**CS 4013 Projects 3/4:**

**Semantic Analysis and**

**Memory Address Computations**

David Glover

January 15, 2015

**Table of Contents**

**Introduction** …………………………………………………………………………………… 1

**Methodology** …………………………………………………………………………………... 1

*L-Attributed Definition* ………………………………………………….…………………........ 1

**Implementation** ……………………………………………………………………………….. 9

**Discussion and Conclusions** ……………………………………………………………... 10

**Bibliography** ….…………………………………………………………………………….... 10

**Appendix I: Sample Inputs and Outputs** …………………………………..……………. 11

**Appendix II: Program Listings** ……………………………………………………………. 14

**Introduction**

The compilation process is comprised of six main phases: lexical analysis, syntax analysis, semantic analysis, intermediate code generation, code optimization, and target code generation. The role of a semantic analyzer is to “[check] the source program for semantic errors and [gather] type information for the subsequent code-generation phase.” (Aho 8) In this project, the semantic analyzer performs type and scope checking, which are folded into the recursive descent parser constructed in Project 2. Memory address computation acts as an extension to scope checking by simply computing the offset of each variable within each scope of the source program.

**Methodology**

The type and scope checking were based on an L-Attributed Definition (LAD), which is illustrated below. LADs heavily rely upon LL(1) grammars using recursive descent parsers, which were created as part of Project 2. Inherited and synthesized attributes play an important role in type checking for LADs because they allow information to be passed between adjacent grammar productions. Scope checking for LADs is performed by using a blue/green node list representing variables and new scope headers, respectively, as well as a green node pointer stack. Finally, address computation is handled during type and scope checking by calculating the amount of memory required for each data type when a variable is declared in a new scope.

**L-Attributed Definition**

1.1 *program* → **program id** (1) **(** *identifier\_list* **)** **;** (2) *program*’ (3)

(1) {offset := 0;

memory\_list.append(id.lexeme, “FFFFFFFF”);

node = create\_green\_node(id, “PGNAME”);

push\_green\_node(node);

identifier\_list.inherited := get\_pointer(node)}

(2) {program’.inherited := identifier\_list.synthesized}

(3) {popGreenStack()}

1.2.1 *program’* → *compound\_statement* **.**

{}

1.2.2 *program’* → (1) *subprogram\_declarations* *compound\_statement* **.**

(1) {subprogram\_declarations.inherited := program’.inherited}

1.2.3 *program’* → (1) *declarations* (2) *program*”

(1) {declarations.inherited := program’.inherited}

(2) {program”.inherited := declarations.synthesized}

1.3.1 *program”* → *compound\_statement* **.**

{}

1.3.2 *program”* → (1) *subprogram\_declarations* *compound\_statement* **.**

(1) {subprogram\_declarations.inherited := program”.inherited}

2.1 *identifier\_list* → **id** (1) *identifier\_list*’ (2)

(1) {memory\_list.append(id.lexeme, “FFFFFFFF”);

if (identifier\_list.inherited <> NULL {

node = create\_blue\_node(id.lexeme, “PGPARM”);

identifier\_list’.inherited := get\_pointer(node);}

else

identifier\_list’.inherited := identifier\_list.inherited}

(2) {identifier\_list.synthesized := identifier\_list’.synthesized}

2.2.1 *identifier\_list’* → **,** **id** (1) *identifier\_list*’1 (2)

(1) {memory\_list.append(id.lexeme, “FFFFFFFF”);

if (identifier\_list’.inherited <> NULL) {

node = create\_blue\_node(id.lexeme, “PGPARM”);

identifier\_list’1.inherited := get\_pointer(node)}

else

identifier\_list’1.inherited := identifier\_list’.inherited}

(2) {identifier\_list’.synthesized := identifier\_list’1.synthesized}

2.2.2 *identifier\_list’* → **ε**

{identifier\_list’.synthesized := identifier\_list’.inherited}

3.1 *declarations* → **var** **id** **:** *type* **;** (1) *declarations*’ (2)

(1) {memory\_list.append(id.lexeme, offset);

offset += type.size;

if (declarations.inherited <> NULL) {

node = create\_blue\_node(id.lexeme, type.type);

declarations’.inherited := get\_pointer(node)}

else {

node = create\_blue\_node(id.lexeme, type.type);

declarations’.inherited := declarations.inherited}}

(2) {declarations.synthesized := declarations’.synthesized}

3.2.1 *declarations’* → **var** **id** **:** *type* **;** (1) *declarations*’1 (2)

(1) {memory\_list.append(id.lexeme, offset);

offset += type.size;

node = create\_blue\_node(id.lexeme, type.type);

declarations’1.inherited := get\_pointer(node)}

(2) {declarations’.synthesized := declarations’1.synthesized}

3.2.2 *declarations’* → **ε**

{declarations’.synthesized := declarations’.inherited}

4.1.1 *type* → *standard\_type* (1)

(1) {type.type := standard\_type.type

type.size := standard\_type.size}

4.1.2 *type* → **array** **[** **num** **..** **num**1 **]** **of** *standard\_type* (1)

(1){if (num.type <> INT || num1.type <> num.type) {

type.type := ERR\*;

type.size := 0;

print(“Array indices must both be numerical integers.”)}

else if (num.val > num1.val) {

type.type := ERR\*;

type.size := 0;

print(“Invalid array range.”)}

else {

if (standard\_type.type = INT)

type.type := AINT;

else if (standard\_type.type = REAL)

type.type := AREAL;

type.size := (num1.val – num.val + 1) \* standard\_type.size}}

5.1.1 *standard\_type* → **integer** (1)

(1) {standard\_type.type := INT;

standard\_type.size := 4}

5.1.2 *standard\_type* → **real** (1)

(1) {standard\_type.type := REAL;

standard\_type.size := 8}

6.1 *subprogram\_declarations* → (1) *subprogram\_declaration* **;** (2) *subprogram\_declarations*’ (3)

(1) {subprogram\_declaration.inherited := subprogram\_declarations.inherited}

(2) {subprogram\_declarations’.inherited := subprogram\_declaration.synthesized}

(3) {subprogram\_declarations.synthesized := subprogram\_declarations’.synthesized}

6.2.1 *subprogram\_declarations’* → (1) *subprogram\_declaration* **;** (2) *subprogram\_declarations*’1 (3)

(1) {subprogram\_declaration.inherited := subprogram\_declarations’.inherited}

(2) {subprogram\_declarations’1.inherited := subprogram\_declaration.synthesized}

(3) {subprogram\_declarations’.synthesized := subprogram\_declarations’1.synthesized}

6.2.2 *subprogram\_declarations’* → **ε**

{}

7.1 *subprogram\_declaration* → (1) *subprogram\_head* (2) *subprogram\_declaration*’ (3)

(1) {subprogram\_head.inherited := subprogram\_declaration.inherited}

(2) {subprogram\_declaration’.inherited := subprogram\_head.synthesized}

(3) {subprogram\_declaration.synthesized := subprogram\_declaration’.synthesized}

7.2.1 *subprogram\_declaration’* → (1) *declarations* (2) *subprogram\_declaration*” (3)

(1) {declarations.inherited := subprogram\_declaration’.inherited}

(2) {subprogram\_declaration”.inherited := declarations.synthesized}

(3) {subprogram\_declaration’.synthesized := subprogram\_declaration”.synthesized}

7.2.2 *subprogram\_declaration’* → *compound\_statement* (1)

(1) {subprogram\_declaration’.synthesized := subprogram\_declaration’.inherited}

7.2.3 *subprogram\_declaration’* → (1) *subprogram\_declarations* *compound\_statement* (2)

(1) {subprogram\_declarations.inherited := subprogram\_declaration’.inherited}

(2) {subprogram\_declaration’.synthesized := subprogram\_declarations.synthesized}

7.3.1 *subprogram\_declaration”* → *compound\_statement* (1)

(1) {subprogram\_declaration”.synthesized := subprogram\_declaration”.inherited}

7.3.2 *subprogram\_declaration”* → (1) *subprogram\_declarations* *compound\_statement* (2)

