# An Implementation of a Recursive Descent Parser

Noha Abuaesh

noha.abuaesh@gmail.com

## Introduction

A recursive descent parser is a top-down parser built from a set of mutually-recursive procedures (or a non-recursive equivalent) where each such procedure usually implements one of the production rules of the grammar. Thus the structure of the resulting program closely mirrors that of the grammar it recognizes. [REF1]

## Problem Specification

In this project, I develop a recursive-descent parser and it will handle syntax errors for the C- grammar of [REF2], appendix A.2. Some modifications were made to the grammar. The syntactic rules related to function declaration, local declaration and function call are removed from the grammar. The deleted rules are 6 ,11,14,17,27,28,29. The modified rules are:

1 - *program* · *type-specifier* **ID (** *params* **)** **{***declaration list compound statement***}.**
3- As is but remove *fun-declaration*

5- *type-specifier* · *int* | *float*|**void**

10- compound-stmt · **{***statement*-list**}**

13- *statement* · *assignment-stmt*| *compound-stmt*| *selection-stmt*| *iteration-stmt*

18- assignment-stmt · *var* = *expression*

20- *expression* · *expression relop additive-expression*| *additive-expression*
26- As is but remove *call*

### Error Handling

If a syntax error occurs while parsing, an error message is reported specifying the line number, the look-ahead token that caused the error, and the expected token. Compilation does not terminate at the first syntax error, but rather continues until the end of file is encountered, generating an error at every unmatched token afterwards. Error recovery is not used.

### Output

The output is a text file(specified by the user) showing the parsing details. Syntactic errors appear in cygwin's shell after the parsing is completed.

## Procedure

The figure below shows an outline for the procedure I used to achieve the recursive decent parser.

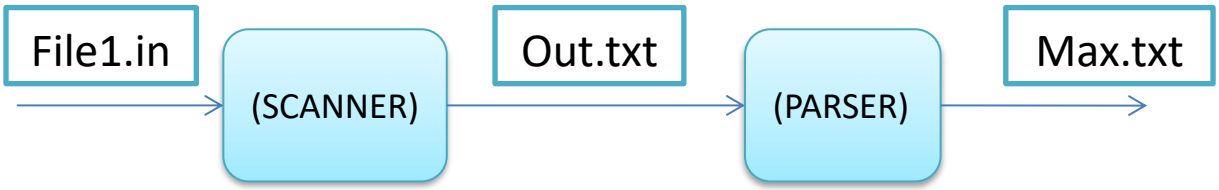



Figure - An outline showing the organization of my recursive descent parser

Below are the steps in detail of how this was done:

1- I used MS Visual Studio to create a C++ program for the recursive descent parser. After ensuring it has no errors, I saved the executable as the parser.exe.

2-To test my program, I scanned a sample C- program(file1.in) in my scanner that I made earlier. The result of scanning is a list of tokens representing the code in file1.in and I called this as out.txt.

3- After that, I passed the scanner's result; output.txt as an input for my recursive descent parser that I wrote in step 1. The result of this step is a text file containing the sequence of the function calls and returns of my parser with the token value displayed at each call and return. This file is called max.txt. If any syntactic errors were discovered while parsing, they are reported directly into cygwin, with the token causing the error and the expected token and the line number.

## Results and Conclusion

After trying my parser on several code samples, I chose to include the example of a program to peform eqlides algorithm to compute gcd(whatever that is); file1.in. This C- program will be parsed twice below; once to discover the errors generated at the first run, and then, a second run to show the successful parsing completed after fixing the errors.

### The First Run of File1.in:

- The following is the contents of the source file file1.in:

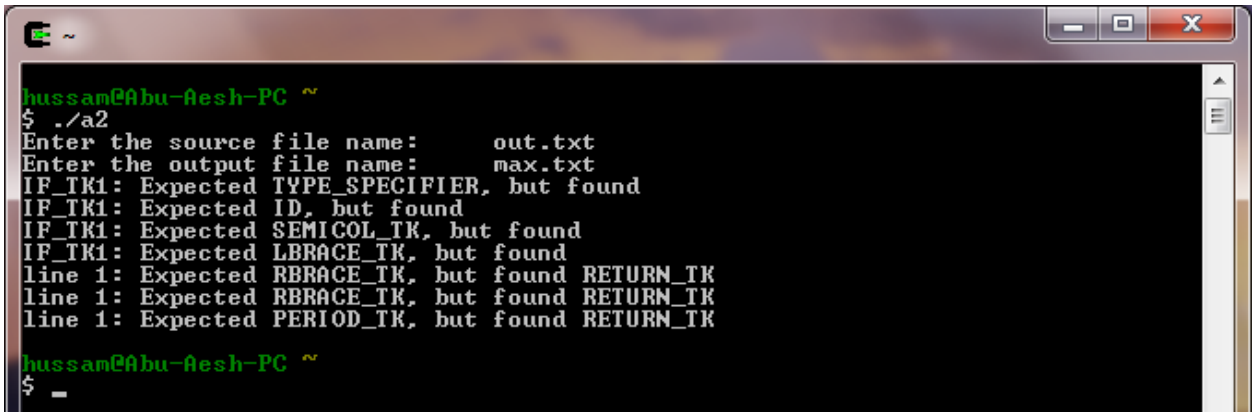
#### File1.in Contents

```
/* a program to peform eqlides algorithm to compute gcd */
int gcd(int u, int v)
{
float x;
{
if (v == 0) return u;
else return gcd(v,u-u/v);
/* u-u/v*v == u mod v */
}}
```

```
void main(void #)
{ int x; int y;
x= input (); y = input90;
output(gcd(x,y));
}
```

- Scanning file1.in , we get:


- Passing the resulting file out.txt into the parser we get:




What happened here is a compound problem that I haven't been lucky with for 2 days ☹ Cygwin appears to be unstable with stderr messaging because what it does here is that it returns the feed back to the beginning of the line before printing the fount token, which overwrites the line number. This is the first part of the problem. The second part is, it always shows the line number as 1, which is obvious in the console at the top. For some reason, the variable "line" remains the same after it is initialized, despite the code I wrote to update it whenever a newline is encountered during scanning. Below is an excerpt of the scan() function responsible for keeping track of the line number, which seems to be never accessed:

```
if(c == '\n'){
        fprintf(outfile, "\n\nNEWLINE SEEN!\n\n");
        line++;
        continue;
}
```

As a result of this un-recognition of new lines in my scan(), '\n' becomes part of the scanned token and is probably what's causing the line return in the error reports back in the cygwin shell.

Anyway, going back to the results, we see an error occurring because an IF token is encountered instead of a semicolon. Because the error report is not giving enough detail about the error position, I made the parsing file show where exactly in the parsing process did the mismatching happen.

Taking a look at the resulting file, we find the following:

### Excerpt of max.txt:

```
PROGRAM:Enter        Next =
TYPE_SPECIFIER
params:Enter         Next = TYPE_SPECIFIER
param_list:Enter             Next = TYPE_SPECIFIER
param:Enter          Next = TYPE_SPECIFIER
param:Leave   Next = COMMA_TK
param:Enter          Next = TYPE_SPECIFIER
param:Leave   Next = RPARA_TK
param_list:Leave     Next = RPARA_TK
params:Leave  Next = RPARA_TK
declaration_list:Enter             Next =
IF_TK
declaration:Enter          Next =
IF_TK
var_declaration:Enter              Next =
IF_TK
PROBLEM: does not match :(
       Next =
IF_TK
PROBLEM: does not match :(
       Next =
IF_TK
…
```

The bottom line, production rule 1(defining program) forces a non-empty list of declarations before starting the compound statement. I fixed this error by adding a simple declaration statement before the if statement to conform with rule 1.

Similarly, other grammar violations were detected in file1.in and were fixed. The main clashes with the grammar causing errors were:

- Return statements- eliminated as rule 17 was deleted per the statement of the assignment.
- Function calls and declarations- eliminated as well because their rules were deleted(rule 6 and rule 27)
- Statements in the program are grouped as 1 compound statement as modified in rule 1.
- Semicolons do not follow statements, only declarations have semicolons at the end.

