

Théorie des langages de programmation







Zoé

517261910035

Théorie des langages de programmation

- 0 Introduction
- 1 Projet 1: Analyse lexicale
 - 1.1 Idées
 - 1.2 Exécution
 - 1.3 Résultats
- 2 Projet 2: Analyse syntaxique
 - 2.1 Idées
 - 2.1.1 Analyse syntaxique avec la méthode BNF
 - 2.1.2 Analyse syntaxique avec automate à pile
 - 2.2 Exécution
 - 2.3 Résultats
- 3 Projet 3: Interprétation
 - 3.1 Idées
 - 3.2 Exécution
 - 3.3 Résultats
- 4 Projet 4: Compilation et Exécution
 - 4.1 Idées
 - 4.1.1 Ajouter dans le tableau de LAC et de VM
 - 4.1.2 Compilation
 - 4.2 Exécution
 - 4.3 Résultats
- 5 Projet 5: Compilation d'une fonction L_{AC}
 - 5.1 Idées
 - 5.1.1 Compilation d'une structure conditionnelle
 - 5.1.2 Traitement des chaîne de caractère
 - 5.1.3 Compilation d'une déclaration
 - 5.2 Exécution
 - 5.3 Résultats
 - 5.3.1 Compilation d'une structure conditionnelle
 - 5.3.2 Traitement des chaîne de caractère
 - 5.3.3 Compilation d'une déclaration
 - 5.3.3.1 La fonction factorielle
 - 5.3.3.2 La fonction de u

0 Introduction

<input type="checkbox"/> 名称	修改日期	类型	大小
<input type="checkbox"/>  Projet1	2021/1/4 7:43	文件夹	
 Projet2	2021/1/4 7:43	文件夹	
 Projet3	2021/1/4 7:43	文件夹	
 Projet4	2021/1/4 7:43	文件夹	
 Projet5	2021/1/4 7:43	文件夹	
 utiles	2021/1/4 7:43	文件夹	

il y a 5 dossiers ici, dans chaque dossier, on applique `gcc document_nom.c -o nom` pour interpréter,

par exemple, dans le Projet1, on applique `gcc Projet1.c -o projet1`.

1 Projet 1: Analyse lexicale

1.1 Idées

Dans le projet 1, on fait l'analyse lexicale de langage de AC. Il y a 4 types de phases dans le langage AC:

- Les commentaires
- Les entiers naturels (on les note par (N))
- Les chaînes de caractères (on les note par (S))
- Les mots (on les note par (M))

Dans l'énoncé, les commentaires sont commencés par les caractères `'\'` ou commencés par les caractères `('` et terminés par `)'`, donc on peut écrire leur regex expression de commentaire en C et les enlever premièrement. Ensuite, les chaînes de caractère sont commencés par `' '` et terminés par `' '`, donc il est aussi facile d'écrire leur regex expression et les trouver et noter. Après, on peut utiliser `' '` pour séparer tous les identificateurs, la regex expression des entiers sont facile à écrire, donc pour chaque identificateur, on le apparie par la regex expression des entiers, s'il a bien concordé, on le note entier, si non, on les note mot.

1.2 Exécution

Dans notre algorithme, pour que le résultat finale soit dans bon ordre, tout d'abord, on apparie les commentaire, dans la fonction d'appariage de commentaires, on apparie les chaîne de caractère et dans la fonction d'appariage de chaîne de caractère, on apparie les identificateurs et on vérifie s'il est un entier.

Les regex expression sont écrites ci-dessous:

```
#define REGEX_COMMENT "(^|[\n| ])\(\( [^\\)]*?\\)|(^|[\n| ])\\\"\\\"[^\n]*\"
#define REGEX_STRING "(^|[\n| ])\\" [^"]*?"
#define REGEX_IDENTIFIER "[^ \n\\\"\\\\\\\\\\(\\)]+"
#define REGEX_INTEGER "(\\"-|\\+)?[0-9]+(\\".[0-9]+)?[ \n]"
```

On définit un `struct` pour stocker la position commencée, la position terminée et le type du mot.

```
typedef struct lexeme_t {
    char *type;
    long lstart;
    long lEnd;
} lexeme_t;
```

La fonction principale est la fonction `find_lac`.

```
int find_lac(char *Readbuffer, char *ReadbufferFin, queue_t *FinalQue) {
    // Definition of 4 regex parser
    regex_t MyComment;
    regex_t MyString;
    regex_t MyIdentifier;
```

```

regex_t MyInteger;
if(regcomp(&MyComment, REGEX_COMMENT, REG_EXTENDED)) exit(2);
if(regcomp(&MyString, REGEX_STRING, REG_EXTENDED)) exit(2);
if(regcomp(&MyIdentifier, REGEX_IDENTIFIER, REG_EXTENDED)) exit(2);
if(regcomp(&MyInteger, REGEX_INTEGER, REG_EXTENDED)) exit(2);
// if failed, get out.

int res = 0;
regmatch_t pMatch[10]; // match results
int MyCommentStart;
int MyCommentEnd;
char *pCommentBegin = Readbuffer;
char *pStringBegin = NULL;
char *pStringFin = NULL;

while (1) {
    // match comments
    res = regexec(&MyComment, pCommentBegin, 1, pMatch, 0);
    if (res == REG_NOMATCH) { // the last comment is extracted
        pStringBegin = pCommentBegin;
        pStringFin = ReadbufferFin;
        // match string from the rest
        find_string(Readbuffer, pStringBegin, pStringFin, &MyString,
&MyIdentifier, &MyInteger, FinalQue);
        break;
    } else {
        MyCommentStart = pMatch[0].rm_so;
        MyCommentEnd = pMatch[0].rm_eo;
        pStringBegin = pCommentBegin;
        pStringFin = pCommentBegin + MyCommentStart;

        char mot[500];
        char* p = mot;
        copy_mid(mot, Readbuffer, MyCommentEnd-MyCommentStart, MyCommentStart);

        // match string in between
        find_string(Readbuffer, pStringBegin, pStringFin, &MyString,
&MyIdentifier, &MyInteger, FinalQue);

        // Move to the next comment
        pCommentBegin += MyCommentEnd;
    }
}

// free
regfree(&MyComment);
regfree(&MyString);
regfree(&MyIdentifier);

return 1;
}

```

Dans la fonction `find_lac`, on trouve la position de commentaire et entre les commentaire on apparie les chaîne de caractère, donc on exécute `find_string`.

```

int find_string(const char *Readbuffer, const char *pStringBegin, const char
*pStringFin,
                regex_t *MyString, regex_t *MyIdentifier, regex_t *MyInteger,
queue_t *FinalQue) {
    int res;
    int offset;
    regmatch_t pMatch[10];
    int MyStringStart;
    int MyStringEnd;
    const char *pIdenBegin = NULL;
    const char *pIdenFin = NULL;

    lexeme_t solu;
    solu.type = (char*)"string";

    while (1) {
        res = regexec(MyString, pStringBegin, 1, pMatch, 0);
        if (res == REG_NOMATCH) {
            // last string is matched, try match identifier in the rest of
characters
            pIdenBegin = pStringBegin;
            pIdenFin = (char *) pStringFin;
            find_iden(Readbuffer, pIdenBegin, pIdenFin, MyIdentifier, MyInteger,
FinalQue);
            break;
        } else {
            MyStringStart = pMatch[0].rm_so;
            MyStringEnd = pMatch[0].rm_eo;
            // out of boundary, try match identifier
            if ((pStringBegin + MyStringEnd) > pStringFin) {
                pIdenBegin = pStringBegin;
                pIdenFin = (char *) pStringFin;
                find_iden(Readbuffer, pIdenBegin, pIdenFin, MyIdentifier,
MyInteger, FinalQue);
                break;
            }
            // set limit
            pIdenBegin = pStringBegin;
            pIdenFin = pStringBegin + MyStringStart;

            offset = pStringBegin - Readbuffer;

            if(offset==0){
                char comp[10];
                copy_mid(comp, Readbuffer, 1, MyStringStart);

                if(strcmp(comp, "\"")==0){
                    solu.lStart = MyStringStart + offset + 2; // begin with "
remove "
                    solu.lEnd = MyStringEnd + offset - 1; // remove "
                }
                else{
                    solu.lStart = MyStringStart + offset + 3; // begin with "\s
remove "
                    solu.lEnd = MyStringEnd + offset - 1; // remove "
                }
            }
        }
    }
}