(1) {subprogram\_declarations.inherited := subprogram\_declaration”.inherited}

(2) {subprogram\_declaration”.synthesized := subprogram\_declarations.synthesized}

8.1 *subprogram\_head* → **procedure id** (1) *subprogram\_head*’ (2)

(1) {memory\_list.append(id.lexeme, “FFFFFFFF”);

offset := 0;

node = create\_green\_node(id.lexeme, “PROC”);

if (subprogram\_head.inherited <> NULL)

subprogram\_head’.inherited := get\_pointer(node)

else

subprogram\_head’.inherited := NULL}

(2) {subprogram\_head.synthesized := subprogram\_head.inherited}

8.2.1 *subprogram\_head’* → (1) *arguments* **;** (2)

(1) {arguments.inherited := subprogram\_head’.inherited}

(2) {subprogram\_head’.synthesized := arguments.synthesized}

8.2.2 *subprogram\_head’* → **;** (1)

(1) {subprogram\_head’.synthesized := subprogram\_head’.inherited}

9.1 *arguments* → (1) **(** *parameter\_list* **)** (2)

(1) {parameter\_list.inherited := arguments.inherited}

(2) {arguments.synthesized := parameter\_list.synthesized}

10.1 *parameter\_list* → **id** **:** *type* (1) *parameter\_list*’ (2)

(1) {memory\_list.append(id.lexeme, “FFFFFFFF”);

if (type.type = “INT”)

node = create\_blue\_node(id.lexeme, “PPINT”);

else if (type.type = “REAL”)

node = create\_blue\_node(id.lexeme, “PPREAL”);

else if (type.type = “AINT”)

node = create\_blue\_node(id.lexeme, “PPAINT”);

else if (type.type = “AREAL”)

node = create\_blue\_node(id.lexeme, “PPAREAL”);

else

node = create\_blue\_node(id.lexeme, “ERR”);

parameter\_list’.inherited := [node.name]}

(2) {parameter\_list.synthesized := parameter\_list’.synthesized}

10.2.1 *parameter\_list’* → **;** **id** **:** *type* (1) *parameter\_list*’1 (2)

(1) {memory\_list.append(id.lexeme, “FFFFFFFF”);

if (type.type = “INT”)

node = create\_blue\_node(id.lexeme, “PPINT”);

else if (type.type = “REAL”)

node = create\_blue\_node(id.lexeme, “PPREAL”);

else if (type.type = “AINT”)

node = create\_blue\_node(id.lexeme, “PPAINT”);

else if (type.type = “AREAL”)

node = create\_blue\_node(id.lexeme, “PPAREAL”);

else

node = create\_blue\_node(id.lexeme, “ERR”);

if (id.lexeme in parameter\_list’.inherited)

print(“Cannot have repeated parameters.”);

inherited.append(node.name);

parameter\_list’1.inherited := parameter\_list’.inherited}

(2) {parameter\_list’.synthesized := parameter\_list’1.synthesized}

10.2.2 *parameter\_list’* → **ε** (1)

(1) {parameter\_list’.synthesized := parameter\_list’.inherited}

11.1 *compound\_statement* → **begin** *compound\_statement*’

{}

11.2.1 *compound\_statement’* → *statement\_list* **end**

{}

11.2.2 *compound\_statement’* → **end**

{}

12.1 *statement\_list* → *statement* *statement\_list*’

{}

12.2.1 *statement\_list’* → **;** *statement* *statement\_list*’1

{}

12.2.2 *statement\_list’* → **ε**

{}

13.1.1 *statement* → *variable* **assignop** *expression* (1)

(1) {if (variable.type <> expression.type && variable.type <> “ERR”)

print(“Assignop operand types mismatched with each other.”)}

13.1.2 *statement* → *procedure\_statement*

{}

13.1.3 *statement* → *compound\_statement*

{}

13.1.4 *statement* → **if** *expression* (1) **then** *statement* *statement*’

(1) {if (expression.type <> “BOOL”)

print(“Conditional expression type is not boolean.”)}

13.1.5 *statement* → **while** *expression* (1) **do** *statement*1

(1) {if (expression.type <> “BOOL”)

print(“Conditional expression type is not boolean.”)}

13.2.1 *statement’* → **else** *statement*

{}

13.2.2 *statement’* → **ε**

{}

14.1 *variable* → **id** (1) *variable*’ (2)

(1) {variable’.inherited := get\_pointer(id.lexeme)}

(2) {variable.type := variable’.type}

14.2.1 *variable’* → **[** *expression* **]** (1)

(1) {if (expression.type = “INT” && variable’.inherited <> NULL) {

if (variable’.inherited.type in [“PPAINT”|“AINT”])

variable’.type := “INT”;

else if (variable’.inherited.type in [“PPAREAL”|“AREAL”])

variable’.type := “REAL”;

else if (variable’.inherited.type = ERR)

variable’.type := “ERR”;

else {

variable’.type := “ERR\*”;

print(“Attempted to find index of non-array variable.”)}}

else {

variable’.type := “ERR\*”;

print(“Attempted to use invalid number or variable as array index.”)}}

14.2.2 *variable’* → **ε** (1)

(1) {if (variable’.inherited <> NULL)

if (variable’.inherited.type in [“INT”|“REAL”|“PPINT”|“PPREAL”|“PROC”])

variable’.type := variable’.inherited.type

else if (variable’.inherited.type = “ERR”)

variable’.type := “ERR”

else

variable’.type := ERR\*

print(“Not an assignable type.”)}

15.1 *procedure\_statement* → **call id** (1) *procedure\_statement*’

(1) {procedure\_statement’.inherited := getPointer(id)

15.2.1 *procedure\_statement’* → (1) **(** *expression\_list* **)**

(1) {expression\_list.inherited := procedure\_statement’.inherited}

15.2.2 *procedure\_statement’* → **ε**

{}

16.1 *expression\_list* → *expression* (1) *expression\_list*’ (2)

(1) {if (expression\_list.inherited <> NULL)

if (get\_num\_params(expression\_list.inherited) == 0)

print(“More than 0 parameters supplied for procedure call.”);

if (expression\_list.inherited.blue\_nodes = NULL || expression.type <> expression\_list.blue\_nodes[0].type[2:])

print(“Type of parameter 1 for procedure call does not match declared type.”);

count := 2;

expression\_list’.inherited := expression\_list.inherited}

(2) {expression\_list.synthesized := expression\_list’.synthesized}

16.2.1 *expression\_list’* → **,** *expression* (1) *expression\_list*’1 (2)

(1) {if (expression\_list’.inherited <> NULL && get\_num\_params(expression\_list’.inherited) > count – 1 && expression\_list’.inherited.blue\_nodes[count – 1].type[2:] <> expression.type)

print(“Type of parameter ” + count + “ for procedure call does not match declared type.”);

count += 1;

expression\_list’1.inherited := expression\_list’.inherited}}

(2) {expression\_list’.synthesized := expression\_list’1.synthesized}

16.2.2 *expression\_list’* → **ε** (1)

(1) {if (expression\_list’.inherited <> NULL && get\_num\_params(expression\_list’.inherited) > count - 1)

print(“Procedure call has too few parameters.”)

else if (expression\_list’.inherited <> NULL && get\_num\_params(expression\_list’.inherited) < count - 1)

print(“Procedure call has too many parameters.”)}

17.1 *expression* → *simple\_expression* (1) *expression*’ (2)

(1) {expression’.inherited := simple\_expression.type}

(2) {expression.type := expression’.type}

17.2.1 *expression’* → **relop** *simple\_expression* (1)

(1) {if (expression’.inherited = simple\_expression.type) {

if (expression’.inherited in [“INT”|“REAL”])

expression’.type := “BOOL”;

else if (expression’.inherited = “ERR\*”)

expression’.type := “ERR”;

else {

expression’.type := “ERR\*”;

print(“Invalid operand types for relop.”)}}

else {

expression’.type := “ERR\*”;

print(“Relop operand types mismatched with each other.”)}}

17.2.2 *expression’* → **ε** (1)

