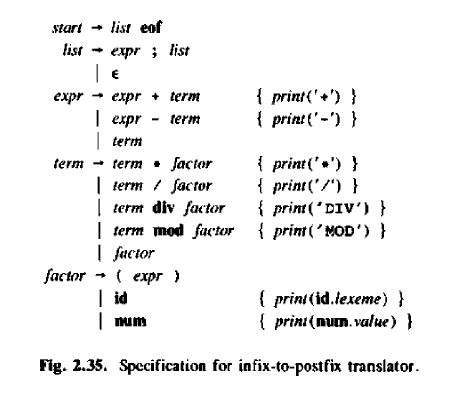
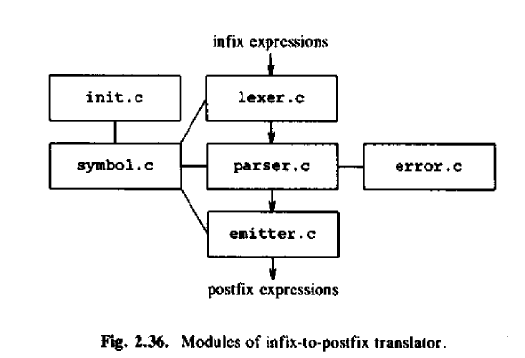
**Experiment No.: 7 Date: 16/11/2020**

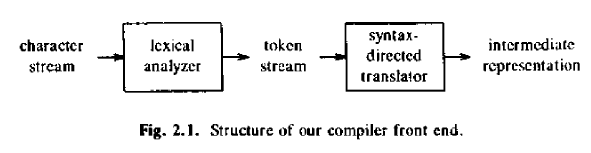
**Lexical Analysis**

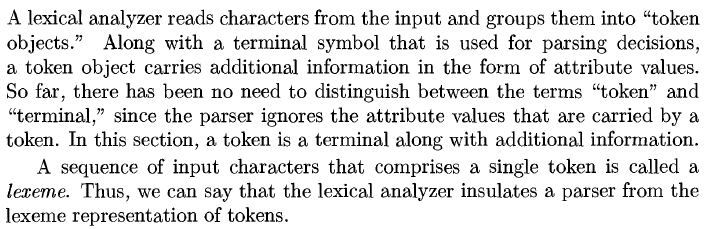
Aim: To write a program to detect tokens from user defined expression.

Theory:

Compiler that translates infix expressions into postfix form.

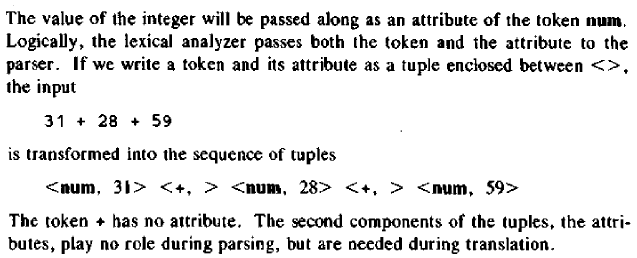
****



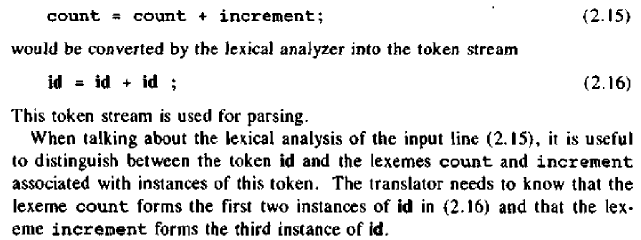


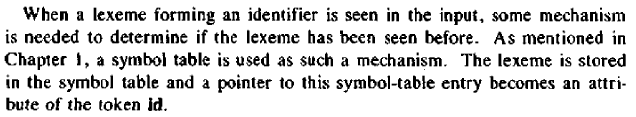
Functions of a lexical analyser

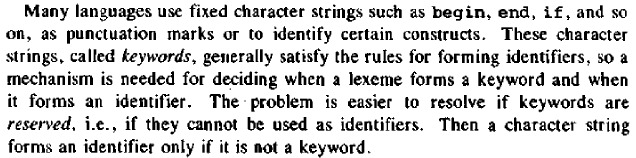
1. Removal of white space and comments
2. Constants – when a sequence of digits appears in the input stream, the lexical analyzer will pass the token **num** to the parser

