**CS4013**

**Project 3 & 4**

A Semantics Analyzer and addressing scheme for a subset of the Pascal programming language

By Carson Ellsworth

0976

**Part 1)** **Introduction:**

For the third project in building a front-end compiler, I was assigned to building a semantic analyzer. Semantic Analyzer’s dissect the meaning of a program to analyze the type information and catch errors within the program based on the programming language specifications, this includes base type checking of variables as well as the scoping rules of the language. For the Semantic Analyzer to work correctly in my one pass implementation of the compiler, I built it upon the underlying structure set up for Syntax analysis in project 2.

The type checking done for this compiler involves six types, integers, reals, array of integers, array of reals, Booleans, and type error. The first four types are able to have variables and function parameters defined with such types. The way to have access to the Boolean type is to use some valid combination of the four using an expression that utilizes a relational operator. And the way to acquire a type error is to simply use an invalid combination of the types.

For the combination of the types to be valid a project parameter was declared so that no mixed mode arithmetic (MMA) would be allowed for the compiler. MMA is the arithmetic that deals with the type analysis of two different types given a single operator. For example, to analyze the proper type of the expression a:= 5 + 5.5; a compiler will need to know how to add two different types together. In this case the addition of a real and integer has been determined to be a real because this helps prevent loss of information.

To properly type check the pascal subset language I also would need to build a scoping system to account for variable and function creations within certain levels of the program, the scoping system helps determine what variable and it’s type is being referred to in a program if there are many instances of that same variable name used. This also allows for the catching of semantic errors of variables being used outside the bounds of where they are allowed to be called from.

Finally, this project also covers generating addresses for variables within local scopes. If the rest of this project is implemented correctly then doing this part is a breeze compared to the rest of the projects. For the project each local scope would calculate an address for the variables within it to use given their type and size. As per the given parameters an integer takes up four bytes and a real takes up eight bytes with their respective arrays taking up a multiple of four or eight depending on their length.

**Part 2)** **Methodology:**

In determining how to make type information flow I encoded all possible types as different integers and made all of the structures in project 2 return integers rather than void. I then worked on paper out how a majority of the information will be transported through the code. To do this I went back to the productions that I derived to make the ll(1) grammar and expanded out each production and its tail to get several trees of the same production.

for instance, one of factors productions was <factor> -> **id** <factor’>. On my paper I would have expanded this out to <factor> -> **id (**<expr\_lst>**)** | **id** | **id [**<expr>**]** from these expansions I then would create separate type tables. For example the table below shows how the information must flow

Table for the production <factor> -> **Id** **[**<expr>**]**

|  |  |  |
| --- | --- | --- |
| Id.t | <expr>.t | Factor.t |
| Int Array | Int | Int |
| Real Array | Int | Real |
| Not array | Int | ERR\*(1) |
| Array | Not Int | ERR\*(2) |

The ERR\* signifies a semantic error, in the case of the compiler, it would inform the programmer in case 1 that a non-array identifier cannot be indexed, and in case 2 it would inform the programmer that an array cannot be indexed with a non-integer type. There are other cases than just these two for this particular example, but the idea is quite simple for a majority of the other tables. There are some cases unique to the language specification. Such as the information passing for the statement procdueres and for function calls.

<stmt> -> <var> **assignop** <expr> | <cmpd\_stmt> | **if** <expr> **then** <stmt> <stmt’>

While an argument could be made for the type of the first procedure, it is unreasonable to assume any type to the totality of the next two procedures that could encompass all the possibilities present. Unless we use the void type to assign a correct statement and the error type to assign mal-formed statements. This solves the problem because now there is a known type to check for when evaluating statement procedures.

Ex. for <stmt> -> <var> **assignop** <expr>

|  |  |  |
| --- | --- | --- |
| <var>.t | <expr>.t | <stmt>.t |
| Int | Int | void |
| real | real | void |
| Array Int | Array Int | void |
| Array Real | Array Real | void |

Function calls were another tricky business, this is because a function can have any number of parameters of any type. This obfuscates the path to being able to properly check and see if a function is called correctly with parameters of the correct type. A solution that I found to solve the issue was to create a linked node system of variables (this will be discussed more in part 3). This system allows me to differentiate between functions, function parameters and normal variables. Which will allow for construction of a relatively easy searching algorithm to return the necessary information to the type checking system.

**Part 3)** **Implementation:**

The biggest part of this project was by far implementing the node structure to correctly setup variables with their type information. The first big challenge in doing so was to correctly create a structure that would allow for all the necessary type information to be held. A single node holds the information of its name, type, node type (function, function parameter, or variable), referred to as color, and pointers to other nodes in front (next), behind (prev), above (parent), or below (child) them.



fun1 next fun2 next var2



child



next



fp1 var1

This example of the node structure shows three colors of nodes, green for functions, yellow for function parameters and blue for regular variables

this type of node structure could be created from a segment of the pascal language as follows.

…

function fun1:integer;

begin

end;

function fun2 (fp1:integer):real;

var var1 : array [1..25] of integer;

begin

end;

var var2 : array [1..2] of real;

…

In the diagram above the nodes that have a next pointer, such as that of fun2 to var2 would also have a pointer going from var2 to fun2 using the structures prev (previous) pointer, the same is also true for nodes with child and parent relationships.

To properly implement the pascal scoping structure, I also needed to create functions that would allow for the traversal of going up and down scope easy. Going up scope luckily was simple. To properly go up scope I needed to find where the production for a functions end was located. This production happened to be in the sub declaration call as it deals with function definition.

<sub\_dec> -> <sub\_head> <sub\_decT>

All my function for going up scope had to do was look for the nearest parent node and make that the top of the stack.

Knowing when to go down scope, while it might sound like an easy task, was actually very complicated. This is because there are many options when it comes to the first variable declared in the scope of a function, this could be a function parameter, variable, nothing, or an error. To overcome this problem, I created a lock system that the compiler uses. When a variable is created it looks to see if there is an available lock for that scope, if so it will be created as a child node to the front node of the stack, however if the lock has been consumed by a variable already then the variable in creation will just stick itself as the next node to the front of the stack. This lock system was able to solve all of my problems in one solution which was amazing with the exception of a function not having any declared variables within its scope. Though to counter this there is a manual lock override done at the same place the scoping up of the function takes place. And with that covers an incredibly simplified explanation of the scoping system.

Type checking from this point on was quite straight forward, I just kept creating my tables to see which productions needed to return what type given the scenarios. For calling functions however it was a bit trickier because I needed to figure out how to account for the parameter issue. Using the scoping system though I used the productions <expr\_lst> and <expr\_lstT> to compare if the amount of parameters present were equal to the amount declared as well as checked their types for continuity. If there was no errors in the parameter calls, then the production would return back to <factorT> -> **(**<expr\_lst>**)** with type void, and because I had access to the functions return type stored, if type void was present then I could safely return the functions type.

After all the type checking was implemented, addressing was incredibly easy to get going. Because project parameters state that in each new scope the addresses are recalculated, starting at zero. It was really easy to create a global variable to hold the current address and accumulate the correct values based on type information. The error handling for this part required me to store a previously calculated version of the address, just in case an error occurred during variable creation the address can still be reverted back to a correct value.

**Part 4)** **Discussion and Conclusions:**

The type checking and scoping for this project was by far the most challenging part of creating the compiler. This required by far the most mental energy from me, but was also the most rewarding when it came to problem solving, and is by far the best and biggest programming project I have worked on. It took me quite a while to learn how to translate this idea of information flow to actual code that worked with my compiler. However once I started grasping the fundamental concepts at play I quickly was able to translate the type information from abstract on paper to actual working code. The error handling is by far the most complicated part of any software and should truly be regarded as an art. The fact that my compiler is not allowed to just crash when a dumb programmer puts in error filled code and still produce meaningful output is a true testament to the resilience that this software is required to have.

**Part 5) References:**

Aho, A., Sethi R., Ullman J. (1986) Compilers Principles, Techniques, and Tools. Reading, MA: Addison-Wesly

**Appendix I: Sample Inputs and Output:**

**No syntax errors**

program test (input, output);

var a : integer;

var b : real;

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

var f : array [25..45] of real;

var g : array [1..2] of integer;

function fun1 : integer;

function fun5 : integer;

begin

end;

begin

end;

function fun2(x: integer; y: real) : real;

var e: real;

var ab : array [1..2500] of real;

var c: integer;

function fun3 : integer;

var g: integer;

begin

end;

function fun6 : integer;

begin

end;

function fun7 : integer;

var g: integer;

begin

end;

function fun4(hello: array [1..2] of integer) : integer;

begin

while(a=hello[2]) do

begin

e:=e

end

end;

begin

b := e + 4.44;

b:= (a mod x) / x;

while ((a >= 1) and ((b <= e) or (not (a = c)))) do

begin

b:= 2.5E2

end;

fun2 := 2.5

end;

begin

b:= fun2(2,f[5]);

if (a>2) then a:= 2 else a := a+2;

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 var f : array [25..45] of real;

6 var g : array [1..2] of integer;

7 function fun1 : integer;

8 function fun5 : integer;

9 begin

10 end;

11 begin

12 end;

13 function fun2(x: integer; y: real) : real;

14 var e: real;

15 var ab : array [1..2500] of real;

16 var c: integer;

17 function fun3 : integer;

18 var g: integer;

19 begin

20 end;

21 function fun6 : integer;

22 begin

23 end;

24 function fun7 : integer;

25 var g: integer;

26 begin

27 end;

28 function fun4(hello: array [1..2] of integer) : integer;

29 begin

30 while(a=hello[2]) do

31 begin

32 e:=e

33 end

34 end;

35 begin

36 b := e + 4.44;

37 b:= (a mod x) / x;

38 while ((a >= 1) and ((b <= e) or (not (a = c)))) do

39 begin

40 b:= 2.5E2

41 end;

42 fun2 := 2.5

43 end;

44

45 begin

46 b:= fun2(2,f[5]);

47 if (a>2) then a:= 2 else a := a+2;

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

49 end.

**Address file**

node name address scope function

a 0 test

b 4 test

c 12 test

f 20 test

g 188 test

e 0 fun2

ab 8 fun2

c 20008 fun2

g 0 fun3

g 0 fun7

**With just semantic errors**

program test (input, output);

var a : integer;

var b : real;

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

var f : array [25..45] of real;

var g : array [1..2] of integer;

function fun1 : integer;

function fun5 : integer;

begin

end;

begin

end;

function fun2(x: integer; y: real) : real;

var e: real;

var ab : array [1..2500] of real;

var c: integer;

function fun3 : integer;

var l: integer;

begin

end;

function fun6 : integer;

begin

l:= 75

end;

function fun7 : integer;

var g: integer;

begin

end;

function fun4(hello: array [1..2] of integer) : integer;

begin

while(a=hello[2.5]) do

begin

e:=c

end

end;

begin

b := e + 4.44;

b:= (a mod x) / x;

while ((a >= 1) and ((b <= e) or (not (a = c)))) do

begin

c:= 2.5E2

end;

fun2 := 2.5

end;

begin

b:= fun2(2,f[5]);

if (a>2) then a:= 2 else a := a+2;

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 var f : array [25..45] of real;

6 var g : array [1..2] of integer;

7 function fun1 : integer;

8 function fun5 : integer;

9 begin

10 end;

11 begin

12 end;

13 function fun2(x: integer; y: real) : real;

14 var e: real;

15 var ab : array [1..2500] of real;

16 var c: integer;

17 function fun3 : integer;

18 var l: integer;

19 begin

20 end;

21 function fun6 : integer;

22 begin

23 l:= 75

SEMERR: Identifier l is either not defined or not within scope

24 end;

25 function fun7 : integer;

26 var g: integer;

27 begin

28 end;

29 function fun4(hello: array [1..2] of integer) : integer;

30 begin

31 while(a=hello[2.5]) do

SEMERR: Identifiers cannot be indexed with a non-integer

SEMERR: invalid operands for operator =, operands must be of type boolean or operands must be of type integer/real

32 begin

33 e:=c

34 end

SEMERR: Type mismatch on assignment expecting type real received type integer

35 end;

36 begin

37 b := e + 4.44;

38 b:= (a mod x) / x;

39 while ((a >= 1) and ((b <= e) or (not (a = c)))) do

40 begin

41 c:= 2.5E2

42 end;

SEMERR: Type mismatch on assignment expecting type integer received type real

43 fun2 := 2.5

44 end;

45

46 begin

47 b:= fun2(2,f[5]);

48 if (a>2) then a:= 2 else a := a+2;

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

50 end.