(1) {expression’.type := expression’.inherited}

18.1.1 *simple\_expression* → *term* (1) *simple\_expression*’ (2)

(1) {simple\_expression’.inherited := term.type}

(2) {simple\_expression.type := simple\_expression’.type}

18.1.2 *simple\_expression* → *sign term* (1) *simple\_expression*’ (2)

(1) {if (term.type in [“INT”|“REAL”])

simple\_expression’.inherited := term.type;

else {

simple\_expression’.inherited := “ERR\*”;

print(“Sign is inappropriate for the given term type.”)}}

(2) {simple\_expression.type := simple\_expression’.type}

18.2.1 *simple\_expression’* → **addop** *term* (1) *simple\_expression*’1 (2)

(1) {if (simple\_expression’.inherited = term.type) {

if ((addop.op = “OR” && term.type <> “BOOL”) || (addop.op <> “OR” && term.type not in [“INT”|“REAL”])) {

simple\_expression’1.inherited := “ERR\*”;

print("Operand types mismatched with addop type.")}

else if (term.type in [“BOOL”|“INT”|“REAL”])

simple\_expression’1.inherited := term.type}

else {

if simple\_expression’.inherited <> “ERR\*”

print(“Operand types mismatched with each other.”);

simple\_expression’1.inherited := “ERR\*”}}

(2) {simple\_expression’.type := simple\_expression’1.type}

18.2.2 *simple\_expression’* → **ε** (1)

(1) {simple\_expression’.type := simple\_expression’.inherited}

19.1 *term* → *factor* (1) *term*’ (2)

(1) {term’.inherited := factor.type}

(2) {term.type := term’.type}

19.2.1 *term’* → **mulop** *factor* (1) *term*’1 (2)

(1) {if (term’.inherited = factor.type) {

if (((mulop.op in [“\*”|“div”|“/”]) && factor.type <> “BOOL”) || (mulop.op = “MOD” && factor.type = “INT”) || (mulop.op = “AND” && factor.type = “BOOL”))

term’1.inherited := factor.type

else

term'1.inherited := “ERR\*”

print("Operand types mismatched with mulop type.")}

else {

if term’.inherited <> “ERR\*”

print("Operand types mismatched with each other.");

term’1.inherited := “ERR\*”}}

(2) {term’.type := term’1.type}

19.2.2 *term’* → **ε** (1)

(1) {term’.type := term’.inherited}

20.1.1 *factor* → **id** (1) *factor*’ (2)

(1) {factor’.inherited := get\_pointer(id)}

(2) {factor.type := factor’.type}

20.1.2 *factor* → **num** (1)

(1) {factor.type := num.type}

20.1.3 *factor* → **(** *expression* **)** (1)

(1) {factor.type := expression.type}

20.1.4 *factor* → **not** *factor*1 (1)

(1) {if (factor1.type in [“BOOL”|“ERR”])

factor.type := factor1.type

else {

factor.type = “ERR\*”;

print(“’not’ used with non-boolean operand.”)}}

20.2.1 *factor’* → **[** *expression* **]** (1)

(1) {if (expression.type = “INT”) {

if (factor’.inherited <> NULL) {

if (factor’.inherited.type in [“PPAINT”|“AINT”])

factor’.type := “INT”;

else if (factor’.inherited in [“PPAREAL”|“AREAL”])

factor’.type := “REAL”;

else {

factor’.type := “ERR\*”;

print(“Expression type is not an array.”)}

else {

factor’.type := “ERR\*”;

print(“Expression type is not an array.”)}}

else {

print(“Attempted to use invalid number or variable as array index.”);

factor’.type := “ERR\*”}}

20.2.2 *factor’* → **ε** (1)

(1) {if (factor’.inherited <> NULL)

if (factor’.inherited.type = “PPINT”)

factor’.type := “INT”;

else if (factor’.inherited.type = “PPREAL”)

factor’.type := “REAL”;

else if (factor’.inherited.type = “PPAINT”)

factor’.type := “AINT”;

else if (factor’.inherited.type = “PPAREAL”)

factor’.type := “AREAL”;

else

factor’.type := factor’.inherited.type}

21.1.1 *sign* → **+**

{}

21.1.2 *sign* → **–**

{}

**Implementation**

My implementation of the semantic analyzer and memory addresses is written using the Python programming language. I began this project by modifying the lexical analyzer to pass the symbol file to the recursive descent parser so it could be modified there. For type checking, I modified the nonterminal symbols in the recursive descent parser to have an inherited attribute, if necessary, and to return Type\_Wrapper data structures, which have attributes type, synthesized, value, and size. These were used to simulate the L-Attributed Definitions listed above.

For scope checking, I created a tree of Nodes and a stack of pointers to green Nodes. Green Nodes represent scopes, such as program and procedure declarations, whereas blue Nodes represent associated parameters and variables declared within each of the scopes. My Node data structures store color, name, type, children, and associated blue Nodes. The Node tree searches for the current Node by using the green Node stack. It begins at the bottom of the stack and matches that Node to the root of the Node tree, and works its way through the stack and tree until the node at the top of the stack is reached.

Finally, for memory address computation, I used a global variable called “offset” to keep track of the current offset for variables in each scope. Integers take 4 bytes, reals take 8 bytes, and arrays take (array\_size \* array\_type) bytes. “offset” is reset to 0 when a new scope is reached. At the end, all of the data is recorded in an output file.

**Discussion and Conclusions**

It took a couple of weeks for me to understand exactly what was expected of me in this project. Once I finally understood how type and scope checking worked, this project became straightforward. The type checking was the most difficult part of this project because I had to think through every possible type that any of the productions could take on and handle them accordingly. Once I completed type checking, I just had to review my notes on scope checking and wrote the code for the blue/green nodes. The memory address computation code was almost trivial because it made use of the code I had already written for the type and scope checking. When I started this project, it seemed complicated and daunting, but after thinking through and reviewing what I needed to do, it became straightforward.

**Bibliography**

Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. 1st ed. Bell Laboratories, 1986. Print.

**Appendix I: Sample Input and Output**

**src\_program**

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

e := a;

a := e - x;

call proc4(b, c)

end;

begin

a := e;

e := c[1.5]

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] + 5, c[5]);

call proc2(c[4], 2);

call proc3(1, 1.2e45);

if (a < 2.1) then a:= 1.1 else a := a + 2.1;

if (b > 4.2) then a := c[a]

end.

**listing\_file**

1 program test (input, output);

2 var a : integer;

3 var b : real;

4 var c : array [1..2] of integer;

5

6 procedure proc1(x: integer; y: real;

7 z: array [1..2] of integer; q: real);

8 var d: integer;

9 begin

10 a := 2;

11 z[a] := 4;

12 c[3] := 3

13 end;

14

15 procedure proc2(x: integer; y: integer);

16 var e: real;

17

18 procedure proc3(n: integer; z: real);

19 var e: integer;

20

21 procedure proc4(a: integer; z: array [1..3] of real);

22 var x: integer;

23 begin

24 e := a;

25 a := e - x;

26 call proc4(b, c)

SEMERR: Type of parameter 1 for call to proc4 does not match declared type.

SEMERR: Type of parameter 2 for call to proc4 does not match declared type.

27 end;

28

29 begin

30 a := e;

31 e := c[1.5]

SEMERR: Attempted to use invalid number or variable as array index.

32 end;

33

34 begin

35 call proc1(x, e, c, b);

36 call proc3(c[1], e);

37 e := e + 4.44;

38 a:= (a mod y) div x;

39 while ((a >= 4) and ((b <= e)

40 or (not (a = c[a])))) do

41 begin

42 a:= c[a] + 1 + %

LEXERR: Unrecognized symbol: %

SYNERR: Expected id, num, not, (; Received %

SEMERR: Addop operand types mismatched with each other.

43 end

44 end;

45

46 begin

47 call proc2(c[4] + 5, c[5]);

48 call proc2(c[4], 2);

49 call proc3(1, 1.2e45);

SEMERR: proc3 was not declared or is outside the current scope.