```

```

        else{
            //printf("string, offset:%d\n",offset);
            solu.lStart = MyStringStart + offset + 3; // remove "
            solu.lEnd = MyStringEnd + offset - 1; // remove "
        }

        // match identifieur
        find_iden(Readbuffer, pIdenBegin, pIdenFin, MyIdentifier, MyInteger,
FinalQue);

        // put into queue
        cqueue_push_back(FinalQue, (void *) &solu, sizeof(solu));
        pStringBegin += MyStringEnd;
    }
}
return 1;
}

```

Dans la fonction `find_lac`, on trouve la position de chaîne de caractère et entre la chaîne de caractère, on apparie les identificateur, donc on exécute `find_iden`.

```

int find_iden(const char *Readbuffer, const char *pIdenBegin, const char
*pIdenFin,
            regex_t *MyIdentifier, regex_t *MyInteger, queue_t *FinalQue) {
    int res;
    int offset;
    regmatch_t pMatch[10];
    int MyIdenStart;
    int MyidenEnd;

    lexeme_t solu;
    solu.type = (char*)"word";

    while (1) {
        res = regexec(MyIdentifier, pIdenBegin, 1, pMatch, 0);
        if (res == REG_NOMATCH) { // No match
            break;
        } else {
            MyIdenStart = pMatch[0].rm_so;
            MyidenEnd = pMatch[0].rm_eo;

            // out of range, break
            if (pIdenBegin + MyidenEnd > pIdenFin) {
                break;
            }

            offset = pIdenBegin - Readbuffer;
            solu.lStart = MyIdenStart + offset;
            solu.lEnd = MyidenEnd + offset;

            // identifier number
            int IdenN=isInteger(Readbuffer + solu.lStart, Readbuffer +
solu.lEnd, MyInteger);
            if (IdenN==1) {
                solu.type = (char*)"number";
            }
            else {

```

```

        solu.type = (char*)"word";
    }

    // put in the queue
    cqueue_push_back(FinalQue, (void *) &solu, sizeof(solu));
    pIdenBegin += MyidenEnd;

}
}
return 1;
}

```

Après on trouve la position de l'identificateur, on vérifie s'il est nombre en utilisant la fonction `isInteger`

```

int isInteger(const char *pBegin, const char *pFin, regex_t *MyInteger) {
    int res;
    regmatch_t pMatch[10];
    int iEnd;

    res = regexec(MyInteger, pBegin, 1, pMatch, 0);
    if (res == REG_NOMATCH) {
        return -1;
    }
    else {
        iEnd = pMatch[0].rm_eo;
        //out of range
        if (pBegin + iEnd - 1 > pFin) {
            return -1;
        } else {
            return 1;
        }
    }
}

```

1.3 Résultats

On teste notre fonction `Projet1.c` sur le document `factorielle.lac`, le document est présenté dans la Figure 1-1:

```

1  \ Fichier "factorielle.lac"
2  ( Ce fichier est un "exemple" étudié pour tester
3  l'analyseur lexical écrit en phase 1 du projet)
4  : fact ( n -- n!)
5      dup 0= |
6      if
7      |     drop 1
8      else
9      |     dup 1- recurse *
10     then ;
11
12 : go ( n -- )
13     " Factorielle" count type
14     dup .
15     " vaut :
16     " count type
17     fact . cr ;
18
19 6 go
20

```

Figure 1-1: Le contenu du document 'factorielle.lac'

Et le résultat est:

```
zoe@ubuntu:~/Desktop/PLT/Projet1$ gcc Projet1.c -o projet1
zoe@ubuntu:~/Desktop/PLT/Projet1$ ./projet1
M(":")->M("fact")->M("dup")->M("0=")->M("if")->M("drop")->N("1")->M("e
lse")->M("dup")->M("1-")->M("recurse")->M("*")->M("then")->M(";")->M("
:")->M("go")->S("Factorielle")->M("count")->M("type")->M("dup")->M(".")
->S("vaut :
")->M("count")->M("type")->M("fact")->M(".")->M("cr")->M(";")->N("6")-
>M("go")->
```

Figure 1-2: Le résultat de la fonction d'analyse lexicale

On a bien le résultat.

2 Projet 2: Analyse syntaxique

2.1 Idées

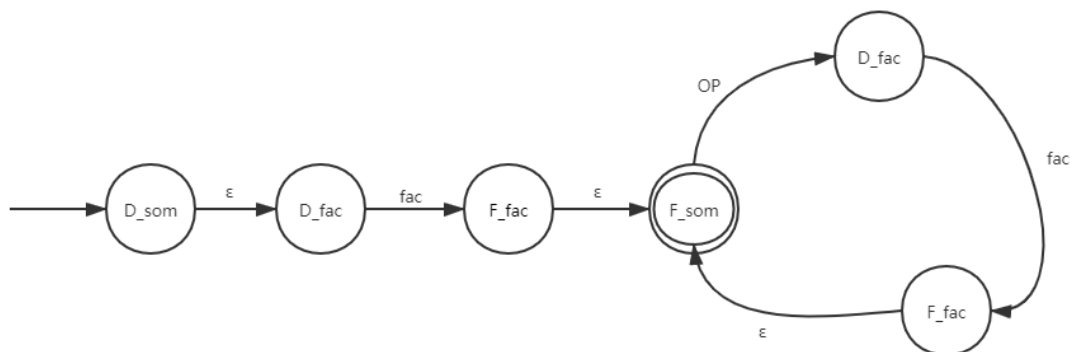
2.1.1 Analyse syntaxique avec la méthode BNF

on fait une analyse syntaxique par un BNF:

```
1 <chiffre> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
2 <naturel> ::= <chiffre> | <chiffre> <naturel>
3 <entier> ::= <naturel> | "-" <naturel>
4 <opérateur> ::= "+" | "-" | "x" | "/"
5 <facteur> ::= <entier> | "(" <facteur> ")"
6 <somme> ::= <facteur> | <somme> <opérateur> <facteur>
```

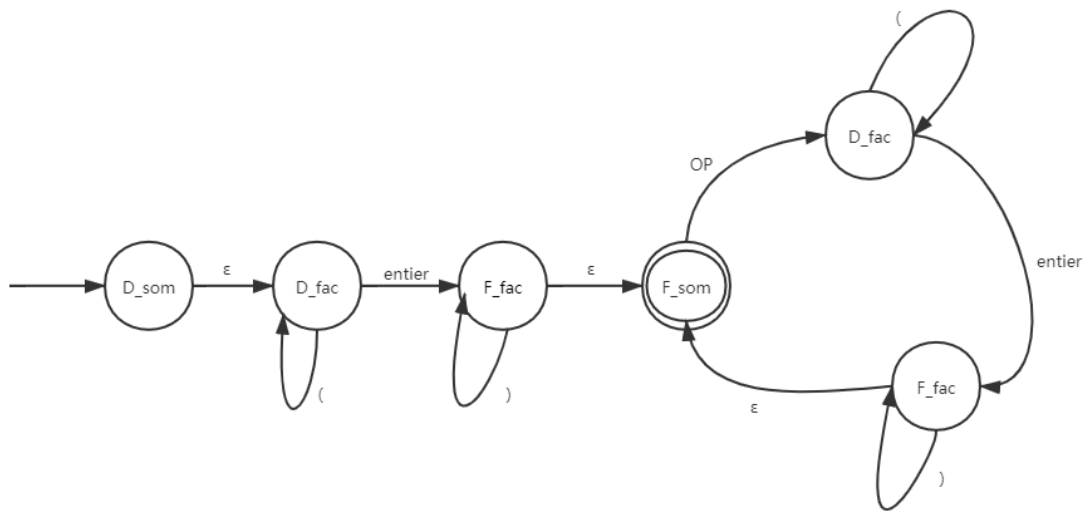
2.1.2 Analyse syntaxique avec automate à pile

