**CS4013**

**Project 2**

A Syntactical Analyzer for a subset of the Pascal programming language

By Carson Ellsworth

0976

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

For the second project in building a front-end compiler, I was assigned to building a syntax analyzer, referred to from on out as SA, for the pascal subset language. SA’s are programs that make sure the structuring of a program is correct, whether that structure makes any sense is irreverent to the SA (as we see in project three the Semantic Analyzer handles the meaning of the program). So the first step in creating the SA is first to determine what counts as correct structure, you could just use the grammar given by the book for the subset language which is very neat and human read-able (HR), but when it came time to actually write code for parsing the program, it would be an absolute nightmare, with backtracking happening all the time. Ideally, we would like a grammar structure that given a single token we know exactly what grammar production to use and can then easily determine if the code has any syntax errors. Thankfully we can “massage” grammar structures, so they have the exact same meaning, but come with the benefits of single token production choosing, however, to accomplish this we must trade in the human readability for computational efficiency. This new grammar structure is called an ll(1) grammar, ll(1) grammars are context free grammars that contain no ambiguity, they are scanned in left to right order and constructs a leftmost derivation of the grammar sentence. The massaging operations we can do to transform the grammar to ll(1) are as follows.

1. Remove Ambiguity from Grammar
   1. removing epsilon productions
2. Remove Left Recursion
3. Perform Left Factoring

By doing these three steps we can turn our pascal grammar into an ll(1) grammar. The act of transforming our grammar does increase the overall size of the grammar but keeps the meaning intact and provides us with the benefits already mentioned.

**Part 2) Methodology:**

To massage the grammar and transform it to an ll(1) grammar we do the steps in the order they are listed. Firstly, we start by identifying the null variables, that is, the productions that have epsilon transitions in them initially, for instance,

<arguments> -> **(** <parameter\_list> **)** | e

because <arguments> can be epsilon that makes it a null variable and a point of ambiguity, therefore we need to remove epsilon from the production.

This process is done by finding all the places in the grammar where the null variables are present and then taking them out and creating two alternate productions accordingly. For example,

<subprogram\_head> -> **function id** <arguments> **:** <std\_type> **;**

should be made into  
 <subprogram\_head> -> **function id** <arguments> **:** <std\_type> **;** | **function id** **:** <std\_type> **;**

by doing this we can preemptively remove the epsilon production and the ambiguity along with it.

Next we want to remove left recursion, to do this we first find all the productions that contain left recursion within them, for example,

<identifier\_list> -> <identifier\_list>**,** **id** | **id**

this production obviously has left recursion as we can see because it can trap itself in an infinite loop of <identifier\_list> and has no guarantee of exiting the recursion with a unique sentence. To massage this, we first label the parts of the production, the alpha and beta parts, the alpha parts are the production possibilities that contain left recursions, but do not include the left recursive production call

<identifier\_list> -> <identifier\_list>**,** **id**

and the beta parts are the production possibilities that do not contain left recursion

<identifier\_list> -> **id**

now that we have identified our parts, we can massage the grammar in such a way to remove the left recursion. To remove the left recursion, we create a new production we call <identifier\_list’>, all the alpha parts of the <identifier\_list> are moved to the new <identifier\_list’> with a right recursive <identifier\_list’> production with an epsilon optional production and look like this,

<identifier\_list’> -> **, id** <identifier\_list’> | e

and <identifier\_list> is just the beta sentence with a right recursive <identifier\_list’> production added on

<identifier\_list> -> **id** <identifier\_list’>

For the final steps in creating the ll(1) grammar we need to be able to look at every single token and choose the correct productions based on a single token (also called a handle). To make single handle productions we need to create a new production with sentences that have all the possible combinations, for example

<factor> -> **id (**<expr>**)** | **id [**<expr>**]**

this is a troublesome production due to the fact that if factor is looking at an id token, it does not know what to choose without looking at another token following id, we fix this by massaging the grammar as such

<factor> -> **id** <factor’>

<factor’> -> **(**<expr>**)** | **[**<expr>**]**

now we can choose just the one factor production with **id** and look at the next token in <factor’> and choose the proper production.

