Compiler Design Project
Part One

Lexical Analysis

Project Guided by :

Project done by :

Menezes Elvis Edward (15CO254)
Somnath Krishna Sarkar (15CO247)

Contents…………………………………………………………………………

Compiler Design

The Compiler Design Process is broken down mainly into two parts namely:

1. Front-End
   a) Lexical Analysis
   b) Syntax Analysis
   c) Semantic Analysis

2. Back-End
   a)Code Optimizer
   b)Code Generator

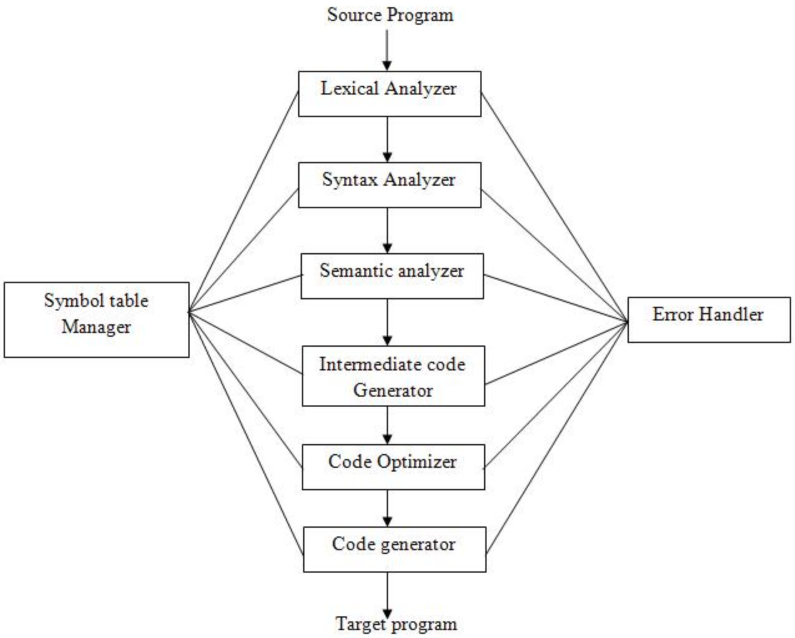


Fig 1.3 Phases of a Compiler

Lexical Analysis:

Lexical Analysis is the first phase of compiler. It is also known as scanner. The input program is converted into a sequence of tokens that can be treated as a unit in the grammar of the programming languages.

Syntax Analysis:

The second phase of the compiler is Syntax Analysis also known as Parsing. The syntactical structure of the given input is checked in this phase, i.e. whether it is the correct syntax. A data structure is built and used in this process called as a Parse tree and also as Syntax tree. The parse tree is constructed by using the predefined Grammar of the language and the input string. If the given input string can be produced with the help of the syntax tree (in the derivation process), the input string is found to be in the correct syntax.

Semantic Analysis:

In this phase the parse tree is verified, if it is meaningful or not and produces a verified parse tree.

Intermediate Code Generator:

A form which can be readily executed by machine is generated in this phase of the compiler. One of the many popular intermediate codes is the Three address code.

Code Optimizer:

In this phase the code is transformed into more efficient code so that it consumes less resources and is executed faster without the meaning of the code not altered. Optimization can be categorized into two types: machine dependent and machine independent.

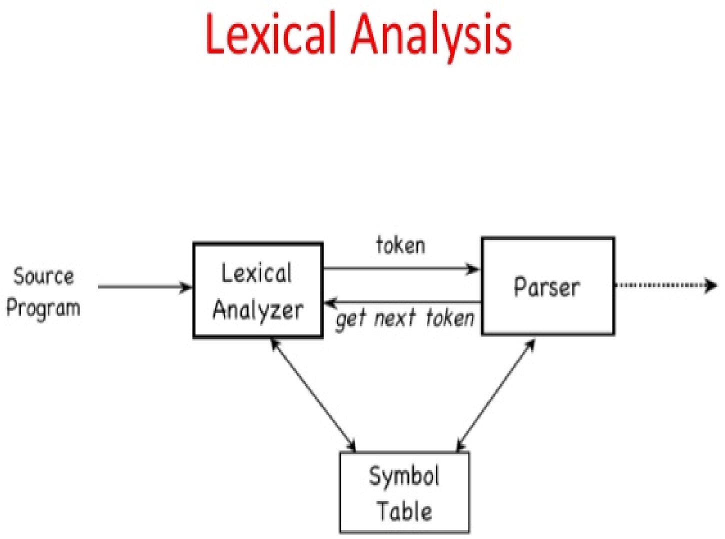
Code Generator:

This is the final stage of the compiler and the main purpose of this stage is write a code that the machine can understand. The output is dependent on the assembler.

This Report mainly focuses on the first phase of the compiler that is the lexical analyzer.

Lexical Analyzer:

Lexical Analysis is the first phase of the compiler also known as the scanner.



The job of the lexical analyzer is to read the characters from the input and group them into "token objects."

The program that performs the above task can be termed as a lexer, tokenizer, or scanner.