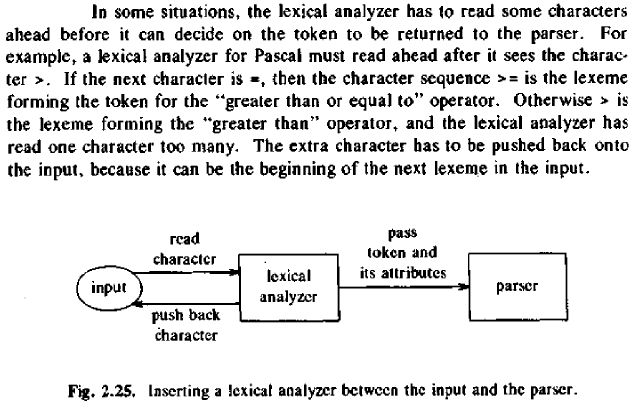


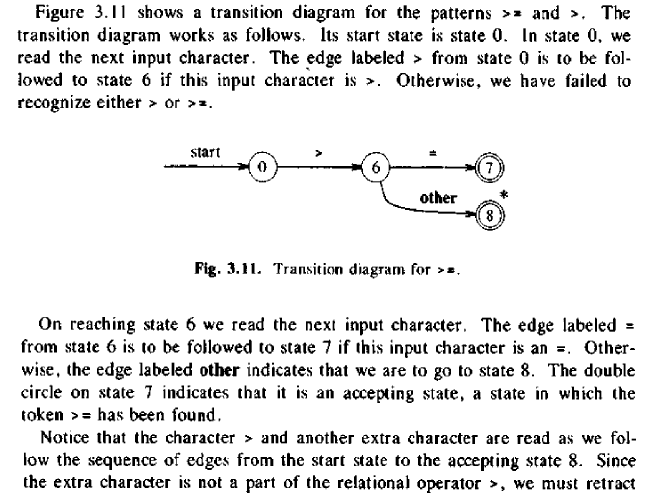
3. Recognizing identifiers and keywords



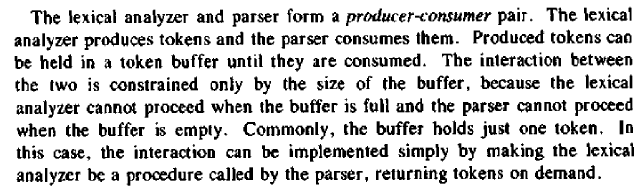


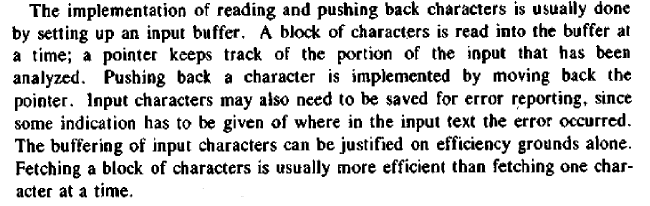




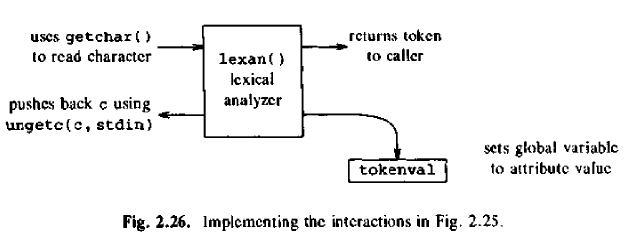






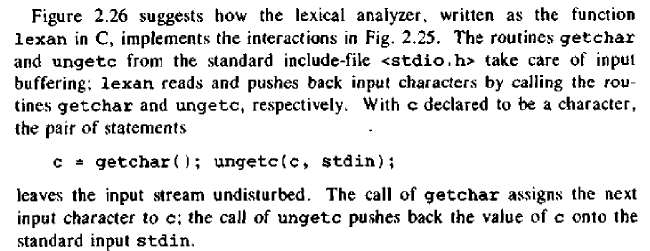


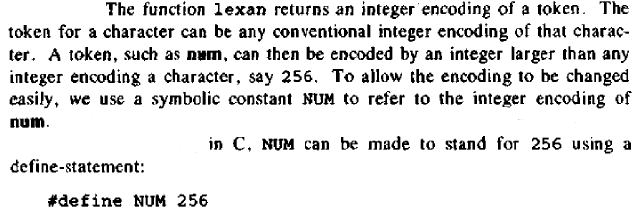
lexer.c

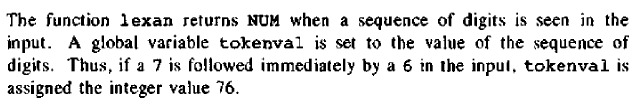


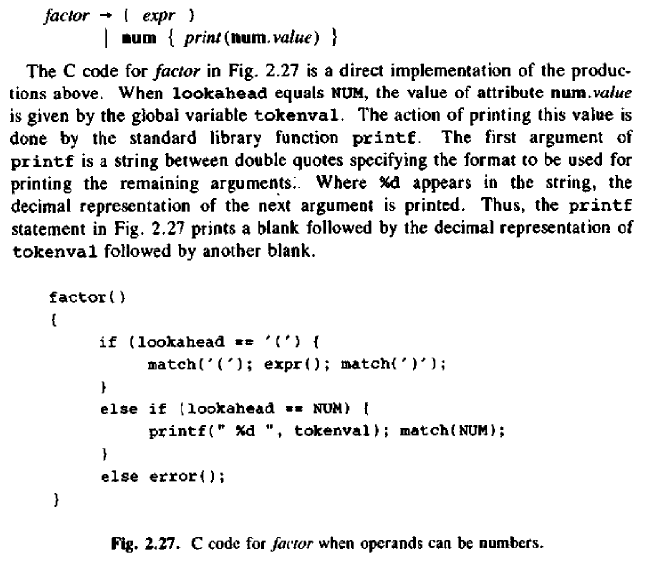
Text

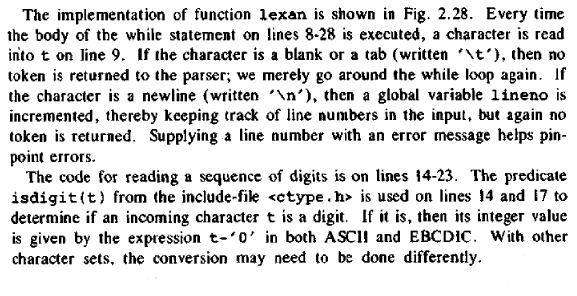
Description automatically generated