The final version of file1.in, renamed to file2.in, that does not generate any errors is shown below:

### File2.in (File1.in after correction):

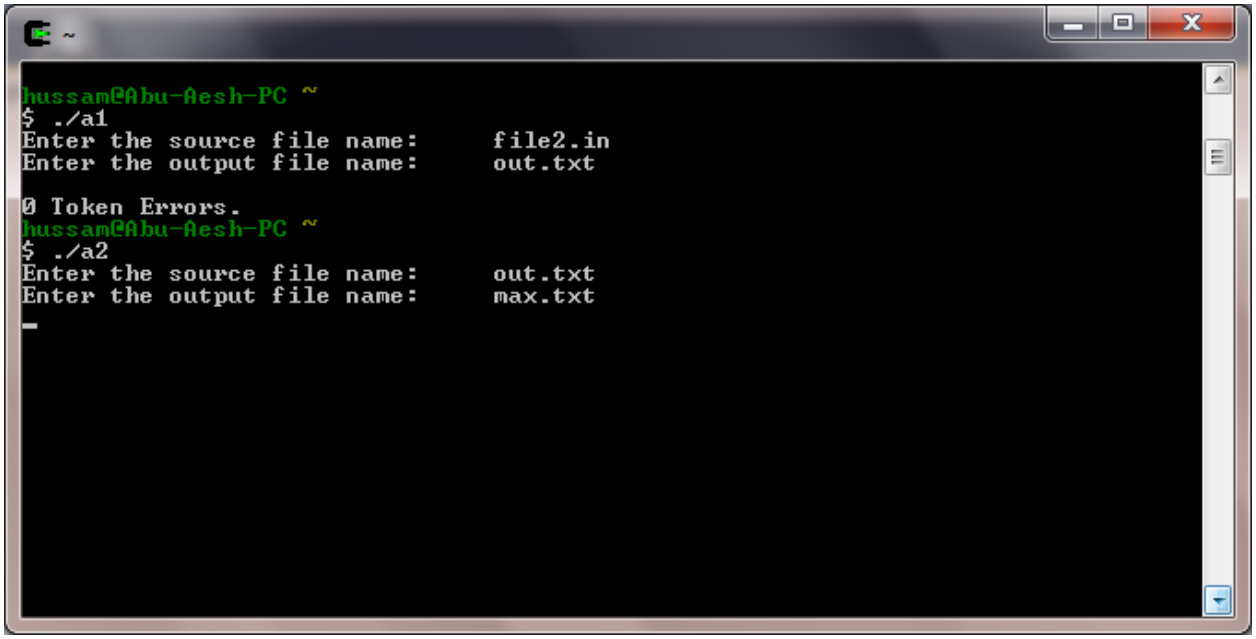
```
/* a program to peform eqlides algorithm to compute gcd */
int gcd(int u, int v)
{
float x;
```

```
{
if (v == 0) x = u
else x = v
/* u-u/v*v == u mod v */
}}
```

### The resulting max.txt(will be pretty long but no mismatches ☺):

```
PROGRAM:Enter        Next =
TYPE_SPECIFIER
params:Enter         Next = TYPE_SPECIFIER
param_list:Enter             Next = TYPE_SPECIFIER
param:Enter          Next = TYPE_SPECIFIER
param:Leave   Next = COMMA_TK
param:Enter          Next = TYPE_SPECIFIER
param:Leave   Next = RPARA_TK
param_list:Leave     Next = RPARA_TK
params:Leave  Next = RPARA_TK
declaration_list:Enter               Next =
TYPE_SPECIFIER
declaration:Enter            Next =
TYPE_SPECIFIER
var_declaration:Enter                Next =
TYPE_SPECIFIER
var_declaration:Leave        Next =
LBRACE_TK
declaration:Leave    Next =
LBRACE_TK
declaration_list:Leave       Next =
LBRACE_TK
compound_stmt:Enter          Next =
LBRACE_TK
statement_list:Enter         Next =
IF_TK
statement:Enter              Next =
IF_TK
selection_stmt:Enter         Next =
IF_TK
expression:Enter             Next = ID
additive_expression:Enter            Next = ID
term:Enter           Next = ID
factor:Enter         Next = ID
var:Enter            Next = ID
var:Leave            Next = RELOP
factor:Leave         Next = RELOP
term:Leave           Next = RELOP
additive_expression:Leave            Next = RELOP
additive_expression:Enter            Next = NUM
term:Enter           Next = NUM
factor:Enter         Next = NUM
factor:Leave         Next = RPARA_TK
term:Leave           Next = RPARA_TK
additive_expression:Leave            Next = RPARA_TK
expression:Leave             Next = RPARA_TK
statement:Enter              Next = ID
assignment_stmt:Enter                Next = ID
var:Enter            Next = ID
var:Leave            Next = ASGNOP
expression:Enter             Next = ID
```

```
additive_expression:Enter          Next = ID
term:Enter           Next = ID
factor:Enter         Next = ID
var:Enter            Next = ID
var:Leave            Next = SEMICOL_TK
factor:Leave         Next = SEMICOL_TK
term:Leave           Next = SEMICOL_TK
additive_expression:Leave          Next = SEMICOL_TK
expression:Leave           Next = SEMICOL_TK
assignment_stmt:Leave              Next = SEMICOL_TK
statement:Leave      Next = SEMICOL_TK
selection_stmt:Leave Next = SEMICOL_TK
statement:Leave      Next = SEMICOL_TK
statement_list:Leave Next = SEMICOL_TK
PROBLEM: does not match :(
       Next = SEMICOL_TK
compound_stmt:Leave  Next = SEMICOL_TK
PROBLEM: does not match :(
       Next = SEMICOL_TK
PROBLEM: does not match :(
       Next = SEMICOL_TK
PROGRAM:Leave Next = SEMICOL_TK
```