Terms generally used in lexical analysis phase:

a) Lexeme: It is a sequence of characters in the source program that matches the pattern for a token and is identified by the analyzer as an instance of that token.
b) Token : It is a pair consisting of a token name and an optional token value.
c) Pattern: It is a description of the form that the lexemes of a token may take. For example, a keyword as a token, the pattern is just the sequence of characters that form the keyword.

The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexemes.

For each lexeme, the lexical analyzer produces as output a token of the form:

(token-name, attribute-value)

For example, suppose a source program contains the assignment statement:

var1 = var2 + var3 * 40

The result for the above after lexical analysis will be:

(identifier, l) (=) (identifier, 2) (+) (identifier, 3) (*) (40)

The parser uses the first components of the tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream.

The lexical analyzer also interacts with the symbol table. Information regarding the kind of identifier may be read from the symbol table by the lexical analyzer to assist it in determining the proper token it must pass to the parser

Lexical Errors:

The lexical analyzer finds all the errors once ran so that the user can make changes accordingly and doesn’t have to run the lex again and again. This results in some disadvantages. When the lexer finds an error, the consumer of the tokens that the lexer produces (e.g., the rest of the compiler) can not usually itself produce a valid result. However, the compiler may try to find other errors in the remaining input, again allowing the user to find several errors in one edit-compile cycle.

The subsequent errors may really be spurious errors caused by lexical error , so the user will have to guess at the validity of every error message except the first, as only the first error message is guaranteed to be a real error.

The lexical analyzer detects an error when it discovers that an input's prefix does not fit the specification of any token class.

Recovery from lexical Errors:

The simplest possible error recovery is to skip the erroneous characters until the lexical analyzer finds another token. This strategy is also called as “panic mode recovery”. This recovery technique may confuse the parser, but in an interactive computing environment it may be quite adequate.

Other possible error-recovery actions are:

1. Delete one character from the remaining input.
2. Insert a missing character into the remaining input.
3. Replace a character by another character.
4. Transpose two adjacent characters.

Functions Covered in the Project:

1. **Scope of the variables** are taken into consideration and accordingly pushed into the symbol table. eg. if variable abc is declared in main and variable with the same name is again declared in a function main.abc and func.abc are pushed considering them different.

2. Errors related to **incorrect variable names** are succesfully identified.

3. **Dangling strings** are successfully identified that is a opening quote for a string literal without a closing quote is considered and given as an error.

4. **Dangling comments** are taken into consideration such as a opening multiline comment /* without a closing */ is given as an error by our lexer.

5. Tokens that cannot be identified and not in the language specifications are also considered and an error for “**Could not identify token**” is given by lexer.

6. **Line number** is outputted for an error detected as to where the error has occurred for the convenience of the user.

7. **Symbol table** is outputted at the end of every successful compilation.

8. **Constants table** is outputted at the end of every successful compilation.

9. The code is converted into the format :
(token-name, attribute-value)

Code:

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
struct hash_node{
        char *str;
        int id;
        struct hash_node* next;
};
const char * keywords[] = {"int","float","char","long","short","double","void","if","else","while","struct","for","switch","case","break","union","return"};
const char * errormsg[] = {"Could not identify token","Dangling Comment","Dangling String Literal"};
const char * literals[] = {"Integer\t","Floating Point","String\t"};
const int MOD = 104729,PRIME = 1009;
struct hash_node* hash_map[2][104729];
char scope_prefix[100],ctmp[100];
char *identifiers[100],*dtypes[100],*constants[100];
int ctypes[100],idline[100],conline[100];
char datatype[100];
int idstack[100],sp,newdef = 0;
int lines = 1;
int kc = 17,idc = 0,opensmall = 0,imp = 0,conc = 0;
int lastdef = 0;
int match,i;
int errors[100],errorlines[100],ec = 0;
int constant_type(char* str){
        if(strstr(str,"\""))
                return 2;
        if(strstr(str,"."))
                return 1;
        return 0;
}
int compute_hash(char* str){
        int len = strlen(str),i,ans = 0;
        for(i = len-1; i>=0; i--){
                ans=(ans*PRIME)%MOD;
                ans+=str[i];
        }
        return ans;
}
int insert_hash(struct hash_node** hash_map, char *s,int lookup, int is_func){
        char *str = s;
        if(newdef||lookup){
                scope_prefix[0] = '\0';
                for(int i = 0; i<sp; i++){
                        strcat(scope_prefix,identifiers[idstack[i]]);
                        strcat(scope_prefix,".");
                }
                strcat(scope_prefix,s);
                str = scope_prefix;
        }
```