50 if (a < 2.1) then a:= 1.1 else a := a + 2.1;

SEMERR: Relop operand types mismatched with each other.

SEMERR: Conditional expression type is not boolean.

SEMERR: Assignop operand types mismatched with each other.

SEMERR: Addop operand types mismatched with each other.

51 if (b > 4.2) then a := c[a]

52 end.

**symbol\_file**

Name Address Type

test a80438cf PGNAME

input 5c9e730e PGPARM

output 3908a528 PGPARM

a c465bfd5 INT

b 9189d214 REAL

c 596df6a3 AINT

proc1 c6befb70 PROC

x 0ad7bf00 PPINT

y c35e06ee PPREAL

z a93f6494 PPAINT

q 632b59bd PPREAL

d d8fc6525 INT

proc2 c391bc03 PROC

e e6d58fe3 REAL

proc3 dcee6ca0 PROC

n 97e0c180 PPINT

proc4 6853c675 PROC

**mem\_addresses**

Name Address

test FFFFFFFF

input FFFFFFFF

output FFFFFFFF

a 0

b 4

c 12

proc1 FFFFFFFF

x FFFFFFFF

y FFFFFFFF

z FFFFFFFF

q FFFFFFFF

d 0

proc2 FFFFFFFF

x FFFFFFFF

y FFFFFFFF

e 0

proc3 FFFFFFFF

n FFFFFFFF

z FFFFFFFF

e 0

proc4 FFFFFFFF

a FFFFFFFF

z FFFFFFFF

x 0

**Appendix II: Program Listings**

**recursive\_descent.py**

'''

@author David Glover

CS 4013 - Compiler Construction

Project 3 and 4 - Type and Scope Checking and

Memory Address Computations

January 15, 2015

'''

from type\_wrapper import \*

from node import \*

class Recursive\_Descent:

def \_\_init\_\_(self, t\_list, listing\_file\_list, listing\_file, symbol\_list, symbol\_file, memory\_file):

global token\_list, token, l\_file\_lines, s\_file\_lines, s\_file\_lines\_types, scope\_tree, green\_node\_stack, m\_file\_lines

token\_list = t\_list

token = token\_list.pop(0)

l\_file\_lines = listing\_file\_list

s\_file\_lines = symbol\_list

s\_file\_lines\_types = []

m\_file\_lines = []

scope\_tree = Node\_Tree()

green\_node\_stack = []

if token\_list: self.program()

self.match('$')

for line in l\_file\_lines:

listing\_file.write(line)

listing\_file.close()

tabs = '\t\t\t'

symbol\_file.write('Name\t\tAddress\t\t\tType\n')

for line in s\_file\_lines\_types:

symbol\_file.write(str(line[0]) + tabs[0:len(tabs) - len(str(line[0])) / 4] + str(line[1]) + '\t\t' + str(line[2]) + '\n')

symbol\_file.close()

memory\_file.write('Name\t\tAddress\n')

for line in m\_file\_lines:

memory\_file.write(str(line[0] + tabs[0:len(tabs) - len(str(line[0])) / 4] + str(line[1]) + '\n'))

memory\_file.close()

def match(self, t):

global token\_list, token

i = 0

try:

while i < len(t) and int(t[i]) != token.token\_type:

i += 1

if i < len(t) and token.token\_type == int(t[i]) and token\_list:

token = token\_list.pop(0)

else:

if t == [1]: t = ['id']

elif t == [2]: t = ['relop']

elif t == [3]: t = ['mulop']

elif t == [4]: t = ['addop']

self.add\_error(token.line\_number, t, 0)

if token\_list: token = token\_list.pop(0)

except:

while i < len(t) and str(t[i]) != token.lexeme:

i += 1

if i < len(t) and token.lexeme == str(t[i]):

if t[i] != '$' and token\_list:

token = token\_list.pop(0)

else:

self.add\_error(token.line\_number, t, 0)

if token\_list: token = token\_list.pop(0)

def synch(self, synch\_set, firsts):

global token, token\_list

self.add\_error(token.line\_number, firsts, 0)

while token.token\_type not in synch\_set and token.lexeme not in synch\_set and token\_list:

token = token\_list.pop(0)

if not token\_list:

self.add\_error(token.line\_number, token.lexeme, 'id')

def add\_error(self, line\_num, exp, err\_type, err\_msg = None):

global l\_file\_lines, token

j = 0

while len(l\_file\_lines[j]) > 0 and ' '.join(l\_file\_lines[j].split()[:1]) != str(line\_num):

j += 1

while len(l\_file\_lines[j]) > 0 and j < len(l\_file\_lines) - 1 and (' '.join(l\_file\_lines[j].split()[:1]) in ['LEXERR:', 'SYNERR:', 'SEMERR:', 'No'] or ' '.join(l\_file\_lines[j + 1].split()[:1]) in ['LEXERR:', 'SYNERR:', 'SEMERR:', 'No']) and ' '.join(l\_file\_lines[j + 1].split()[:1]) != str(line\_num + 1):

j += 1

if err\_type == 0:

l\_file\_lines.insert(j + 1, 'SYNERR: Expected ' + ', '.join(str(e) for e in exp) + '; Received ' + token.lexeme + '\n')

elif err\_type == 1:

l\_file\_lines.insert(j + 1, 'No more tokens available. Synch failed while looking for ' + token.lexeme + '\n')

elif err\_type == 2:

l\_file\_lines.insert(j + 1, 'SEMERR: ' + err\_msg + '\n')

def add\_type(self, p\_id, p\_type):

global s\_file\_lines, s\_file\_lines\_types

i = 0

while i < len(s\_file\_lines) and p\_id.lexeme != s\_file\_lines[i][0]:

i += 1

if s\_file\_lines and i < len(s\_file\_lines) and p\_id.lexeme == s\_file\_lines[i][0]:

for s in s\_file\_lines\_types:

if p\_id.lexeme == s[0]:

return

s\_file\_lines\_types.append([p\_id.lexeme, s\_file\_lines[i][1], p\_type])

def get\_num\_params(self, node):

tmp = []

for n in node.blue\_nodes:

if n.type[:2] == 'PP':

tmp.append(node)

return len(tmp)

def get\_pointer(self, search\_node, l\_n):

global green\_node\_stack, scope\_tree

scope = []

scope.append(green\_node\_stack[0])

for c in scope\_tree.search(green\_node\_stack).children:

scope.append(c)

for a in green\_node\_stack[1:]:

scope.append(a)

if search\_node.color == 'blue':

for node in scope:

for b in node.blue\_nodes:

if search\_node.name == b.name and search\_node.color == b.color:

return b

else:

if len(green\_node\_stack) > 1:

for s in scope\_tree.search(green\_node\_stack[1:]).children:

scope.append(s)

for node in scope:

if node.name == search\_node.name:

return node

self.add\_error(l\_n, None, 2, search\_node.name + ' was not declared or is outside the current scope.')

return None

def push\_green\_node(self, node, l\_n):

global green\_node\_stack

green\_node\_stack.insert(0, node)

def push\_blue\_node(self, s\_node, node, l\_n):

global green\_node\_stack

for b in green\_node\_stack[0].blue\_nodes:

if node.name == b.name and node.color == b.color:

self.add\_error(l\_n, None, 2, node.name + ' has already been declared in this scope.')

s\_node.blue\_nodes.append(node)

def program(self):

global token, offset, scope\_tree, green\_node\_stack, m\_file\_lines

program\_w = Type\_Wrapper(None, None, None, 0)

if token.lexeme == 'program':

self.match(['program'])

p\_id = token

self.match([1])

self.add\_type(p\_id, 'PGNAME')

offset = 0

m\_file\_lines.append([p\_id.lexeme, 'FFFFFFFF'])

node = Node('green', p\_id.lexeme, 'PGNAME')

scope\_tree.insert\_node(node, green\_node\_stack)

self.push\_green\_node(node, p\_id.line\_number)

self.match(['('])

il\_w = self.identifier\_list(self.get\_pointer(Node('green', p\_id.lexeme, 'PGNAME'), p\_id.line\_number))

self.match([')'])

self.match([';'])

self.program2(il\_w.synthesized)

green\_node\_stack.pop(0)

else:

self.synch(['$', 'program'], ['program'])

return program\_w

def program2(self, inherited):

global token

program2\_w = Type\_Wrapper(None, None, None, 0)

if token.lexeme == 'begin':

self.compound\_statement()

self.match(['.'])