**Address file**

node name address scope function

a 0 test

b 4 test

c 12 test

f 20 test

g 188 test

e 0 fun2

ab 8 fun2

c 20008 fun2

l 0 fun3

g 0 fun7

**with semantic, syntax errors and lexical errors**

program test (input, output);

var a : integer;

var b : real;

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

var f : array [25..45] of real;

var ggggggggggggg : array [1..2] of integer;

function fun2(x: integer; y: real) : real;

var e: real;

var ab : array [1..2500] of real;

var c: integer;

begin

b := e + 4.44;

b:= (a mod x) / x;

while ((a >= 1) and ((b <= e) or (not (a = c)))) do

begin

c:= 2.5E2

end;

fun2 := 2.55555555555555555555

end;

begin

b:= fun2(2.5,f[5]);

if (a>2) then a:= 2 else a := a+2;

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 var f : array [25..45] of real;

6 var ggggggggggggg : array [1..2] of integer;

LEXERR: Identifier is greater than 10 characters ggggggggggggg

SYNERR: token mismatch, expecting token Identifier instead received token ggggggggggggg

7 function fun2(x: integer; y: real) : real;

8 var e: real;

9 var ab : array [1..2500] of real;

10 var c: integer;

11 begin

12 b := e + 4.44;

13 b:= (a mod x) / x;

14 while ((a >= 1) and ((b <= e) or (not (a = c)))) do

15 begin

16 c:= 2.5E2

17 end;

SEMERR: Type mismatch on assignment expecting type integer received type real

18 fun2 := 2.55555555555555555555

LEXERR: Real number back is greater than 5 digits: 2.555555555...

SYNERR: tok mismatch expecting Identifier, Number, not, +, -, (, instead received 2.555555555...

19 end;

SEMERR: Type mismatch on assignment expecting type real received type type error

20

21 begin

22 b:= fun2(2.5,f[5]);

SEMERR: Function parameter type mismatch expecting integer, received real

SEMERR: Type mismatch on assignment expecting type real received type type error

23 if (a>2) then a:= 2 else a := a+2;

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

25 end.

Address file

node name address scope function

a 0 test

b 4 test

c 12 test

f 20 test

e 196 fun2

ab 204 fun2

c 20204 fun2

**Appendix II: Program Listing:**

**Parser.c**

#ifndef PARSER\_H

#define PARSER\_H

#include <stdio.h>

#include "../LexicalAnalyzer/lexAnalyze.c"

#include "../lexeme.c"

#include "../node/nodetest.c"

#include "../tokcode2str.c"

struct Lexeme tok;

struct Lexeme temp; //temp value that needs to be outside tok\_match as to not be destroyed by function return

struct Lexeme tok\_ret;

struct Lexeme bad\_ret;

struct node\_stack parsestack;

stack\_ptr st = &parsestack;

FILE \*token;

FILE \*list;

FILE \*file;

FILE \*trace;

FILE \*nodes;

FILE \*ad;

int scope\_flag;

int scope\_level=0;

//function call variables

int param\_count;//parameters present in call

int f\_params;//amount of parameters specified by actual function

char f\_name[15];

//addressing node variables

int addr;

void decs();

int std\_type();

void sub\_decs();

int type();

int cmpd\_stmt();

int stmt();

int expr\_lst();

int expr();

struct Lexeme tok\_match(int t,int val){

//int t = lex.tkn;

//int type;

bad\_ret.type = TYPEERR;

memcpy(&temp,&tok,sizeof tok);

//tok\_lex = &temp;//need to memcpy this instead of just straight addressing, or just use a temp lexeme that copies the current token

printf("tokcmp %s %s\n",tokcode2str(tok.tkn), tokcode2str(t));

if(t == tok.tkn){

if(t != EOF && val == 0 && tok.tkn != EOF){

if(tok.tkn == ID ){

printf("lexeme: %s, type: %d\n",tok.word,get\_id\_type(st,tok.word));

tok = get\_next\_token(file,list,token);

return temp; //all good here

}

else if(tok.tkn == NUM){

printf("lexeme: %s, type: %d\n",tok.word,tok.type);

tok = get\_next\_token(file,list,token);

return temp;

}

else{

tok = get\_next\_token(file,list,token);

return temp; //all good here

}

}

if(t == EOF){

//exit(0);

}

if(val != 0 && tok.attr.val == val){

tok = get\_next\_token(file,list,token);

return temp;

}

if(val != 0 && tok.attr.val != val){

printf("Sync Error!! Expecting %s instead received token %s \n",tokcode2str(t),tok.word);

fprintf(list,"SYNERR: token mismatch, expecting token %s instead received token %s\n",tokcode2str(t),tok.word);

tok = get\_next\_token(file,list,token);

return bad\_ret;

}

else{

//exit(0);

//end of parse

//stop parse here some how

}

}//end t == tok

else{

printf("Sync Error!! Expecting %s instead received token %s \n",tokcode2str(t),tok.word);

fprintf(list,"SYNERR: token mismatch, expecting token %s instead received token %s\n",tokcode2str(t),tok.word);

tok = get\_next\_token(file,list,token);

return bad\_ret;

}

}

void sign(){

fprintf(trace, "%s %s\n","sign", tok.word);

switch (tok.tkn) {

case ADDOP:

if(tok.attr.val == ADDOP\_PL){

tok\_match(ADDOP,ADDOP\_PL);

break;

}

if(tok.attr.val == ADDOP\_MN){

tok\_match(ADDOP,ADDOP\_MN);

break;

}

default:

fprintf(list,"SYNERR: tok mismatch expecting %s, %s, instead received %s\n","+","-",tok.word);

while (tok.tkn!=ID && tok.tkn!=NUM && tok.tkn!=NOT && strcmp(tok.word,"(") != 0) {

tok = get\_next\_token(file,list,token);

fprintf(trace, "sign: trying to sync on %s\n", tok.word);

}

fprintf(trace, "sign: sync on token %s\n",tok.word );

}

}

int factorT(int i\_val){

int r\_val;

int f\_val;

char brack[15];

fprintf(trace, "%s %s\n","factorT",tok.word );

switch (tok.tkn) {

//start epsilons

case MULOP:

r\_val=i\_val;

if(i\_val == TYPEERR){

printf("leaving factorT with type error\n");

fprintf(trace, "%s %s\n","factorT","IMM TYPEERR RET");

r\_val = TYPEERR;

return r\_val;

}

return r\_val;

case ADDOP:

r\_val=i\_val;

if(i\_val == TYPEERR){

printf("leaving factorT with type error\n");

fprintf(trace, "%s %s\n","factorT","IMM TYPEERR RET");

r\_val = TYPEERR;

return r\_val;

}

return r\_val;

case RELOP:

r\_val=i\_val;

if(i\_val == TYPEERR){

printf("leaving factorT with type error\n");

fprintf(trace, "%s %s\n","factorT","IMM TYPEERR RET");

r\_val = TYPEERR;

return r\_val;

}

return r\_val;

case SEMICOLON:

r\_val=i\_val;

if(i\_val == TYPEERR){

printf("leaving factorT with type error\n");

fprintf(trace, "%s %s\n","factorT","IMM TYPEERR RET");

r\_val = TYPEERR;

return r\_val;

}

return r\_val;

case END:

r\_val=i\_val;

if(i\_val == TYPEERR){

printf("leaving factorT with type error\n");

fprintf(trace, "%s %s\n","factorT","IMM TYPEERR RET");

r\_val = TYPEERR;

return r\_val;

}

return r\_val;

case ELSE:

r\_val=i\_val;

if(i\_val == TYPEERR){

printf("leaving factorT with type error\n");

fprintf(trace, "%s %s\n","factorT","IMM TYPEERR RET");

r\_val = TYPEERR;

return r\_val;

}

return r\_val;

case THEN:

r\_val=i\_val;

if(i\_val == TYPEERR){

printf("leaving factorT with type error\n");

fprintf(trace, "%s %s\n","factorT","IMM TYPEERR RET");

r\_val = TYPEERR;

return r\_val;

}

return r\_val;

case DO:

r\_val=i\_val;

if(i\_val == TYPEERR){

printf("leaving factorT with type error\n");

fprintf(trace, "%s %s\n","factorT","IMM TYPEERR RET");

r\_val = TYPEERR;

return r\_val;

}

return r\_val;

case COMMA:

r\_val=i\_val;

if(i\_val == TYPEERR){

printf("leaving factorT with type error\n");

fprintf(trace, "%s %s\n","factorT","IMM TYPEERR RET");

r\_val = TYPEERR;

return r\_val;

}

return r\_val;

//end epsilons

case PAREN: //function call !!!

if(tok.tkn == PAREN && tok.attr.val == PAREN\_OPEN){

tok\_match(PAREN,PAREN\_OPEN);

r\_val = expr\_lst(); //type void, but the individual types for parameters must be checked for the function call

//only a function call is going to use this expr\_lst

tok\_match(PAREN,PAREN\_CLOSE);

return r\_val;

}

case BRACK: //array assign or look up, but all this needs to do is give back type info

if(tok.tkn == BRACK && tok.attr.val == BRACK\_OPEN){

tok\_match(BRACK,BRACK\_OPEN);

f\_val = expr();//must be an integer type

if(f\_val == TYPEINT){

if(i\_val == TYPEARR\_INT){

r\_val = TYPEINT;

}

else if(i\_val == TYPEARR\_REAL){

r\_val = TYPEREAL;

}

else{

fprintf(list,"SEMERR: Non-array identifiers cannot be indexed\n");

r\_val = TYPEERR;

}

}

else if(f\_val != TYPEINT) {

fprintf(list, "SEMERR: Identifiers cannot be indexed with a non-integer\n" );

r\_val = TYPEERR;

}

tok\_ret = tok\_match(BRACK,BRACK\_CLOSE);

strcpy(brack,tok\_ret.word);

if(i\_val == TYPEERR){

printf("leaving factorT with type error\n");

fprintf(trace, "%s %s\n","factorT","IMM TYPEERR RET");

r\_val = TYPEERR;

return r\_val;

