CC INTERNAL-I

1. Implement Lexical analyzer / Scanner using C.

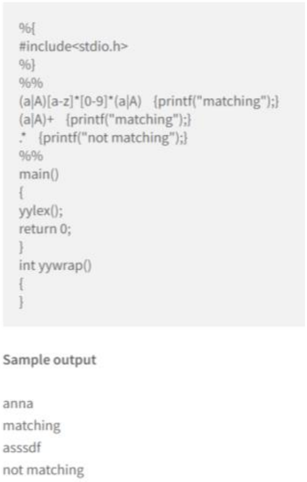
```
#include <stdbool.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

// Returns 'true' if the character is a DELIMITER.
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);
}

// Returns 'true' if the character is an OPERATOR.
bool isOperator(char ch)
{
        if (ch == '+' || ch == '-' || ch == '*' ||
                ch == '/' || ch == '>' || ch == '<' ||
                ch == '=')
                return (true);
        return (false);
}

// Returns 'true' if the string is a VALID IDENTIFIER.
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);
}

// Returns 'true' if the string is a KEYWORD.
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"))
                return (true);
        return (false);
}

// Returns 'true' if the string is an INTEGER.
```

```
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);
}

// Returns 'true' if the string is a REAL NUMBER.
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);
}

// Extracts the SUBSTRING.
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);
}

// Parsing the input STRING.
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]);

            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);

            else if (isInteger(subStr) == true)
                printf("'%s' IS AN INTEGER\n", subStr);

            else if (isRealNumber(subStr) == true)
                printf("'%s' IS A REAL NUMBER\n", subStr);

            else if (validIdentifier(subStr) == true
                    && isDelimiter(str[right - 1]) == false)
                printf("'%s' IS A VALID IDENTIFIER\n", subStr);

            else if (validIdentifier(subStr) == false
                    && isDelimiter(str[right - 1]) == false)
                printf("'%s' IS NOT A VALID IDENTIFIER\n", subStr);
            left = right;
        }
    }
    return;
}

// DRIVER FUNCTION
int main()
{
    // maximum length of string is 100 here
    char str[100] = "int a = b + 1c; ";

    parse(str); // calling the parse function

    return (0);
}
```

2. Lex program to recognize String ending with 00.

```
 %%
[0-9]*00{printf(“string accepted”);
[0-9]*{printf(“string rejected”);}
%%
main()
{
yylex();
}
int yywrap()
{
return 1;
}
```

3. Lex Program to recognize the strings which are starting and ending with ‘a’

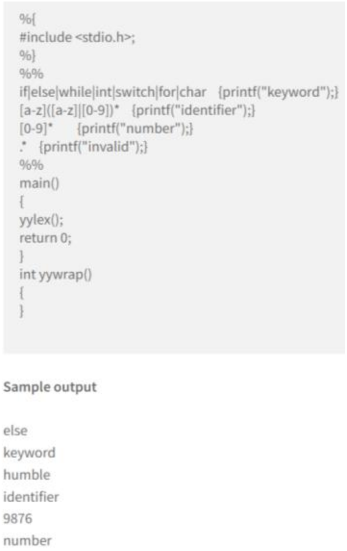


```
%{
#include<stdio.h>
%}
%%
(a|A)[a-z]*[0-9]*(a|A)   {printf("matching");}
(a|A)+   {printf("matching");}
.*   {printf("not matching");}
%%
main()
{
yylex();
return 0;
}
int yywrap()
{
}
```

Sample output

```
anna
matching
asssdf
not matching
```

4. Lex program to recognize Keywords.



```
%{
#include <stdio.h>;
%}
%%
if|else|while|int|switch|for|char   {printf("keyword");}
[a-z]([a-z]|[0-9])*   {printf("identifier");}
[0-9]*      {printf("number");}
.*   {printf("invalid");}
%%
main()
{
yylex();
return 0;
}
int yywrap()
{
}
```

Sample output

```
else
keyword
humble
identifier
9876
number
```

5. Lex Program to recognize the numbers which has 1 in its 5th position from right.

```
%%
[1-9]*1[1-9]{4}  {printf(“satisfying”);}
%%
```

6. Lex program to recognize Identifiers.

4 lo chusko pooooo

7. Lex program to assign line numbers for source code.

```
/* Program to add line numbers
to a given file*/
%{
int line_number = 1; // initializing line number to 1
%}

/* simple name definitions to simplify the scanner specification name definition of line*/

%%
{line} { printf("%10d %s", line_number++, yytext); }

/* whenever a line is encountered increment count*/

/* 10 specifies the padding from left side to present the line numbers*/

/* yytext The text of the matched pattern is stored in this variable (char*)*/

%%

int yywrap(){}

int main(int argc, char*argv[])
{
extern FILE *yyin; // yyin as pointer of File type

yyin = fopen("testtest.c","r"); /* yyin points to the file testtest.c and opens it in read mode.*/

yylex(); // The function that starts the analysis.

return 0;
}
```

