
     TODO investigate
throw_lex_error(LEXERR, 1,
    initial, start - initial);
}
return start;
}
```

Listing 69: mulop.c

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

int mulop(Token* storage, char
    * str, int start)
{
storage -> attribute = MULOP;
if (str[start] == '*')
{
storage -> aspect = 0;
start++;
} else if (str[start] == '/')
{
storage -> aspect = 1;
start++;
}

return start;
}
```

Listing 70: numbers.c

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

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

// Assumes that "str" is valid
     as an integer.
char* parseNum(LinkedList*
    chars, bool real)
{
char* num = malloc((chars ->
    size + 1) * sizeof(char));
size_t count = chars -> size;
num[count--] = 0;
struct node* node = chars ->
    head;
while (node)
{
num[count--] = *(char *)node
    -> data;
node = node -> next;
}

return num;
}

```

```
double parseReal(LinkedList* digits)
{
char* array = parseNum(digits, true);
double val = strtod(array, NULL);
free(array);
return val;
}

int parseInt(LinkedList* digits)
{
char* array = parseNum(digits, false);
int val = (int) strtol(array, NULL, 10);
free(array);
return val;
}

int intMachine(Token* storage, char* str, int start)
{
storage -> attribute = NUM;

bool errored = false;
int initial = start;

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

if (start - initial > 10)
{
errored = true;
throw_lex_error(LEXERR, 2, initial, start - initial);
}
if (start > initial + 1 && str[initial] == '0')
{
errored = true;
throw_lex_error(LEXERR, 7, initial, start - initial);
}
// TODO investigate (all of these machines)
/*if (errored)
        storage -> attribute = NOOP;
    else*/ if (start > initial) // It's a proper integer!
{
storage -> aspect = 0;
storage -> int_val = parseInt(digits);
}

return start;
}

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

int initial = start;
bool errored = false;

int intPart = 0;
int fracPart = 0;

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

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

```

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


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

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

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

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

return start;
}

int longRealMachine(Token* storage, char* str, int start)
{
storage -> attribute = NUM;

int initial = start;
bool errored = false;

int intPart = 0;
int fracPart = 0;
int expPart = 0;

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

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

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

```
return initial;


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

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

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


// LONG REAL part
int signum = 0;

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

if (str[start] == ’+’ || str[start] == ’-’)
{
signum++;
add(digits, &str[start++], sizeof(char*));
}

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

expPart = start - (initial + fracPart + intPart + signum + 2);

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


// Now, we check for errors.
if (intPart > 5)
{
throw_lex_error(LEXERR, 3, initial, start - initial);
errored = true;
}
if (fracPart > 5)
{
throw_lex_error(LEXERR, 4, initial, start - initial);
errored = true;
}
if (str[initial] == ’0’ && intPart > 1) // Leading zero!
{
throw_lex_error(LEXERR, 8, initial, start - initial);
errored = true;
}
if (str[start - expPart - 2] == ’0’ && fracPart > 1) // Trailing zero in real!
{
throw_lex_error(LEXERR, 9, initial, start - initial);
errored = true;
}
if (expPart > 2) // Exponent too long!
{
throw_lex_error(LEXERR, 5, initial, start - initial);
errored = true;
}
if (str[start - expPart] == ’0’) // Leading zero in exponent!
{
throw_lex_error(LEXERR, 10, initial, start - initial);
errored = true;
}
/*
    if (errored)
        storage -> attribute =
        NOOP;
    else*/
{
storage -> aspect = 1;
storage -> real_val =
    parseReal(digits);
}

return start;
}
```

Listing 71: relop.c

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

int relop(Token* storage, char
    * str, int start)
{
storage -> attribute = RELOP;
char next = str[start];
switch (next) {
case ’<’:
start++;
if (str[start] == ’=’)
{
storage -> aspect = 1;
start++;
} else if (str[start] == ’>’)
{
storage -> aspect = 5;
start++;
} else {
storage -> aspect = 0;
}
break;

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

case ’>’:
start++;
if (str[start] == ’=’)
{
storage -> aspect = 4;
start++;
} else {
storage -> aspect = 3;
}
break;

default: break; // Do not
    increment; continue on to
    the next machine.
}

return start;
}
```

Listing 72: whitespace.c

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

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

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

Listing 73: tokenizer.c

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

#include "tokenizer.h"
#include "../dataStructures/
    linkedList/linkedlist.h"
```

```
#include "machines/machines.h"
#include "../errorHandler/
    errorHandler.h"
#include "../globals/globals.h
    "

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

// Initialization stuff
static bool initialized =
    false;

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

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

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

// Unrecognized symbol error.
    This error is manual
    because it takes
// the place of a lexeme,
    rather than being processed
     during one.
throw_lex_error(LEXERR, 0,
    START, 1);
//current -> attribute = NOOP;
START++;
return current;
} else {
fprintf(stderr, "%s\n", "
    Tokenizer not initialized.
    Aborting.");
return NULL;
}
}