As seen above, the file file2.in was parsed successfully without producing errors ☺

## References


[REF1] Wikipedia Definition of Recursive Descent Parser: http://en.wikipedia.org/wiki/Recursive_descent_parser

[REF2] Kenneth Louden, Compiler Construction: Principles and Practice, Course Technology.

# Appendix A

## A2.c

```
#include <stdio.h>
//#include <stdlib.h>
#include <string.h>
//#include<string>
//#include "scan.h"
/*global vars:*/
enum TokenType {ELSE_TK=0, IF_TK, INT_TK, FLOAT_TK, RETURN_TK, VOID_TK, TYPE_SPECIFIER,
WHILE_TK, SEMICOL_TK,
              COMMA_TK, PERIOD_TK, LPARA_TK, RPARA_TK, LBRACKET_TK, RBRACKET_TK,
LBRACE_TK,
              RBRACE_TK, LCOMMENT_TK, RCOMMENT_TK, ADDOP, MULOP, RELOP, ASGNOP, ID, NUM};
char* tokens[]={"ELSE_TK", "IF_TK", "INT_TK", "FLOAT_TK", "RETURN_TK", "VOID_TK",
"TYPE_SPECIFIER", "WHILE_TK", "SEMICOL_TK",
              "COMMA_TK", "PERIOD_TK", "LPARA_TK", "RPARA_TK", "LBRACKET_TK",
"RBRACKET_TK", "LBRACE_TK",
              "RBRACE_TK", "LCOMMENT_TK", "RCOMMENT_TK", "ADDOP", "MULOP", "RELOP",
"ASGNOP", "ID", "NUM"};
char token[20] = "";
int line = 1;
FILE *srcfile;
FILE *outfile;
//int srcfile_marker = 0; //marks the position that scan has reached so far
/*function prototypes*/
void program();
void declaration_list();
void declaration();
void var_declaration();
//void type_specifier();
void params();
void param_list();
void param();
void compound_stmt();
void statement_list();
void statement();
void selection_stmt();
void iteration_stmt();
void assignment_stmt();
void var();
void expression();
//void relop();
void additive_expression();
//void addop();
void term();
//void mulop();
void factor();

void lex_print(char *);
void syn_err(char *, char *);
void match(TokenType);
void scan();
//char* token2string(TokenType);
//TokenType string2token(char*);
//======PROGRAM=============================================================
void program()
```

```
{
	lex_print("PROGRAM:Enter\t");
//{type_specifier}{blank_str}{identifier}"("{params}")"{blank_str}"{"{declaration_list}{b
lank_str}{compound_stmt}"}""."
match(TYPE_SPECIFIER);
//match(BLANK_STR);
match(ID);
match(LPARA_TK);
params();
match(RPARA_TK);
//match(BLANK_STR);
match(LBRACE_TK);
declaration_list();
//match(BLANK_STR);
compound_stmt();
match(RBRACE_TK);
match(PERIOD_TK);
lex_print("PROGRAM:Leave");
}
//======DECLARATION_LIST=================================================================
void declaration_list()
{
	lex_print("declaration_list:Enter\t");
//({declaration_list}{blank_str}{declaration})|{declaration}
declaration();
while(strstr(token, tokens[TYPE_SPECIFIER]) != 0){
	//match(BLANK_STR);
	declaration();
	}
lex_print("declaration_list:Leave");
}
//======DECLARATION======================================================================
void declaration()
{
	lex_print("declaration:Enter\t");
//{var_declaration}
var_declaration();
lex_print("declaration:Leave");
}
//======VAR_DECLARATION==================================================================
void var_declaration()
{
	lex_print("var_declaration:Enter\t");
//{type_specifier}{blank_str}{identifier}";"|{type_specifier}{blank_str}{identifier}"["nu
mber"]";
match(TYPE_SPECIFIER);
match(ID);
if(strstr(token, tokens[LBRACKET_TK]) != '\0'){
	match(LBRACKET_TK);
	match(NUM);
	match(RBRACKET_TK);
	}
match(SEMICOL_TK);
lex_print("var_declaration:Leave");
}
/*//======TYPE_SPECIFIER===================================================================
void type_specifier()
{
```

```
//[iI][nN][tT]|[fF][lL][oO][aA][tT]|[vV][oO][iI][dD]
case token is
"i","I":      match(INT_TK);
"f","F":      match(FLOAT_TK);
"v","V":      match(VOID_TK);
others:              syn_err("a type specifier: int, float or void", token);
}*/
//======PARAMS=========================================================================