}

printf("array type return: %d\n",r\_val);

return r\_val;

}

if(tok.attr.val == PAREN\_CLOSE || tok.attr.val == BRACK\_CLOSE){

//these two cases are also part of epsilons

r\_val=i\_val;

if(i\_val == TYPEERR){

printf("leaving factorT with type error\n");

fprintf(trace, "%s %s\n","factorT","IMM TYPEERR RET");

r\_val = TYPEERR;

return r\_val;

}

return r\_val;

}

default:

fprintf(list,"SYNERR: tok mismatch expecting %s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s, instead received %s\n","\* / div mod and","+ - or","< > <= >= <> ==",";","end","else","then","do",",","]",")",tok.word);

while (tok.tkn != MULOP && tok.tkn != ADDOP && tok.tkn != RELOP && tok.tkn != SEMICOLON && tok.tkn != END && tok.tkn != ELSE && tok.tkn != THEN && tok.tkn != DO && tok.tkn != COMMA && strcmp(tok.word,")") != 0 && strcmp(tok.word,"]") != 0) {

tok = get\_next\_token(file,list,token);

fprintf(trace, "factorT: trying to sync on %s\n", tok.word);

}

fprintf(trace, "factorT: sync on token %s\n", tok.word);

return TYPEERR;

}

}

//NEED TO FIX RVALUE tok\_match stuff

int factor(){

fprintf(trace, "%s %s\n","factor",tok.word );

int r\_val = 0;

char name[15];

switch (tok.tkn) {

case ID:

tok\_ret = tok\_match(ID,0);

strcpy(name,tok\_ret.word);

int i\_val = get\_id\_type(st,tok\_ret.word);

printf("factor get\_id\_type for %s: %d\n", tok\_ret.word,i\_val);

if(i\_val == TYPEERR){

fprintf(list, "SEMERR: No id with name %s\n", name);

}

r\_val = factorT(i\_val);

return r\_val;

case NUM:

tok\_ret = tok\_match(NUM,0);

return tok\_ret.type;

case NOT:

tok\_match(NOT,0); r\_val = factor();

if(r\_val == TYPEBOOL){

return TYPEBOOL;

}

else{

fprintf(list,"SEMERR: type mismatch, expecting type: %s received: %s\n",tokcode2str(TYPEBOOL),tokcode2str(r\_val));

return TYPEERR;

}

case PAREN:

if(tok.attr.val == PAREN\_OPEN){

tok\_match(PAREN,PAREN\_OPEN); int r\_val = expr(); tok\_match(PAREN,PAREN\_CLOSE);

return r\_val;

}

default:

fprintf(list,"SYNERR: tok mismatch expecting %s,%s,%s,%s, instead received %s\n","Identifier","Number","not","(",tok.word);

while (tok.tkn != MULOP && tok.tkn != ADDOP && tok.tkn != RELOP && tok.tkn != SEMICOLON && tok.tkn != END && tok.tkn != ELSE && tok.tkn != THEN && tok.tkn != DO && tok.tkn != COMMA && strcmp(tok.word,")") != 0 && strcmp(tok.word,"]") != 0) {

tok = get\_next\_token(file,list,token);

fprintf(trace, "factor: trying to sync on %s\n", tok.word);

}

fprintf(trace, "factor: sync on token %s\n", tok.word);

}

return TYPEERR;

}

int termT(int i\_val){

//the i\_val is the factor type information

int r\_val;

int f\_val;

char mulop[15];

fprintf(trace, "%s %s\n","termT",tok.word );

switch (tok.tkn) {

case MULOP:

tok\_ret = tok\_match(MULOP,0); strcpy(mulop,tok\_ret.word);

f\_val = factor();

printf("termT i\_val: %d f\_val: %d mulop: %s\n",i\_val,f\_val,mulop);

if(strcmp(mulop,"and") == 0){

// "and" needs bool

if(i\_val == TYPEBOOL && f\_val == TYPEBOOL){

r\_val = TYPEBOOL;

}

else if(i\_val == TYPEERR || f\_val == TYPEERR){

r\_val = TYPEERR;

}

else{

fprintf(list, "SEMERR: invalid operands for operator and, require type boolean\n" );

r\_val = TYPEERR;

}

printf("mulop AND encountered\n");

}

else if(strcmp(mulop,"mod") == 0){

// "mod"

if(i\_val == TYPEINT && f\_val == TYPEINT){

r\_val = TYPEINT;

}

else{

fprintf(list, "SEMERR: invalid operands for operator mod, require type integer\n" );

r\_val = TYPEERR;

}

printf("mulop MOD encountered\n");

}

else if(strcmp(mulop,"\*") == 0){

// "\*"

if(i\_val == TYPEINT && f\_val == TYPEINT){

r\_val = TYPEINT;

}

else if(i\_val == TYPEINT && f\_val == TYPEREAL){

r\_val = TYPEREAL;

}

else if(i\_val == TYPEREAL && f\_val == TYPEINT){

r\_val = TYPEREAL;

}

else if(i\_val == TYPEREAL && f\_val == TYPEREAL){

r\_val = TYPEREAL;

}

else{

fprintf(list, "SEMERR: invalid operands for operator \*, require either type real or integer\n");

r\_val = TYPEERR;

}

printf("mulop \* encountered\n");

}

else if(strcmp(mulop,"/") == 0){

// "/"

if(i\_val == TYPEINT && f\_val == TYPEINT){

r\_val = TYPEREAL;

}

else if(i\_val == TYPEINT && f\_val == TYPEREAL){

//r\_val = TYPEREAL;

r\_val = TYPEERR;

fprintf(list, "SEMERR: Mixed mode arithmetic is not supported\n" );

}

else if(i\_val == TYPEREAL && f\_val == TYPEINT){

//r\_val = TYPEREAL;

r\_val = TYPEERR;

fprintf(list, "SEMERR: Mixed mode arithmetic is not supported\n" );

}

else if(i\_val == TYPEREAL && f\_val == TYPEREAL){

r\_val = TYPEREAL;

}

else{

fprintf(list, "SEMERR: invalid operands for operator /, require either type real or integer\n" );

r\_val = TYPEERR;

}

printf("mulop / encountered\n");

}

else if(strcmp(mulop,"div") == 0){

//DIV

if(i\_val == TYPEINT && f\_val == TYPEINT){

r\_val = TYPEINT;

}

else{

fprintf(list, "SEMERR: invalid operands for operator div, require type integer\n" );

r\_val = TYPEERR;

}

printf("mulop DIV encountered\n");

}

r\_val = termT(r\_val);

if(i\_val == TYPEERR){

printf("leaving termT with type error\n");

fprintf(trace, "%s %s\n","termT","IMM TYPEERR RET");

r\_val = TYPEERR;

return r\_val;

}

return r\_val;

case ADDOP: case RELOP: case SEMICOLON: case END: case ELSE: case THEN: case DO: case COMMA:

r\_val = i\_val;

if(i\_val == TYPEERR){

printf("leaving termT with type error\n");

fprintf(trace, "%s %s\n","termT","IMM TYPEERR RET");

r\_val = TYPEERR;

return r\_val;

}

return r\_val;

case BRACK: case PAREN:

if(tok.attr.val == BRACK\_CLOSE || tok.attr.val == PAREN\_CLOSE){

r\_val = i\_val;

if(i\_val == TYPEERR){

printf("leaving termT with type error\n");

fprintf(trace, "%s %s\n","termT","IMM TYPEERR RET");

r\_val = TYPEERR;

return r\_val;

}

return r\_val;

}

default:

fprintf(list,"SYNERR: tok mismatch expecting %s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s, instead received %s\n","\* / div mod and","+ - or","< > <= >= <> ==",";","end","else","then","do",",","]",")",tok.word);

while (tok.tkn != ADDOP && tok.tkn != RELOP && strcmp(tok.word,";") != 0 && tok.tkn != END && tok.tkn != ELSE && tok.tkn != THEN && tok.tkn != DO && tok.tkn != COMMA && strcmp(tok.word,")") != 0 && strcmp(tok.word,"]") != 0) {

tok = get\_next\_token(file,list,token);

fprintf(trace, "termT: trying to sync on %s\n", tok.word);

}

fprintf(trace, "termT: sync on token %s\n", tok.word);

return TYPEERR;

}

}

int term(){

int r\_val;

int f\_val;

fprintf(trace, "%s %s\n","term" ,tok.word);

switch (tok.tkn) {

case ID: case NUM: case NOT:;

f\_val = factor(); r\_val = termT(f\_val);

return r\_val;

case PAREN:

if(tok.attr.val == PAREN\_OPEN){

f\_val = factor(); r\_val = termT(f\_val);

return r\_val;

}

default:

fprintf(list,"SYNERR: tok mismatch expecting %s,%s,%s,%s, instead received %s\n","Identifier","Number","not","(",tok.word);

while (tok.tkn != ADDOP && tok.tkn != RELOP && tok.tkn != SEMICOLON && tok.tkn != END && tok.tkn != ELSE && tok.tkn != THEN && tok.tkn != DO && tok.tkn != COMMA && strcmp(tok.word,")") != 0 && strcmp(tok.word,"]") != 0) {

tok = get\_next\_token(file,list,token);

fprintf(trace, "term: trying to sync on %s\n", tok.word);

}

fprintf(trace, "term: sync on token %s\n", tok.word);

return TYPEERR;

}

}

int smpl\_exprT(int i\_val){

int r\_val;

int f\_val;

char addop[15];

fprintf(trace, "%s %s\n","smpl\_exprT",tok.word );

switch (tok.tkn) {

case ADDOP:

tok\_ret = tok\_match(ADDOP,0); strcpy(addop,tok\_ret.word);

f\_val = term();

printf("ADDOP: %s i\_val: %d f\_val: %d\n",addop,i\_val,f\_val);

if(strcmp(addop,"or")==0){

if(i\_val == TYPEBOOL && f\_val == TYPEBOOL){

r\_val = TYPEBOOL;

}

else if(i\_val == TYPEERR || f\_val == TYPEERR){

r\_val = TYPEERR;

}

else{

fprintf(list, "SEMERR: invalid simple expression involving the or of non boolean element\n" );

r\_val = TYPEERR;

}

}

if(strcmp(addop,"+") == 0 || strcmp(addop,"-") == 0){

if(i\_val == TYPEINT && f\_val == TYPEINT){

r\_val = TYPEINT;

}

else if(i\_val == TYPEREAL && f\_val == TYPEREAL){

r\_val = TYPEREAL;

}

else if(i\_val == TYPEREAL && f\_val == TYPEINT){

//r\_val = TYPEREAL;

r\_val = TYPEERR;

fprintf(list, "SEMERR: Mixed mode arithmetic is not supported\n" );

}

else if(i\_val == TYPEINT && f\_val == TYPEREAL){

//r\_val = TYPEREAL;

r\_val = TYPEERR;

fprintf(list, "SEMERR: Mixed mode arithmetic is not supported\n" );

}

else if(i\_val == TYPEERR || f\_val == TYPEERR){

r\_val = TYPEERR;

}

else{

fprintf(list, "SEMERR: invalid simple expression involving a boolean with an non-boolean operator (+,-)\n" );

r\_val = TYPEERR;