Le BNF peut être transformé en un automate à pile. On le construit de haut en bas. D'abord, on applique la règle du `somme`:

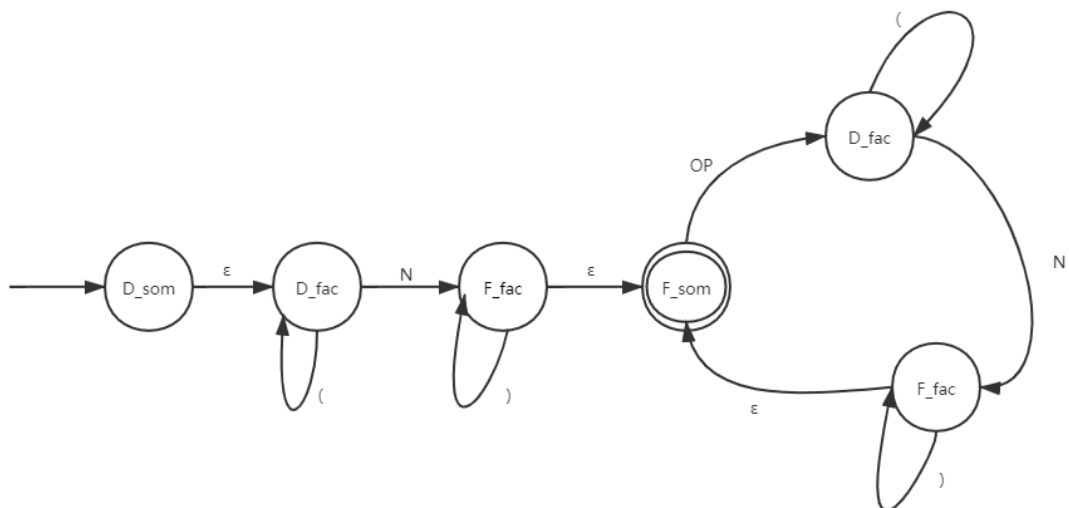


Où **D** et **F** sont respectivement les abréviations de début et fin.

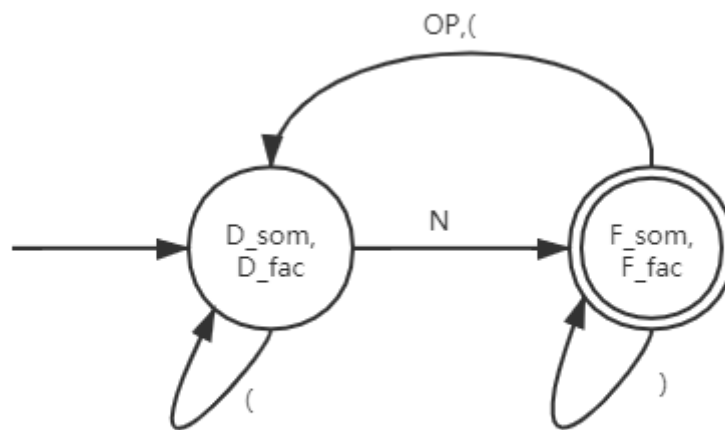
Après, on applique les règles de `facteur`:



Les entiers sont le type de `NOMBRE` dans notre algorithme, donc on peut avoir:



Finalement, on transforme l'automate fini non déterministe en automate fini déterministe:



2.2 Exécution

Tout d'abord, on fait l'analyse syntaxique avec la méthode BNF, si la fonction est syntaxiquement correcte, on transforme la notation infixée en notation postfixée par une pile, ici, on n'utilise pas arbre pour la transformer.

La fonction pour trouver si la fonction est syntaxiquement correcte est ce qui dans le moodle, on juste modifie un peu pour noter le nombre négative dans le type de entier.

```

int synta(char *s){
    int i = 0, j = 0; // i est l'indice de parcours de s, j est le nombre de
chiffres lus
    resultat.N = 0;
    lexeme lu;
    .
    .
    .
    do // L'analyseur lexical proprement dit
        switch(s[i])
        {
            case '0': case '1': case '2': case '3': case '4': case '5': case
'6': case '7': case '8': case '9':
                j++;
                break;
            case '-': // if we find the negative number, we add dans le type
number.
                if (i!=0 &&s[i-1]=='('){
                    int begin = i;
                    while(isdigit(s[i+1])){
                        j++;i++;
                        if (s[i+1]=='\0') break;
                    }
                    if(begin==i){
                        add_divers(OP);break;
                    }
                }
                else{
                    i++;j+=1;
                    add_nombre();
                }
            }
        }
    }
}

```

```

        i--;j=0;
        break;
    }
}
else{
    add_divers(OP);break;
}
case '+': case '*': case '/':
    add_divers(OP);break;
case '(':
    add_divers(Par_ou);break;
case ')':
    add_divers(Par_fe);break;
default:
    printf("Erreur !");exit(1);
}
while (s[++i] != 0);

// La ligne peut se terminer par un nombre qui n'a pas été encore pris en
compte
add_nombre();

if (resultat.N>1 && strcmp(resultat.tab[0].valeur,"-")==0){
    printf("\n\nL'expression %s est syntaxiquement %s", s, somme(1,
resultat.N-1) ? "correcte\n" : "incorrecte\n");
    return somme(1, resultat.N-1);
}
else{
    printf("\n\nL'expression %s est syntaxiquement %s", s, somme(0,
resultat.N-1) ? "correcte\n" : "incorrecte\n");
    return somme(0, resultat.N-1);
}
}

```

on stocker le résultat dans `resultat`, après, on utilise la pile pour transformer la notation infixée en notation postfixée. On l'appelle `exp_to_postfix`.

```

// convert in-fix expression to post-fix
char *exp_to_postfix(tllexeme resultat){

    Cstack *s = cstack_init();
    char e;
    char *res=malloc(100*sizeof(char));
    char *p = res;
    int len_res = 0;

    if (strcmp(resultat.tab[0].valeur,"-")==0){ // if the first element '-', we
add 0 in the begin
        *(p++) = '0';
        len_res++;
        *(p++) = ' ';
        len_res++;
    }

    for(int i=0;i<resultat.N;i++){ // add number to the array
        if(resultat.tab[i].type==NOMBRE){

```

```

        int num_len=strlen(resultat.tab[i].valeur);
        for(int j=0;j<num_len;j++){
            *(p++)=resultat.tab[i].valeur[j];
            len_res++;
        }
        *(p++) = ' ';
        len_res++;
    }
    else if (strcmp(resultat.tab[i].valeur,"")==0){
        e = cstack_pop(s);
        while( '(' != e ){
            *(p++) = e;
            len_res++;
            *(p++) = ' ';
            len_res++;
            e = cstack_pop(s);
        }
    }
    else if ( strcmp(resultat.tab[i].valeur,"+")==0 ||
strcmp(resultat.tab[i].valeur,"-")==0){
        if ( !s->size ){ // if the stack is vide, we push the operator into
stack.

            cstack_push(s,resultat.tab[i].valeur[0]);
        }
        else{
            do{ // if the stack is not vide
                e = cstack_pop(s);
                if ( '('==e ){ // if the first element is '(', we do
nothing.

                    cstack_push(s,e);
                }
                else{ // otherwise, we add the element to the array
                    *(p++) = e;
                    len_res++;
                    *(p++) = ' ';
                    len_res++;
                }
            }while( s->size && '('!=e );
            cstack_push(s,resultat.tab[i].valeur[0]);
        }
    }

    else if ( strcmp(resultat.tab[i].valeur,"*")==0 ||
strcmp(resultat.tab[i].valeur,"/")==0 || strcmp(resultat.tab[i].valeur,"(")==0 )
{
        cstack_push(s,resultat.tab[i].valeur[0]); // push operator to the
stack

    }

}

while( s->size){ // add the rest of operand to the array
    e = cstack_pop(s);
    *(p++) = e;
    len_res++;
    *(p++) = ' ';
    len_res++;
}
res[len_res] = '\0';
return res;