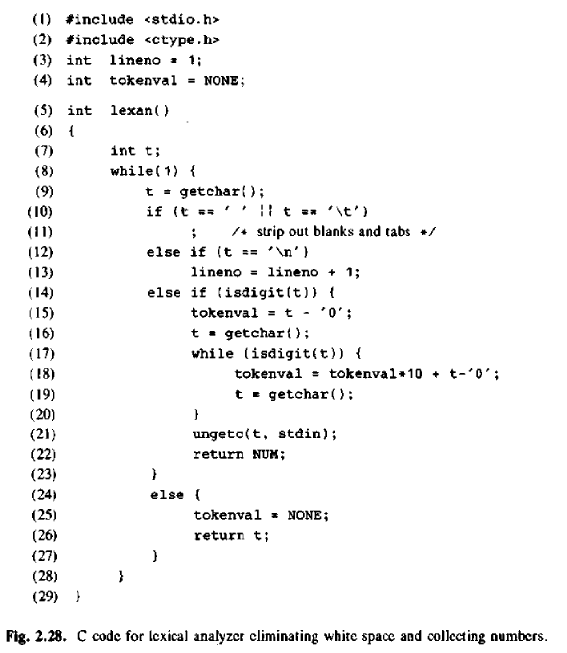












**References:**

1. Aho, Ulman and Sethi; Compilers, Principles, techniques and tools; Pearson Education Asia, ISBN: 81-7808-046-X.

PROGRAM:

#include <stdbool.h>

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

struct entry

{

char type[10];

char token[10];

struct entry \*next;

};

struct entry \*start, \*end;

int lookup(char token[]) {

struct entry \*temp = start;

while (temp != NULL)

{

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

return 1;