elif token.lexeme == 'procedure':

self.subprogram\_declarations(inherited)

self.compound\_statement()

self.match(['.'])

elif token.lexeme == 'var':

d\_w = self.declarations(inherited)

self.program3(d\_w.synthesized)

else:

self.synch(['$', 'begin', 'procedure', 'var'], ['begin', 'procedure', 'var'])

return program2\_w

def program3(self, inherited):

global token

program3\_w = Type\_Wrapper(None, None, None, 0)

if token.lexeme == 'begin':

self.compound\_statement()

self.match(['.'])

elif token.lexeme == 'procedure':

self.subprogram\_declarations(inherited)

self.compound\_statement()

self.match(['.'])

else:

self.synch(['$', 'begin', 'procedure'], ['begin', 'procedure'])

return program3\_w

def identifier\_list(self, inherited):

global token, scope\_tree, green\_node\_stack, m\_file\_lines

identifier\_list\_w = Type\_Wrapper(None, None, None, 0)

if token.token\_type == 1:

p\_id = token

self.add\_type(p\_id, 'PGPARM')

self.match([1])

m\_file\_lines.append([p\_id.lexeme, 'FFFFFFFF'])

if inherited:

node = scope\_tree.search(green\_node\_stack)

self.push\_blue\_node(node, Node('blue', p\_id.lexeme, 'PGPARM'), p\_id.line\_number)

il2\_w = self.identifier\_list2(self.get\_pointer(Node('blue', p\_id.lexeme, 'PGPARM'), p\_id.line\_number))

else:

il2\_w = self.identifier\_list2(inherited)

identifier\_list\_w.synthesized = il2\_w.synthesized

else:

self.synch(['$', 1, ')'], ['id'])

return identifier\_list\_w

def identifier\_list2(self, inherited):

global token, scope\_tree, green\_node\_stack, m\_file\_lines

identifier\_list2\_w = Type\_Wrapper(None, None, None, 0)

if token.lexeme == ',':

self.match([','])

p\_id = token

self.add\_type(p\_id, 'PGPARM')

self.match([1])

m\_file\_lines.append([p\_id.lexeme, 'FFFFFFFF'])

if inherited:

node = scope\_tree.search(green\_node\_stack)

self.push\_blue\_node(node, Node('blue', p\_id.lexeme, 'PGPARM'), p\_id.line\_number)

il2\_w = self.identifier\_list2(self.get\_pointer(Node('blue', p\_id.lexeme, 'PGPARM'), p\_id.line\_number))

else:

il2\_w = self.identifier\_list2(inherited)

identifier\_list2\_w.synthesized = il2\_w.synthesized

elif token.lexeme == ')':

identifier\_list2\_w.synthesized = inherited

else:

self.synch(['$', ',', ')'], [','])

return identifier\_list2\_w

def declarations(self, inherited):

global token, offset, m\_file\_lines

declarations\_w = Type\_Wrapper(None, None, None, 0)

if token.lexeme == 'var':

self.match(['var'])

p\_id = token

self.match([1])

self.match([':'])

t\_w = self.type()

self.match([';'])

m\_file\_lines.append([p\_id.lexeme, offset])

offset += t\_w.size

self.add\_type(p\_id, t\_w.type)

if inherited:

node = scope\_tree.search(green\_node\_stack)

self.push\_blue\_node(node, Node('blue', p\_id.lexeme, t\_w.type), p\_id.line\_number)

d2\_w = self.declarations2(self.get\_pointer(Node('blue', p\_id.lexeme, t\_w.type), p\_id.line\_number))

else:

node = scope\_tree.search(green\_node\_stack)

self.push\_blue\_node(node, Node('blue', p\_id.lexeme, t\_w.type), p\_id.line\_number)

d2\_w = self.declarations2(inherited)

declarations\_w.synthesized = d2\_w.synthesized

else:

self.synch(['$', 'var', 'begin', 'procedure'], ['var'])

return declarations\_w

def declarations2(self, inherited):

global token, offset, m\_file\_lines

declarations2\_w = Type\_Wrapper(None, None, None, 0)

if token.lexeme == 'var':

self.match(['var'])

p\_id = token

self.match([1])

self.match([':'])

t\_w = self.type()

self.match([';'])

m\_file\_lines.append([p\_id.lexeme, offset])

offset += t\_w.size

self.add\_type(p\_id, t\_w.type)

node = scope\_tree.search(green\_node\_stack)

self.push\_blue\_node(node, Node('blue', p\_id.lexeme, t\_w.type), p\_id.line\_number)

d2\_w = self.declarations2(self.get\_pointer(Node('blue', p\_id.lexeme, t\_w.type), p\_id.line\_number))

declarations2\_w.synthesized = d2\_w.synthesized

elif token.lexeme in ['procedure', 'begin']:

declarations2\_w.synthesized = inherited

else:

self.synch(['$', 'var', 'begin', 'procedure'], ['var'])

return declarations2\_w

def type(self):

global token

type\_w = Type\_Wrapper(None, None, None, 0)

if token.lexeme in ['integer', 'real']:

st\_w = self.standard\_type()

type\_w.type = st\_w.type

type\_w.size = st\_w.size

elif token.lexeme == 'array':

self.match(['array'])

self.match(['['])

num1 = token

self.match([32, 33, 34])

self.match(['..'])

num2 = token

self.match([32, 33, 34])

rbrack = token

self.match([']'])

self.match(['of'])

st\_w = self.standard\_type()

try:

if int(num1.lexeme) > int(num2.lexeme):

type\_w.type = 'ERR\*'

type\_w.size = 0

self.add\_error(rbrack.line\_number, None, 2, 'Invalid array range.')

else:

if st\_w.type in ['INT', 'REAL']:

type\_w.type = 'A' + st\_w.type

type\_w.size = (int(num2.lexeme) - int(num1.lexeme) + 1) \* int(st\_w.size)

except:

type\_w.type = 'ERR\*'

type\_w.size = 0

self.add\_error(num2.line\_number, None, 2, 'Array indices must both be numerical integers.')

else:

self.synch(['$', 'integer', 'real', 'array', ';', ')'], ['integer', 'real', 'array'])

return type\_w

def standard\_type(self):

global token

standard\_type\_w = Type\_Wrapper(None, None, None, 0)

if token.lexeme == 'integer':

self.match(['integer'])

standard\_type\_w.type = 'INT'

standard\_type\_w.size = 4

elif token.lexeme == 'real':

self.match(['real'])

standard\_type\_w.type = 'REAL'

standard\_type\_w.size = 8

else:

self.synch(['$', 'integer', 'real', ';', ')'], ['integer', 'real'])

return standard\_type\_w

def subprogram\_declarations(self, inherited):

global token

subprogram\_declarations\_w = Type\_Wrapper(None, None, None, 0)

if token.lexeme == 'procedure':

sd\_w = self.subprogram\_declaration(inherited)

self.match([';'])

sd2\_w = self.subprogram\_declarations2(sd\_w.synthesized)

subprogram\_declarations\_w.synthesized = sd2\_w.synthesized

else:

self.synch(['$', 'procedure', 'begin'], ['procedure'])

return subprogram\_declarations\_w

def subprogram\_declarations2(self, inherited):

global token

subprogram\_declarations2\_w = Type\_Wrapper(None, None, None, 0)

if token.lexeme == 'procedure':

sd\_w = self.subprogram\_declaration(inherited)

self.match([';'])

sd2\_w = self.subprogram\_declarations2(sd\_w.synthesized)

subprogram\_declarations2\_w.synthesized = sd2\_w.synthesized

elif token.lexeme != 'begin':

self.synch(['$', 'procedure', 'begin'], ['procedure'])

return subprogram\_declarations2\_w

def subprogram\_declaration(self, inherited):