8. Implement lexical analyzer in Lex.

```
%{
int COMMENT=0;
%}
identifier [a-zA-Z][a-zA-Z0-9]*
%%
#.* {printf("\n%s is a preprocessor directive",yytext);}
int |
float |
char |
double |
while |
for |
struct |
typedef |
do |
if |
break |
continue |
void |
switch |
return |
else |
goto {printf("\n\t%s is a keyword",yytext);}
"/*" {COMMENT=1;}{printf("\n\t %s is a COMMENT",yytext);}
{identifier}\( {if(!COMMENT)printf("\nFUNCTION \n\t%s",yytext);}
\{  {if(!COMMENT)printf("\n BLOCK BEGINS");}
\}  {if(!COMMENT)printf("BLOCK ENDS ");}
```

```
{identifier}(\[[0-9]*\])? {if(!COMMENT) printf("\n %s IDENTIFIER",yytext);}
\".*\" {if(!COMMENT)printf("\n\t %s is a STRING",yytext);}
[0-9]+ {if(!COMMENT) printf("\n %s is a NUMBER ",yytext);}
\)(\:)? {if(!COMMENT)printf("\n\t");ECHO;printf("\n");}
\( ECHO;
= {if(!COMMENT)printf("\n\t %s is an ASSIGNMENT OPERATOR",yytext);}
\<= |
\>= |
\< |
== |
\> {if(!COMMENT) printf("\n\t%s is a RELATIONAL OPERATOR",yytext);}
%%

int main(int argc, char **argv)
{
FILE *file;
file=fopen("var.c","r");
if(!file)
{
printf("could not open the file");
exit(0);
}
yyin=file;
yylex();
printf("\n");
return(0);
}
int yywrap()
{
return(1);
}
```