}

temp = temp->next;

}

return 0;

}

void delete(char token[]) {

struct entry \*temp = start;

struct entry \*tempp = start;

if (!lookup(token)) {

printf("Token does not exist\n");

return;

}

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

start = temp->next;

free(temp);

return;

}

while (temp != NULL)

{

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

tempp->next = temp->next;

free(temp);

return;

}

tempp = temp;

temp = temp->next;

}

}

int insert(char type[], char token[]) {

if (strcmp(type, "id") == 0) {

if (lookup(token)) {

printf("Token %s already exists\n", token);

return 0;

}

}

struct entry \*node = (struct entry\*)malloc(sizeof(struct entry));

strcpy\_s(node->token, 10, token);

strcpy\_s(node->type, 10, type);

node->next = NULL;

if (start == NULL) {

start = end = node;

}

else

{

end->next = node;

end = node;

}

return 1;

}

void print() {

struct entry \*temp = start;

printf("\nSymbol table");

while (temp != NULL)

{

printf("\nToken : %s\tType: %s", temp->token, temp->type);

temp = temp->next;

}

}

bool isDelimiter(char ch)

{

if (ch == ' ' || ch == '+' || ch == '-' || ch == '\*' ||

ch == '/' || ch == ',' || ch == ';' || ch == '>' ||

ch == '<' || ch == '=' || ch == '(' || ch == ')' ||

ch == '[' || ch == ']' || ch == '{' || ch == '}')

return (true);

return (false);

}

bool isOperator(char ch)

{

if (ch == '+' || ch == '-' || ch == '\*' ||

ch == '/' || ch == '>' || ch == '<' ||

ch == '=')

return (true);

return (false);

}

bool validIdentifier(char\* str)

{

if (str[0] == '0' || str[0] == '1' || str[0] == '2' ||

str[0] == '3' || str[0] == '4' || str[0] == '5' ||

str[0] == '6' || str[0] == '7' || str[0] == '8' ||

str[0] == '9' || isDelimiter(str[0]) == true)

return (false);

return (true);

}

bool isKeyword(char\* str)

{

if (!strcmp(str, "if") || !strcmp(str, "else") ||

!strcmp(str, "while") || !strcmp(str, "do") ||

!strcmp(str, "break") ||

!strcmp(str, "continue") || !strcmp(str, "int")

|| !strcmp(str, "double") || !strcmp(str, "float")

|| !strcmp(str, "return") || !strcmp(str, "char")

|| !strcmp(str, "case") || !strcmp(str, "char")

|| !strcmp(str, "sizeof") || !strcmp(str, "long")

|| !strcmp(str, "short") || !strcmp(str, "typedef")

|| !strcmp(str, "switch") || !strcmp(str, "unsigned")

|| !strcmp(str, "void") || !strcmp(str, "static")

|| !strcmp(str, "struct") || !strcmp(str, "goto")

|| !strcmp(str, "true") || !strcmp(str, "false"))

return (true);

return (false);

}

bool isInteger(char\* str)

{

int i, len = strlen(str);

if (len == 0)

return (false);

for (i = 0; i < len; i++) {

if (str[i] != '0' && str[i] != '1' && str[i] != '2'

&& str[i] != '3' && str[i] != '4' && str[i] != '5'

&& str[i] != '6' && str[i] != '7' && str[i] != '8'

&& str[i] != '9' || (str[i] == '-' && i > 0))

return (false);

}

return (true);

}

bool isRealNumber(char\* str)

{

int i, len = strlen(str);

bool hasDecimal = false;

if (len == 0)

return (false);

for (i = 0; i < len; i++) {

if (str[i] != '0' && str[i] != '1' && str[i] != '2'

&& str[i] != '3' && str[i] != '4' && str[i] != '5'

&& str[i] != '6' && str[i] != '7' && str[i] != '8'

&& str[i] != '9' && str[i] != '.' ||

(str[i] == '-' && i > 0))

return (false);

if (str[i] == '.')

hasDecimal = true;

}

return (hasDecimal);

}

char\* subString(char\* str, int left, int right)

{

int i;

char\* subStr = (char\*)malloc(

sizeof(char) \* (right - left + 2));

for (i = left; i <= right; i++)

subStr[i - left] = str[i];

subStr[right - left + 1] = '\0';

return (subStr);