Token* getNextToken()
{
Token* next = malloc(sizeof(*
    next));
do {
next = generateNextToken();
} while (next -> attribute ==
    NOOP);

return next;
}
```

Listing 74: tokens.c

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

#include "../errorHandler/
    errorHandler.h"
#include "tokens.h"

const char* catNames[] = {"
    NOOP", "FILEEND", "ASSIGNOP
    ", "RELOP", "ID",
 "CONTROL", "ADDOP", "MULOP", "
     WS", "ARRAY", "TYPE",
 "VAR", "NUM", "PUNC", "GROUP",
      "INVERSE",
 "LEXERR", "SYNERR", "SEMERR"};

const char* typeNames[] = {"
    ERR", "REAL", "INT", "BOOL"
    , "PROGRAM",
 "PROGRAM_PARAMETER", "
     PROCEDURE",
 "INT ARRAY", "REAL ARRAY"};

const Token eof_tok = {FILEEND
    , 0, false, 0, 0};
const Token lparen_tok = {
    GROUP, 0, true, 0, 0};
const Token rparen_tok = {
    GROUP, 1, true, 0, 0};
const Token plus_tok = {ADDOP,
     0, true, 0, 0};
const Token comma_tok = {PUNC,
     0, true, 0, 0};
const Token minus_tok = {ADDOP
    , 1, true, 0, 0};
const Token semic_tok = {PUNC,
     1, true, 0, 0};
const Token colon_tok = {TYPE,
     0, true, 0, 0};
const Token dotdot_tok = {
    ARRAY, 1, true, 0, 0};
const Token period_tok = {PUNC
    , 2, true, 0, 0};
const Token lbrac_tok = {GROUP
    , 2, true, 0, 0};
const Token rbrac_tok = {GROUP
    , 3, true, 0, 0};
const Token addop_tok = {ADDOP
    , 0, false, 0, 0};
const Token array_tok = {ARRAY
    , 0, true, 0, 0};
const Token assignop_tok = {
    ASSIGNOP, 0, true, 0, 0};
const Token begin_tok = {
    CONTROL, 0, true, 0, 0};
const Token call_tok = {
    CONTROL, 10, true, 0, 0};
const Token do_tok = {CONTROL,
     1, true, 0, 0};
const Token else_tok = {
    CONTROL, 2, true, 0, 0};
const Token end_tok = {CONTROL
    , 3, true, 0, 0};
const Token id_tok = {ID, 0,
    false, 0, 0};
const Token if_tok = {CONTROL,
     5, true, 0, 0};
const Token integer_tok = {
    TYPE, 1, true, 0, 0};
const Token integer_val_tok =
    {NUM, 0, true, 0, 0};
const Token of_tok = {ARRAY,
    2, true, 0, 0};
const Token real_val_tok = {
    NUM, 1, true, 0, 0};
const Token mulop_tok = {MULOP
    , 0, false, 0, 0};
const Token not_tok = {INVERSE
    , 0, true, 0, 0};
const Token num_tok = {NUM, 0,
     false, 0, 0};
const Token procedure_tok = {
    CONTROL, 6, true, 0, 0};
const Token program_tok = {
    CONTROL, 7, true, 0, 0};
const Token real_tok = {TYPE,
    2, true, 0, 0};
const Token relop_tok = {RELOP
    , 0, false, 0, 0};
const Token then_tok = {
    CONTROL, 8, true, 0, 0};
const Token var_tok = {VAR, 0,
     true, 0, 0};
const Token while_tok = {
    CONTROL, 9, true, 0, 0};

static const char* lexes[] = {
    "(", ")", "+", ",", "-", ";
    ", ":", "[", "]", "addop",
"array", "assignop", "begin",
    "call", "do", "else",
"end", "ID", "if", "integer",
    "mulop", "not",
"num", "procedure", "program",
     "real", "relop",
"then", "var", "while", "EOF",
     "..", ":", ".",
 "int value", "of", "real value
     "};

static const Token* tokens[] =
     {&lparen_tok, &rparen_tok,
     &plus_tok, &comma_tok, &
    minus_tok, &semic_tok,
&colon_tok, &lbrac_tok, &
    rbrac_tok, &addop_tok, &
    array_tok, &assignop_tok,
&begin_tok, &call_tok, &do_tok
    , &else_tok, &end_tok, &
    id_tok,
&if_tok, &integer_tok, &
    mulop_tok, &not_tok, &
    num_tok,
&procedure_tok, &program_tok,
    &real_tok, &relop_tok, &
    then_tok,
&var_tok, &while_tok, &eof_tok
    , &dotdot_tok,
&colon_tok, &period_tok, &
    integer_val_tok,
&of_tok, &real_val_tok};

const Token* getTokenFromLex(
    char* lex) {
for (int i = 0;i < sizeof(
    lexes); i++) {
if (strcmp(lexes[i], lex) ==
    0)
return tokens[i];
}

return NULL;
}

const char* getLexFromToken(
    Token* token, bool strict)
    {
switch (token -> attribute) {
case FILEEND: return "EOF";
case ASSIGNOP: return ":=";

case RELOP: if (strict)
switch (token -> aspect) {
case 0: return "<";
case 1: return "<=";
case 2: return "=";
case 3: return ">";
case 4: return ">=";
case 5: return "<>";
}
else return "RELOP";

case ID: return "ID";

case CONTROL: if (!strict)
    return "CONTROL"; else
switch (token -> aspect) {
case 0: return "begin";
case 1: return "do";
case 2: return "else";
case 3: return "end";
case 4: return "function";
case 5: return "if";
case 6: return "procedure";
case 7: return "program";
case 8: return "then";
case 9: return "while";
case 10: return "call";
}

case ADDOP: if (!strict) return "ADDOP"; else
switch (token -> aspect) {
case 0: return "+";
case 1: return "-";
}

case MULOP: if (!strict) return "MULOP"; else
switch (token -> aspect) {
case 0: return "*";
case 1: return "/";
}

case ARRAY: if (!strict) return "ARRAY"; else
switch (token -> aspect) {
case 0: return "array";
case 1: return "..";
case 2: return "of";
}

case TYPE: switch (token -> aspect) {