Finally we need to compute the first and follows of the grammar to generate recognize the handles (firsts) and assist in error recovery (follows). To recognize the handles of each production we need to look at the first terminal token of the production, if a production has another production as its first, then use the sub productions terminal tokens as the firsts.

first of <expr>: **+ - id num not**

<expr\_list> production : <expr\_list> -> <expr> <expr\_list’>

So the first for <expr > are the firsts for <expr\_list>

follows allow us to find the next correct production in the source code to correctly keep on parsing the file, this means our error recovery method will just be a sort of panic and spit out till a possible recovery location is found and let the programmer deal with their issues.

consider the production for <compound\_statement>

<compound\_statement> -> **begin end** | **begin** <optional\_statement> **end**

here we can see that one of the possible follows for the <optional\_statement> production is the terminal lexeme **end** because it follows the completion of the possible sentences generated by <optional\_statement>. So we can assume we should start parsing after we have reached a valid follow or end of file.

By performing these massages to the grammar we can turn our rinky dinky grammar to a robust and efficient ll(1) grammar.

**Part 3) Implementation:**

For the implementation of the syntax parser, we are going to make a Recursive Descent Parser (RDP). The reason we choose an RDP is because it is the easiest parser to create based on an ll(1) grammar. To transform this grammar to C code I needed to first realize that each unique production is its own function, the different possible sentences for each production can be represented by a switch case statement based on the firsts of each production and the sentences that succeed it. Because we do not live in a perfect world, we are also going to have to consider the possibility of errors in the syntax of the source code and the recovery needed to keep parsing the code after the error. To accomplish the ideas set in part 2 we will create a while loop that looks for any of the correct follows associated with the production that created the error and keep grabbing new tokens till one is attained, at which point the method will return.

**Part 4) Discussion and Conclusions:**

Creating the RDP was a fun and challenging project, the translation of all my paperwork to functioning code was a satisfying accomplishment and really shows the differences between abstract ideas and actual implementation of the abstractions.

While the error handling first appears quite simple (and it is) it keeps the programmer responsible for the program and its meaning, so the compiler does not have to worry about changing the true meaning of any source program given.

**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;

function fun1(x:integer; y:real;

z:array [1..2] of integer;

q: real) : integer;

var d: integer;

begin

a:= 2;

z[a] := 4;

c[2] := 3;

fun1 := c[1]

end;

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

var e: real;

function fun3(n: integer; z: real) : integer;

var e: integer;

begin

a:= e;

e:= c[e];

fun3 := 3

end;

begin

a:= fun1(x, e, c, b);

x:= fun3(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;

fun2 := 2.5

end;

begin

b:= fun2(c[4], c[5]);

b:= fun2(c[4],2);

if (a < 2) then a:= 1 else a := a + 2;

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

end.

Syntax Errors

program test (input, output);

var a : integer;

var b : real;

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

function fun1(x:integer; y:real;

z:array [1..2] of integer;

q: real) : integer;

var d: integer;

begin

a:= 2;

z[a] := 4;

c[2] := 3;

fun1 := c[1]

end;

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

var e: real;

function fun3(n: ; z: real) : integer;

var e: integer;

begin

a:= e;

e:= c[e];

fun3 3

end;

begin

a:= fun1(x, e, c, b);

x:= fun3(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;

fun2 := 2.5

end;

b:= fun2(c[4], c[5]);

b:= fun2(c[4],2);

if (a < 2) then a:= 1 else a := a + 2;

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

end.

result

1 program test (input, output);

2 var a : integer;

3 var b : real;

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

5

6 function fun1(x:integer; y:real;

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

8 q: real) : integer;

9 var d: integer;

10 begin

11 a:= 2;

12 z[a] := 4;

13 c[2] := 3;

14 fun1 := c[1]

15 end;

16

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

18 var e: real;

19

20 function fun3(n: ; z: real) : integer;

tok mismatch expecting integer, real, array, instead recieved ;

21 var e: integer;

22 begin

23 a:= e;

24 e:= c[e];

25 fun3 3

tok mismatch expecting :=, [, instead recieved 3

26 end;

27

28 begin

29 a:= fun1(x, e, c, b);

30 x:= fun3(c[1], e);

31 e := e + 4.44;

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

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

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

35 begin

36 a:= c[a] + 1

37 end;

38 fun2 := 2.5

39 end;

40

41

42 b:= fun2(c[4], c[5]);

tok mismatch expecting function, begin, instead recieved b

43 b:= fun2(c[4],2);

44 if (a < 2) then a:= 1 else a := a + 2;

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

46 end.

