

Mini projet: MINI COMPELATEUR EN LANGAGE C

AZZIM Taha
DATA ENGINEERING-INE1 INPT

10 Avril 2020

But : Le but de ce rapport est de résumer et expliquer l'ensemble des étapes et des fonctions utilisées pour construire un analyseur lexicale et syntaxique du langage Pascal en langage C.

1 ANALYSEUR LEXICAL

Définissons les variables globales du programme:

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

char Car_cour;
FILE *f;
int affichage = 1;

char mot_cle[][20]= {"program", "const", "var", "begin", "end", "if", "then", "while", "do", "read", "write", "else", "repeat", "until", "for", "into", "downto", "case", "of"};

char symboles_speciaux[16]= {';', '.', '+', '-', '*', '/', ',', ':', '<', '>', '<=', '>=', '<>', '(', ')', '{', '=', ' '};

char CODE_LEX_CHAR[][20] = {"PROGRAM_TOKEN", "CONST_TOKEN", "VAR_TOKEN", "BEGIN_TOKEN", "END_TOKEN", "IF_TOKEN", "THEN_TOKEN", "WHILE_TOKEN", "DO_TOKEN", "READ_TOKEN", "WRITE_TOKEN", "ELSE_TOKEN", "REPEAT_TOKEN", "UNTIL_TOKEN", "FOR_TOKEN", "INTO_TOKEN", "DOWNTO_TOKEN", "CAS_TOKEN", "OF_TOKEN", "PV_TOKEN", "PT_TOKEN", "PLUS_TOKEN", "MOINS_TOKEN", "MULT_TOKEN", "DIV_TOKEN", "VIR_TOKEN", "AFF_TOKEN", "INF_TOKEN", "INFEG_TOKEN", "SUP_TOKEN", "SUPEG_TOKEN", "DIFF_TOKEN", "PO_TOKEN", "PF_TOKEN", "PL_TOKEN", "FIN_TOKEN", "EG_TOKEN", "ERREUR_TOKEN", "ID_TOKEN", "NUM_TOKEN"};

typedef enum {PROGRAM_TOKEN, CONST_TOKEN, VAR_TOKEN, BEGIN_TOKEN, END_TOKEN, IF_TOKEN, THEN_TOKEN, WHILE_TOKEN, DO_TOKEN, READ_TOKEN, WRITE_TOKEN, ELSE_TOKEN, REPEAT_TOKEN, UNTIL_TOKEN, FOR_TOKEN, INTO_TOKEN, DOWNTO_TOKEN, CASE_TOKEN, OF_TOKEN, PV_TOKEN, PT_TOKEN, PLUS_TOKEN, MOINS_TOKEN, MULT_TOKEN, DIV_TOKEN, VIR_TOKEN, AFF_TOKEN, INF_TOKEN, INFEG_TOKEN, SUP_TOKEN, SUPEG_TOKEN, DIFF_TOKEN, PO_TOKEN, PF_TOKEN, PL_TOKEN, FIN_TOKEN, EG_TOKEN, ERREUR_TOKEN, ID_TOKEN, NUM_TOKEN, COMMENT_TOKEN}CODE_LEX;

typedef struct {CODE_LEX CODE;
char NOM[20];
}TSym_Cour;

TSym_Cour SYM_COUR;
```

Le programme PASCAL doit être lu par le compilateur à partir d'un fichier text (*PASCAL.txt* par exemple). On doit donc premièrement définir la fonction *Lire_Car* avec laquelle on va lire les caractères du fichier text en tenant compte de la condition : pas de distinction entre minuscule et majuscule.

```
void Lire_Car(){
    Car_cour = tolower(fgetc(f));
}
```

1.1 LECTURE ET RECONNAISSANCE CAR PAR CAR

On va vous présenter quelques fonctions utilisées dans ce programme avec une petite explication de leur fonctionnement, le reste des fonctions serait présenté après :

- *est_separateur* pour verifier si Car.cour s'agit d'un des separateurs (*espace blanc ou retour chariot*). La fonction retourne 1 si oui et 0 sinon.

```
int est_separateur(){
    int resultat=0;
    if ((Car_cour==' ')||(Car_cour=='\n')||(Car_Cour=='\t'))
        resultat=1;

    return resultat;
}
```

- *est_lettre* pour verifier si Car.cour s'agit d'une lettre ou non. La fonction retourne 1 si oui et 0 sinon.

```
int est_lettre(){
    int resultat = 0;
    if((Car_cour >= 'a') && (Car_cour <= 'z'))
        resultat = 1;
    return resultat;
}
```

- *est_symbole_special* pour verifier si Car.cour s'agit d'un des symboles spéciaux qui sont définit dans la liste *symboles_speciaux*[[20]. La fonction retourne 1 si oui et 0 sinon

```
int est_symbole_special(){
    int i,resultat = 0;
    if(memchr(symboles_speciaux, Car_cour, sizeof(
        symboles_speciaux)))
        resultat = 1;
    return resultat;
}
```

- Lire_mot pour la lecture des mots
Pendant chaque boucle. La variable SYM_COUR doit contenir 2 variables.
 - char NOM[20]: qui contient la catégorie extraite (soit mot, nombre, caractère spécial ...).
 - CODE_LEX CODE: qui est le code (TOKEN) de cette catégorie (ID_TOKEN ,NUM_TOKEN ...).