case 0: return ":";
case 1: return "integer";
case 2: return "real";
}

case VAR: switch (token -> aspect) {
case 0: return "var";
}

case NUM: if (!strict) return "a number"; else
switch (token -> aspect) {
case 0: return "integer value";
case 1: return "real value";
}

case PUNC: switch (token -> aspect) {
case 0: return ",";
case 1: return ";";
case 2: return ".";
}

case GROUP: switch (token -> aspect) {
case 0: return "(";
case 1: return ")";
case 2: return "[";
case 3: return "]";
}

case INVERSE: switch (token -> aspect) {
case 0: return "not";
}

case NOOP:
case WS:
case LEXERR:
case SYNERR:
case SEMERR: return "An error in the compiler has occurred.";
}
}

// Returns true if the tokens are equivalent, false otherwise
bool tokens_equal(const Token* p1, Token* p2, bool strict) {
return p1 -> attribute == p2 -> attribute &&
(!strict || p1 -> aspect == p2 -> aspect);
}

LangType convert_to_array(LangType type) {
char* errorMessage;
switch (type) {
case INT: return AINT;
case REAL: return AREAL;
```

```

// Type mismatch!!
default:errorMessage= calloc
    (150, sizeof(*errorMessage)
    );
sprintf(errorMessage, "Attempt
     to create array using type
     %s; must use integer or
    real instead!", typeNames[
    type]);
throw_sem_error(errorMessage);

case ERR: return ERR;
}
}

LangType convert_from_array(
    LangType type) {
char* errorMessage;
switch (type) {
case AINT: return INT;
case AREAL: return REAL;


default: errorMessage = calloc
    (100, sizeof(*errorMessage)
    );
 sprintf(errorMessage, "Attempt
      to index variable of type
      %s!", typeNames[type]);
 throw_sem_error(errorMessage);
case ERR: return ERR;
}
}

static LangType
    assignop_lookup(LangType
    first, LangType second) {
char* errorMessage;
if (first == ERR || second ==
    ERR) // just an err
return ERR;
else if (first != INT && first
     != REAL) {
errorMessage= calloc(100,
    sizeof(*errorMessage));
sprintf(errorMessage, "Cannot
    assign values to variables
    of type %s!", typeNames[
    first]);
throw_sem_error(errorMessage);
return ERR;
}
else if (second != INT &&
    second != REAL) {
errorMessage= calloc(100,
    sizeof(*errorMessage));
sprintf(errorMessage, "Attempt
     to assign %s value; only
    reals and integers can be
    assigned!", typeNames[
    second]);
throw_sem_error(errorMessage);
return ERR;
}
else if (first != second) {
errorMessage= calloc(100,
    sizeof(*errorMessage));
sprintf(errorMessage, "Attempt
     to convert type %s into
    type %s in assignment!",
    typeNames[first], typeNames
    [second]);
throw_sem_error(errorMessage);
return ERR;
}

return NULL;
}

static LangType relop_lookup(
    LangType first, LangType
    second) {
char* errorMessage;
if (first == second && (first
    == INT || first == REAL))
return BOOL;
else if (first != ERR &&
    second != ERR) {
errorMessage= calloc(100,
    sizeof(*errorMessage));
sprintf(errorMessage, "Attempt
     to compare incompatible
    types %s and %s!",
    typeNames[first], typeNames
    [second]);
throw_sem_error(errorMessage);
}

return ERR;
}

static LangType addop_lookup(
    LangType first, LangType
    second, int opcode) {
char* errorMessage;
switch (opcode) {
case 0:
case 1: if (first == second &&
     (first == INT || first ==
    REAL))
return first;
else if (first != ERR &&
    second != ERR) {
errorMessage= calloc(100,
    sizeof(*errorMessage));
sprintf(errorMessage, "Attempt
     to add incompatible types
    %s and %s!", typeNames[
    first], typeNames[second]);
throw_sem_error(errorMessage);
return ERR;
}

return ERR;


case 2: if (first == second &&
     first == BOOL)
return BOOL;
else if (first != ERR &&
    second != ERR) {
errorMessage= calloc(100,
    sizeof(*errorMessage));
sprintf(errorMessage, "
    Expected BOOL and BOOL for
    use with ’or’, received %s
    and %s!", typeNames[first],
     typeNames[second]);
throw_sem_error(errorMessage);
}

return ERR;

default: return NULL;
}
}

static LangType mulop_lookup(
    LangType first, LangType
    second, int opcode) {
char* errorMessage;

switch (opcode) {
case 0:
case 1: if (first == second &&
     (first == INT || first ==
    REAL))
return first;
else if ((first == REAL &&
    second == INT)
 || (first == INT && second ==
     REAL)) {
 errorMessage= calloc(100,
     sizeof(*errorMessage));
 sprintf(errorMessage, "Attempt
      to multiply or divide
     incompatible types %s and
     %s!", typeNames[first],
     typeNames[second]);
 throw_sem_error(errorMessage);
}
else if (first != ERR &&
    second != ERR) {
errorMessage= calloc(100,
    sizeof(*errorMessage));
sprintf(errorMessage, "
    Expceted ints or reals for
    multiplication, received %s
     and %s!", typeNames[first
    ], typeNames[second]);
throw_sem_error(errorMessage);
}