```

```
}
```

2.3 Résultats

On teste notre fonction `Projet2.c` sur le document `test_projet2.lac`, le document est présenté ci-dessous:

```
1 " -(1-2)+(3-4)*(-5)" calculate .
2 " 1+2+3-2*8/4-9" calculate .
3 " -(-0008)" calculate .
4 " 2+3-" calculate .
5
```

Figure 2-1: Le contenu du document 'test_projet2.lac'

Et le résultat est:

```
zoe@ubuntu:~/Desktop/PLT/Projet2$ gcc Projet2.c -o projet2
zoe@ubuntu:~/Desktop/PLT/Projet2$ ./projet2
S("-(1-2)+(3-4)*(-5)")->M("calculate")->M(".")->S("1+2+3-2*8/4-9")->M(
"calculate")->M(".")->S("-(-0008)")->M("calculate")->M(".")->S("2+3-")
->M("calculate")->M(".")->

L'expression -(1-2)+(3-4)*(-5) est syntaxiquement correcte
exp_post: 0 1 2 - - 3 4 - -5 * +
6

L'expression 1+2+3-2*8/4-9 est syntaxiquement correcte
exp_post: 1 2 + 3 + 2 8 4 / * - 9 -
-7

L'expression -(-0008) est syntaxiquement correcte
exp_post: 0 -0008 -
8

L'expression 2+3- est syntaxiquement incorrecte
```

Figure 2-1: Le résultat du calculateur

On a bien le résultat.

3 Projet 3: Interprétation

3.1 Idées

Pour interpréter, Tout d'abord, on définit les fonction de base ,après, on met le nom de la fonction de base dans le tableau de symbole et met les fonctions dans le tableau de processeur, après on met les positions des fonctions dans le processeur à la machine virtuelle. La règle d'ajouter dans le tableau de symbole est : Premièrement, on ajoute la longueur du mot, après on ajoute son caractère de nom un par un, et puis, on ajoute le Cfa de la fonction qui représente la

position dans la machine virtuelle, enfin, on ajoute le Nfa de ce mot qui représente la position commencé du nom.

Un exemple est présenté dans la figure 3-1.

Numéro	0	1	2	3	4	5	6	7	8	9	10	11
LAC	0	1	43	0	1	4	115	119	97	112	2	5
Signification	rien	len(+)	+	Cfa(+)	Nfa(+)	len(swap)	s	w	a	p	Cfa(swap)	Nfa(swap)
Numéro	0	1	2	3								
VM	0	[0]	0	[1]							Cfa	
Signification	base	+	base	swap							Nfa	

Figure 3-1: Un exemple d'ajouter le tableau de LAC et VM

3.2 Exécution

Tout d'abord, on crée le tableau de symbole noté `LAC`, la machine virtuelle noté `VM` et le tableau dans le tableau de processeur noté `processeur`

```
#define longueur 500
int LAC[longueur]={0}; // symbol table
int pointer_LAC=1;

int VM[longueur]={0}; // VM table
int pointer_VM=0;

typedef void (*base)(void); // processor
base processeur[50];
int pointer_processeur=0;
```

Après, on définit les fonction de base, par exemple, les fonction *dup*, *drop*, *+*, *** et .

```
void func_dup(){
    Elementype copy = cstack_gettop(s_donnee);
    cstack_push(s_donnee,copy);
}

void func_drop(){
    Elementype out;
    out = cstack_pop(s_donnee);
}

void func_plus(){
    Elementype e1 = cstack_pop(s_donnee);
    Elementype e2 = cstack_pop(s_donnee);
    cstack_push(s_donnee,e2+e1);
}

void func_multiply(){
    Elementype e1 = cstack_pop(s_donnee);
    Elementype e2 = cstack_pop(s_donnee);
    cstack_push(s_donnee,e1*e2);
}
```

```

void func_point(){
    Elemtyp e1 = cstack_pop(s_donnee);
    printf("%d\n",e1);
}

```

Ensuite, on écrit la fonction `declare_base`

```

void declare_base(char nom[],void *function_nom){
    // add to LAC
    int PBegin = pointer_LAC; // Nfa
    size_t len = strlen(nom); // length
    LAC[pointer_LAC++] = len;
    for(int i=0;i<len;i++){ // add chaque character
        LAC[pointer_LAC++] = nom[i];
    }
    LAC[pointer_LAC++] = pointer_VM;
    LAC[pointer_LAC++] = PBegin;

    // add to VM
    VM[pointer_VM++]=0; // syndicate base function
    VM[pointer_VM++]=pointer_processeur;
    // add to processor
    processeur[pointer_processeur++]=function_nom;
}

```

Après, pour exécution, tout d'abord, on utilise la fonction du projet 1 pour faire analyse lexicale, quand on trouve les nombres, on les met dans le pile de donnée, quand on trouve le mot, on vérifier si la fonction trouvée est existante en utilisant le tableau de symbole, si la fonction est définit, on l'exécute, sinon, on lève un erreur.

```

void execution(char * psReadBuffer, CQueue * pqueRes, CStack *stack){
    QueueNode * tmp = pqueRes->pHead;
    lexeme_t res;
    int flag = FALSE;
    while (TRUE) {
        res = *(lexeme_t *)tmp->pData;
        char ele[50]; // copy into a char.
        copy_mid(ele,psReadBuffer,(int)(res.lEnd-res.lStart),res.lStart);

        if (strcmp(res.type,"number")==0){ // if number, push in the statistic
stack
            cstack_push(stack,atoi(ele));
        }
        else if(strcmp(res.type,"word")==0){ //if word, find if it exists.
            int VM_value=find_func(ele);
            if(VM_value== -1) { // not exist, exit.
                printf("Function not exist\n");
                exit(1);
            }
            else{
                if(VM[VM_value]==0){ // exist, execute.
                    processeur[VM[VM_value+1]]();
                }
            }
        }
    }
}

```

```

        tmp = tmp->pNext;
        if (flag == TRUE) break;
        if (tmp == tmp->pNext) {
            flag = TRUE;
        }
    }
    printf("\n");
}

```

La fonction pour trouver si la fonction soit bien défini est appelé `find_func`.

```

int find_func(char *exp){
    int pointer_now=pointer_LAC-1; // every index of position Nfa
    while(pointer_now>0){ // while not the first one
        pointer_now = LAC[pointer_now];
        if (strlen(exp)==LAC[pointer_now]){ // compare the string length
            int tmp = pointer_now+1;
            int flag=1;
            char *p=exp;
            while(*p!='\0'){ // compare each character
                if(*p!=LAC[tmp]){
                    flag=0;
                    break;
                }
                tmp++; p++;
            }
            if (flag==1){ // successfully find, return the Cfa
                int VM_value=LAC[pointer_now+LAC[pointer_now]+1];
                return VM_value;
            }
        }
        pointer_now--;
    }
    return -1;
}

```

3.3 Résultats

On ajoute les fonction de base dans le tableau de symbole et la machine virtuelle.

```

int main(){

    //establish pile
    s_donnee=cstack_init();
    s_retour=cstack_init();
    char *nom[10]={"+", "*", "-", "dup", "drop", "swap", "count", ".", "type", "="};
    void *function_list[10] =
{func_plus, func_multiply, func_minus, func_dup, func_drop,

func_swap, func_count, func_point, func_type, func_equal};

    for(int i=0; i<10; i++){
        declare_base(nom[i], function_list[i]);
    }
    return 0;
}