```
	int len = strlen(str);
	int hashval = compute_hash(str);
	struct hash_node** p = &hash_map[hashval];
	while(*p){
		if(!strcmp((*p)->str,str))
			return 0-((*p)->id);
		(*p)++;
	}
	if(lookup)
		return insert_hash(hash_map,s,0,is_func);
	(*p) = (struct hash_node*)malloc(sizeof(struct hash_node));
	(*p)->str = (char *)malloc(len+1);
	strcpy((*p)->str,str);
	(*p)->id = idc;
	identifiers[idc] = (*p)->str;
	dtypes[idc] = (char *)malloc(strlen(datatype)+1);
	idline[idc] = lines;
	strcpy(dtypes[idc],datatype);
	if(is_func)
		strcat(dtypes[idc],"()");
	idc++;
	if(!opensmall)
		imp=idc;
	(*p)->next = NULL;
	return (*p)->id;
}
int insert_constant_hash(struct hash_node** hash_map, char *s){
	char *str = s;
	int len = strlen(str);
	int hashval = compute_hash(str);
	struct hash_node** p = &hash_map[hashval];
	while(*p){
		if(!strcmp((*p)->str,str))
			return 0-((*p)->id);
		(*p)++;
	}
	(*p) = (struct hash_node*)malloc(sizeof(struct hash_node));
	(*p)->str = (char *)malloc(len+1);
	strcpy((*p)->str,str);
	(*p)->id = conc;
	constants[conc] = (*p)->str;
	ctypes[conc] = constant_type(str);
	conline[conc] = lines;
	conc++;
	(*p)->next = NULL;
	return (*p)->id;
}
void init(){
	int i,j;
	for(j = 0; j<2; j++)
		for(int i = 0; i<MOD; i++)
			hash_map[j][i] = NULL;
	scope_prefix[0] = '\0';
	sp = 0;
	datatype[0] = '\0';
```

```
}
void push(int x){
        idstack[sp++] = x;
}
int pop(){
        return idstack[--sp];
}
int isempty(){
        return (sp==0);
}
%}
letter [a-zA-Z]
digit [0-9]
comment "//".*
multicomment "/*"([^*]|\*[^/])*"*/"
danglingcomment "/*"
preprocessor "#".*
ws [\t ]+
newline [\n]
datatype ("int"|"float"|"double"|"long"|"short"|"char"|"struct"|"union"|"void")\**
identifier {letter}({digit}|{letter}|[_])*
if_keyword "if"
else_keyword "else"
while_keyword "while"
switch_keyword "switch"|"case"|"break"
for_keyword "for"
return_keyword "return"
keyword {if_keyword}|{else_keyword}|{while_keyword}|{switch_keyword}|{return_keyword}
intliteral {digit}+
floatliteral [+-]?(({digit}+("."{digit}*)?)|("."{digit}+))
stringliteral "\""(\\.|[^\\"])*["]
literal {intliteral}|{floatliteral}|{stringliteral}
operator "+="|"++"|"--"|"-="|"*="|"/="|"^="|"&="|"|="|"=="|"&&"|"||"|"->"|"!="|"<="|">="|
[+\-*/=\^&|<>]
separator [;]
startscope [{(]
endscope [)}]
comma [,]
%Start start_clean start_literal start_identifier start_keyword
%%
{comment} {printf("COMMENT"); BEGIN start_clean;}
{multicomment} {
        printf("MULTICOMMENT");
        for(int i = 0; i<yyleng; i++){
                if(yytext[i]=='\n')
                        lines++;
        }
        BEGIN start_clean;
}
{danglingcomment} {errorlines[ec] = lines; errors[ec++] = 1; BEGIN start_clean;}
{preprocessor} {printf("PREPROCESSOR"); BEGIN start_clean;}
{ws} {ECHO; BEGIN start_clean;}
{newline} {ECHO; lines++; BEGIN start_clean;};
<start_clean>{identifier} {
        int is_func = 0,i,rev=0;
        ctmp[rev++] = input();
        while((ctmp[rev-1]==' '||ctmp[rev-1]=='\t'||ctmp[rev-1]=='\n')){
                ctmp[rev++] = input();
        }
        if(ctmp[rev-1]=='(')
                is_func = 1;
        char *yycopy = (char *)malloc(yyleng+1);
        strcpy(yycopy,yytext);
        for(i=rev-1;i>=0; i--)
                unput(ctmp[i]);
        match = 0;
        for(i = 0; i<kc; i++){
                if(!strcmp(yycopy,keywords[i])){
                        match = 1;
                        break;
                }
        }
        if(match){
                free(yycopy);
                REJECT;
        }else{
                lastdef = insert_hash(hash_map[0],yycopy,1,is_func);
                printf("(identifier,%d)",abs(lastdef));
                newdef = 0;
                free(yycopy);
                BEGIN start_identifier;
        }
}
<start_clean>{literal} {
        printf("(constant,%d)",abs(insert_constant_hash(hash_map[1],yytext)));
        BEGIN start_literal;
}
{operator} {printf("(operator,%s)",yytext); BEGIN start_clean;}
<start_clean>{datatype} {
        printf("(datatype,%s)",yytext);
        BEGIN start_keyword;
        strcpy(datatype,yytext);
        newdef = 1;
}
<start_clean>{keyword} {
        printf("(keyword,%s)",yytext);
        BEGIN start_keyword;
}
{startscope} {
        printf("(separator,%s)",yytext);
        BEGIN start_clean;
        push(imp-1);
        if(yytext[0]=='(')
                opensmall++;
}
{endscope} {
        printf("(separator,%s)",yytext);
        BEGIN start_clean;
        pop();
        if(yytext[0]==')')
```

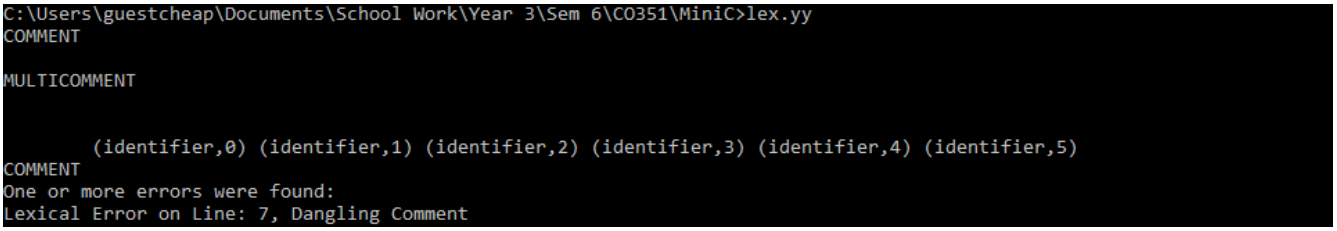
```
                opensmall--;
}
{separator} {
        printf("(separator,%s)",yytext);
        BEGIN start_clean;
        datatype[0] = 0;
        newdef = 0;
}
{comma} {
        printf("(separator,,)");
        BEGIN start_clean;
        if(datatype[0])
                newdef = 1;
}
. {
        if(yytext[0]=='"')
                errors[ec] = 2;
        else
                errors[ec] = 0;
        errorlines[ec++] = lines;
}
%%
void printerrors(){
        for(int i = 0; i<ec; i++)
                printf("Lexical Error on Line: %d, %s\n",errorlines[i],errormsg[errors[i]]);
}
void printidentifiers(){
        printf("\t\tIdentifier No.\tIdentifier Name\tIdentifier Type\tLine\n");
        for(int i = 0; i<idc; i++)
                printf("\t\t%d\t\t%s\t\t%s\t\t%d\n",i,identifiers[i],dtypes[i],idline[i]);
}
void printconstants(){
        printf("\t\tConstant No.\tConstant Value\tConstant Type\t\tLine\n");
        for(int i = 0; i<conc; i++)
                printf("\t\t%d\t\t%s\t\t%s\t\t%d\n",i,constants[i],literals[ctypes[i]],conline[i]);
}
int main(){
        init();
        yyin = fopen("testFiles/dataType_test.c","r");
        yylex();
        printf("\n");
        if(ec){
                printf("One or more errors were found:\n");
                        printerrors();
        }
        if(!ec){
                printidentifiers();
                printf("\n");
                printconstants();
        }
        return 0;
}
int yywrap(){
        return 1;
}
```

Test Cases

Case 1: Comments