}

}

r\_val = smpl\_exprT(r\_val);

if(i\_val == TYPEERR){

printf("leaving smpl\_exprT with type error\n");

fprintf(trace, "%s %s\n","smpl\_exprT","IMM TYPEERR RET");

r\_val = TYPEERR;

return r\_val;

}

return r\_val;

case RELOP: case SEMICOLON: case END: case ELSE: case THEN: case DO: case COMMA:

r\_val = i\_val;

if(i\_val == TYPEERR){

printf("leaving smpl\_exprT with type error\n");

fprintf(trace, "%s %s\n","smpl\_exprT","IMM TYPEERR RET");

r\_val = TYPEERR;

return r\_val;

}

return r\_val;

case BRACK: case PAREN:

if(tok.attr.val == BRACK\_CLOSE || tok.attr.val == PAREN\_CLOSE){

r\_val = i\_val;

if(i\_val == TYPEERR){

printf("leaving smpl\_exprT with type error\n");

fprintf(trace, "%s %s\n","smpl\_exprT","IMM TYPEERR RET");

r\_val = TYPEERR;

return r\_val;

}

return r\_val;

}

default:

fprintf(list,"SYNERR: tok mismatch expecting %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, instead received %s\n","+ - or","< > <= >= <> =",";","end","else","then","do",";","]",")",tok.word);

while (strcmp(tok.word,")") != 0) {

tok = get\_next\_token(file,list,token);

fprintf(trace, "smpl\_exprT: trying to sync on %s\n", tok.word);

}

fprintf(trace, "smpl\_exprT: sync on token %s\n", tok.word);

return TYPEERR;

}

}

int smpl\_expr(){

fprintf(trace, "%s %s\n","smpl\_expr",tok.word );

int f\_val;

int r\_val;

switch (tok.tkn) {

case ID: case NUM: case NOT:

f\_val = term(); r\_val = smpl\_exprT(f\_val);

return r\_val;

case ADDOP://no booleans allowed

if(tok.attr.val == ADDOP\_PL || tok.attr.val == ADDOP\_MN){

sign(); f\_val = term(); r\_val = smpl\_exprT(f\_val);

if(f\_val == TYPEBOOL || r\_val == TYPEBOOL){

fprintf(list, "SEMERR: Boolean with +/- has no valid meaning\n");

r\_val = TYPEERR;

}

printf("smpl\_expr +- r\_val: %d\n",r\_val );

return r\_val;

}

case PAREN:

if(tok.attr.val == PAREN\_OPEN){

f\_val = term(); r\_val = smpl\_exprT(f\_val);

printf("smpl\_expr ( r\_val: %d\n",r\_val );

return r\_val;

}

default:

fprintf(list,"SYNERR: tok mismatch expecting %s,%s,%s,%s,%s, instead received %s\n","Identifier","Number","not","+","-",tok.word);

while (tok.tkn != RELOP && tok.tkn != SEMICOLON && tok.tkn != END && tok.tkn != ELSE && tok.tkn != THEN && tok.tkn != DO && tok.tkn != COMMA && strcmp(tok.word,"]") != 0 && strcmp(tok.word,")") != 0 ) {

tok = get\_next\_token(file,list,token);

fprintf(trace, "smpl\_expr: trying to sync on %s\n", tok.word);

}

fprintf(trace, "smpl\_expr: sync on token %s\n", tok.word);

return TYPEERR;

}

}

int exprT(int i\_val){

int f\_val;

int r\_val;

char relop[15];

fprintf(trace, "%s %s\n","exprT" ,tok.word);

switch (tok.tkn) {

case RELOP:

tok\_ret = tok\_match(RELOP,0);

strcpy(relop,tok\_ret.word);

f\_val = smpl\_expr();

printf("relop: %s i\_val: %d f\_val: %d\n",relop,i\_val,f\_val);

if(strcmp(relop,"=")==0){

//printf("going into = relop with types i\_val: %d f\_val: %s\n", i\_val,f\_val);

if(i\_val == TYPEBOOL && f\_val == TYPEBOOL){

r\_val = TYPEBOOL;

}

else if ((i\_val == TYPEINT && f\_val == TYPEINT) || (i\_val == TYPEREAL || f\_val == TYPEREAL)) {

r\_val = TYPEBOOL;

}

else {

fprintf(list, "SEMERR: invalid operands for operator =, operands must be of type boolean or operands must be of type integer/real\n");

r\_val = TYPEERR;

}

}

else {

if(i\_val == TYPEINT && f\_val == TYPEINT){

r\_val = TYPEBOOL;

}

else if(i\_val == TYPEREAL && f\_val == TYPEINT){

//r\_val = TYPEBOOL;

r\_val = TYPEERR;

fprintf(list, "SEMERR: Mixed mode arithmetic is not supported\n" );

}

else if(i\_val == TYPEINT && f\_val == TYPEREAL){

//r\_val = TYPEBOOL;

r\_val = TYPEERR;

fprintf(list, "SEMERR: Mixed mode arithmetic is not supported\n" );

}

else if(i\_val == TYPEREAL && f\_val == TYPEREAL){

r\_val = TYPEBOOL;

}

else if(i\_val == TYPEERR || f\_val == TYPEERR){

//THIS MAKES SURE ERRORS ARE NOT SPAMMED

r\_val = TYPEERR;

}

else {

fprintf(list,"SEMERR: invalid operands for inequality operator %s, operands must be of the type integer or real\n",relop);

r\_val = TYPEERR;

}

}

if(i\_val == TYPEERR){

printf("leaving exprT with type error\n");

fprintf(trace, "%s %s\n","exprT","IMM TYPEERR RET");

r\_val = TYPEERR;

return r\_val;

}

return r\_val;

case SEMICOLON: case END: case ELSE: case THEN: case DO: case COMMA:

r\_val = i\_val;

if(i\_val == TYPEERR){

printf("leaving exprT with type error\n");

fprintf(trace, "%s %s\n","exprT","IMM TYPEERR RET");

r\_val = TYPEERR;

return r\_val;

}

return r\_val;

case BRACK:

if(tok.attr.val == BRACK\_CLOSE){

r\_val = i\_val;

if(i\_val == TYPEERR){

printf("leaving exprT with type error\n");

fprintf(trace, "%s %s\n","exprT","IMM TYPEERR RET");

r\_val = TYPEERR;

return r\_val;

}

return r\_val;

}

case PAREN:

if(tok.attr.val == PAREN\_CLOSE){

r\_val = i\_val;

if(i\_val == TYPEERR){

printf("leaving exprT with type error\n");

fprintf(trace, "%s %s\n","exprT","IMM TYPEERR RET");

r\_val = TYPEERR;

return r\_val;

}

return r\_val;

}

default:

fprintf(list,"SYNERR: SYNERR: tok mismatch expecting %s, instead received %s\n","< > <= >= <> = ; , ) ] end else then do",tok.word);

while (tok.tkn != COMMA && tok.tkn != SEMICOLON && tok.tkn != END && tok.tkn != ELSE && tok.tkn != THEN && tok.tkn != DO && strcmp(tok.word, ")") != 0 && strcmp(tok.word, "]") != 0) {

tok = get\_next\_token(file,list,token);

fprintf(trace, "exprT: trying to sync on %s\n", tok.word);

}

fprintf(trace, "exprT: sync on token %s\n", tok.word);

return TYPEERR;

}

}

int expr(){

int r\_val;

int f\_val;

fprintf(trace, "%s %s\n","expr",tok.word );

printf("expr: %d,%s\n",tok.tkn,tok.word);

printf("%s %s\n","expr",tok.word);

switch (tok.tkn) {

case NUM: case ID: case NOT:

f\_val = smpl\_expr(); r\_val = exprT(f\_val);

return r\_val;

case ADDOP:

if(strcmp(tok.word,"+")==0){

f\_val = smpl\_expr(); r\_val = exprT(f\_val);

return r\_val;

}

if(strcmp(tok.word,"-")==0){

f\_val = smpl\_expr(); r\_val = exprT(f\_val);

return r\_val;

}

case PAREN:

if(strcmp(tok.word,"(")==0){

f\_val = smpl\_expr(); r\_val = exprT(f\_val);

return r\_val;

}

default:

fprintf(list,"SYNERR: tok mismatch expecting %s, %s, %s, %s, %s, %s, instead received %s\n","Identifier","Number","not","+","-","(",tok.word);

while (tok.tkn != COMMA && tok.tkn != SEMICOLON && tok.tkn != END && tok.tkn != ELSE && tok.tkn != THEN && tok.tkn != DO && strcmp(tok.word, ")") != 0 && strcmp(tok.word, "]") != 0) {

tok = get\_next\_token(file,list,token);

fprintf(trace, "expr: trying to sync on %s\n", tok.word);

}

fprintf(trace, "expr: sync on token %s\n", tok.word);

return TYPEERR;

}

}

int expr\_lstT(int i\_val){

int f\_val;

int r\_val;

int exp\_type;

fprintf(trace, "%s %s\n","expr\_lstT",tok.word );

switch (tok.tkn) {

case COMMA:

param\_count++;

exp\_type = get\_func\_param\_type(st,f\_name,param\_count);

tok\_match(COMMA,0); f\_val = expr();

if(exp\_type != f\_val){

fprintf(list, "SEMERR: Function parameter type mismatch expecting %s, received %s\n", tokcode2str(exp\_type),tokcode2str(f\_val));

f\_val = TYPEERR;

}

r\_val = expr\_lstT(f\_val);

if(i\_val == TYPEERR){

printf("leaving expr\_lstT with type error\n");

fprintf(trace, "%s %s\n","expr\_lstT","IMM TYPEERR RET");

r\_val = TYPEERR;

return r\_val;

}

return r\_val;

case PAREN://end of function call, check for param\_count match

if(tok.attr.val == PAREN\_CLOSE){

if(param\_count > f\_params){

fprintf(list, "SEMERR: function call has too many arguements %d given, %d required\n", param\_count,f\_params);

}

if(param\_count < f\_params){

fprintf(list, "SEMERR: Function call has too few arguements %d given, %d required\n", param\_count,f\_params);

}

if(i\_val == TYPEERR){

printf("leaving expr\_lstT with type error\n");

fprintf(trace, "%s %s\n","expr\_lstT","IMM TYPEERR RET");

r\_val = TYPEERR;

return r\_val;

}

r\_val = i\_val;

return r\_val;

}

default:

fprintf(list,"SYNERR: tok mismatch expecting %s, %s, instead received %s\n",",",")",tok.word);

while (strcmp(tok.word,")") != 0) {

tok = get\_next\_token(file,list,token);

fprintf(trace, "expr\_lstT: trying to sync on %s\n", tok.word);

}

fprintf(trace, "expr\_lstT: sync on token %s\n", tok.word);

return TYPEERR;

}

}

int expr\_lst(){

int f\_val;

int r\_val;

int exp\_type;

int f\_found=1;

strcpy(f\_name,tok\_ret.word);

int f\_type = get\_id\_type(st,f\_name);

param\_count=0;

f\_params = get\_func\_param\_count(st,f\_name);

fprintf(trace, "%s %s\n","expr\_lst" ,tok.word);

if(in\_scope(st,f\_name)==-1){

fprintf(list,"SEMERR: function %s either not declared or not within scope\n",f\_name);

f\_found =0;