```

le résultat est présenté dans la figure 3-2 et 3-3.

```

----- LAC -----
position: 1, value: 1, signification: len
position: 2, value: 43, signification: +
position: 3, value: 0, signification: Cfa
position: 4, value: 1, signification: Nfa
position: 5, value: 1, signification: len
position: 6, value: 42, signification: *
position: 7, value: 2, signification: Nfa
position: 8, value: 5, signification: Cfa
position: 9, value: 1, signification: len
position: 10, value: 45, signification: -
position: 11, value: 4, signification: Cfa
position: 12, value: 9, signification: Nfa
position: 13, value: 3, signification: len
position: 14, value: 100, signification: d
position: 15, value: 117, signification: Nfa
position: 16, value: 112, signification: p
position: 17, value: 6, signification: Cfa
position: 18, value: 13, signification: Nfa
position: 19, value: 4, signification: len
position: 20, value: 100, signification: d
position: 21, value: 114, signification: r
position: 22, value: 111, signification: o
position: 23, value: 112, signification: p
position: 24, value: 8, signification: Cfa
position: 25, value: 19, signification: Nfa
position: 26, value: 4, signification: len
position: 27, value: 115, signification: s
position: 28, value: 119, signification: w
position: 29, value: 97, signification: a
position: 30, value: 112, signification: p
position: 31, value: 10, signification: Cfa
position: 32, value: 26, signification: Nfa
position: 33, value: 5, signification: len
position: 34, value: 99, signification: c
position: 35, value: 111, signification: o
position: 36, value: 117, signification: u
position: 37, value: 110, signification: n
position: 38, value: 116, signification: t
position: 39, value: 12, signification: Cfa
position: 40, value: 33, signification: Nfa
position: 41, value: 1, signification: len
position: 42, value: 46, signification: .
position: 43, value: 14, signification: Cfa
position: 44, value: 41, signification: Nfa
position: 45, value: 4, signification: len
position: 46, value: 116, signification: t
position: 47, value: 121, signification: y
position: 48, value: 112, signification: p
position: 49, value: 101, signification: e
position: 50, value: 16, signification: Cfa
position: 51, value: 45, signification: Nfa
position: 52, value: 1, signification: len
position: 53, value: 61, signification: =
position: 54, value: 18, signification: Cfa
position: 55, value: 52, signification: Nfa

```

Figure 3-2 : Le résultat de LAC

```

----- VM -----
position: 0, value: 0, signification: fonction de base
position: 1, value: 0, signification: +
position: 2, value: 0, signification: fonction de base
position: 3, value: 1, signification: *
position: 4, value: 0, signification: fonction de base
position: 5, value: 2, signification: -
position: 6, value: 0, signification: fonction de base
position: 7, value: 3, signification: dup
position: 8, value: 0, signification: fonction de base
position: 9, value: 4, signification: drop
position: 10, value: 0, signification: fonction de base
position: 11, value: 5, signification: swap
position: 12, value: 0, signification: fonction de base
position: 13, value: 6, signification: count
position: 14, value: 0, signification: fonction de base
position: 15, value: 7, signification: .
position: 16, value: 0, signification: fonction de base
position: 17, value: 8, signification: type
position: 18, value: 0, signification: fonction de base
position: 19, value: 9, signification: =

```

Figure 3-3 : Le résultat de VM

Après, on teste notre algorithme sur le document `test_projet3.lac`, ce document contient deux lignes, tout les fonctions dans la première ligne sont définies, il faut calculer le résultat. Pour la deuxième ligne, la fonction `$` n'est pas défini, donc il faut lever un erreur.

1	2 3 4 + * .
2	2 3 4 5 + \$

Figure 3-4 : Le contenu du document 'test_projet3.lac'


```
zoe@ubuntu:~/Desktop/PLT/Projet3$ gcc Projet3.c -o projet3
zoe@ubuntu:~/Desktop/PLT/Projet3$ ./projet3
N("2")->N("3")->N("4")->M("+")->M("*")->M(".")->N("2")->N("3")->N("4")
->N("5")->M("+")->M("$")->
14
Function not exist
```

Figure 3-5 : Le resultat

4 Projet 4: Compilation et Exécution

4.1 Idées

4.1.1 Ajouter dans le tableau de LAC et de VM

Après définir la fonction de base, on va ajouter des fonctions de LAC qui sont composées par les fonctions de base. Premièrement, on ajoute le nom de la fonction dans le tableau de symbole comme le façon d'ajouter la fonction de base, après on ajoute la fonction de lac dans la machine virtuelle en utilisant les Cfa de fonction de base.

Tous les définition de la fonction est commercé par le caractère `:` et terminé par le caractère `;`, le premier mot après le caractère `:` est sa nom, le suivant est sa définition. On ajoute la fonction dans VM mot par mot, quand on lit un nombre dans la fonction, on ajoute un `<lit>` (fonction de base) avant ce nombre, à la fin de la fonction de base on ajoute un `<fin>` (fonction de base).

Example :

Si on va ajouter les fonctions:

```
: incr 1 + ;
: 2+. incr incr . :
```

le nom de la première fonction et deuxième fonction est `incr` et `2+`, et ses définitions sont le suivant. On doit former un tableau de symbole et un tableau de VM présenté dans la figure 4-1.

Numéro	0	1	2	3	4	5	6	7	8	9	10	11	12
LAC	0	1	43	0	1	1	46	2	5	5	60	108	105
Signification	rien	len(+)	+	Cfa(+)	Nfa(+)	len(.)	.	Cfa(.)	Nfa(.)	len(<lit>)	<	l	i
Numéro	13	14	15	16	17	18	19	20	21	22	23	24	25
LAC	116	62	4	9	5	60	102	105	110	62	6	17	4
Signification	t	>	Cfa(<lit>)	Nfa(<lit>)	len(<fin>)	<	f	i	n	>	Cfa(<fin>)	Nfa(<fin>)	len(incr)
Numéro	26	27	28	29	30	31	32	33	34	35	36	37	
LAC	105	110	99	114	8	25	3	50	43	46	13	32	
Signification	i	n	c	r	Cfa(incr)	Nfa(incr)	len(2+.)	2	+	.	Cfa(2+.)	Nfa(2+.)	
Numéro	0	1	2	3	4	5	6	7					
VM	0	[0]	0	[1]	0	[2]	0	[3]					
Signification	base	+	base	.	base	<lit>	base	<fin>					
Numéro	8	9	10	11	12	13	14	15	16	17			
VM	1	4	1	0	6	1	8	8	2	6			
Signification	LAC	(<lit>)	nombre	(+)	(<fin>)	LAC	(incr)	(incr)	(.)	(<fin>)			
				incr				2+.					

Figure 4-1 : Le tableau de LAC et VM pour la fonction de LAC

4.1.2 Compilation

Avec le tableau de VM et le tableau de LAC, on peut compiler une fonction, quand on voit un mot, on lit le tableau de LAC, si le nom de fonction n'est pas dans LAC, on lève un erreur. Si le nom de fonction est dans LAC, on trouve sa Cfa et on utilise le tableau de VM pour exécuter:

- Si le VM[Cfa]=0, la fonction est fonction de base, on exécute la fonction dans la position VM[Cfa+1].
- Si le VM[Cfa]=1, la fonction est fonction de LAC, on empile Cfa+1 dans la pile de retour et on exécute VM[Cfa+1].
- Quand la pile de retour n'est pas vide, on dépile la pile de retour la valeur z, ensuite on empile z+1 dans la pile de retour et on exécute VM[Z+1]. On répète ce procédure juste qu'à la pile de retour est vide.

4.2 Exécution

Les fonctions de base et sont définies ci-dessous:

```
void func_lit(){
    Elemtyp iAddr = cstack_pop(s_retour);
    iAddr+=1;
    cstack_push(s_donnee, VM[iAddr]);
    cstack_push(s_retour, iAddr);
}

void func_fin(){
    cstack_pop(s_retour);
}
```

Si on lit le mot <lit> on dépile la pile de retour (iAddr) et empile la iAddr+1 dans la pile de retour, et puis, on empile VM[iAddr+1] dans la pile de donnée, car élément après <lit> est toujours un nombre. Si on lit le mot <fin>, on dépile la pile de retour.