**Appendix II: Program Listing:**

#ifndef PARSER\_H

#define PARSER\_H

#include <stdio.h>

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

#include "../lexeme.c"

struct Lexeme tok;

FILE \*token;

FILE \*list;

FILE \*file;

void decs();

void std\_type();

void sub\_decs();

void type();

void cmpd\_stmt();

void stmt();

void expr\_lst();

void expr();

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

//int t = lex.tkn;

int type;

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

if(t == tok.tkn){

if(t != EOF && val == 0){

if(tok.tkn == ID ){

type = tok.type;

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

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

return type; //all good here

}

else if(tok.tkn == NUM){

type = tok.type;

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

return type;

}

else{

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

return 0; //all good here

}

}

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

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

return 0;

}

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

printf("Sync Error!! Expecting token %d attr %d instead recieved token %s \n",t,val,tok.word);

//THIS ALSO NEEDS A PRINTOUT TO THE LISTING FILE

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

}

else{

exit(0);

//end of parse

//stop parse here some how

}

}//end t == tok

else{

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

//THIS ALSO NEEDS A PRINTOUT TO THE LISTING FILE

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

}

return -1;

}

void sign(){

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,"tok mismatch expecting %s, %s, instead recieved %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);

}

}

}

void factorT(){

switch (tok.tkn) {

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

break;

case PAREN:

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

tok\_match(PAREN,PAREN\_OPEN); expr\_lst(); tok\_match(PAREN,PAREN\_CLOSE);

break;

}

case BRACK:

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

tok\_match(BRACK,BRACK\_OPEN); expr(); tok\_match(BRACK,BRACK\_CLOSE);

break;

}

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

default:

fprintf(list,"tok mismatch expecting %s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s, instead recieved %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);

}

}

}

void factor(){

switch (tok.tkn) {

case ID:

tok\_match(ID,0); factorT();

break;

case NUM:

tok\_match(NUM,0);

break;

case NOT:

tok\_match(NOT,0); factorT();

break;

case PAREN:

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

tok\_match(PAREN,PAREN\_OPEN); expr(); tok\_match(PAREN,PAREN\_CLOSE);

break;

}

default:

fprintf(list,"tok mismatch expecting %s,%s,%s,%s, instead recieved %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);

}

}

}

void termT(){

switch (tok.tkn) {

case MULOP:

tok\_match(MULOP,0); factor(); termT();

break;

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

break;

case BRACK: case PAREN:

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

break;

}

default:

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

}

}

}

void term(){

switch (tok.tkn) {

case ID: case NUM: case NOT:

factor(); termT();

break;

case PAREN:

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

factor(); termT();

break;

}

default:

fprintf(list,"tok mismatch expecting %s,%s,%s,%s, instead recieved %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);

}

}

}

void smpl\_exprT(){

switch (tok.tkn) {

case ADDOP:

tok\_match(ADDOP,0); term(); smpl\_exprT();

break;

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

break;

case BRACK: case PAREN:

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

default:

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

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

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

}

}

}

void smpl\_expr(){

switch (tok.tkn) {

case ID: case NUM: case NOT:

term(); smpl\_exprT();

break;

case ADDOP:

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

sign(); term(); smpl\_exprT();

break;

}

case PAREN:

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

term(); smpl\_exprT();

break;

}

default:

fprintf(list,"tok mismatch expecting %s,%s,%s,%s,%s, instead recieved %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);

}

}

}

void exprT(){

switch (tok.tkn) {

case RELOP:

tok\_match(RELOP,0); smpl\_expr();

break;

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

break;

case BRACK:

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

case PAREN:

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

default:

fprintf(list,"tok mismatch expecting %s, instead recieved %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);

}

}

}

void expr(){

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

switch (tok.tkn) {

case NUM: case ID: case NOT:

smpl\_expr(); exprT();

break;

case ADDOP:

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

smpl\_expr(); exprT();

break;

}

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

smpl\_expr(); exprT();

break;

}

case PAREN:

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

smpl\_expr(); exprT();

break;

}

default:

fprintf(list,"tok mismatch expecting %s, %s, %s, %s, %s, %s, instead recieved %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);

}

}

}

void expr\_lstT(){

switch (tok.tkn) {

case COMMA:

tok\_match(COMMA,0); expr(); expr\_lstT();

break;

case PAREN:

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

break;

}

default:

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

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

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

}

}

}

void expr\_lst(){

switch (tok.tkn) {

case NUM: case ID: case NOT:

expr(); expr\_lstT();

break;

case ADDOP:

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

expr(); expr\_lstT();

break;

}

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

expr(); expr\_lstT();

break;

}

case PAREN:

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

expr(); expr\_lstT();

break;

}

default:

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

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

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

}

}

}

void varT(){

switch (tok.tkn) {

case ASSIGNOP:

break;

case BRACK:

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

tok\_match(BRACK,BRACK\_OPEN); expr(); tok\_match(BRACK,BRACK\_CLOSE);

break;

}

default:

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

while (tok.tkn != ASSIGNOP) {

//this is the cause of the infinite loop that happens when procedure statements

//are tried against this grammar or other wacky sentences that begin with an identity

printf("Possible procedure statement encountered\n");

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

if (tok.tkn == EOF){

exit(0);

}

}

}

}

void var(){

switch (tok.tkn) {

case ID:

tok\_match(ID,0); varT();

break;

default:

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

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

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

}

}

}

void stmtT(){

switch (tok.tkn) {

case ELSE:

tok\_match(ELSE,0); stmt();

break;

case SEMICOLON: case END:

break;

default:

fprintf(list,"tok mismatch expecting %s, %s, instead recieved %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);

}

}

}

void stmt(){

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

switch (tok.tkn) {

case ID:

var(); tok\_match(ASSIGNOP,0); expr();

break;

case BEGIN:

cmpd\_stmt();

break;

case IF:

tok\_match(IF,0); expr(); tok\_match(THEN,0); stmt(); stmtT();

break;

case WHILE:

tok\_match(WHILE,0); expr(); tok\_match(DO,0); stmt();

break;

default:

fprintf(list,"tok mismatch expecting %s,%s,%s,%s instead recieved %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);

}

}

}

void stmt\_lstT(){

switch (tok.tkn) {

case SEMICOLON:

tok\_match(SEMICOLON,0); stmt(); stmt\_lstT();

break;

case END:

break;

default:

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

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

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

}

}

}

void stmt\_lst(){

switch (tok.tkn) {

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

stmt(); stmt\_lstT();

break;

default:

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

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

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

}

}

}

void opt\_stmt(){

switch (tok.tkn) {

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

stmt\_lst();

break;

default:

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

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

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

}

}

}

void cmpd\_stmtT(){

switch (tok.tkn) {

case END:

tok\_match(END,0);

break;

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

opt\_stmt(); tok\_match(END,0);

break;

default:

fprintf(list,"tok mismatch expecting %s,%s,%s,%s,%s instead recieved %s\n","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);

}

}

}

void cmpd\_stmt(){

switch (tok.tkn) {

case BEGIN:

tok\_match(BEGIN,0); cmpd\_stmtT();

break;

default:

fprintf(list,"tok mismatch expecting %s, instead recieved %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);

}

}

}

void param\_lstT(){

switch (tok.tkn) {

case SEMICOLON:

tok\_match(SEMICOLON,0); tok\_match(ID,0); tok\_match(COLON,0); type(); param\_lstT();

break;

case PAREN:

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

break;

}

default:

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

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

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

}

}

}

void param\_lst(){

switch (tok.tkn) {

case ID:

tok\_match(ID,0); tok\_match(COLON,0); type(); param\_lstT();

break;

default:

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

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

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

}

}

}

void args(){

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,"tok mismatch expecting %s, instead recieved %s\n","(",tok.word);

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

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

}

}

}

void sub\_headT(){

switch (tok.tkn) {

case COLON:

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

break;

case PAREN:

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

args(); tok\_match(COLON,0); std\_type(); tok\_match(SEMICOLON,0); break;

}

default:

fprintf(list,"tok mismatch expecting %s, %s, instead recieved %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);

}

}

}

void sub\_head(){

switch (tok.tkn) {

case FUNCTION:

tok\_match(FUNCTION,0); tok\_match(ID,0); sub\_headT();

break;

default:

fprintf(list,"tok mismatch expecting %s, instead recieved %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);

}

}

}

void sub\_decTT(){

switch (tok.tkn) {

case FUNCTION:

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

break;

case BEGIN:

cmpd\_stmt();

break;

default:

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

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

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

}

}

}

void sub\_decT(){

switch (tok.tkn) {

case VAR:

decs(); sub\_decTT();

break;

case FUNCTION:

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

break;

case BEGIN:

cmpd\_stmt();

break;

default:

fprintf(list,"tok mismatch expecting %s, %s, %s, instead recieved %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);

}

}

}

void sub\_dec(){

switch (tok.tkn) {

case FUNCTION:

sub\_head(); sub\_decT();

break;

default:

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

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

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

}

}

}

void sub\_decsT(){

switch (tok.tkn) {

case FUNCTION:

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

break;

case BEGIN:

break;

default:

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

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

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

}

}

}

void sub\_decs(){

switch (tok.tkn) {

case FUNCTION:

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

break;

default:

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

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

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

}

}

}

void std\_type(){

char syncSet[] = {';'};

switch(tok.tkn){

case INTEGER:

tok\_match(INTEGER,0);

break;

case REAL:

tok\_match(REAL,0);

break;

default:

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

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

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

}

}

}

void type(){

switch (tok.tkn) {

case INTEGER: case REAL:

std\_type();

break;

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\_match(NUM,0); tok\_match(DOTDOT,0); tok\_match(NUM,0); tok\_match(BRACK,BRACK\_CLOSE);

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

break;

default:

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

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

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

}

}

}

void decsT(){

switch(tok.tkn){

case VAR:

tok\_match(VAR, 0); tok\_match(ID,0); tok\_match(COLON, 0); type(); tok\_match(SEMICOLON,0); decsT();

break;

case FUNCTION: case BEGIN:

break;

default:

fprintf(list,"tok mismatch expecting %s %s %s instead recieved %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);

}

}

}

void decs(){

switch(tok.tkn){

case VAR:

tok\_match(VAR, 0); tok\_match(ID,0); tok\_match(COLON, 0); type(); tok\_match(SEMICOLON,0); decsT();

break;

default:

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

while(/\*!tok\_match(FUNCTION,0) || !tok\_match(BEGIN,0) ||\*/strcmp(tok.word,"function") != 0 || strcmp(tok.word,"begin") != 0){

//print error message

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

}

}

}

void id\_lstT(){

char syncSet[] = {')'};

switch(tok.tkn){

case COMMA:

//printf("match comma\n");

tok\_match(COMMA,0); tok\_match(ID,0); id\_lstT();

break;

case PAREN:

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

break;

}

default:

//print error message

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

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

//print error message

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

}

}

}

void id\_lst(){

char syncSet[] = {')'};

switch(tok.tkn){

case ID:

//printf("ID\_LST fucnt\n");

tok\_match(ID,0); id\_lstT();

break;

default:

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

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

//print error message

//printf("no match\n");

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

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

}

}

}

void prgmTT(){

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, "tok mismatch expecting %s instead recieved %s\n","function begin",tok.word);

while(tok.tkn != EOF){

//print error message

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

}

}

}

void prgmT(){

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

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

while (tok.tkn != EOF){

//print error message

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

}

}

}

void prgm(){

switch(tok.tkn){

case PROGRAM:

tok\_match(PROGRAM,0); tok\_match(ID,0); tok\_match(PAREN, PAREN\_OPEN); id\_lst(); tok\_match(PAREN,PAREN\_CLOSE); tok\_match(SEMICOLON,0); prgmT();

break;

default: //this will be the otherwise statement

while (tok.tkn != EOF){

//print error message

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

}

}

}

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

file = f;

list = l;

token = t;

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

prgm();

tok\_match(EOF,0);

}

#endif