```
// This is a valid single line comment

/*
        This is a valid multi-line comment
*/

/*
        This is an invalid multiline comment
//
```

Output:

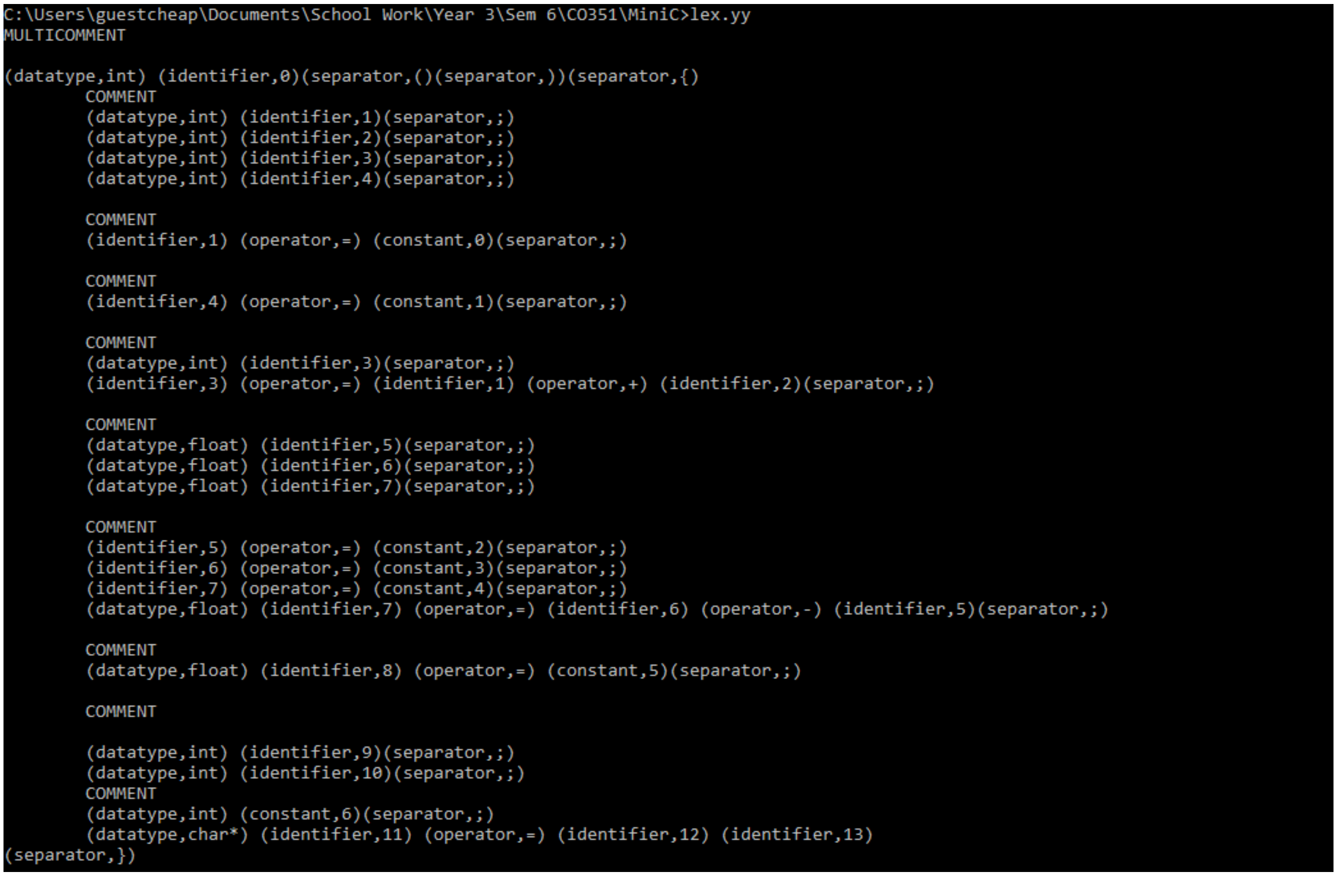