Ensuite, on ajoute le nom de fonction dans le tableau de symbole et la fonction dans la machine virtuelle en utilisant la fonction `declare_LAC`

```
void declare_LAC(char *function[], int length){
    // add to LAC
    int PBegin = pointer_LAC;
    size_t len = strlen(function[0]); //first one is the name of function
    LAC[pointer_LAC++] = len;
    for(int i=0; i<len; i++){
        LAC[pointer_LAC++] = function[0][i];
    }
    LAC[pointer_LAC++] = pointer_VM;
    LAC[pointer_LAC++] = PBegin;

    //add to VM
    VM[pointer_VM++] = 1;
    for(int j=1; j<length; j++){
        int func=find_func(function[j]);
        printf("function nom: %s, find func: %d\n", function[j], func);
        VM[pointer_VM++] = func;
        if(strcmp(function[j], "<lit>")==0){ // the element behind <lit> is
always a number.
            j++;
        }
    }
}
```

```

        printf("digit, function nom: %s\n",function[j]);
        VM[pointer_VM++]=atoi(function[j]);
    }
}
}

```

Pour exécuter, on définit la fonction `execution`

```

void execution(int vm_index){
    if (vm_index==-1){
        printf("No Function");
        exit(1);
    }
    if(VM[vm_index]==0){
        processeur[VM[vm_index+1]]();
    }
    else if(VM[vm_index]==1){
        cstack_push(s_retour,vm_index+1);
        execution(VM[vm_index+1]);
        while(s_retour->size!=0){
            int z = cstack_pop(s_retour);
            cstack_push(s_retour,z+1);
            execution(VM[z+1]);
        }
    }
}
}

```

La fonction `compilation` est presque la même chose que la fonction `execution` dans **Projet 3**, on modifie un peu pour déclarer la fonction de LAC quand on lit le caractère `:`, on crée `char *function[200]` et on met la suite dans ce char jusqu'à on lit le caractère `;`, après, on les met dans la fonction `declare_LAC`.

```

void compilation(char * psReadBuffer, CQueue * pqueRes){
    QueueNode * tmp = pqueRes->pHead;
    lexeme_t res;
    int flag = TRUE;
    while (flag) {
        res = *(lexeme_t *)tmp->pData;
        char ele[50]; // copy into a char.
        copy_mid(ele,psReadBuffer,(int)(res.lEnd-res.lStart),res.lStart);
        if (strcmp(res.type,"word")==0){
            if(strcmp(ele,":")==0){ // when we see :, we begin to define a
function.
                int flag2 = TRUE;
                char *function[200];
                int len=0;
                while(flag2){
                    tmp = tmp->pNext;
                    lexeme_t function_LAC;
                    function_LAC = *(lexeme_t *)tmp->pData;
                    char mot[50];
                    copy_mid(mot,psReadBuffer,(int)(function_LAC.lEnd-
function_LAC.lStart),function_LAC.lStart);
                    if (strcmp(function_LAC.type,"number")==0){
                        function[len++] = "<lit>"; // see number add <lit>
before

```

```

        function[len]=malloc(strlen(mot)*sizeof(char));
        strcpy(function[len++],mot);
    }
    else if(strcmp(function_LAC.type,"string")==0){
        function[len++] = "<str>";
        function[len]=malloc(strlen(mot)*sizeof(char));
        strcpy(function[len++],mot);
    }
    else if(strcmp(function_LAC.type,"word")==0){
        if (strcmp(mot,";")==0){ //when we see ;, the function
terninate.

            function[len++] = "<fin>";
            flag2 = FALSE;
        }
        else{
            function[len]=malloc(strlen(mot)*sizeof(char));
            strcpy(function[len++],mot);
        }
    }
}
declare_LAC(function,len); // add function LAC into LAC and VM
}
else{
    int VM_index=find_func(ele);
    execution(VM_index);
}
}
else if(strcmp(res.type,"number")==0){ // if number, push in the
statistic stack
    cstack_push(s_donnee,atoi(ele));
}
if (tmp == tmp->pNext) {
    flag = FALSE;
}
else tmp = tmp->pNext;
}
printf("\n");
}

```

4.3 Résultats

On utilise le document `test_projet4.lac` pour tester notre algorithme.

```

1  : incr 1 + ;
2  : 2+. incr incr . ;
3  5 2+.
4

```

Figure 4-2 : Le contenu de document test_projet4.lac

Après exécuter les deux premier lignes, le résultat est présenté dans la figure 4-3 et 4-4.

```

-----LAC-----
position: 0, value: 0, signification: fini
position: 1, value: 1, signification: len
position: 2, value: 43, signification: +
position: 3, value: 0, signification: Cfa
position: 4, value: 1, signification: Nfa
position: 5, value: 1, signification: len
position: 6, value: 46, signification: .
position: 7, value: 2, signification: Cfa
position: 8, value: 5, signification: Nfa
position: 9, value: 5, signification: len
position: 10, value: 60, signification: <
position: 11, value: 108, signification: l
position: 12, value: 105, signification: i
position: 13, value: 116, signification: t
position: 14, value: 62, signification: >
position: 15, value: 4, signification: Cfa
position: 16, value: 9, signification: Nfa
position: 17, value: 5, signification: len
position: 18, value: 60, signification: <
position: 19, value: 102, signification: f
position: 20, value: 105, signification: i
position: 21, value: 110, signification: n
position: 22, value: 62, signification: >
position: 23, value: 6, signification: Cfa
position: 24, value: 17, signification: len
position: 25, value: 4, signification: len
position: 26, value: 105, signification: i
position: 27, value: 110, signification: n
position: 28, value: 99, signification: c
position: 29, value: 114, signification: r
position: 30, value: 8, signification: Cfa
position: 31, value: 25, signification: Nfa
position: 32, value: 3, signification: len
position: 33, value: 50, signification: 2
position: 34, value: 43, signification: +
position: 35, value: 46, signification: .
position: 36, value: 13, signification: Cfa
position: 37, value: 32, signification: Nfa

```

Figure 4-3 : Le resultat de LAC

```

-----VM-----
position: 0, value: 0
position: 1, value: 0
position: 2, value: 0
position: 3, value: 1
position: 4, value: 0
position: 5, value: 2
position: 6, value: 0
position: 7, value: 3
position: 8, value: 1
position: 9, value: 4
position: 10, value: 1
position: 11, value: 0
position: 12, value: 6
position: 13, value: 1
position: 14, value: 8
position: 15, value: 8
position: 16, value: 2
position: 17, value: 6

```

Figure 4-3 : Le resultat de VM

Ce qui correspond à ce qui on a besoin (Figure 4-1), donc le résultat est bon.

Après la compilation de la troisième ligne `5 2+.`, le résultat vaut 7, ce qui est aussi bon.

```

zoe@ubuntu:~/Desktop/PLT/Projet4$ gcc Projet4.c -o projet4
zoe@ubuntu:~/Desktop/PLT/Projet4$ ./projet4
M(":")->M("incr")->N("1")->M("+")->M(";")->M(":")->M("2+.")->M("incr")
->M("incr")->M(".")->M(";")->N("5")->M("2+.")->
7

```

5 Projet 5: Compilation d'une fonction L_{AC}

5.1 Idées

5.1.1 Compilation d'une structure conditionnelle

Pour compiler une structure conditionnelle, tout d'abord on ajoute les fonction de base `if`, `else` et `then` dans le tableau de symbole et la machine virtuelle, quand on définit une fonction qui va utiliser la structure conditionnelle, on doit ajouter le codage dans la machine virtuelle et ajouter le nom de la fonction dans le tableau de symbole. Le règle est, après la position de Cfa de

if, on doit ajouter l'index de Cfa de then , après la position de Cfa de else, on ajoute l'index de Cfa de then.

Exemple :

Si la fonction est défini par

```
: cond 0 = if 1 else 2 then ;
```

Le tableau de LAC et VM est présenté dans la figure 5-1.