}

switch (tok.tkn) {

case NUM: case ID: case NOT:

param\_count++;

exp\_type = get\_func\_param\_type(st,f\_name,param\_count);

f\_val = expr();

if(exp\_type != f\_val){

if(f\_found)

fprintf(list, "SEMERR: Function parameter type mismatch expecting %s, received %s\n", tokcode2str(exp\_type),tokcode2str(f\_val));

f\_val = TYPEERR;

}

r\_val = expr\_lstT(f\_val);

if(r\_val != TYPEERR){

r\_val = f\_type;

}

return r\_val;

case ADDOP:

if(tok.attr.val == ADDOP\_PL){

param\_count++;

exp\_type = get\_func\_param\_type(st,f\_name,param\_count);

f\_val = expr();

if(exp\_type != f\_val){

if(f\_found)

fprintf(list, "SEMERR: Function parameter type mismatch expecting %s, received %s\n", tokcode2str(exp\_type),tokcode2str(f\_val));

f\_val = TYPEERR;

}

r\_val = expr\_lstT(f\_val);

if(r\_val != TYPEERR){

r\_val = f\_type;

}

return r\_val;

}

if(tok.attr.val == ADDOP\_MN){

param\_count++;

exp\_type = get\_func\_param\_type(st,f\_name,param\_count);

f\_val = expr();

if(exp\_type != f\_val){

if(f\_found)

fprintf(list, "SEMERR: Function parameter type mismatch expecting %s, received %s\n", tokcode2str(exp\_type),tokcode2str(f\_val));

f\_val = TYPEERR;

}

r\_val = expr\_lstT(f\_val);

if(r\_val != TYPEERR){

r\_val = f\_type;

}

return r\_val;

}

case PAREN:

if(tok.attr.val == PAREN\_OPEN){

param\_count++;

exp\_type = get\_func\_param\_type(st,f\_name,param\_count);

f\_val = expr();

if(exp\_type != f\_val){

if(f\_found)

fprintf(list, "SEMERR: Function parameter type mismatch expecting %s, received %s\n", tokcode2str(exp\_type),tokcode2str(f\_val));

f\_val = TYPEERR;

}

r\_val = expr\_lstT(f\_val);

if(r\_val != TYPEERR){

r\_val = f\_type;

}

return r\_val;

}

default:

fprintf(list,"SYNERR: tok mismatch expecting %s, %s, %s, %s, %s, %s, instead received %s\n","Identifier","Number","not","+","-","(",tok.word);

while (strcmp(tok.word,")") != 0) {

tok = get\_next\_token(file,list,token);

fprintf(trace, "expr\_lst: trying to sync on %s\n", tok.word);

}

fprintf(trace, "expr\_lst: sync on token %s\n", tok.word);

return TYPEERR;

}

}

int varT(int i\_val){

int r\_val;

int f\_val;

fprintf(trace, "%s %s\n","varT",tok.word );

switch (tok.tkn) {

case ASSIGNOP:

r\_val = i\_val;

if(i\_val == TYPEERR){

printf("leaving varT with type error\n");

fprintf(trace, "%s %s\n","varT","IMM TYPEERR RET");

r\_val = TYPEERR;

return r\_val;

}

return r\_val;

case BRACK:

if(tok.attr.val == BRACK\_OPEN){

f\_val = expr();//must be an integer type

if(f\_val == TYPEINT){

if(i\_val == TYPEARR\_INT){

r\_val = TYPEINT;

}

else if(i\_val == TYPEARR\_REAL){

r\_val = TYPEREAL;

}

else{

r\_val = TYPEERR;

fprintf(list,"SEMERR: Non-array identifiers cannot be indexed\n");

}

}

else if(f\_val != TYPEINT) {

fprintf(list, "SEMERR: Array cannot be indexed with non-integer\n" );

r\_val = TYPEERR;

}

tok\_match(BRACK,BRACK\_CLOSE);

if(i\_val == TYPEERR){

printf("leaving varT with type error\n");

fprintf(trace, "%s %s\n","varT","IMM TYPEERR RET");

r\_val = TYPEERR;

return r\_val;

}

return r\_val;

}

default:

fprintf(list,"SYNERR: tok mismatch expecting %s, %s, instead received %s\n",":=","[",tok.word);

while (tok.tkn != ASSIGNOP) {

tok = get\_next\_token(file,list,token);

fprintf(trace, "varT: trying to sync on %s\n", tok.word);

}

fprintf(trace, "varT: sync on token %s\n", tok.word);

return TYPEERR;

}

}

int var(){

int r\_val;

char name[15];

fprintf(trace, "%s %s\n","var",tok.word );

switch (tok.tkn) {

case ID:

tok\_ret = tok\_match(ID,0);

strcpy(name,tok\_ret.word);

int i\_val = get\_id\_type(st,name);

printf("stmt before expr1\n");

if(i\_val == TYPEERR){

fprintf(list, "SEMERR: Identifier %s is either not defined or not within scope\n", name);

}

r\_val = varT(i\_val);

return r\_val;

default:

fprintf(list,"SYNERR: tok mismatch expecting Identifier, instead received %s\n",tok.word);

while (strcmp(tok.word,":=") != 0) {

tok = get\_next\_token(file,list,token);

fprintf(trace, "var: trying to sync on %s\n", tok.word);

}

fprintf(trace, "var: sync on token %s\n", tok.word);

return TYPEERR;

}

}

int stmtT(int i\_val){//need to return type something on correct stmtT

int r\_val;

int f\_val;

fprintf(trace, "%s %s\n","stmtT",tok.word );

switch (tok.tkn) {

case ELSE:

tok\_match(ELSE,0); f\_val = stmt();

if(f\_val == TYPEVOID){

r\_val = TYPEVOID;

}

else {

r\_val = TYPEERR;

//print error in stmt function

}

return r\_val;

case SEMICOLON: case END:

r\_val = i\_val;

return r\_val;

default:

fprintf(list,"SYNERR: tok mismatch expecting %s, %s, instead received %s\n","else",";",tok.word);

while (strcmp(tok.word,"else") != 0 && strcmp(tok.word,";") != 0 && strcmp(tok.word,"end") != 0) {

tok = get\_next\_token(file,list,token);

fprintf(trace, "stmtT: trying to sync on %s\n", tok.word);

}

fprintf(trace, "stmtT: sync on token %s\n", tok.word);

//return TYPEERR;

}

}

int stmt(){//need to return type void on correct stmt

int r\_val;

int f\_val;

int f2\_val;

fprintf(trace, "%s %s\n","stmt",tok.word );

//printf("%d,%d,%s\n",tok.tkn,tok.attr.val,tok.word);

switch (tok.tkn) {

case ID:

f\_val = var();printf("stmt before expr1\n"); tok\_match(ASSIGNOP,0);

printf("stmt before expr2\n");

f2\_val = expr();

if(f\_val == TYPEINT && f2\_val == TYPEINT){

r\_val = TYPEVOID;

}

else if(f\_val == TYPEREAL && f2\_val == TYPEREAL){

r\_val = TYPEVOID;

}

else{

if(f\_val != TYPEERR){

fprintf(list, "SEMERR: Type mismatch on assignment expecting type %s received type %s\n",tokcode2str(f\_val),tokcode2str(f2\_val));

}

else{

//this semantic error is handled within the var function

}

r\_val = TYPEERR;

}

return r\_val;

case BEGIN:

f\_val = cmpd\_stmt();

if(f\_val == TYPEVOID){

r\_val = TYPEVOID;

}

return r\_val;

case IF:

tok\_match(IF,0); f\_val = expr(); tok\_match(THEN,0); f2\_val = stmt(); r\_val = stmtT(f2\_val);

if(f\_val == TYPEBOOL){

if(r\_val == TYPEVOID){

//for r\_val to be type void, f2\_val must be type void

r\_val = TYPEVOID;

}

else if(f2\_val != TYPEVOID){

r\_val = TYPEERR;

}

else if(r\_val != TYPEVOID){

r\_val = TYPEERR;

}

}

else{

r\_val = TYPEERR;

fprintf(list, "SEMERR: if statement cannot be evaluated with non-boolean expression\n");

}

return r\_val;

case WHILE:

tok\_match(WHILE,0); f\_val = expr(); tok\_match(DO,0); f2\_val = stmt();

if(f\_val == TYPEBOOL){

if(f2\_val == TYPEVOID){

r\_val = TYPEVOID;

}

else{

//handled within other statement stuff

r\_val = TYPEERR;

}

}

return r\_val;

default:

fprintf(list,"SYNERR: tok mismatch expecting %s,%s,%s,%s instead received %s\n","Identifier","begin","if","while",tok.word);

while (strcmp(tok.word,"else") != 0 && strcmp(tok.word,";") != 0 && strcmp(tok.word,"end") != 0) {

tok = get\_next\_token(file,list,token);

fprintf(trace, "stmt: trying to sync on %s\n", tok.word);

}

fprintf(trace, "stmt: sync on token %s\n", tok.word);

//return TYPEERR;

}

}

int stmt\_lstT(int i\_val){

int f\_val;

int r\_val;

fprintf(trace, "%s %s\n","stmt\_lstT" ,tok.word);

switch (tok.tkn) {

case SEMICOLON:

tok\_match(SEMICOLON,0); f\_val = stmt(); r\_val = stmt\_lstT(f\_val);

return r\_val;

case END:

fprintf(trace, "%s\n","stmt\_lstT end break no tok consumption" );

r\_val = i\_val;

return r\_val;

default:

fprintf(list,"SYNERR: tok mismatch expecting %s,%s, instead received %s\n",";","end",tok.word);

while (strcmp(tok.word,"end") != 0) {

tok = get\_next\_token(file,list,token);

fprintf(trace, "stmt\_lstT: trying to sync on %s\n", tok.word);

printf("Stuck here with tok: %d\n",tok.tkn);

}

fprintf(trace, "stmt\_lstT: sync on token %s\n", tok.word);

return TYPEERR;

}

}

int stmt\_lst(){

int r\_val;

int f\_val;

fprintf(trace, "%s %s\n","stmt\_lst",tok.word );

switch (tok.tkn) {

case ID: case BEGIN: case IF: case WHILE:

f\_val = stmt(); r\_val = stmt\_lstT(f\_val);

return r\_val;

default:

fprintf(list,"SYNERR: tok mismatch expecting %s,%s,%s,%s, instead received %s\n","Identifier","begin","if","while",tok.word);

while (strcmp(tok.word,"end") != 0) {

tok = get\_next\_token(file,list,token);

fprintf(trace, "stmt\_lst: trying to sync on %s\n", tok.word);

}

fprintf(trace, "stmt\_lst: sync on token %s\n", tok.word);

return TYPEERR;

}

}

int opt\_stmt(){

int r\_val;

fprintf(trace, "%s %s\n","opt\_stmt",tok.word );

switch (tok.tkn) {

case ID: case BEGIN: case IF: case WHILE:

r\_val = stmt\_lst();

return r\_val;

default:

fprintf(list,"SYNERR: tok mismatch expecting %s,%s,%s,%s, instead received %s\n","Identifier","begin","if","while",tok.word);

while (strcmp(tok.word,"end") != 0) {

tok = get\_next\_token(file,list,token);

fprintf(trace, "opt\_stmt: trying to sync on %s\n", tok.word);

}

fprintf(trace, "opt\_stmt: sync on token %s\n", tok.word);

return TYPEERR;