global token, green\_node\_stack

subprogram\_declaration\_w = Type\_Wrapper(None, None, None, 0)

if token.lexeme == 'procedure':

sh\_w = self.subprogram\_head(inherited)

sd2\_w = self.subprogram\_declaration2(sh\_w.synthesized)

green\_node\_stack.pop(0)

subprogram\_declaration\_w.synthesized = sd2\_w.synthesized

else:

self.synch(['$', 'procedure', ';'], ['procedure'])

return subprogram\_declaration\_w

def subprogram\_declaration2(self, inherited):

global token

subprogram\_declaration2\_w = Type\_Wrapper(None, None, None, 0)

if token.lexeme == 'var':

d\_w = self.declarations(inherited)

sd3\_w = self.subprogram\_declaration3(d\_w.synthesized)

subprogram\_declaration2\_w.synthesized = sd3\_w.synthesized

elif token.lexeme == 'begin':

self.compound\_statement()

subprogram\_declaration2\_w.synthesized = inherited

elif token.lexeme == 'procedure':

sd\_w = self.subprogram\_declarations(inherited)

self.compound\_statement()

subprogram\_declaration2\_w.synthesized = sd\_w.synthesized

else:

self.synch(['$', 'var', 'begin', 'procedure', ';'], ['var', 'begin', 'procedure'])

return subprogram\_declaration2\_w

def subprogram\_declaration3(self, inherited):

global token

subprogram\_declaration3\_w = Type\_Wrapper(None, None, None, 0)

if token.lexeme == 'begin':

self.compound\_statement()

subprogram\_declaration3\_w.synthesized = inherited

elif token.lexeme == 'procedure':

sd\_w = self.subprogram\_declarations(inherited)

self.compound\_statement()

subprogram\_declaration3\_w.synthesized = sd\_w.synthesized

else:

self.synch(['$', 'begin', 'procedure', ';'], ['begin', 'procedure'])

return subprogram\_declaration3\_w

def subprogram\_head(self, inherited):

global token, offset, m\_file\_lines

subprogram\_head\_w = Type\_Wrapper(None, None, None, 0)

if token.lexeme == 'procedure':

self.match(['procedure'])

p\_id = token

self.add\_type(p\_id, 'PROC')

self.match([1])

m\_file\_lines.append([p\_id.lexeme, 'FFFFFFFF'])

offset = 0

new\_node = Node('green', p\_id.lexeme, 'PROC')

scope\_tree.insert\_node(new\_node, green\_node\_stack)

self.push\_green\_node(new\_node, p\_id.line\_number)

if inherited:

sh2\_w = self.subprogram\_head2(self.get\_pointer(Node('green', p\_id.lexeme, 'PROC'), p\_id.line\_number))

else:

sh2\_w = self.subprogram\_head2(None)

subprogram\_head\_w.synthesized = inherited

else:

self.synch(['$', 'procedure', 'var', 'begin'], ['procedure'])

return subprogram\_head\_w

def subprogram\_head2(self, inherited):

global token, green\_node\_stack

subprogram\_head2\_w = Type\_Wrapper(None, None, None, 0)

if token.lexeme == '(':

a\_w = self.arguments(inherited)

self.match([';'])

subprogram\_head2\_w.synthesized = a\_w.synthesized

elif token.lexeme == ';':

self.match([';'])

subprogram\_head2\_w.synthesized = inherited

else:

self.synch(['$', '(', ';', 'var', 'begin', 'procedure'], ['(', ';'])

return subprogram\_head2\_w

def arguments(self, inherited):

global token

arguments\_w = Type\_Wrapper(None, None, None, 0)

if token.lexeme == '(':

self.match(['('])

pl\_w = self.parameter\_list(inherited)

self.match([')'])

arguments\_w.synthesized = pl\_w.synthesized

else:

self.synch(['$', '(', ';'], ['('])

return arguments\_w

def parameter\_list(self, inherited):

global token, green\_node\_stack, m\_file\_lines

parameter\_list\_w = Type\_Wrapper(None, None, None, 0)

if token.token\_type == 1:

p\_id = token

self.match([1])

self.match([':'])

t\_w = self.type()

m\_file\_lines.append([p\_id.lexeme, 'FFFFFFFF'])

if t\_w.type in ['INT', 'REAL', 'AINT', 'AREAL']:

t\_w.type = 'PP' + t\_w.type

node = scope\_tree.search(green\_node\_stack)

self.push\_blue\_node(node, Node('blue', p\_id.lexeme, t\_w.type), p\_id.line\_number)

else:

t\_w.type = 'ERR'

node = scope\_tree.search(green\_node\_stack)

self.push\_blue\_node(node, Node('blue', p\_id.lexeme, 'ERR'), p\_id.line\_number)

self.add\_type(p\_id, t\_w.type)

pl2\_w = self.parameter\_list2([node.name])

parameter\_list\_w.synthesized = pl2\_w.synthesized

else:

self.synch(['$', 1, ')'], ['id'])

return parameter\_list\_w

def parameter\_list2(self, inherited):

global token

parameter\_list2\_w = Type\_Wrapper(None, None, None, 0)

if token.lexeme == ';':

self.match([';'])

p\_id = token

self.match([1])

self.match([':'])

t\_w = self.type()

m\_file\_lines.append([p\_id.lexeme, 'FFFFFFFF'])

if t\_w.type in ['INT', 'REAL', 'AINT', 'AREAL']:

t\_w.type = 'PP' + t\_w.type

node = scope\_tree.search(green\_node\_stack)

self.push\_blue\_node(node, Node('blue', p\_id.lexeme, t\_w.type), p\_id.line\_number)

else:

t\_w.type = 'ERR'

node = scope\_tree.search(green\_node\_stack)

self.push\_blue\_node(node, Node('blue', p\_id.lexeme, 'ERR'), p\_id.line\_number)

self.add\_type(p\_id, t\_w.type)

if p\_id.lexeme in inherited:

self.add\_error(token.line\_number, None, 2, 'Cannot have repeated parameters.')

inherited.append(node.name)

pl2\_w = self.parameter\_list2(inherited)

parameter\_list2\_w.synthesized = pl2\_w.synthesized

elif token.lexeme == ')':

parameter\_list2\_w.synthesized = inherited

else:

self.synch(['$', ';', ')'], [';'])

return parameter\_list2\_w

def compound\_statement(self):

global token

compound\_statement\_w = Type\_Wrapper(None, None, None, 0)

if token.lexeme == 'begin':

self.match(['begin'])

self.compound\_statement2()

else:

self.synch(['$', 'begin', '.', 'else', ';', 'end'], ['begin'])

return compound\_statement\_w

def compound\_statement2(self):

global token

compound\_statement2\_w = Type\_Wrapper(None, None, None, 0)

if token.token\_type == 1 or token.lexeme in ['begin', 'if', 'while', 'call']:

self.statement\_list()

self.match(['end'])

elif token.lexeme == 'end':

self.match(['end'])

else:

self.synch(['$', 1, 'call', 'begin', 'if', 'while', 'end', '.', 'else', ';'], ['id', 'begin', 'if', 'while', 'call'])

return compound\_statement2\_w

def statement\_list(self):

global token

statement\_list\_w = Type\_Wrapper(None, None, None, 0)

if token.token\_type == 1 or token.lexeme in ['begin', 'if', 'while', 'call']:

self.statement()

self.statement\_list2()

else:

self.synch(['$', 1, 'call', 'begin', 'if', 'while', 'end'], ['id', 'begin', 'if', 'while', 'call'])

return statement\_list\_w

def statement\_list2(self):

global token

statement\_list2\_w = Type\_Wrapper(None, None, None, 0)

if token.lexeme == ';':

self.match([';'])

self.statement()

self.statement\_list2()

elif token.lexeme != 'end':

self.synch(['$', ';', 'end'], [';'])

return statement\_list2\_w

def statement(self):

global token

statement\_w = Type\_Wrapper(None, None, None, 0)

if token.token\_type == 1:

v\_w = self.variable()

self.match([30])

p\_id = token

e\_w = self.expression()

tmp1 = v\_w.type

tmp2 = e\_w.type

if tmp1[:2] == 'PP': tmp1 = tmp1[2:]

if tmp2[:2] == 'PP': tmp2 = tmp2[2:]

if tmp1 != tmp2 and tmp1 not in ['ERR', 'ERR\*'] and tmp2 not in ['ERR', 'ERR\*']:

self.add\_error(p\_id.line\_number, None, 2, 'Assignop operand types mismatched with each other.')