}

void parse(char\* str)

{

int left = 0, right = 0;

int len = strlen(str);

while (right <= len && left <= right) {

if (isDelimiter(str[right]) == false)

right++;

if (isDelimiter(str[right]) == true && left == right) {

if (isOperator(str[right]) == true) {

printf("'%c' IS AN OPERATOR\n", str[right]);

char a[10];

a[0] = str[right];

a[1] = '\0';

insert("operator", a);

}

right++;

left = right;

}

else if (isDelimiter(str[right]) == true && left != right

|| (right == len && left != right)) {

char\* subStr = subString(str, left, right - 1);

if (isKeyword(subStr) == true) {

printf("'%s' is a Keyword\n", subStr);

insert("keyword", subStr);

}

else if (isInteger(subStr) == true) {

printf("'%s' is an Integer\n", subStr);

insert("integer", subStr);

}

else if (isRealNumber(subStr) == true) {

printf("'%s' is a Real Number\n", subStr);

insert("real no", subStr);

}

else if (validIdentifier(subStr) == true

&& isDelimiter(str[right - 1]) == false) {

printf("'%s' is a valid Identifier\n", subStr);

insert("id", subStr);

}

else if (validIdentifier(subStr) == false

&& isDelimiter(str[right - 1]) == false)

printf("'%s' is an invalid Identifier\n", subStr);

left = right;

}

}

return;}

int main()

{

char str[100] = "int ab = x + y; ";

start = end = NULL;

parse(str);

print();

int choice;

char token[10];

while (1) {

printf("\nEnter choice : \n1.Insert\n2.Search\n3.Delete\n4.Display table\n");

scanf\_s("%d", &choice);

switch (choice) {

case 1:

printf("Insert Token : ");

scanf\_s("%s", token, 10);

parse(token);

break;

case 2:

printf("Token : ");

scanf\_s("%s", token, 10);

if (lookup(token))

printf("Token found\n");

else

printf("Token not found\n");

break;

case 3:

printf("Delete Token : ");

scanf\_s("%s", token, 10);

delete(token);

break;

case 4:

print();

printf("\n");

break;

}

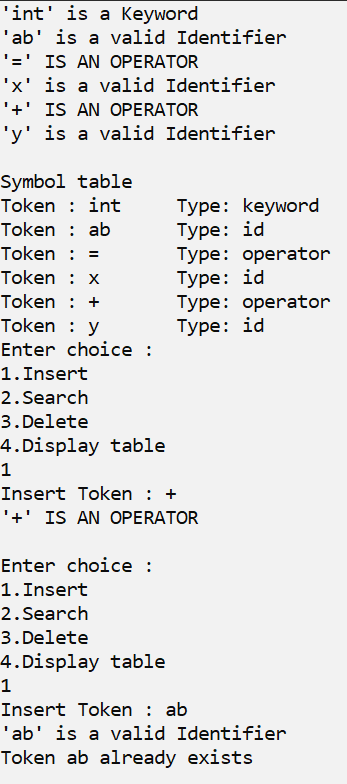
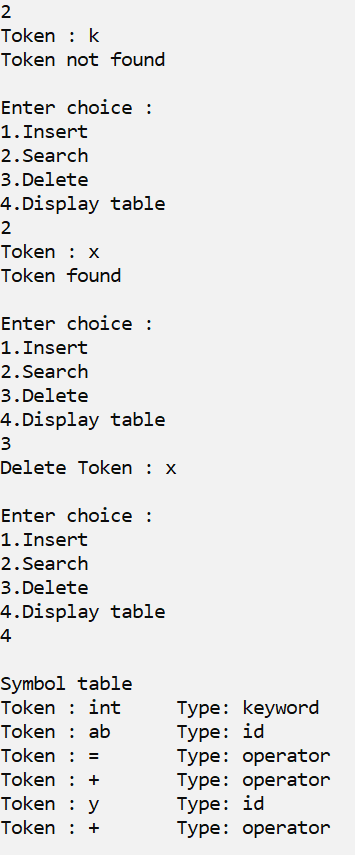
}

\_getch();

return (0);

}

OUTPUT:



Conclusion: program to detect tokens was successfully implemented