void params()
{
       lex_print("params:Enter\t");
//{param_list}|[vV][oO][iI][dD]
/*case token is
TYPE_SPECIFIER:            param_list();
"v","V":            match(VOID);
others:                    syn_err("parameters list or void", token);
*/
param_list();
lex_print("params:Leave");
}
//======PARAM_LIST=====================================================================
void param_list()
{
       lex_print("param_list:Enter\t");
//{param_list}","{param}|{param}
//This means it accepts for example a param list like: void x, void, int y
param();
while(strstr(token, tokens[COMMA_TK]) != '\0'){
       match(COMMA_TK);
       param();
       }
lex_print("param_list:Leave");
}
//======PARAM=========================================================================
void param()
{
       lex_print("param:Enter\t");
//{type_specifier}{blank_str}{identifier}|{type_specifier}{blank_str}{identifier}"[""]"
match(TYPE_SPECIFIER);
//match(BLANK_STR);
match(ID);
if(strstr(token, tokens[LBRACKET_TK]) != '\0'){
       match(LBRACKET_TK);
       match(RBRACKET_TK);
       }
lex_print("param:Leave");
}
//======COMPOUND_STMT=================================================================
void compound_stmt()
{
       lex_print("compound_stmt:Enter\t");
//"{"{statement_list}"}"
match(LBRACE_TK);
statement_list();
match(RBRACE_TK);
lex_print("compound_stmt:Leave");
}
//======STATEMENT_LIST================================================================
```

```
void statement_list()
{
	lex_print("statement_list:Enter\t");
//{statement_list} {statement}|""
//if(token == ID || token == LBRACE_TK || token == IF_TK || token == WHILE_TK)
statement(); //put in mind that a statement can be nothing, which grants 0+ statements in
a statement list
lex_print("statement_list:Leave");
}
//======STATEMENT=====================================================================
void statement()
{
	lex_print("statement:Enter\t");
//{assignment_stmt}|{compound_stmt}|{selection_stmt}|{iteration_stmt} or nothing.
/*switch(token){
	case ID://assignment statement
		assignment_stmt();
	case LBRACE_TK://compound statement
		compound_stmt();
	case IF_TK://selection statement
		selection_stmt();
	case WHILE_TK://iteration statement
		iteration_stmt();
	default://null statement to fix the nullable rule for statement_list, ignore
		return; //ignore
}*/
	if(strstr(token, tokens[ID]) != '\0')	assignment_stmt();	else{
		if(strstr(token, tokens[LBRACE_TK]) != '\0')	compound_stmt(); else{
			if(strstr(token, tokens[IF_TK]) != '\0') selection_stmt(); else{
				if(strstr(token, tokens[WHILE_TK]) != '\0')
	iteration_stmt(); //else return;
			}
		}
	}

lex_print("statement:Leave");
}
//======SELECTION_STMT=================================================================
void selection_stmt()
{
	lex_print("selection_stmt:Enter\t");
//[iI][fF]{blank_str}"("{expression}")"{statement}|[iI][fF]{blank_str}"("{expression}")"{
statement}{blank_str}[eE][lL][sS][eE]{blank_str}{statement}
match(IF_TK);
match(LPARA_TK);
expression();
match(RPARA_TK);
statement();
if(strstr(token, tokens[ELSE_TK]) != '\0'){
	match(ELSE_TK);
	statement();
	}
lex_print("selection_stmt:Leave");
}
//======ITERATION_STMT=================================================================
void iteration_stmt()
{
	lex_print("iteration_stmt:Enter\t");
```

```
//[wW][hH][iI][lL][eE]{blank_str}"("{expression}")"{statement}
match(WHILE_TK);
match(LPARA_TK);
expression();
match(RPARA_TK);
statement();
lex_print("iteration_stmt:Leave\t");
}
//======ASSIGNMENT_STMT=================================================================
void assignment_stmt()
{
	lex_print("assignment_stmt:Enter\t");
//{var}"="{expression}
var();
match(ASGNOP);
expression();
lex_print("assignment_stmt:Leave\t");
}