}

}

int cmpd\_stmtT(){

int r\_val;

fprintf(trace, "%s %s\n","cmpd\_stmtT" ,tok.word);

switch (tok.tkn) {

case END:

tok\_match(END,0);

r\_val = TYPEVOID;

return r\_val;

case BEGIN: case IF: case WHILE: case ID:

r\_val = opt\_stmt(); tok\_match(END,0);

return r\_val;

default:

fprintf(list,"SYNERR: tok mismatch expecting %s,%s,%s,%s,%s,%s instead received %s\n","Identifier","end","begin","if","while","do",tok.word);

while (strcmp(tok.word,".") != 0 && strcmp(tok.word,"function") != 0 && strcmp(tok.word,"begin") != 0 && strcmp(tok.word,";") != 0 && strcmp(tok.word,"end") != 0) {

tok = get\_next\_token(file,list,token);

fprintf(trace, "cmpd\_stmtT: trying to sync on %s\n", tok.word);

}

fprintf(trace, "cmpd\_stmtT: sync on token %s\n", tok.word);

return TYPEERR;

}

}

int cmpd\_stmt(){

int r\_val;

fprintf(trace, "%s %s\n","cmpd\_stmt",tok.word );

switch (tok.tkn) {

case BEGIN:

tok\_match(BEGIN,0); r\_val = cmpd\_stmtT();

return r\_val;

default:

fprintf(list,"SYNERR: tok mismatch expecting %s, instead received %s\n","begin",tok.word);

while (strcmp(tok.word,".") != 0 && strcmp(tok.word,"function") != 0 && strcmp(tok.word,"begin") != 0 && strcmp(tok.word,";") != 0 && strcmp(tok.word,"end") != 0) {

tok = get\_next\_token(file,list,token);

fprintf(trace, "cmpd\_stmt: trying to sync on %s\n", tok.word);

}

fprintf(trace, "cmpd\_stmt: sync on token %s\n", tok.word);

return TYPEERR;

}

}

//THIS IS A BLUE NODE FABRICATOR

void param\_lstT(){

int f\_val;

int gn=1;

int scope\_in =0;

char name[15];

int addr\_skip = addr;

int lex\_bad = 0;

fprintf(trace, "%s %s\n","param\_lstT",tok.word);

switch (tok.tkn) {

case SEMICOLON:

tok\_match(SEMICOLON,0);

if(scope\_flag == 1){

scope\_in = 1;//go down a level in scope

scope\_level++;

addr = 0;

fprintf(ad, "\n" );

scope\_flag = 0;

}

tok\_ret = tok\_match(ID,0);

lex\_bad = tok\_ret.type;

strcpy(name,tok\_ret.word);

tok\_match(COLON,0); f\_val = type();

addr=addr\_skip;

if(f\_val != TYPEERR && lex\_bad != TYPEERR){

gn = check\_add\_node(st,name,f\_val,YELLOW,scope\_in);

}

if(gn != 0){

fprintf(list, "SEMERR: variable %s already declared in scope\n",name );

if(scope\_in == 1){

scope\_flag = 1;

}

}

if(f\_val != TYPEERR && gn == 0){

fprintf(nodes,"%-15s %-15s %-15s\n",name,tokcode2str(f\_val), near\_parent(st,name));

}

param\_lstT();

break;

case PAREN:

if(tok.attr.val == PAREN\_CLOSE){

break;

}

default:

fprintf(list,"SYNERR: tok mismatch expecting %s, %s, instead received %s\n",";",")",tok.word);

while (strcmp(tok.word,")") != 0) {

tok = get\_next\_token(file,list,token);

fprintf(trace, "param\_lstT: trying to sync on %s\n", tok.word);

}

fprintf(trace, "param\_lstT: sync on token %s\n", tok.word);

//return TYPEERR;

}

}

//THIS IS A BLUE NODE FABRICATOR

void param\_lst(){

int f\_val;

int scope\_in = 0;

int gn=1;

char name[15];

int addr\_skip=addr;

int lex\_bad=0;

fprintf(trace, "%s %s\n","param\_lst",tok.word );

switch (tok.tkn) {

case ID:

if(scope\_flag == 1){

scope\_in = 1;//go down a level in scope

scope\_level++;

addr = 0;

fprintf(ad, "\n" );

//var\_ad = 0;

scope\_flag = 0;

}

tok\_ret = tok\_match(ID,0);

lex\_bad = tok\_ret.type;

strcpy(name,tok\_ret.word);

tok\_match(COLON,0); f\_val = type();

addr=addr\_skip;

if(f\_val != TYPEERR && lex\_bad != TYPEERR){

gn = check\_add\_node(st,name,f\_val,YELLOW,scope\_in);

}

if(gn != 0){

fprintf(list, "SEMERR: variable %s already declared in scope\n",name );

if(scope\_in == 1){

scope\_flag = 1;

}

}

if(f\_val != TYPEERR && gn == 0){

//fprintf(ad,"%-15s %-15d\n",name,var\_ad);

fprintf(nodes,"%-15s %-15s %-15s\n",name,tokcode2str(f\_val), near\_parent(st,name));

}

param\_lstT();

break;

default:

fprintf(list,"SYNERR: tok mismatch expecting Identifier, instead received %s\n",tok.word);

while (strcmp(tok.word,")") != 0) {

tok = get\_next\_token(file,list,token);

fprintf(trace, "param\_lst: trying to sync on %s\n", tok.word);

}

fprintf(trace, "param\_lst: sync on token %s\n", tok.word);

//return TYPEERR;

}

}

void args(){

fprintf(trace, "%s %s\n","args",tok.word);

switch (tok.tkn) {

case PAREN:

if (tok.attr.val == PAREN\_OPEN){

tok\_match(PAREN,PAREN\_OPEN); param\_lst(); tok\_match(PAREN,PAREN\_CLOSE); break;

}

default:

fprintf(list,"SYNERR: tok mismatch expecting %s, instead received %s\n","(",tok.word);

while (strcmp(tok.word,":") != 0) {

tok = get\_next\_token(file,list,token);

fprintf(trace, "args: trying to sync on %s\n", tok.word);

}

fprintf(trace, "args: sync on token %s\n", tok.word);

//return TYPEERR;

}

}

//GREEN NODE FABRICATOR

int sub\_headT(){

fprintf(trace, "%s %s\n","sub\_headT",tok.word);

int r\_val;

int f\_val;

switch (tok.tkn) {

case COLON:

tok\_match(COLON,0); f\_val = std\_type(); tok\_match(SEMICOLON,0);

if(f\_val == TYPEINT ){

r\_val = TYPEINT;

}

else if(f\_val == TYPEREAL){

r\_val = TYPEINT;

}

else{

fprintf(list,"SEMERR: Invalid type for function, valid types are %s or %s\n",tokcode2str(TYPEINT),tokcode2str(TYPEREAL));

r\_val = TYPEERR;

}

return r\_val;

case PAREN:

if (tok.attr.val == PAREN\_OPEN){

args(); tok\_match(COLON,0); f\_val = std\_type(); tok\_match(SEMICOLON,0);

if(f\_val == TYPEINT ){

r\_val = TYPEINT;

}

else if(f\_val == TYPEREAL){

r\_val = TYPEREAL;

}

else{

fprintf(list,"SEMERR: Invalid type for function, valid types are %s or %s\n",tokcode2str(TYPEINT),tokcode2str(TYPEREAL));

r\_val = TYPEERR;

}

return r\_val;

}

default:

fprintf(list,"SYNERR: tok mismatch expecting %s, %s, instead received %s\n",":","(",tok.word);

while(strcmp(tok.word,"var") != 0 && strcmp(tok.word,"begin") != 0 && strcmp(tok.word,"function") != 0){

tok = get\_next\_token(file,list,token);

fprintf(trace, "sub\_headT: trying to sync on %s\n", tok.word);

}

fprintf(trace, "sub\_headT: sync on token %s\n", tok.word);

return TYPEERR;

}

}

//GREEN NODE FABRICATOR

int sub\_head(){

fprintf(trace, "%s %s\n","sub\_head" ,tok.word);

int f\_val; //function return type

int r\_val;

int scope\_in = 0;

char func\_name[15];

switch (tok.tkn) {

case FUNCTION:

if(scope\_flag == 1){

scope\_in = 1;//go down a level in scope

scope\_level++;

addr = 0;

scope\_flag = 0;

}

tok\_match(FUNCTION,0); tok\_ret = tok\_match(ID,0);

strcpy(func\_name,tok\_ret.word);

check\_add\_node(st,tok\_ret.word,TYPEVOID,GREEN,scope\_in);

fprintf(nodes,"%-15s %-15s %-15s\n",func\_name,"TBD", near\_parent(st,func\_name));

scope\_flag=1;

f\_val = sub\_headT();

set\_id\_type(st,func\_name,f\_val);

if(f\_val == TYPEVOID){

r\_val = TYPEVOID;

}

else{

r\_val = TYPEERR;

}

return r\_val;

default:

fprintf(list,"SYNERR: tok mismatch expecting %s, instead received %s\n","function",tok.word);

while(strcmp(tok.word,"var") != 0 && strcmp(tok.word,"begin") != 0 && strcmp(tok.word,"function") != 0){

tok = get\_next\_token(file,list,token);

fprintf(trace, "sub\_head: trying to sync on %s\n", tok.word);

}

fprintf(trace, "sub\_head: sync on token %s\n", tok.word);

return TYPEERR;

}

}

void sub\_decTT(){

fprintf(trace, "%s %s\n","sub\_decTT",tok.word);

switch (tok.tkn) {

case FUNCTION:

sub\_decs(); cmpd\_stmt();

break;

case BEGIN:

cmpd\_stmt();

break;

default:

fprintf(list,"SYNERR: tok mismatch expecting %s, %s, instead received %s\n","function","begin",tok.word);

while(strcmp(tok.word,"begin") != 0 && strcmp(tok.word,"function") != 0){

tok = get\_next\_token(file,list,token);

fprintf(trace, "sub\_decTT: trying to sync on %s\n", tok.word);

}

fprintf(trace, "sub\_decTT: sync on token %s\n", tok.word);

//return TYPEERR;

}

}

void sub\_decT(){

fprintf(trace, "%s %s\n","sub\_decT" ,tok.word);

switch (tok.tkn) {

case VAR:

decs(); sub\_decTT();

break;

case FUNCTION:

sub\_decs(); cmpd\_stmt();

break;

case BEGIN:

cmpd\_stmt();

break;

default:

fprintf(list,"SYNERR: tok mismatch expecting %s, %s, %s, instead received %s\n","var","function","begin",tok.word);

while(strcmp(tok.word,"begin") != 0 && strcmp(tok.word,"function") != 0){

tok = get\_next\_token(file,list,token);

fprintf(trace, "sub\_decT: trying to sync on %s\n", tok.word);

}

fprintf(trace, "sub\_decT: sync on token %s\n", tok.word);

//return TYPEERR;

}

}

void sub\_dec(){

fprintf(trace, "%s %s\n","sub\_dec" ,tok.word);

switch (tok.tkn) {

case FUNCTION:

sub\_head(); sub\_decT();//upscope here!!!!

if(scope\_level > 1 && scope\_flag == 0){

up\_scope(st);