```
C:\Users\guestcheap\Documents\School Work\Year 3\Sem 6\CO351\MiniC>lex.yy
COMMENT

MULTICOMMENT


        (identifier,0) (identifier,1) (identifier,2) (identifier,3) (identifier,4) (identifier,5)
COMMENT
One or more errors were found:
Lexical Error on Line: 7, Dangling Comment
```

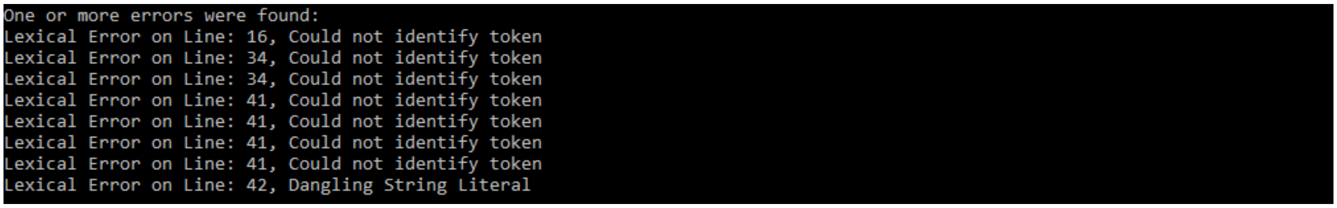
Case 2: Variables and Constants

```
/*
        This is a program to test the compiler for handling basic datatypes, comments and basic
arithmetic operations
*/

int main(){
        //Int dataType declarations
        int a;
        int b;
        int c;
        int k;

        // Correct test case for int dataType
        a = 10;

        //Incorrect test case for int dataType
        k = 2x;

        //Arithmetic operations on int dataType
        int c;
        c = a + b;

        //float dataType declarations
        float e;
        float f;
        float g;

        //Correct test case for float dataTypes
        e = 1.00;
        f = 2.00;
        g = 3;
        float g = f - e;

        //Incorrect test case for float dataType
        float x = 1.0.1;