elif token.lexeme == 'call':

self.procedure\_statement()

elif token.lexeme == 'begin':

self.compound\_statement()

elif token.lexeme == 'if':

self.match(['if'])

e\_w = self.expression()

if e\_w.type != 'BOOL':

self.add\_error(token.line\_number, None, 2, 'Conditional expression type is not boolean.')

self.match(['then'])

self.statement()

self.statement2()

elif token.lexeme == 'while':

self.match(['while'])

e\_w = self.expression()

if e\_w.type != 'BOOL':

self.add\_error(token.line\_number, None, 2, 'Conditional expression type is not boolean.')

self.match(['do'])

self.statement()

else:

self.synch(['$', 1, 'call', 'begin', 'if', 'while', 'else', ';', 'end'], ['id', 'call', 'begin', 'if', 'while'])

return statement\_w

def statement2(self):

global token

statement2\_w = Type\_Wrapper(None, None, None, 0)

if token.lexeme == 'else':

self.match(['else'])

self.statement()

elif token.lexeme not in [';', 'end']:

self.synch(['$', 'else', ';', 'end'], ['else'])

return statement2\_w

def variable(self):

global token

variable\_w = Type\_Wrapper(None, None, None, 0)

if token.token\_type == 1:

p\_id = token

self.match([1])

v2\_w = self.variable2(self.get\_pointer(Node('blue', p\_id.lexeme, None), p\_id.line\_number), p\_id.lexeme)

variable\_w.type = v2\_w.type

else:

self.synch(['$', 1, 30], ['id'])

return variable\_w

def variable2(self, inherited, p\_id):

global token

variable2\_w = Type\_Wrapper(None, None, None, 0)

if token.lexeme == '[':

self.match(['['])

e\_w = self.expression()

self.match([']'])

if e\_w.type == 'INT' and inherited:

if inherited.type in ['PPAINT', 'AINT']:

variable2\_w.type = 'INT'

elif inherited.type in ['PPAREAL', 'AREAL']:

variable2\_w.type = 'REAL'

elif inherited.type == 'ERR':

variable2\_w.type = 'ERR'

else:

variable2\_w.type = 'ERR\*'

self.add\_error(token.line\_number, None, 2, 'Attempted to find index of non-array variable.')

else:

variable2\_w.type = 'ERR\*'

self.add\_error(token.line\_number, None, 2, 'Attempted to use invalid number or variable as array index.')

elif token.token\_type == 30:

if inherited:

if inherited.type in ['INT', 'REAL', 'PPINT', 'PPREAL', 'PROC']:

variable2\_w.type = inherited.type

elif inherited.type == 'ERR':

variable2\_w.type = 'ERR'

else:

variable2\_w.type = 'ERR\*'

self.add\_error(token.line\_number, None, 2, p\_id + ' is not an assignable type.')

else:

variable2\_w.type = 'ERR\*'

self.add\_error(token.line\_number, None, 2, p\_id + ' is not an assignable type.')

else:

self.synch(['$', '[', 30], ['['])

return variable2\_w

def procedure\_statement(self):

global token, green\_node\_stack, scope\_tree

procedure\_statement\_w = Type\_Wrapper(None, None, None, 0)

if token.lexeme == 'call':

self.match(['call'])

p\_id = token

self.match([1])

self.procedure\_statement2(self.get\_pointer(Node('green', p\_id.lexeme, 'PROC'), p\_id.line\_number))

else:

self.synch(['$', 'call', 'else', ';', 'end'], ['call'])

return procedure\_statement\_w

def procedure\_statement2(self, inherited):

global token

procedure\_statement2\_w = Type\_Wrapper(None, None, None, 0)

if token.lexeme == '(':

self.match(['('])

self.expression\_list(inherited)

self.match(')')

elif token.lexeme not in [';', 'end', 'else']:

self.synch(['$', '(', 'else', ';', 'end'], ['('])

return procedure\_statement2\_w

def expression\_list(self, inherited):

global token, scope\_tree, green\_node\_stack

expression\_list\_w = Type\_Wrapper(None, None, None, 0)

if token.token\_type in [1, 32, 33, 34] or token.lexeme in ['not', '+', '-', '(']:

e\_w = self.expression()

if inherited:

if self.get\_num\_params(inherited) == 0:

self.add\_error(token.line\_number, None, 2, 'More than 0 parameters supplied for call to ' + inherited.name + '.')

if not inherited.blue\_nodes or e\_w.type != inherited.blue\_nodes[0].type[2:]:

self.add\_error(token.line\_number, None, 2, 'Type of parameter 1 for call to ' + inherited.name + ' does not match declared type.')

el2\_w = self.expression\_list2(inherited, 2)

expression\_list\_w.synthesized = el2\_w.synthesized

else:

self.synch(['$', 1, 32, 33, 34, '(', 'not', '+', '-', ')'], ['id', 'num', 'not', '+', '-', '('])

return expression\_list\_w

def expression\_list2(self, inherited, count):

global token, scope\_tree

expression\_list2\_w = Type\_Wrapper(None, None, None, 0)

if token.lexeme == ',':

self.match([','])

e\_w = self.expression()

if inherited and self.get\_num\_params(inherited) > count - 1 and inherited.blue\_nodes[count - 1].type[2:] != e\_w.type:

self.add\_error(token.line\_number, None, 2, 'Type of parameter ' + str(count) + ' for call to ' + inherited.name + ' does not match declared type.')

count += 1

el2\_w = self.expression\_list2(inherited, count)

expression\_list2\_w.synthesized = el2\_w.synthesized

elif token.lexeme == ')':

if inherited and self.get\_num\_params(inherited) > count - 1:

self.add\_error(token.line\_number, None, 2, 'Procedure call to ' + inherited.name + ' has too few parameters.')

elif inherited and self.get\_num\_params(inherited) < count - 1:

self.add\_error(token.line\_number, None, 2, 'Procedure call to ' + inherited.name + ' has too many parameters.')

else:

self.synch(['$', ',', ')'], [','])

return expression\_list2\_w

def expression(self):

global token

expression\_w = Type\_Wrapper(None, None, None, 0)

if token.token\_type in [1, 32, 33, 34] or token.lexeme in ['not', '+', '-', '(']:

p\_id = token

se\_w = self.simple\_expression()

e2\_w = self.expression2(se\_w.type)

expression\_w.type = e2\_w.type

else:

self.synch(['$', 1, 32, 33, 34, '(', 'not', '+', '-', ']', ')', ',', 'then', 'do', 'else', ';', 'end'], ['id', 'num', 'not', '+', '-', '('])

return expression\_w

def expression2(self, inherited):

global token

expression2\_w = Type\_Wrapper(None, None, None, 0)

if token.token\_type == 2:

self.match([2])

se\_w = self.simple\_expression()

if inherited == se\_w.type:

if inherited in ['INT', 'REAL']:

expression2\_w.type = 'BOOL'

elif inherited == 'ERR\*':

expression2\_w.type = 'ERR'

else:

expression2\_w.type = 'ERR\*'

self.add\_error(token.line\_number, None, 2, 'Invalid operand types for relop.')

else:

expression2\_w.type = 'ERR\*'

self.add\_error(token.line\_number, None, 2, 'Relop operand types mismatched with each other.')