}

scope\_flag = 0;

break;

default:

fprintf(list,"SYNERR: tok mismatch expecting %s instead received %s\n","function",tok.word);

while(strcmp(tok.word,"begin") != 0 && strcmp(tok.word,"function") != 0){

tok = get\_next\_token(file,list,token);

fprintf(trace, "sub\_dec: trying to sync on %s\n", tok.word);

}

fprintf(trace, "sub\_dec: sync on token %s\n", tok.word);

//return TYPEERR;

}

}

void sub\_decsT(){

fprintf(trace, "%s %s\n","sub\_decsT" ,tok.word);

switch (tok.tkn) {

case FUNCTION:

sub\_dec(); tok\_match(SEMICOLON,0); sub\_decsT();

break;

case BEGIN:

break;

default:

fprintf(list,"SYNERR: tok mismatch expecting %s, %s, instead received %s\n","function","begin",tok.word);

while(strcmp(tok.word,"begin") != 0){

tok = get\_next\_token(file,list,token);

fprintf(trace, "sub\_decsT: trying to sync on %s\n", tok.word);

}

fprintf(trace, "sub\_decsT: sync on token %s\n", tok.word);

//return TYPEERR;

}

}

void sub\_decs(){

fprintf(trace, "%s %s\n","sub\_decs" ,tok.word );

switch (tok.tkn) {

case FUNCTION:

sub\_dec(); tok\_match(SEMICOLON,0); sub\_decsT();

break;

default:

fprintf(list,"SYNERR: tok mismatch expecting %s instead received %s\n","function",tok.word);

while(strcmp(tok.word,"begin") != 0){

tok = get\_next\_token(file,list,token);

fprintf(trace, "sub\_decs: trying to sync on %s\n", tok.word);

}

fprintf(trace, "sub\_decs: sync on token %s\n", tok.word);

//return TYPEERR;

}

}

int std\_type(){

fprintf(trace, "%s %s\n","std\_type" ,tok.word);

switch(tok.tkn){

case INTEGER://assign identifiers type integer or real here

tok\_match(INTEGER,0);

return TYPEINT;

case REAL:

tok\_match(REAL,0);

return TYPEREAL;

default:

fprintf(list,"SYNERR: tok mismatch expecting ; instead received %s\n",tok.word);

while(strcmp(tok.word,";") != 0){

tok = get\_next\_token(file,list,token);

fprintf(trace, "std\_type: trying to sync on %s\n", tok.word);

}

fprintf(trace, "std\_type: sync on token %s\n", tok.word);

return TYPEERR;

}

}

int type(){

fprintf(trace, "%s %s\n","type" ,tok.word);

int s;

int arr\_check;

int err\_flag=0;

int backup\_addr = addr;

switch (tok.tkn) {

case INTEGER: case REAL:

s = std\_type();

if(s == TYPEINT){

addr = addr + 4;

}

else if(s == TYPEREAL){

addr = addr + 8;

}

else{

addr = backup\_addr;

}

return s;

case ARRAY:

//for type checking in p3 the two num values need to be of type integer

tok\_match(ARRAY,0); tok\_match(BRACK,BRACK\_OPEN);

tok\_ret = tok\_match(NUM,0);

arr\_check = tok\_ret.type;

int num1 = atoi(tok\_ret.word);

if(arr\_check != TYPEINT){

err\_flag=1;

fprintf(list, "SEMERR: non integer number used for array creation\n" );

}

tok\_match(DOTDOT,0);

tok\_ret = tok\_match(NUM,0);

arr\_check = tok\_ret.type;

int num2 = atoi(tok\_ret.word);

if(arr\_check != TYPEINT){

err\_flag=1;

fprintf(list, "SEMERR: non integer number used for array creation\n" );

}

tok\_match(BRACK,BRACK\_CLOSE);

tok\_match(OF,0); s = std\_type();

if (s != TYPEERR){

s = s + 3;//type is transformed into respective array type

}

if(err\_flag){

s=TYPEERR;

addr = backup\_addr;

}

if(num1 > num2){

addr = backup\_addr;

fprintf(list, "SEMERR: the first index of an array cannot be greater than the second\n" );

s=TYPEERR;

}

else{

if(s-3 == TYPEINT){

addr = addr + ((num2 - num1)+1)\*4;

}

else if(s-3 == TYPEREAL){

addr = addr + ((num2 - num1)+1)\*8;

}

else{

addr = backup\_addr;

}

}

return s;

default:

fprintf(list,"SYNERR: tok mismatch expecting %s, %s, %s, instead received %s\n","integer","real","array",tok.word);

while(strcmp(tok.word,";") != 0){

tok = get\_next\_token(file,list,token);

fprintf(trace, "type: trying to sync on %s\n", tok.word);

}

fprintf(trace, "type: sync on token %s\n", tok.word);

return TYPEERR;

}

}

void decsT(){

int f\_val;

fprintf(trace, "%s %s\n","decsT" ,tok.word);

int node\_color = BLUE;

char name[15];

int gn=1;

int scope\_in = 0;

int var\_ad = addr;

int lex\_bad=0;

switch(tok.tkn){

case VAR://grab a variable declaration from here

if(scope\_flag == 1){

scope\_in = 1;//go down a level in scope

scope\_level++;

addr = 0;

var\_ad = 0;

fprintf(ad, "\n" );

scope\_flag = 0;

}

tok\_match(VAR, 0); tok\_ret = tok\_match(ID,0);

lex\_bad = tok\_ret.type;

strcpy(name,tok\_ret.word);

tok\_match(COLON, 0); f\_val = type();

if(f\_val != TYPEERR && lex\_bad != TYPEERR){

gn = check\_add\_node(st,name,f\_val,BLUE,scope\_in);//create blue node

}

if(gn < 0){

fprintf(list, "SEMERR: variable %s already declared in scope\n",name );

if(scope\_in == 1){

scope\_flag = 1;

}

}

if(f\_val != TYPEERR && gn == 0){

fprintf(ad,"%-15s %-15d %-15s\n",name,var\_ad,near\_parent(st,name));

fprintf(nodes,"%-15s %-15s %-15s\n",name,tokcode2str(f\_val), near\_parent(st,name));

}

tok\_match(SEMICOLON,0);

decsT();

break;

case FUNCTION: case BEGIN:

break;

default:

fprintf(list,"SYNERR: tok mismatch expecting %s %s %s instead received %s\n","var","function","begin",tok.word);

while(strcmp(tok.word,"function") != 0 && strcmp(tok.word,"begin") != 0){

//print error message

tok = get\_next\_token(file,list,token);

fprintf(trace, "decsT: trying to sync on %s\n", tok.word);

}

fprintf(trace, "decsT: sync on token %s\n", tok.word);

//return TYPEERR;

}

}

void decs(){

int f\_val;

int scope\_in = 0;

char name[15];

int gn=1;

int var\_ad=addr;

int lex\_bad=0;

fprintf(trace, "%s %s\n","decs" ,tok.word);

switch(tok.tkn){

case VAR:

//function parameters present

if(scope\_flag == 1){

scope\_in = 1;//go down a level in scope

scope\_level++;

addr = 0;

var\_ad = 0;

fprintf(ad, "\n" );

scope\_flag = 0;

}

//create a blue node here and attach it to what the last function (green node) to put it into scope

tok\_match(VAR, 0); tok\_ret = tok\_match(ID,0);

lex\_bad = tok\_ret.type;

strcpy(name,tok\_ret.word);

tok\_match(COLON, 0); f\_val = type();

//down scope value must be 1 on the first declaration

if(f\_val != TYPEERR && lex\_bad != TYPEERR){

gn = check\_add\_node(st,name,f\_val,BLUE,scope\_in);//create blue node

}

if(gn != 0){

fprintf(list, "SEMERR: variable %s already declared in scope\n",name );

if(scope\_in == 1){

scope\_flag = 1;

}

}

if(f\_val != TYPEERR && gn == 0){

fprintf(ad,"%-15s %-15d %-15s\n",name,var\_ad,near\_parent(st,name));

fprintf(nodes,"%-15s %-15s %-15s\n",name,tokcode2str(f\_val), near\_parent(st,name));

}

tok\_match(SEMICOLON,0);

decsT();

break;

default:

fprintf(list,"SYNERR: tok mismatch expecting %s instead received %s\n","var",tok.word);

while(strcmp(tok.word,"function") != 0 && strcmp(tok.word,"begin") != 0){

//print error message

tok = get\_next\_token(file,list,token);

fprintf(trace, "decs: trying to sync on %s\n", tok.word);

}

fprintf(trace, "decs: sync on token %s\n", tok.word);

//return TYPEERR;

}

}

void id\_lstT(){

fprintf(trace, "%s %s\n","id\_lstT" ,tok.word);

char syncSet[] = {')'};

char name[15];

char p\_name[15];

int scope\_in =0;

switch(tok.tkn){

case COMMA:

if(scope\_flag == 1){

scope\_in = 1;//does not go down a level in scope

scope\_level++;

addr = 0;

scope\_flag = 0;

}

tok\_match(COMMA,0); tok\_ret = tok\_match(ID,0);

strcpy(name,tok\_ret.word);

if(tok\_ret.type != TYPEERR){

check\_add\_node(st,tok\_ret.word,TYPEVOID,BLUE,scope\_in);

//fprintf(nodes,"%-15s %-15s %-15s\n",name,tokcode2str(TYPEVOID),near\_parent(st,name));

strcpy(p\_name,near\_parent(st,name));

fprintf(nodes,"%-15s %-15s %-15s\n",name,tokcode2str(TYPEVOID),p\_name);

}

else{

if(scope\_in==1){

scope\_flag = 1;

}

}

id\_lstT();

break;

case PAREN:

if(tok.attr.val == PAREN\_CLOSE){

break;

}

default:

//print error message

fprintf(list,"SYNERR: tok mismatch expecting %s instead received %s\n",syncSet,tok.word);

while(strcmp(tok.word,")") != 0){//paren close match

//print error message

tok = get\_next\_token(file,list,token);

fprintf(trace, "id\_lstT: trying to sync on %s\n", tok.word);

}

fprintf(trace, "id\_lstT: sync on token %s\n", tok.word);

//return TYPEERR;

}

}

void id\_lst(){

fprintf(trace, "%s %s\n","id\_lst" ,tok.word);

char syncSet[] = {')'};

int scope\_in = 0;

char name[15];

char p\_name[15];

switch(tok.tkn){

case ID:

if(scope\_flag == 1){

scope\_in = 1;//does not go down a level in scope

scope\_level++;

addr = 0;

scope\_flag = 0;

}

tok\_ret = tok\_match(ID,0); //need to go down scope otherwise

strcpy(name,tok\_ret.word);

if(tok\_ret.type != TYPEERR){

check\_add\_node(st,tok\_ret.word,TYPEVOID,BLUE,scope\_in);

strcpy(p\_name,near\_parent(st,name));

fprintf(nodes,"%-15s %-15s %-15s\n",name,tokcode2str(TYPEVOID),p\_name);

}

else{

if(scope\_in == 1){

scope\_flag =1;

}

}

id\_lstT();

break;

default:

fprintf(list,"SYNERR: tok mismatch expecting %s instead received %s\n","Identifier",tok.word);

while(strcmp(tok.word,")") != 0){

//print error message

tok = get\_next\_token(file,list,token);

fprintf(trace, "id\_lst: trying to sync on %s\n", tok.word);