9. Write a program to find first and follow set of the variable in the given productions.

```
// C program to calculate the First and
// Follow sets of a given grammar
#include<stdio.h>
#include<ctype.h>
#include<string.h>

// Functions to calculate Follow
void followfirst(char, int, int);
void follow(char c);

// Function to calculate First
void findfirst(char, int, int);

int count, n = 0;
```

```
// Stores the final result
// of the First Sets
char calc_first[10][100];

// Stores the final result
// of the Follow Sets
char calc_follow[10][100];
int m = 0;

// Stores the production rules
char production[10][10];
char f[10], first[10];
int k;
char ck;
int e;

int main(int argc, char **argv)
{
	int jm = 0;
	int km = 0;
	int i, choice;
	char c, ch;
	count = 8;

	// The Input grammar
	strcpy(production[0], "E=TR");
	strcpy(production[1], "R=+TR");
	strcpy(production[2], "R=#");
	strcpy(production[3], "T=FY");
	strcpy(production[4], "Y=*FY");
	strcpy(production[5], "Y=#");
	strcpy(production[6], "F=(E)");
	strcpy(production[7], "F=i");

	int kay;
	char done[count];
	int ptr = -1;

	// Initializing the calc_first array
	for(k = 0; k < count; k++) {
		for(kay = 0; kay < 100; kay++) {
			calc_first[k][kay] = '!';
		}
	}
	int point1 = 0, point2, xxx;

	for(k = 0; k < count; k++)
	{
		c = production[k][0];
		point2 = 0;
		xxx = 0;
```

```
        // Checking if First of c has
        // already been calculated
        for(kay = 0; kay <= ptr; kay++)
            if(c == done[kay])
                xxx = 1;

        if (xxx == 1)
            continue;

        // Function call
        findfirst(c, 0, 0);
        ptr += 1;

        // Adding c to the calculated list
        done[ptr] = c;
        printf("\n First(%c) = { ", c);
        calc_first[point1][point2++] = c;

        // Printing the First Sets of the grammar
        for(i = 0 + jm; i < n; i++) {
            int lark = 0, chk = 0;

            for(lark = 0; lark < point2; lark++) {

                if (first[i] == calc_first[point1][lark])
                {
                    chk = 1;
                    break;
                }
            }
            if(chk == 0)
            {
                printf("%c, ", first[i]);
                calc_first[point1][point2++] = first[i];
            }
        }
        printf("}\n");
        jm = n;
        point1++;
    }
    printf("\n");
    printf("-----------------------------------------------\n\n");
    char donee[count];
    ptr = -1;

    // Initializing the calc_follow array
    for(k = 0; k < count; k++) {
        for(kay = 0; kay < 100; kay++) {
            calc_follow[k][kay] = '!';
        }
    }
    point1 = 0;
    int land = 0;
    for(e = 0; e < count; e++)
    {
        ck = production[e][0];
        point2 = 0;
        xxx = 0;

        // Checking if Follow of ck
        // has already been calculated
        for(kay = 0; kay <= ptr; kay++)
            if(ck == donee[kay])
                xxx = 1;

        if (xxx == 1)
            continue;
        land += 1;

        // Function call
        follow(ck);
        ptr += 1;

        // Adding ck to the calculated list
        donee[ptr] = ck;
        printf(" Follow(%c) = { ", ck);
        calc_follow[point1][point2++] = ck;

        // Printing the Follow Sets of the grammar
        for(i = 0 + km; i < m; i++) {
            int lark = 0, chk = 0;
            for(lark = 0; lark < point2; lark++)
            {
                if (f[i] == calc_follow[point1][lark])
                {
                    chk = 1;
                    break;
                }
            }
            if(chk == 0)
            {
                printf("%c, ", f[i]);
                calc_follow[point1][point2++] = f[i];
            }
        }
        printf(" }\n\n");
        km = m;
        point1++;
    }
}
```

```
void follow(char c)
{
    int i, j;

    // Adding "$" to the follow
    // set of the start symbol
    if(production[0][0] == c) {
        f[m++] = '$';
    }
    for(i = 0; i < 10; i++)
    {
        for(j = 2;j < 10; j++)
        {
            if(production[i][j] == c)
            {
                if(production[i][j+1] != '\0')
                {
                    // Calculate the first of the next
                    // Non-Terminal in the production
                    followfirst(production[i][j+1], i, (j+2));
                }

                if(production[i][j+1]=='\0' && c!=production[i][0])
                {
                    // Calculate the follow of the Non-Terminal
                    // in the L.H.S. of the production
                    follow(production[i][0]);
                }
            }
        }
    }
}

void findfirst(char c, int q1, int q2)
{
    int j;

    // The case where we
    // encounter a Terminal
    if(!(isupper(c))) {
        first[n++] = c;
    }
    for(j = 0; j < count; j++)
    {
        if(production[j][0] == c)
        {
            if(production[j][2] == '#')
            {
                if(production[q1][q2] == '\0')
                    first[n++] = '#';
                else if(production[q1][q2] != '\0'
                        && (q1 != 0 || q2 != 0))
                {
                    // Recursion to calculate First of New
                    // Non-Terminal we encounter after epsilon
                    findfirst(production[q1][q2], q1, (q2+1));
                }
                else
                    first[n++] = '#';
            }
            else if(!isupper(production[j][2]))
            {
                first[n++] = production[j][2];
            }
            else
            {
                // Recursion to calculate First of
                // New Non-Terminal we encounter
                // at the beginning
                findfirst(production[j][2], j, 3);
            }
        }
    }
}

void followfirst(char c, int c1, int c2)
{
    int k;

    // The case where we encounter
    // a Terminal
    if(!(isupper(c)))
        f[m++] = c;
    else
    {
        int i = 0, j = 1;
        for(i = 0; i < count; i++)
        {
            if(calc_first[i][0] == c)
                break;
        }

        //Including the First set of the
        // Non-Terminal in the Follow of
        // the original query
        while(calc_first[i][j] != '!')
        {
            if(calc_first[i][j] != '#')
            {
                f[m++] = calc_first[i][j];
            }
            else
            {
                if(production[c1][c2] == '\0')
                {
                    // Case where we reach the
                    // end of a production
                    follow(production[c1][0]);
                }
                else
                {
                    // Recursion to the next symbol
                    // in case we encounter a "#"
                    followfirst(production[c1][c2], c1, c2+1);
                }
            }
            j++;
        }}}
```