```

```
return ERR;


case 2: if (first == second && first == BOOL) // and
return BOOL;
else if (first != ERR && second != ERR)
{
errorMessage= calloc(100, sizeof(*errorMessage));
sprintf(errorMessage, "Expected BOOL and BOOL for use with 'and', received %s and %s!", typeNames[first], typeNames[second]);
throw_sem_error(errorMessage);
}

return ERR;

case 3: // div; mod
case 4: if (first == second && first == INT)
return INT;
else if (first != ERR && second != ERR) {
errorMessage= calloc(100, sizeof(*errorMessage));
sprintf(errorMessage,
"Integers required with %s, received %s and %s!",
opcode == 3 ? "div" : "mod", typeNames[first],
typeNames[second]);
throw_sem_error(errorMessage);
}

return ERR;

default: return NULL;
}
}

static LangType not_lookup(LangType first, LangType second) {
char* errorMessage;

if (first == BOOL)
return BOOL;
else if (first != ERR)
{
errorMessage= calloc(100, sizeof(*errorMessage));
sprintf(errorMessage, "Expected BOOL use with 'not', received %s!", typeNames[first]);
throw_sem_error(errorMessage);
}

return ERR;
}

static LangType array_lookup(LangType first, LangType second) {
if (first == second && first == INT)
return INT;
else if (first != ERR)
{
char* errorMessage = calloc(100, sizeof(*errorMessage));
sprintf(errorMessage, "Attempt to index variable of type %s!", typeNames[first]);
throw_sem_error(errorMessage);
} else if (second != ERR){
char* errorMessage = calloc(100, sizeof(*errorMessage));
sprintf(errorMessage, "Attempt to use variable of type %s to index array!", typeNames[second]);
throw_sem_error(errorMessage);
}

return ERR;
```

```
}

LangType type_lookup(LangType
    first, LangType second,
    Token* op) {
if (first == ERR || second ==
    ERR || op == NULL)
return ERR;

switch (op -> attribute) {
// Operations which are
    meaninngless
case NOOP:
case LEXERR:
case SYNERR:
case SEMERR:
case GROUP:
case PUNC:
case FILEEND:
case ID:
case CONTROL:
case WS:
case TYPE:
case VAR:
case NUM: return NULL;

case ASSIGNOP: return
    assignop_lookup(first,
    second);
case RELOP: return
    relop_lookup(first, second)
    ;
case ADDOP: return
    addop_lookup(first, second,
     op -> aspect);
case ARRAY: return
    array_lookup(first, second)
    ;
case MULOP: return
    mulop_lookup(first, second,
     op -> aspect);
case INVERSE: return
    not_lookup(first, second);

}
}
```