//======VAR=============================================================================
void var()
{
	lex_print("var:Enter\t");
//{identifier}|{identifier}"["{expression}"]"
match(ID);
if(strstr(token, tokens[LBRACKET_TK]) != '\0'){
	match(LBRACKET_TK);
	expression();
	match(RBRACKET_TK);
	}
lex_print("var:Leave\t");
}
//======EXPRESSION======================================================================
void expression()
{
	lex_print("expression:Enter\t");
//{expression}{relop}{additive_expression}|{additive_expression}
additive_expression();
while(strstr(token, tokens[RELOP]) != '\0'){
	match(RELOP);
	additive_expression();
	}
lex_print("expression:Leave\t");
}
//======RELOP===========================================================================
/*void relop()
{
//"<="|"<"|">="|">"|"=="|"!="
}*/
//======ADDITIVE_EXPRESSION=============================================================
void additive_expression()
{
	lex_print("additive_expression:Enter\t");
//{additive_expression}{addop}{term}|{term}
term();
while(strstr(token, tokens[ADDOP]) != '\0'){
	match(ADDOP);
	term();
	}
```

```
lex_print("additive_expression:Leave\t");
}
//======ADDOP=============================================================================
/*void addop()
{
//"+"|"-"
}*/
//======TERM=============================================================================
void term()
{
	lex_print("term:Enter\t");
//{term}{mulop}{factor}|{factor}
factor();
while(strstr(token, tokens[MULOP]) != '\0'){
	match(MULOP);
	factor();
	}
lex_print("term:Leave\t");
}
//======MULOP=============================================================================
/*void mulop()
{
//"*"|"/"
}*/
//======FACTOR=============================================================================
void factor()
{
	lex_print("factor:Enter\t");
//"("{expression}")"|{var}|{number}
/*switch(token){
	case LPARA_TK:
		{match(LPARA_TK);
		expression();
		match(RPARA_TK);
		}
	case ID:		var();
	case NUM:		match(NUM);
	default:	syn_err("left paranthesis, variable or a number", token);
	}*/
	if(strstr(token, tokens[LPARA_TK]) != '\0'){
		match(LPARA_TK);
		expression();
		match(RPARA_TK);
	} else{
		if(strstr(token, tokens[ID]) != '\0') var(); else{
			if(strstr(token, tokens[NUM]) != '\0') match(NUM); else
				syn_err("left paranthesis, variable or a number", token);
		}
	}
	lex_print("factor:Leave\t");
}
//=========================================================================================
//======MATCH=============================================================================
//=========================================================================================
void match(TokenType expected_token)
{
	//fprintf(stderr, "matching %s...  ", tokens[expected_token]);
if(strstr(token, tokens[expected_token]) != '\0'){
```

```
	//lex_print("matched successfully :)\n");
	//strcpy(token,scan());
	scan();
}
else{
	lex_print("PROBLEM: does not match :(\n");
	//syntax error
	syn_err(tokens[expected_token], token);
}
if(strcmp(token, "") == 0)  //end of file reached--useless
	return;
}
//=============================================================================
void scan(){
	//returns next token, keeps track and increments the line number,
	//remembers the position it has reached reading since last  call(s)
	char t[50] = "";
	char c;
	int i = 0;

	//for(int i = 0; i <= srcfile_bookmark; i++) fscanf(srcfile, "%c", &c);

	//fscanf(srcfile, "%c", &c);
	//fprintf(outfile, "\nJust scanned: %c\n\n", c);
	//convert c from character into string
	//string str; str.append(1, c);
	//concat the result to t
	//append(t, c);
	//srcfile_bookmark++;
		while(c != EOF){
			fscanf(srcfile, "%c", &c);
			//fprintf(outfile, "\nJust scanned: %c\n\n", c);

			if(c == '\n'){
				fprintf(outfile, "\n\nNEWLINE SEEN!\n\n");