}

fprintf(trace, "id\_lst: sync on token %s\n", tok.word);

//return TYPEERR;

}

}

void prgmTT(){

fprintf(trace, "%s %s\n","prgmTT" ,tok.word );

switch(tok.tkn){

case FUNCTION:

sub\_decs(); cmpd\_stmt(); tok\_match(DOT,0);

break;

case BEGIN:

cmpd\_stmt(); tok\_match(DOT,0);

break;

default:

fprintf(list, "SYNERR: tok mismatch expecting %s instead received %s\n","function begin",tok.word);

while(tok.tkn != EOF){

//print error message

tok = get\_next\_token(file,list,token);

fprintf(trace, "prgmTT: trying to sync on %s\n", tok.word);

}

fprintf(trace, "prgmTT: sync on token %s\n", tok.word);

//return TYPEERR;

}

}

void prgmT(){

fprintf(trace, "%s %s\n","prgmT" ,tok.word);

switch(tok.tkn){

case VAR:

decs(); prgmTT();

break;

case FUNCTION:

sub\_decs(); cmpd\_stmt(); tok\_match(DOT,0);

break;

case BEGIN:

cmpd\_stmt(); tok\_match(DOT,0);

break;

default: // this is the otherwise statement

fprintf(list, "SYNERR: tok mismatch expecting %s instead received %s\n","var, function, begin",tok.word);

while (tok.tkn != EOF){

//print error message

tok = get\_next\_token(file,list,token);

fprintf(trace, "prgmT: trying to sync on %s\n", tok.word);

}

fprintf(trace, "prgmT: sync on token %s\n", tok.word);

//return TYPEERR;

}

}

void prgm(){

fprintf(trace, "%s %s\n","prgm" ,tok.word);

char name[15];

switch(tok.tkn){

case PROGRAM:

scope\_flag = 1;

tok\_match(PROGRAM,0); tok\_ret = tok\_match(ID,0);

strcpy(name,tok\_ret.word);

tok\_match(PAREN, PAREN\_OPEN);

check\_add\_node(st,name,TYPEVOID,GREEN,0);//create program node here

//fprintf(nodes,"%-15s %-15s %-15s\n",name,tokcode2str(TYPEVOID),near\_parent(st,tok\_ret.word));

fprintf(nodes,"%-15s ",name);

fprintf(nodes , "%-15s ",tokcode2str(TYPEVOID) );

fprintf(nodes, "%-15s\n", near\_parent(st,name));

id\_lst(); tok\_match(PAREN,PAREN\_CLOSE); tok\_match(SEMICOLON,0);

prgmT();

break;

default: //this will be the otherwise statement

fprintf(list, "SYNERR: tok mismatch expecting %s instead received %s\n","program",tok.word);

while (tok.tkn != EOF){

//print error message

tok = get\_next\_token(file,list,token);

fprintf(trace, "prgm: trying to sync on %s\n", tok.word);

}

fprintf(trace, "prgm: sync on token %s\n", tok.word);

//return TYPEERR;

}

}

void parse(FILE \*f,FILE \*l, FILE \*t, FILE \*tr,FILE \*n,FILE \*a){

file = f;

list = l;

token = t;

trace = tr;

nodes = n;

ad = a;

fprintf(nodes, "%-15s %-15s %-15s\n","nodes:","type:","NP:" );

fprintf(ad, "%-15s %-15s %-15s\n","node name","address","scope function");

tok = get\_next\_token(file,list,token);

prgm();

tok\_match(EOF,0);

print\_stack(st);

}

#endif

**Node.c**

#ifndef NODE\_H

#define NODE\_H

/\*

What needs to be done:

proper working nodes in terms of creation addressing printing

node scope structure

What is done:

rough creation of node structure as well as some functionality

nodes can be created and added to a blue node stack

Possible problems:

does the blue node need to be able to have children nodes?

answer: No not if blue nodes are made children of the function nodes they are apart of

solution: blue nodes need to be able to be children

\*/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include "../lexeme.c"

#define GREEN 101 //function or program

#define BLUE 100 //declared variable

#define YELLOW 99 //function parameter/variable

/\*

All functions that have parameters require the first parameter to be treated as a child node

\*/

char np\_name[15];

struct node{

int type;

int color;

char name[15];

struct node \*next;

struct node \*prev;

struct node \*child;

struct node \*parent;

};

typedef struct node \*node\_ptr;

struct node\_stack{

struct node \*stackbase;

struct node \*stacktop;

};

typedef struct node\_stack \*stack\_ptr;

/\*

this just adds a node by moving the stack top pointer arround

\*/

void add\_make\_node(stack\_ptr st, char name[15], int type, int color,int down\_scope){

node\_ptr new\_node = (node\_ptr)malloc(sizeof(struct node));//create new\_node

strcpy(new\_node->name,name);

new\_node->color = color;

new\_node->type = type;

if (st->stackbase == NULL){

printf("Setting up Stack\n" );

st->stackbase = new\_node;

st->stacktop = new\_node;

}

else if(down\_scope){

st->stacktop->child = new\_node;

new\_node->parent = st->stacktop;

st->stacktop = new\_node;

}

else{

st->stacktop->next = new\_node;

new\_node->prev = st->stacktop;

st->stacktop = new\_node;

}

}

/\*

This is the method used for creating new nodes for the stack

\*/

int check\_add\_node(stack\_ptr st,char name[15],int type, int color,int down\_scope){

//only check within local scope

node\_ptr temp = st->stacktop;

if (st->stacktop != NULL){

while(strcmp(temp->name,name)!=0){

//not same keep searching

if(temp->prev == NULL){

if(temp->parent != NULL){

printf("reached base of scope, valid to add node\n");

add\_make\_node(st,name,type,color,down\_scope);

return 0;

}

else{

if(temp == st->stackbase){

printf("reached base node, valid to add node\n");

add\_make\_node(st,name,type,color,down\_scope);

return 0;

}

else{

//throw

}

}

}

else{

temp = temp->prev;

}

}

//found node with same name say no node no

printf("Found Node with same name within scope, cannot create new node\n");

return -1;

}

else{

add\_make\_node(st,name,type,color,down\_scope);

return 0;

}

}

node\_ptr pop\_node(stack\_ptr st){

node\_ptr cur = st->stacktop;

node\_ptr prev;

if(cur != NULL){

if (cur->prev == NULL){

printf("end of scope, going up scope level\n");

if(cur->parent != NULL) {

prev = cur->parent;

prev->child = NULL;

st->stacktop = prev;

}

else {

printf("No parent node\n");

st->stacktop = NULL;

st->stackbase = NULL;

}

printf("poped node: %s\n",cur->name);

return cur;

}

else {

prev = cur->prev;

prev->next = NULL;

st->stacktop = prev;

printf("poped node: %s\n",cur->name);

return cur;

}

}

else{

printf("Stack is empty!! Returning NULL\n");

return st->stacktop;

}

}

void up\_scope(stack\_ptr st){

node\_ptr temp = st->stacktop;

if(temp == NULL) {

printf("Stack is empty\n" );

return;

}

while (temp->parent == NULL){

if(temp == st->stackbase){

printf("Stackbase reached\n");

return;

}

temp=temp->prev;

}

temp = temp->parent;

st->stacktop = temp;

printf("scope up new stacktop: %s\n",temp->name);

}

/\*

this method searches a stack within a scope level to see if a node is present in the stact

returns pointer to node if found NULL otherwise

\*/

node\_ptr up\_stream\_node\_find(stack\_ptr st, char name[15]){

node\_ptr temp = st->stacktop;

if (temp == NULL) return NULL;

while (temp != st->stackbase){

if(strcmp(temp->name,name) == 0){

//congrats a node within scope level is Found

return temp;

}

if(temp->prev == NULL){

//go up to parent

temp = temp->parent;

}

else{temp = temp->prev;}

}

temp = st->stackbase;

if(strcmp(temp->name,name) == 0){

//congrats a node within scope level is Found

return temp;

}

return NULL;

}

char\* near\_parent(stack\_ptr st, char name[15]){

node\_ptr temp = up\_stream\_node\_find(st,name);

if(temp == NULL){

printf("UNABLE TO FIND NODE\n");

return NULL;

}

while(temp->parent == NULL){

if(temp->prev == NULL){

printf("UNABLE TO FIND PARENT\n");

return "NA";

}

temp=temp->prev;

}

temp = temp->parent;

strcpy(np\_name,temp->name);

printf("TEST: np\_name %s!!!!!!!!!!!!!!!\n", np\_name);

return np\_name;

}

int get\_id\_type(stack\_ptr st, char name[15]){

node\_ptr temp = up\_stream\_node\_find(st,name);

if(temp == NULL){

printf("NO ID WITH NAME %s WITHIN SCOPE\n", name);

return TYPEERR;

}

return temp->type;

}

int set\_id\_type(stack\_ptr st, char name[15],int type){

node\_ptr temp = up\_stream\_node\_find(st,name);

if(temp == NULL){

printf("NO ID WITH NAME %s\n",name);

return -1;

}

temp->type = type;

return 0;

}

int in\_scope(stack\_ptr st, char name[15]){

node\_ptr temp = up\_stream\_node\_find(st,name);

if(temp == NULL){

return -1;//no node in scope

}

return 1;

}

int get\_func\_param\_count(stack\_ptr st, char name[15]){

node\_ptr temp = up\_stream\_node\_find(st,name);

int params = 0;

if(temp == NULL){

printf("no func with name %s\n",name );

return -1;

}

if(temp->color != GREEN){

printf("%s is not a function\n",name);

return -2;

}

if(temp->child == NULL){

return 0;

}

if(temp->child->color != YELLOW){

return 0;

}

//the first child node must now be yellow

temp = temp->child;

params++;

while(temp->color == YELLOW && temp->next != NULL){

temp=temp->next;

if(temp->color!=YELLOW){

break;

}

params++;

}

return params;

}

int get\_func\_param\_type(stack\_ptr st, char name[15],int param\_num){

node\_ptr temp = up\_stream\_node\_find(st,name);

if(temp == NULL){

printf("no func with name %s\n",name );

return -1;

}

int max\_param\_num = get\_func\_param\_count(st,name);

if(max\_param\_num == 0){

return -2;//no parameters for function

}

if(max\_param\_num < param\_num){

return -3;//too many function parameters in function call

}

int itter=1;

temp = temp->child;

while(param\_num != itter){

temp = temp->next;

itter++;

}

return temp->type;

}

void print\_stack(stack\_ptr st){

node\_ptr temp = st->stackbase;

if(st->stackbase != NULL){

printf("stackbase: %s\nstacktop: %s\n",st->stackbase->name,st->stacktop->name);

while (temp->next != st->stacktop){

if(temp->next == NULL) {

if(temp->child != NULL){

printf("node name: %s type: %d color: %d\n",temp->name,temp->type,temp->color);

temp = temp->child;

continue;

}

else break;

}

printf("node name: %s type: %d color: %d\n",temp->name,temp->type,temp->color);

temp = temp->next;

}

printf("node name: %s type: %d color: %d\n",temp->name,temp->type,temp->color);

printf("node name: %s type: %d color: %d\n",st->stacktop->name,st->stacktop->type,st->stacktop->color);

}

else{

printf("Empty Stack\n");

}

}

#endif