elif token.lexeme in ['end', 'then', 'else', 'do', ')', ']', ',', ';']:

if inherited:

expression2\_w.type = inherited

else:

expression2\_w.type = 'ERR\*'

else:

self.synch(['$', 2, ']', ')', ',', 'then', 'do', 'else', ';', 'end'], ['relop'])

return expression2\_w

def simple\_expression(self):

global token

simple\_expression\_w = Type\_Wrapper(None, None, None, 0)

if token.token\_type in [1, 32, 33, 34] or token.lexeme in ['not', '(']:

t\_w = self.term()

se2\_w = self.simple\_expression2(t\_w.type)

simple\_expression\_w.type = se2\_w.type

elif token.lexeme in ['+', '-']:

self.sign()

t\_w = self.term()

if t\_w.type in ['INT', 'REAL']:

se2\_w = self.simple\_expression2(t\_w.type)

else:

se2\_w = self.simple\_expression2('ERR\*')

self.add\_error(token.line\_number, None, 2, 'Sign is inappropriate for the given term type.')

simple\_expression\_w.type = se2\_w.type

else:

self.synch(['$', 1, 32, 33, 34, '(', 'not', '+', '-', 2, ']', ')', ',', 'then', 'do', 'else', ';', 'end'], ['id', 'num', 'not', '(', '+', '-'])

return simple\_expression\_w

def simple\_expression2(self, inherited):

global token

simple\_expression2\_w = Type\_Wrapper(None, None, None, 0)

if token.token\_type == 4:

addop = token

self.match([4])

t\_w = self.term()

if inherited == t\_w.type:

if (addop.lexeme.lower() == 'or' and t\_w.type != 'BOOL') or (addop.lexeme.lower() != 'or' and t\_w.type not in ['INT', 'REAL']):

se2\_w = self.simple\_expression2('ERR\*')

self.add\_error(token.line\_number, None, 2, 'Operand types mismatched with addop type.')

elif t\_w.type in ['BOOL', 'INT', 'REAL']:

se2\_w = self.simple\_expression2(t\_w.type)

else:

if inherited != 'ERR\*':

self.add\_error(addop.line\_number, None, 2, 'Addop operand types mismatched with each other.')

se2\_w = self.simple\_expression2('ERR\*')

simple\_expression2\_w.type = se2\_w.type

elif token.token\_type == 2 or token.lexeme in ['end', 'then', 'else', 'do', ')', ']', ',', ';']:

simple\_expression2\_w.type = inherited

else:

self.synch(['$', 4, 2, ']', ')', ',', 'then', 'do', 'else', ';', 'end'], ['addop'])

return simple\_expression2\_w

def term(self):

global token

term\_w = Type\_Wrapper(None, None, None, 0)

if token.token\_type in [1, 32, 33, 34] or token.lexeme in ['not', '(']:

f\_w = self.factor()

t2\_w = self.term2(f\_w.type)

term\_w.type = t2\_w.type

else:

self.synch(['$', 1, 32, 33, 34, '(', 'not', 4, 2, ']', ')', ',', 'then', 'do', 'else', ';', 'end'], ['id', 'num', 'not', '('])

return term\_w

def term2(self, inherited):

global token

term2\_w = Type\_Wrapper(None, None, None, 0)

if token.token\_type == 3:

mulop = token.lexeme.lower()

self.match([3])

f\_w = self.factor()

if inherited == f\_w.type:

if (mulop in ['\*', 'div', '/'] and f\_w.type != 'BOOL') or (mulop == 'mod' and f\_w.type == 'INT') or (mulop == 'and' and f\_w.type == 'BOOL'):

t2\_w = self.term2(f\_w.type)

else:

if f\_w.type != 'ERR\*':

self.add\_error(token.line\_number, None, 2, 'Operand types mismatched with mulop type.')

t2\_w = self.term2('ERR\*')

else:

if inherited != 'ERR\*':

self.add\_error(token.line\_number, None, 2, 'Mulop operand types mismatched with each other.')

t2\_w = self.term2('ERR\*')

term2\_w.type = t2\_w.type

elif token.token\_type in [2, 4] or token.lexeme in ['end', 'then', 'else', 'do', ')', ']', ',', ';']:

term2\_w.type = inherited

else:

self.synch(['$', 3, 4, 2, ']', ')', ',', 'then', 'do', 'else', ';', 'end'], ['mulop'])

return term2\_w

def factor(self):

global token

factor\_w = Type\_Wrapper(None, None, None, 0)

if token.token\_type == 1:

p\_id = token

self.match([1])

f2\_w = self.factor2(self.get\_pointer(Node('blue', p\_id.lexeme, None), p\_id.line\_number), p\_id)

factor\_w.type = f2\_w.type

elif token.token\_type in [32, 33, 34]:

num = token

self.match([32, 33, 34])

if num.token\_type == 33:

factor\_w.type = 'INT'

elif num.token\_type in [32, 34]:

factor\_w.type = 'REAL'

elif token.lexeme == '(':

self.match(['('])

e\_w = self.expression()

self.match([')'])

factor\_w.type = e\_w.type

elif token.lexeme == 'not':

self.match(['not'])

f\_w = self.factor()

if f\_w.type in ['BOOL', 'ERR']:

factor\_w.type = f\_w.type

else:

factor\_w.type = 'ERR\*'

self.add\_error(token.line\_number, None, 2, '\'not\' used with non-boolean operand.')

else:

self.synch(['$', 1, 32, 33, 34, '(', 'not', 3, 4, 2, ']', ')', ',', 'then', 'do', 'else', ';', 'end'], ['id', 'num', '(', 'not'])

return factor\_w

def factor2(self, inherited, p\_id):

global token

factor2\_w = Type\_Wrapper(None, None, None, 0)

if token.lexeme == '[':

self.match(['['])

e\_w = self.expression()

rbrack = token

self.match([']'])

if e\_w.type == 'INT':

if inherited:

if inherited.type in ['PPAINT', 'AINT']:

factor2\_w.type = 'INT'

elif inherited.type in ['PPAREAL', 'AREAL']:

factor2\_w.type = 'REAL'

else:

factor2\_w.type = 'ERR\*'

self.add\_error(rbrack.line\_number, None, 2, 'Expression type is not an array.')

else:

factor2\_w.type = 'ERR\*'

self.add\_error(rbrack.line\_number, None, 2, 'Expression type is not an array.')

else:

factor2\_w.type = 'ERR\*'

self.add\_error(rbrack.line\_number, None, 2, 'Attempted to use invalid number or variable as array index.')

elif token.token\_type in [2, 3, 4] or token.lexeme in ['end', 'then', 'else', 'do', ')', ']', ',', ';']:

if inherited:

if inherited.type in ['PPINT', 'PPREAL', 'PPAINT', 'PPAREAL']:

factor2\_w.type = inherited.type[2:]

else:

factor2\_w.type = inherited.type

else:

self.synch(['$', '[', 3, 4, 2, ']', ')', ',', 'then', 'do', 'else', ';', 'end'], ['['])

return factor2\_w

def sign(self):

global token

if token.lexeme == '+':

self.match(['+'])

elif token.lexeme == '-':

self.match(['-'])

else:

self.synch(['$', '+', '-', 1, 32, 33, 34, '(', 'not'], ['+', '-'])

**type\_wrapper.py**

class Type\_Wrapper():

def \_\_init\_\_(self, t, synth, v, s):

self.type = t

self.synthesized = synth

self.value = v

self.size = s

**node.py**

class Node():

def \_\_init\_\_(self, c, n, t):

self.color = c

self.name = n

self.type = t

self.children = []

self.blue\_nodes = []

def \_\_eq\_\_(self, other):

return self.color == other.color and self.name == other.name and self.type == other.type

class Node\_Tree():

def \_\_init\_\_(self):

self.root = None

def insert\_node(self, new\_node, green\_node\_stack):

if len(green\_node\_stack) == 0:

self.root = new\_node

else:

node = self.search(green\_node\_stack)

node.children.append(new\_node)

def search(self, green\_node\_stack):

target\_node = green\_node\_stack[0]

curr\_node = green\_node\_stack[len(green\_node\_stack) - 1]

if len(green\_node\_stack) == 1 and curr\_node == target\_node:

return curr\_node

else:

for child in curr\_node.children:

node = self.search(green\_node\_stack[:len(green\_node\_stack) - 1])

if node == child:

return node

return node