10. Write a program to find follow set of the variable in the given productions.

Above…………

11. Write a program for Recursive descent Parsing for expression grammar.

```
#include<stdio.h>
#include<string.h>
int E(),Edash(),T(),Tdash(),F();
char *ip;
char string[50];
int main()
{
printf("Enter the string\n");
scanf("%s",string);
ip=string;
printf("\n\nInput\tAction\n--------------------------------\n");

if(E() && ip=="\0"){
printf("\n--------------------------------\n");
printf("\n String is successfully parsed\n");
}
else{
printf("\n--------------------------------\n");
printf("Error in parsing String\n");
}
}
int E()
{
printf("%s\tE->TE' \n",ip);
if(T())
{
if(Edash())
{
return 1;
}
else
return 0;
}
else
return 0;
}
int Edash()
{
if(*ip=='+')
{
printf("%s\tE'->+TE' \n",ip);
ip++;
if(T())
{
if(Edash())
{
return 1;
}
else
return 0;
}
else
return 0;
}
else
{
printf("%s\tE'->^ \n",ip);
return 1;
}
}
int T()
{
printf("%s\tT->FT' \n",ip);
if(F())
{

if(Tdash())
{
return 1;
}
else
return 0;
}
else
return 0;
}
int Tdash()
{
if(*ip=='*')
{
```

```
printf("%s\tT'->*FT' \n",ip);
ip++;
if(F())
{
if(Tdash())
{
return 1;
}
else
return 0;
}
else
return 0;
}
else
{
printf("%s\tT'->^ \n",ip);
return 1;
}
}
int F()
{
if(*ip=='(')
{
printf("%s\tF->(E) \n",ip);
ip++;
if(E())
{
if(*ip==')')
{
ip++;
return 0;
}
else
return 0;
}
else
return 0;
}

else if(*ip=='i')
{
ip++;
printf("%s\tF->id \n",ip);
return 1;
}
else
return 0;
}
```