Numéro	0	1	2	3	4	5	6	7	8	9	10	11	12
LAC	0	1	61	0	1	5	60	108	105	116	62	2	5
Signification	rien	len(=)	=	Cfa(=)	Nfa(=)	len(<lit>)	<	l	i	t	>	Cfa(<lit>)	Nfa(<lit>)
Numéro	13	14	15	16	17	18	19	20	21	22	23	24	25
LAC	5	60	102	105	110	62	4	13	2	105	102	6	21
Signification	len(<fin>)	<	f	i	n	>	Cfa(<fin>)	Nfa(<fin>)	len(if)	i	f	Cfa(if)	Nfa(if)
Numéro	26	27	28	29	30	31	32	33	34	35	36	37	38
LAC	4	101	108	115	101	8	26	4	116	104	101	110	10
Signification	len(else)	e	l	s	e	Cfa(else)	Nfa(else)	len(then)	t	h	e	n	Cfa(then)
Numéro	39	40	41	42	43	44	45	46					
LAC	33	4	99	111	110	100	12	40					
Signification	Nfa(then)	len(cond)	c	o	n	d	Cfa(cond)	Nfa(cond)					
Numéro	0	1	2	3	4	5	6	7	8	9	10	11	12
VM	0	[0]	0	[1]	0	[2]	0	[3]	0	[4]	0	[5]	1
Signification	base	=	base	<lit>	base	<fin>	base	if	base	else	base	then	LAC
Numéro	13	14	15	16	17	18	19	20	21	22	23	24	25
VM	2	0	0	6	21	2	1	8	24	2	2	10	4
Signification	(<lit>)	nombre	(=)	(if)		(<lit>)	nombre	(else)		(<lit>)	nombre	(then)	(<fin>)

Figure 5-1 : Un exemple de condition en machine virtuelle

5.1.2 Traitement des chaîne de caractère

Pour taper des caractères de la chaîne de caractère, on peut définir la fonction de base <str> , count et type . Dans la fonction, quand on lit une chaîne de caractère dans le document on ajoute un <str> avant dans la machine virtuelle, ensuite, on ajoute sa longueur et les caractères de élément un par un, à la fin, on ajoute <fin> .

Exemple :

Si la fonction est défini par

```
: essai "Bonjour" count type ;
```

Le tableau de LAC et VM est présenté dans la figure 5-2.

Numéro	0	1	2	3	4	5	6	7	8	9	10
LAC	0	5	99	111	117	110	116	0	1	4	116
Signification	rien	len(count)	c	o	u	n	t	Cfa(count)	Nfa(count)	len(type)	t
Numéro	11	12	13	14	15	16	17	18	19	20	21
LAC	121	112	101	2	9	5	60	102	105	110	62
Signification	y	p	e	Cfa(type)	Nfa(type)	len(<fin>)	<	f	i	n	>
Numéro	22	23	24	25	26	27	28	29	30	31	32
LAC	4	13	5	60	115	116	114	62	6	24	5
Signification	Cfa(<fin>)	Nfa(<fin>)	len(<str>)	<	s	t	r	>	Cfa(<str>)	Nfa(<str>)	len(essai)
Numéro	33	34	35	36	37	38	39				
LAC	101	115	115	97	105	8	32				
Signification	e	s	s	a	i	Cfa(essai)	Nfa(essai)				
Numéro	0	1	2	3	4	5	6	7	8	9	10
VM	0	[0]	0	[1]	0	[2]	0	[3]	1	6	7
Signification	base	count	base	type	base	<fin>	base	<str>	LAC	(<str>)	len(Bonjour)
Numéro	11	12	13	14	15	16	17	18	19	20	
VM	66	111	110	106	111	117	114	0	2	4	
Signification	B	o	n	j	o	u	r	(count)	(type)	(<fin>)	

Figure 5-2 : Un exemple de taper string en machine virtuelle

5.1.3 Compilation d'une déclaration

- On peut ajouter la fonction de base `recurse`, la fonction `recurse` rien fait, mais quand on lit le mot `recurse` dans la fonction, dans la position du VM, on ajoute le Cfa de sa fonction.

Exemple :

Si la fonction est défini par

```
: add 2 + recurse ;
```

Le tableau de LAC et VM est présenté dans la figure 5-3.

Numéro	0	1	2	3	4	5	6	7	8	9	10	11	12
LAC	0	1	43	0	1	7	114	101	99	117	114	115	101
Signification	rien	len(+)	+	Cfa(+)	Nfa(+)	len(recurse)	r	e	c	u	r	s	e
Numéro	13	14	15	16	17	18	19	20	21	22	23	24	
LAC	5	5	5	60	108	105	116	62	4	15	5	60	
Signification	Cfa(recurse)	Nfa(recurse)	len(<lit>)	<	l	i	t	>	Cfa(<lit>)	Nfa(<lit>)	len(<fin>)	<	
Numéro	25	26	27	28	29	30	31	32	33	34	35	36	
LAC	102	105	110	62	6	23	3	97	100	100	8	31	
Signification	f	i	n	>	Cfa(<fin>)	Nfa(<fin>)	len(add)	a	d	d	Cfa(add)	Nfa(add)	
Numéro	0	1	2	3	4	5	6	7					
VM	0	[0]	0	[1]	0	[2]	0	[3]					
Signification	base	+	base	recurse	base	<lit>	base	<fin>					
Numéro	8	9	10	11	12	13							
VM	1	4	2	0	8	6							
Signification	LAC	(<lit>)	nombre	(+)		(<fin>)							

Figure 5-3 : Un exemple de recurse en machine virtuelle

- On ajoute les fonctions de LAC `0=`, `1=` et `cr`.

```
: 0= 0 = ;
: 1= 1 - ;
: cr 2 ;
```

- Quand on lit le mot `defer`, on définit le mot suivant come une fonction de LAC. avec une fonction de base `erreur`.

```
: mot erreur ;
```

- Les fonctions `'` et `is` ne fonctionnent qu'en mode interprété. Quand on lit `'`, on trouve le Cfa de la fonction qui suit et le note par `Cfa_first`, quand on lit `is`, on trouve le Cfa de la fonction suivant et le note par `Cfa_second`, après, on remplace `VM[Cfa_second+1]=Cfa_first`.

5.2 Exécution

Premièrement, on définit les fonctions de base `if`, `else`, `then`, `<str>`, `count`, `type` et `erreur`.

```
void func_if(){
    int iCond=cstack_pop(s_donnee);
    int iAddr;
    if (iCond>0){
        iAddr=cstack_pop(s_retour);
        cstack_push(s_retour,iAddr+1);
    }
    else{
        iAddr=cstack_pop(s_retour);
        cstack_push(s_retour,VM[iAddr+1]);
    }
}

void func_else(){
    int iAddr = cstack_pop(s_retour);
    cstack_push(s_retour,VM[iAddr+1]);
}

void func_then(){
    ;
}

void func_str(){
    Elemtyp iAddr = cstack_pop(s_retour);
    iAddr +=1;
    cstack_push(s_donnee,iAddr);
    cstack_push(s_retour,iAddr+VM[iAddr]);
}

void func_count(){
    int iAddr=cstack_pop(s_donnee);
    int len = VM[iAddr];
    cstack_push(s_donnee,iAddr+1);
    cstack_push(s_donnee,len);
}

void func_type(){
    Elemtyp len,iAddr;
    len = cstack_pop(s_donnee);
    iAddr = cstack_pop(s_donnee);
    for(int i=0;i<len;i++){
        printf("%c",VM[iAddr+i]);
    }
    printf(" ");
}

void func_erreur(){
    printf("function pas defini\n");
}
```



```
}
```

Après, on modifie un peu la fonction `declare_LAC`, en ajoutant les cas de `str`, `if`, `else` et `then`.

```
void declare_LAC(char *function[],int length){
    // add to LAC
    int PBegin = pointer_LAC;
    size_t len = strlen(function[0]); // first one is the name of function
    LAC[pointer_LAC++] = len;
    for(int i=0;i<len;i++){
        LAC[pointer_LAC++] = function[0][i];
    }
    LAC[pointer_LAC++] = pointer_VM;
    LAC[pointer_LAC++] = PBegin;

    //add to VM
    int if_next_index;
    int else_next_index;
    int recurse_index;
    recurse_index=pointer_VM;
    VM[pointer_VM++]=1;
    for(int j=1;j<length;j++){
        int func=find_func(function[j]);
        VM[pointer_VM++]=func;
        if(strcmp(function[j],"<lit>")==0){ // the element behind <lit> is
always a number.
            j++;
            VM[pointer_VM++]=atoi(function[j]);
        }
        else if(strcmp(function[j],"<str>")==0){ // the element behind <str> is
always a string.
            j++;
            int str_l=strlen(function[j]);
            VM[pointer_VM++]=str_l;
            for (int m=0;m<str_l;m++){
                VM[pointer_VM++]=function[j][m];
            }
        }
        else if(strcmp(function[j],"if")==0){
            if_next_index=pointer_VM++;
        }
        else if(strcmp(function[j],"else")==0){
            else_next_index=pointer_VM++;
            VM[if_next_index]=else_next_index;
        }
        else if(strcmp(function[j],"then")==0){
            VM[else_next_index]=pointer_VM-1;
        }
        else if(strcmp(function[j],"recurse")==0){
            VM[pointer_VM-1]=recurse_index;
        }
    }
}
```