				line++;
				continue;
			}
			if(c == ' ') {
				//fprintf(outfile, "\n\nSPACE SEEN!\n\n");
				if(i == 0)
					continue;
				else break;
			}
			//append c to t
			//t = t + c;
			t[i] = c;
			i++;
			//srcfile_bookmark++;
		}
		t[i] = '\0';
		//fprintf(outfile, "\n\nt is now = %s\n", t);
		strcpy(token, t);
		//fprintf(outfile, "FROM SCAN: token is now = %s\nline %d", token, line);
	//return t;
}
//=============================================================================
char* token2string(TokenType t)
```

```
{
	/*enum TokenType {ELSE_TK=300, IF_TK, INT_TK, FLOAT_TK, RETURN_TK, VOID_TK,
TYPE_SPECIFIER, WHILE_TK, SEMICOL_TK,
		COMMA_TK, PERIOD_TK, LPARA_TK, RPARA_TK, LBRACKET_TK, RBRACKET_TK,
LBRACE_TK,
		RBRACE_TK, LCOMMENT_TK, RCOMMENT_TK, ADDOP, MULOP, RELOP, ASGNOP, ID,
NUM};*/
	char* map[] = {"else", "if", "int", "float", "return", "void", "type specifier",
		"while", ";", ",", ".", "(", ")", "[", "]", "{", "}", "/*", "*/", "+ or -",
		"* or /", "relational operator", "identifier", "number"};

	return map[t];

}
//=============================================================================
/*TokenType string2token(char* s)
{
	TokenType t;
	switch(s){
	case "ELSE_TK":			t = ELSE_TK;
	case "IF_TK":		t = IF_TK;
	case "TYPE_SPECIFIER":		t = TYPE_SPECIFIER;
	case "RETURN_TK":		t = RETURN_TK;
	case "WHILE_TK":		t = WHILE_TK;
	case "SEMICOL_TK":		t = SEMICOL_TK;
	case "COMMA_TK":		t = COMMA_TK;
	case "PERIOD_TK":		t = PERIOD_TK;
	case "LPARA_TK":		t = LPARA_TK;
	case "RPARA_TK":		t = RPARA_TK;
	case "LBRACKET_TK":		t = LBRACKET_TK;
	case "LBRACE_TK":		t = LBRACE_TK;
	case "RBRACE_TK":		t = RBRACE_TK;
	case "LCOMMENT_TK":		t = LCOMMENT_TK;
	case "RCOMMENT_TK":		t = RCOMMENT_TK;
	case "ADDOP":		t = ADDOP;
	case "MULOP":		t = MULOP;
	case "RELOP":		t = RELOP;
	case "ASGNOP":			t = ASGNOP;
	case "ID":		t = ID;
	case "NUM":		t = NUM;
	//default:
	}

	return t;
}*/
//=============================================================================
void syn_err(char* s1, char* s2){
	fprintf(stderr, "line %d:\tExpected %s, but found %s", line, s1, s2);
	fprintf(stderr, "\n");
	//total_errors++;
}
//=============================================================================
void lex_print(char* str){
	fprintf(outfile, "%s\tNext = %s", str, token);
	fprintf(outfile, "\n");
}
/*//=============================================================================
void lex_err(char* s1, char* s2){
```

```
	fprintf(stderr, "line %d, character %d:\t%s%s\n", line, epos, s1, s2);
	total_errors++;
}
/*int yywrap(){
	return 1;
}*/
//=============================================================================
bool lex_init(char* src, char* out){
	srcfile = fopen(src, "r");
	outfile = fopen(out, "w");
	bool done = true;
	if(srcfile == 0){
		printf("Can't open %s\n", src);
		done = false;
	}else //read the file name
	if(outfile == false){
		printf("Can't write to %s", out);
		done = false;
	}
	return done;
}
//=============================================================================
//=============================================================================
int main(int argc, char **argv) {
	 ++argv; --argc;
	/* if (argc > 0) {
	     srcfile = fopen(argv[0], "r");
	 } else {
	     srcfile = stdin;
	 }*/
	 //char* srcfile;
	 //char* outfile;

	 char f1[50]="";
	 char f2[50]="";

	 printf("Enter the source file name:\t");
	 scanf("%s", f1);
		//gets(srcfile);

	 printf("Enter the output file name:\t");
	 scanf("%s", f2);
	 //gets(outfile);

	 if(lex_init(f1,f2) == true){
			//yylex();
			//strcpy(token,scan());
			scan();
			//fprintf(outfile, "\n\nFROM MAIN: t is now = %s\n", token);
			program();
		}
		else   printf("Cannot initialize files.\n");

		//printf("\n%d Token Errors.", total_errors);
	return 0;
    }
```