        //TestCases for checking the valid variable conventions

        int abc;
        int abc_9;
        //Invalid Case:
        int 9_abc;
        char* str = "Invalid string
}
```

Output:



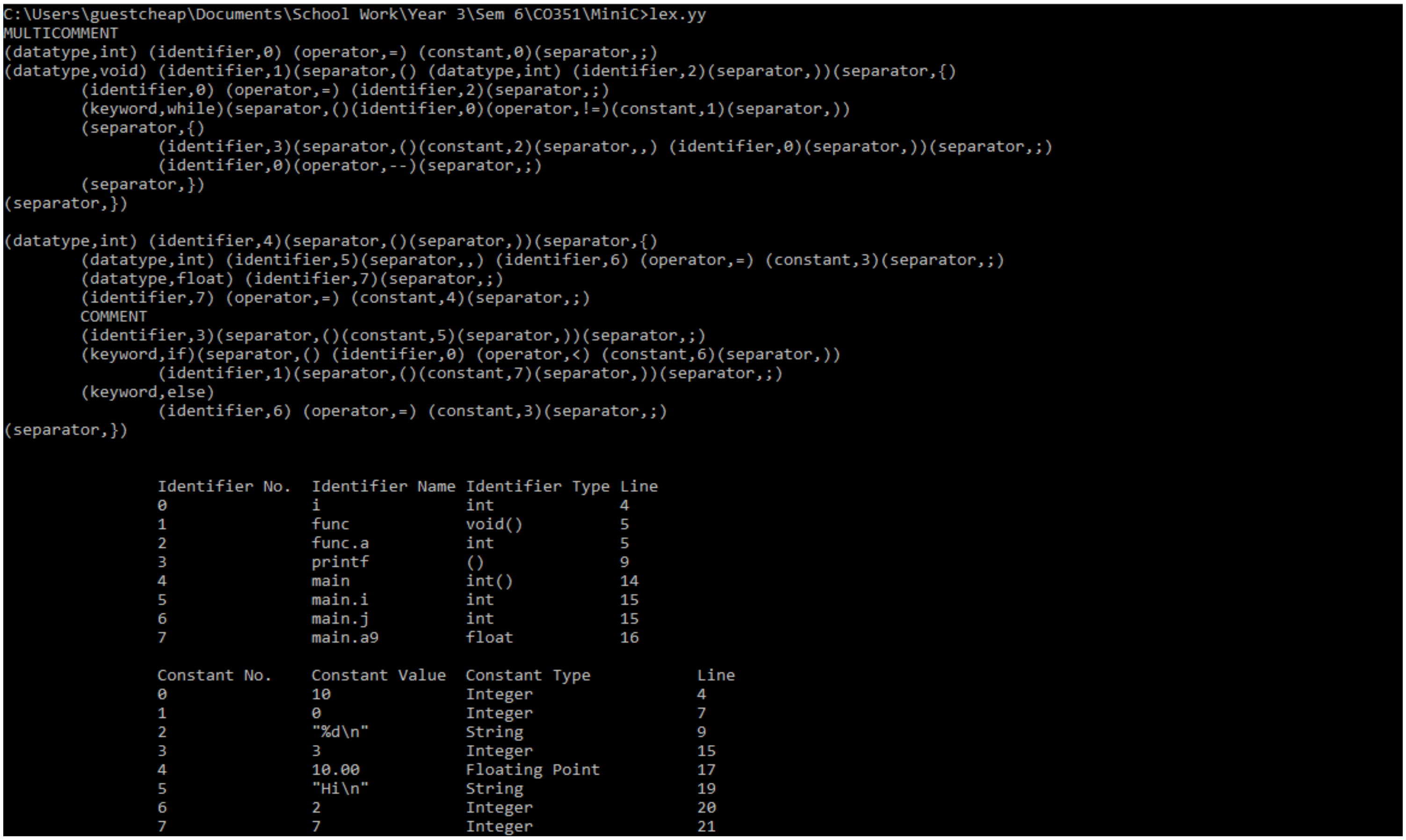
```
C:\Users\guestcheap\Documents\School Work\Year 3\Sem 6\CO351\MiniC>lex.yy
MULTICOMMENT

(datatype,int) (identifier,0)(separator,()(separator,))(separator,{)
        COMMENT
        (datatype,int) (identifier,1)(separator,;)
        (datatype,int) (identifier,2)(separator,;)
        (datatype,int) (identifier,3)(separator,;)
        (datatype,int) (identifier,4)(separator,;)

        COMMENT
        (identifier,1) (operator,=) (constant,0)(separator,;)

        COMMENT
        (identifier,4) (operator,=) (constant,1)(separator,;)

        COMMENT
        (datatype,int) (identifier,3)(separator,;)
        (identifier,3) (operator,=) (identifier,1) (operator,+) (identifier,2)(separator,;)

        COMMENT
        (datatype,float) (identifier,5)(separator,;)
        (datatype,float) (identifier,6)(separator,;)
        (datatype,float) (identifier,7)(separator,;)

        COMMENT
        (identifier,5) (operator,=) (constant,2)(separator,;)
        (identifier,6) (operator,=) (constant,3)(separator,;)
        (identifier,7) (operator,=) (constant,4)(separator,;)
        (datatype,float) (identifier,7) (operator,=) (identifier,6) (operator,-) (identifier,5)(separator,;)

        COMMENT
        (datatype,float) (identifier,8) (operator,=) (constant,5)(separator,;)

        COMMENT

        (datatype,int) (identifier,9)(separator,;)
        (datatype,int) (identifier,10)(separator,;)
        COMMENT
        (datatype,int) (constant,6)(separator,;)
        (datatype,char*) (identifier,11) (operator,=) (identifier,12) (identifier,13)
(separator,})
```



```
One or more errors were found:
Lexical Error on Line: 16, Could not identify token
Lexical Error on Line: 34, Could not identify token
Lexical Error on Line: 34, Could not identify token
Lexical Error on Line: 41, Could not identify token
Lexical Error on Line: 41, Could not identify token
Lexical Error on Line: 41, Could not identify token
Lexical Error on Line: 41, Could not identify token
Lexical Error on Line: 42, Dangling String Literal
```

Case 3: Function Test