On modifie un peu de la fonction `compilation` dans le **Projet4**.

```

void compilation(char * psReadBuffer, CQueue * pqueRes){
    QueueNode * tmp = pqueRes->pHead;
    lexeme_t res;
    int flag = TRUE;
    while (flag) {
        res = *(lexeme_t *)tmp->pData;
        lexeme_t function_LAC;
        char ele[50];           // copy into a char.
        copy_mid(ele,psReadBuffer,(int)(res.lEnd-res.lStart),res.lStart);
        if (strcmp(res.type,"word")==0){
            if(strcmp(ele,"defer")==0){
                char *function[10];
                int len=0;
                tmp = tmp->pNext;
                function_LAC = *(lexeme_t *)tmp->pData;
                char mot[50];
                copy_mid(mot,psReadBuffer,(int)(function_LAC.lEnd-
function_LAC.lStart),function_LAC.lStart);
                function[len]=malloc(strlen(mot)*sizeof(char));
                strcpy(function[len++],mot);
                function[len++] = "erreur";
                function[len++] = "<fin>";
                declare_LAC(function,len);
            }
            else if(strcmp(ele,"")==0){
                char *function[10];
                int len=0;
                tmp = tmp->pNext;
                function_LAC = *(lexeme_t *)tmp->pData;
                char first[50];
                copy_mid(first,psReadBuffer,(int)(function_LAC.lEnd-
function_LAC.lStart),function_LAC.lStart);
                int Cfa_first=find_func(first);

                tmp = tmp->pNext;
                function_LAC = *(lexeme_t *)tmp->pData;
                char is[50];
                copy_mid(is,psReadBuffer,(int)(function_LAC.lEnd-
function_LAC.lStart),function_LAC.lStart);

                if (strcmp(is,"is")==0){
                    tmp = tmp->pNext;
                    function_LAC = *(lexeme_t *)tmp->pData;
                    char second[50];
                    copy_mid(second,psReadBuffer,(int)(function_LAC.lEnd-
function_LAC.lStart),function_LAC.lStart);
                    int Cfa_second=find_func(second);
                    VM[Cfa_second+1]=Cfa_first;
                }
            }
            else if(strcmp(ele,":")==0){
                ...
            }
            ...
        }
        printf("\n");
    }
}

```

5.3 Résultats

5.3.1 Compilation d'une structure conditionnelle

On utilise le document `test_projet5.lac` pour tester la fonction conditionnelle, le contenu de document est présenté dans la figure 5-4.

```
: essai 0 = if 1 else then ;
: incr 1 + ;
: 2+. incr incr . ;

7 8 + 2+.
0 essai .
19 essai .
```

Figure 5-4 : le contenu de document `test1.lac

Le résultat est présenté ci-dessous:

```
zoe@ubuntu:~/Desktop/PLT/Projet5$ gcc Projet5.c -o projet5
zoe@ubuntu:~/Desktop/PLT/Projet5$ ./projet5
M(":")->M("essai")->N("0")->M("=")->M("if")->N("1")->M("else")->M("the
n")->M(";")->M(":")->M("incr")->N("1")->M("+")->M(";")->M(":")->M("2+.
")->M("incr")->M("incr")->M(".")->M(";")->N("7")->N("8")->M("+")->M("2
+.")->N("0")->M("essai")->M(".")->N("19")->M("essai")->M(".")->
17
1
-1
```

Figure 5-5 : Le résultat de la fontion conditionnelle

Ce qui est correct.

5.3.2 Traitement des chaîne de caractère

Pour tester la fonction de traitement des chaîne de caractère, le contenu de document est présenté dans la figure 5-6.

```
1 : hi ( --- ) " Bonjour" count type ;
2 hi
```

Figure 5-6 : le contenu de document 'test2.lac'

Le résultat est présenté ci-dessous:

```
zoe@ubuntu:~/Desktop/PLT/Projet5$ gcc Projet5.c -o projet5
zoe@ubuntu:~/Desktop/PLT/Projet5$ ./projet5
M(":")->M("hi")->S("Bonjour")->M("count")->M("type")->M(";")->M("hi")->
>
Bonjour
```

Figure 5-7 : Le résultat de taper les chaîne de caractère

Ce qui est correct.

5.3.3 Compilation d'une déclaration

5.3.3.1 La fonction factorielle

Maintenant, on peut compiler la fonction dans le **projet1** en ajoutant la définition de les fonctions `0=`, `1=` et `recurse`.

```
1 \ Fichier "factorielle.lac"
2 ( Ce fichier est un "exemple" étudié pour tester
3 l'analyseur lexical écrit en phase 1 du projet)
4 : 0= 0 = ;
5 : 1= 1 - ;
6 : cr 2 ;
7 : fact ( n -- n!)
8 |   dup 0=
9 |   if
10 |   |   drop 1
11 |   else
12 |   |   dup 1= recurse *
13 |   then ;
14
15 : go ( n -- )
16 |   " Factorielle" count type
17 |   dup .
18 |   " vaut :
19 " count type
20 |   fact . cr ;
21
22 6 go
```

Figure 5-8: Le contenu du document 'factorielle.lac'

Le résultat est présenté dans la figure 5-9:

```
zoe@ubuntu:~/Desktop/PLT/Projet5$ gcc Projet5.c -o projet5
zoe@ubuntu:~/Desktop/PLT/Projet5$ ./projet5
M(":")->M("0=")->N("0")->M("=")->M(";")->M(":")->M("1-")->N("1")->M("-")->M(";")->M(":")->M("cr")->N("2")->M(";")->M(":")->M("fact")->M("dup")->M("0=")->M("if")->M("drop")->N("1")->M("else")->M("dup")->M("1-")->M("recurse")->M("*")->M("then")->M(";")->M(":")->M("go")->S("Factorielle")->M("count")->M("type")->M("dup")->M(".")->S("vaut :")->M("count")->M("type")->M("fact")->M(".")->M("cr")->M(";")->N("6")->M("go")->
Factorielle 6
vaut :
720
```

Figure 5-9 : Le résultat de la fonction factorielle

Ce qui est correct.

5.3.3.2 La fonction de u

Maintenant, on peut compiler la fonction u dans le document 'function_u.lac' en ajoutant la définition de les fonctions 0=, 1= et recurse.

```
defer u \ Déclaration de u

: 0= 0 = ;
: 1= 1 - ;
: v ( n -- v[n] )
dup 0= if drop 1 else dup 1- u 2 * swap 1- recurse - then ;

: [u] ( n -- u[n] )
dup 0= if drop 1 else dup 1- u swap 1- v + then ;

' [u] is u \ Résolution de la déclaration

4 u .
```

Figure 5-10: Le contenu du document 'factorielle.lac'

Le résultat est présenté dans la figure 5-11:

```
zoe@ubuntu:~/Desktop/PLT/Projet5$ gcc Projet5.c -o projet5
zoe@ubuntu:~/Desktop/PLT/Projet5$ ./projet5
M("defer")->M("u")->M(":")->M("0=")->N("0")->M("=")->M(";")->M(":")->M("1-")->N("1")->M("-")->M(";")->M(":")->M("v")->M("dup")->M("0=")->M("if")->M("drop")->N("1")->M("else")->M("dup")->M("1-")->M("u")->N("2")->M("*")->M("swap")->M("1-")->M("recurse")->M("-")->M("then")->M(";")->M(":")->M("[u]")->M("dup")->M("0=")->M("if")->M("drop")->N("1")->M("else")->M("dup")->M("1-")->M("u")->M("swap")->M("1-")->M("v")->M("+")->M("then")->M(";")->M("'")->M("[u]")->M("is")->M("u")->N("4")->M("u")->M(".")->
9
```

Figure 5-11 : Le résultat de la fonction de u

On obtient le bon résultat.