## A1.l(minor changes made)

```
%{

#include <stdio.h>

//global static int line = 1;

//global static int position = 1;

//global int error_count =0;

//int errors_list[100][3] ; //type, line , character position

//static bool unclosed_comment_err = false;

const char* errors_catalog[] ={

                "Unclosed comment encountered. Expected end of comment, found end of file.",

                "Nested comments are not allowed.",

                "Undefined symbol. Use only the characters defined by the language.",

                "Wrong identifier. Identifiers must start with a letter and end with a letter or a digit.",

                "Wrong number. Numbers must end with a digit."

                };

static int line = 1;

static int position = 1;

static int error_count =0;

static int error_list[100][3] ;

static bool unclosed_comment_err = false;

%}

%%

[eE][lL][sS][eE]                printf("ELSE");

[iI][fF]                        printf("IF");

[iI][nN][tT]                    printf("INT");

[rR][eE][tT][uU][rR][nN]        printf("RETURN");
```

```
[vV][oO][iI][dD]                printf("VOID");

[wW][hH][iI][lL][eE]            printf("WHILE");

\+                              printf("ADDOP: ADDITION");

\-                              printf("ADDOP: SUBTRACTION");

\*                              printf("MULOP: MULTIPLICATION");

\/                              printf("MULOP: DIVISION");

\<                              printf("RELOP: LESS THAN");

\<=                             printf("RELOP: LESS OR EQUAL");

\>                              printf("RELOP: GREATER THAN");

\>=                             printf("RELOP: GREATER OR EQUAL");

==                              printf("RELOP: EQUAL");

!=                              printf("RELOP: NOT EQUAL");

=                               printf("ASSIGNMENT OPERATOR");

;                               printf("SEMICOLON ");

\,                              printf("COMMA");

\(                              printf("OPEN PARANTHESIS");

\)                              printf("CLOSE PARANTHESIS");

\[                              printf("OPEN BRACKET");

\]                              printf("CLOSE BRACKET");

\{                              printf("OPEN BRACE");

\}                              printf("END BRACE");

"\/\*"([^"\*\/"])*"\*\/"        printf("COMMENT");

"\/\*"[^\*]*"\/\*"([^"\*\/"])*"\*\/"        printf("NESTED COMMENTS");
        error_list[error_count][0]=1;  error_list[error_count][1]=line;
        error_list[error_count][2]=position; error_count++;
```

```
\/\*([^"\*\/"])*            printf("UNCLOSED COMMENT"); printf("UNCLOSED COMMENT");
        error_list[error_count][0]=0;  error_list[error_count][1]=line;
        error_list[error_count][2]=position; error_count++;

\n                                  line++; position=1;

[^\n]                               position ++;

[ \t]+                              {//ignore whitespace};

[^symbol]                           error_list[error_count][0] =  2; error_list[error_count][1]=line;
        error_list[error_count][2] = position; error_count++;

%%

 int main(int argc, char **argv) {

     ++argv; --argc;                 /* skip over program name */

     if (argc > 0) { yyin = fopen(argv[0], "r");  } else { yyin = stdin; }

     yylex();

  }
```