```
/*
	This is a program to test almost all the functionalities provided by the compiler and mainly function calls
*/
int i = 10;
void func( int a){
	i = a;
	while(i!=0)
	{
		printf("%d\n", i);
		i--;
	}
}

int main(){
	int i, j = 3;
	float a9;
	a9 = 10.00;
	//Valid function calls
	printf("Hi\n");
	if( i < 2)
		func(7);
	else
		j = 3;
}
```

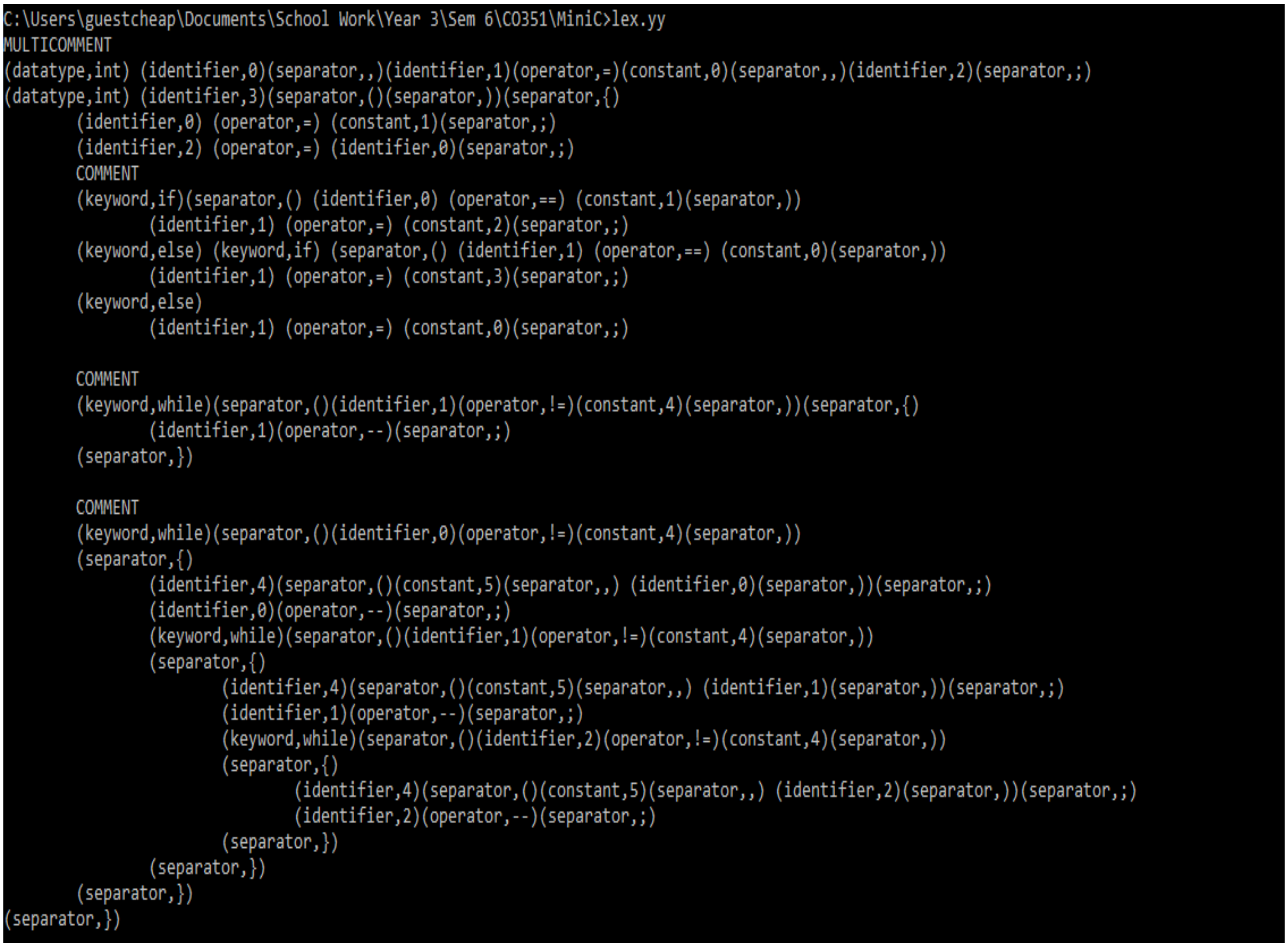


```
C:\Users\guestcheap\Documents\School Work\Year 3\Sem 6\CO351\MiniC>lex.yy
MULTICOMMENT
(datatype,int) (identifier,0) (operator,=) (constant,0)(separator,;)
(datatype,void) (identifier,1)(separator,() (datatype,int) (identifier,2)(separator,))(separator,{)
        (identifier,0) (operator,=) (identifier,2)(separator,;)
        (keyword,while)(separator,()(identifier,0)(operator,!=)(constant,1)(separator,))
        (separator,{)
                (identifier,3)(separator,()(constant,2)(separator,,) (identifier,0)(separator,))(separator,;)
                (identifier,0)(operator,--)(separator,;)
        (separator,})
(separator,})

(datatype,int) (identifier,4)(separator,()(separator,))(separator,{)
        (datatype,int) (identifier,5)(separator,,) (identifier,6) (operator,=) (constant,3)(separator,;)
        (datatype,float) (identifier,7)(separator,;)
        (identifier,7) (operator,=) (constant,4)(separator,;)
        COMMENT
        (identifier,3)(separator,()(constant,5)(separator,))(separator,;)
        (keyword,if)(separator,() (identifier,0) (operator,<) (constant,6)(separator,))
                (identifier,1)(separator,()(constant,7)(separator,))(separator,;)
        (keyword,else)
                (identifier,6) (operator,=) (constant,3)(separator,;)
(separator,})