12. Implement LL(1) Parser.

```
#include<stdio.h>
#include<string.h>
#define TSIZE 128
// table[i][j] stores the index of production that must be applied on ith
varible if the input is jth nonterminal
int table[100][TSIZE];
// stores all list of terminals the ASCII value if use to index terminals
terminal[i] = 1 means the character with ASCII value is a terminal
char terminal[TSIZE];
// stores all list of terminals only Upper case letters from 'A' to 'Z'
can be nonterminals nonterminal[i] means ith alphabet is present as
nonterminal is the grammar
char nonterminal[26];
//structure to hold each production str[] stores the production len is the
length of production
struct product {
char str[100];
int len;
}pro[20];
// no of productions in form A->ß
int no_pro;
char first[26][TSIZE];
char follow[26][TSIZE];
// stores first of each production in form A->ß
char first_rhs[100][TSIZE];
// check if the symbol is nonterminal
int isNT(char c) {
return c >= 'A' && c <= 'Z';
}
// reading data from the file
void readFromFile() {
FILE* fptr;
fptr = fopen("text.txt", "r");
char buffer[255];
int i;
int j;
while (fgets(buffer, sizeof(buffer), fptr)) {
printf("%s", buffer);
j = 0;
nonterminal[buffer[0] - 'A'] = 1;
```

```
for (i = 0; i < strlen(buffer) - 1; ++i) {
if (buffer[i] == '|') {
++no_pro;
pro[no_pro - 1].str[j] = '\0';
pro[no_pro - 1].len = j;
pro[no_pro].str[0] = pro[no_pro - 1].str[0];
pro[no_pro].str[1] = pro[no_pro - 1].str[1];
pro[no_pro].str[2] = pro[no_pro - 1].str[2];
j = 3;
}
else {
pro[no_pro].str[j] = buffer[i];
++j;
if (!isNT(buffer[i]) && buffer[i] != '-' && buffer[i] != '>') {
terminal[buffer[i]] = 1;
}
}
}
pro[no_pro].len = j;
++no_pro;
}
}
void add_FIRST_A_to_FOLLOW_B(char A, char B) {
int i;
for (i = 0; i < TSIZE; ++i) {
if (i != '^')
follow[B - 'A'][i] = follow[B - 'A'][i] || first[A - 'A'][i];
}
}
void add_FOLLOW_A_to_FOLLOW_B(char A, char B) {
int i;
for (i = 0; i < TSIZE; ++i) {
if (i != '^')
follow[B - 'A'][i] = follow[B - 'A'][i] || follow[A - 'A'][i];
}
}
void FOLLOW() {
int t = 0;
int i, j, k, x;
while (t++ < no_pro) {
```

```
for (k = 0; k < 26; ++k) {
if (!nonterminal[k]) continue;
char nt = k + 'A';
for (i = 0; i < no_pro; ++i) {
for (j = 3; j < pro[i].len; ++j) {
if (nt == pro[i].str[j]) {
for (x = j + 1; x < pro[i].len; ++x) {
char sc = pro[i].str[x];
if (isNT(sc)) {
add_FIRST_A_to_FOLLOW_B(sc, nt);
if (first[sc - 'A']['^'])
continue;
}
else {
follow[nt - 'A'][sc] = 1;
}
break;
}
if (x == pro[i].len)
add_FOLLOW_A_to_FOLLOW_B(pro[i].str[0], nt);
}
}
}
}
}
}
void add_FIRST_A_to_FIRST_B(char A, char B) {
int i;
for (i = 0; i < TSIZE; ++i) {
if (i != '^') {
first[B - 'A'][i] = first[A - 'A'][i] || first[B - 'A'][i];
}
}
}
void FIRST() {
int i, j;
int t = 0;
while (t < no_pro) {
for (i = 0; i < no_pro; ++i) {
for (j = 3; j < pro[i].len; ++j) {
```

```
char sc = pro[i].str[j];
if (isNT(sc)) {
add_FIRST_A_to_FIRST_B(sc, pro[i].str[0]);
if (first[sc - 'A']['^'])
continue;
}
else {
first[pro[i].str[0] - 'A'][sc] = 1;
}
break;
}
if (j == pro[i].len)
first[pro[i].str[0] - 'A']['^'] = 1;
}
++t;
}
}
void add_FIRST_A_to_FIRST_RHS__B(char A, int B) {
int i;
for (i = 0; i < TSIZE; ++i) {
if (i != '^')
first_rhs[B][i] = first[A - 'A'][i] || first_rhs[B][i];
}
}
// Calculates FIRST(ß) for each A->ß
void FIRST_RHS() {
int i, j;
int t = 0;
while (t < no_pro) {
for (i = 0; i < no_pro; ++i) {
for (j = 3; j < pro[i].len; ++j) {
char sc = pro[i].str[j];
if (isNT(sc)) {
add_FIRST_A_to_FIRST_RHS__B(sc, i);
if (first[sc - 'A']['^'])
continue;
}
else {
first_rhs[i][sc] = 1;
}
```

```
break;
}
if (j == pro[i].len)
first_rhs[i]['^'] = 1;
}
++t;
}
}
int main() {
readFromFile();
follow[pro[0].str[0] - 'A']['$'] = 1;
FIRST();
FOLLOW();
FIRST_RHS();
int i, j, k;
// display first of each variable
printf("\n");
for (i = 0; i < no_pro; ++i) {
if (i == 0 || (pro[i - 1].str[0] != pro[i].str[0])) {
char c = pro[i].str[0];
printf("FIRST OF %c: ", c);
for (j = 0; j < TSIZE; ++j) {
if (first[c - 'A'][j]) {
printf("%c ", j);
}
}
printf("\n");
}
}
// display follow of each variable
printf("\n");
for (i = 0; i < no_pro; ++i) {
if (i == 0 || (pro[i - 1].str[0] != pro[i].str[0])) {
char c = pro[i].str[0];
printf("FOLLOW OF %c: ", c);
for (j = 0; j < TSIZE; ++j) {
if (follow[c - 'A'][j]) {
printf("%c ", j);
}
}
```

```
printf("\n");
}
}
// display first of each variable ß
// in form A->ß
printf("\n");
for (i = 0; i < no_pro; ++i) {
printf("FIRST OF %s: ", pro[i].str);
for (j = 0; j < TSIZE; ++j) {
if (first_rhs[i][j]) {
printf("%c ", j);
}
}
printf("\n");
}
// the parse table contains '$'
// set terminal['$'] = 1
// to include '$' in the parse table
terminal['$'] = 1;
// the parse table do not read '^'
// as input
// so we set terminal['^'] = 0
// to remove '^' from terminals
terminal['^'] = 0;
// printing parse table
printf("\n");
printf("\n\t**************** LL(1) PARSING TABLE *******************\n");
printf("\t--------------------------------------------------------\n");
printf("%-10s", "");
for (i = 0; i < TSIZE; ++i) {
if (terminal[i]) printf("%-10c", i);
}
printf("\n");
int p = 0;
for (i = 0; i < no_pro; ++i) {
if (i != 0 && (pro[i].str[0] != pro[i - 1].str[0]))
p = p + 1;
for (j = 0; j < TSIZE; ++j) {
if (first_rhs[i][j] && j != '^') {
table[p][j] = i + 1;
}
else if (first_rhs[i]['^']) {
for (k = 0; k < TSIZE; ++k) {
if (follow[pro[i].str[0] - 'A'][k]) {
table[p][k] = i + 1;
}
}
}
}
}
k = 0;
for (i = 0; i < no_pro; ++i) {
if (i == 0 || (pro[i - 1].str[0] != pro[i].str[0])) {
printf("%-10c", pro[i].str[0]);
for (j = 0; j < TSIZE; ++j) {
if (table[k][j]) {
printf("%-10s", pro[table[k][j] - 1].str);
}
else if (terminal[j]) {
printf("%-10s", "");
}
}
++k;
printf("\n");
}
}
}
```