                Identifier No.  Identifier Name Identifier Type Line
                0               i               int             4
                1               func            void()          5
                2               func.a          int             5
                3               printf          ()              9
                4               main            int()           14
                5               main.i          int             15
                6               main.j          int             15
                7               main.a9         float           16

                Constant No.    Constant Value  Constant Type           Line
                0               10              Integer                 4
                1               0               Integer                 7
                2               "%d\n"          String                  9
                3               3               Integer                 15
                4               10.00           Floating Point          17
                5               "Hi\n"          String                  19
                6               2               Integer                 20
                7               7               Integer                 21
```

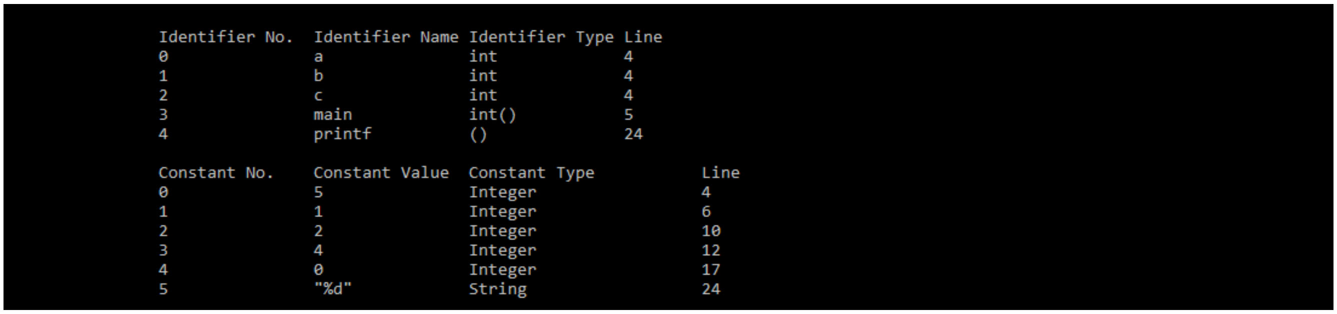
Case 4: Loop Test

```
/*
    This is a program to test the compiler for handling loops, datatype, comments and basic arithmetic operations
*/
int a,b=5,c;
int main(){
    a = 1;
    c = a;
    // Correct test case for If else loop
    if( a == 1)
        b = 2;
    else if ( b == 5)
        b = 4;
    else
        b = 5;

    //Valid while loop statement
    while(b!=0){
        b--;
    }

    // TestCase for checking the correct working of nested loops
    while(a!=0)
    {
        printf("%d", a);
        a--;
        while(b!=0)
        {
            printf("%d", b);
            b--;
            while(c!=0)
            {
                printf("%d", c);
                c--;
            }
        }
    }
```

Output:



```
C:\Users\guestcheap\Documents\School Work\Year 3\Sem 6\CO351\MiniC>lex.yy
MULTICOMMENT
(datatype,int) (identifier,0)(separator,,)(identifier,1)(operator,=)(constant,0)(separator,,)(identifier,2)(separator,;)
(datatype,int) (identifier,3)(separator,()(separator,))(separator,{)
        (identifier,0) (operator,=) (constant,1)(separator,;)
        (identifier,2) (operator,=) (identifier,0)(separator,;)
        COMMENT
        (keyword,if)(separator,() (identifier,0) (operator,==) (constant,1)(separator,))
                (identifier,1) (operator,=) (constant,2)(separator,;)
        (keyword,else) (keyword,if) (separator,() (identifier,1) (operator,==) (constant,0)(separator,))
                (identifier,1) (operator,=) (constant,3)(separator,;)
        (keyword,else)
                (identifier,1) (operator,=) (constant,0)(separator,;)

        COMMENT
        (keyword,while)(separator,()(identifier,1)(operator,!=)(constant,4)(separator,))(separator,{)
                (identifier,1)(operator,--)(separator,;)
        (separator,})

        COMMENT
        (keyword,while)(separator,()(identifier,0)(operator,!=)(constant,4)(separator,))
        (separator,{)
                (identifier,4)(separator,()(constant,5)(separator,,) (identifier,0)(separator,))(separator,;)
                (identifier,0)(operator,--)(separator,;)
                (keyword,while)(separator,()(identifier,1)(operator,!=)(constant,4)(separator,))
                (separator,{)
                        (identifier,4)(separator,()(constant,5)(separator,,) (identifier,1)(separator,))(separator,;)
                        (identifier,1)(operator,--)(separator,;)
                        (keyword,while)(separator,()(identifier,2)(operator,!=)(constant,4)(separator,))
                        (separator,{)
                                (identifier,4)(separator,()(constant,5)(separator,,) (identifier,2)(separator,))(separator,;)
                                (identifier,2)(operator,--)(separator,;)
                        (separator,})
                (separator,})
        (separator,})
(separator,})
```



```
                Identifier No.  Identifier Name Identifier Type Line
                0               a               int             4
                1               b               int             4
                2               c               int             4
                3               main            int()           5
                4               printf          ()              24

                Constant No.    Constant Value  Constant Type           Line
                0               5               Integer                 4
                1               1               Integer                 6
                2               2               Integer                 10
                3               4               Integer                 12
                4               0               Integer                 17
                5               "%d"            String                  24
```