```

void Lire_mot(){
    int i = 0;
    while((est_symbole_special() != 1) && (est_separateur() !=
        1) && (Car_cour != EOF)){
        SYM_COUR.NOM[i] = Car_cour;
        Lire_Car();
        i++;
    }
    SYM_COUR.NOM[i] = '\0';
}

```

Dans cette fonction, on n'a pas affecter à SYM_COUR.CODE sa valeur. De ce fait, on définit la fonction mot_cle_nom. Si le mot est un mot clé la fonction retourne son CODE, sinon elle retourne ID_TOKEN.

- mot_cle_nom

```

void mot_cle_nom(){
    int i, id=38;
    for(i=0 ; i<20 ; i++){
        if(strcmp(SYM_COUR.NOM, mot_cle[i])==0)
            id = i;
    }
    SYM_COUR.CODE = (CODE_LEX)id;
}

```

- Lire_nombre pour la lecture des nombres

```

void Lire_nombre(){
    int i = 0;
    while((est_symbole_special() != 1) && (est_separateur() !=
        1) && (est_lettre() == 0) && (Car_cour != EOF)){
        SYM_COUR.NOM[i] = Car_cour;
        Lire_Car();
        i++;
    }
    SYM_COUR.NOM[i] = '\0';
    SYM_COUR.CODE = NUM_TOKEN;
}

```

- Affichier_TOKEN pour affichage.

```

|| void Affichier_TOKEN() {
||     printf("%s\n", CODE_LEX_CHAR[SYM_COUR.CODE]);
|| }

```

Il nous reste la fonction des symboles spéciaux. Dans la plupart des cas, la lecture d'un symbole spécial implique l'affectation du code du symbole à la variable SYM_COUR.CODE qui est le cas des symboles :

```

- "."
- "+"
- "_"
- "/"
- "*"
- ","
- "("
- ")"
- EOF
- ";"

```

Mais pour les autres, on risque de confuser entre "<" et "<=" par exemple ou entre "<" et "<>".

Dans cette fonction on va inclure les commentaires aussi, ce qui justifie la présence de "{" dans les symboles spéciaux.

On va vous expliquer la fonction étape par étape. D'abord pour les cas évidents:

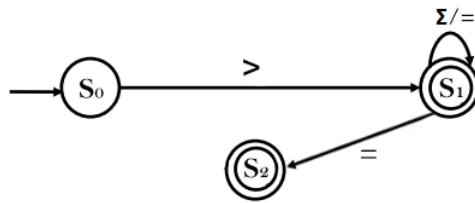
```

void Lire_sym(){
    switch(Car_cour){
        case ';':
            SYM_COUR.CODE = PV_TOKEN;
            Lire_Car();
            break;
        case ',':
            SYM_COUR.CODE = PT_TOKEN;
            Lire_Car();
            break;
        case '+':
            SYM_COUR.CODE = PLUS_TOKEN;
            Lire_Car();
            break;
        case '-':
            SYM_COUR.CODE = MOINS_TOKEN;
            Lire_Car();
            break;
        case '*':
            SYM_COUR.CODE = MULT_TOKEN;
            Lire_Car();
            break;
        case '/':
            SYM_COUR.CODE = DIV_TOKEN;
            Lire_Car();
            break;
        case ', ':
            SYM_COUR.CODE = VIR_TOKEN;
            Lire_Car();
            break;
        case '>=':
            SYM_COUR.CODE = EG_TOKEN;
            Lire_Car();
            break;
        case '(':
            SYM_COUR.CODE = PO_TOKEN;
            Lire_Car();
            break;
        case ')':
            SYM_COUR.CODE = PF_TOKEN;
            Lire_Car();
            break;
        case EOF:
            SYM_COUR.CODE = FIN_TOKEN;
            Lire_Car();
            break;
    }
}

```

Pour les autres cas, on va les expliquer en s'appuyant sur les automates.

La confusion entre les deux symboles $>$ et $>=$ est représentée par l'automate (valable juste pour les mots composés de 2 caractères):

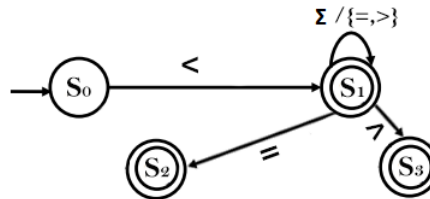


S_1 indique l'état où on doit choisir $>$ alors que S_2 indique celle de $>=$. Le code serait comme suit:

```

case '>':
    Lire_Car();
    if(Car_cour == '>='){
        SYM_COUR.CODE = SUPEG_TOKEN;
    }
    else{SYM_COUR.CODE = SUP_TOKEN;}
    Lire_Car();
    break;
  
```

La confusion entre $>$, $>=$ et $<>$ serait résolue par l'automate (Aussi valable juste pour les mots composés de 2 caractères):



S_1 indique l'état où on doit choisir $<$, S_2 indique celle de $<=$ et S_3 pour $<>$. Le code serait comme suit:

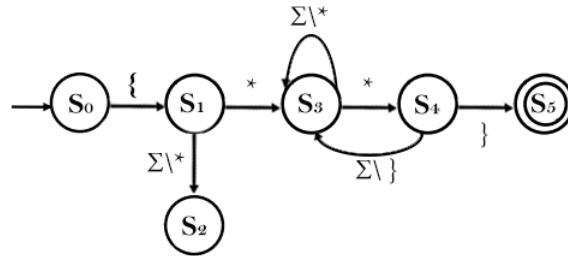
```

case '<':
    Lire_Car();
    if(Car_cour == '>='){
        SYM_COUR.CODE = INFEG_TOKEN;
        Lire_Car();
    }
    else if(Car_cour == '>') {
        SYM_COUR.CODE = DIFF_TOKEN;
        Lire_Car();
    }
    else{
        SYM_COUR.CODE = INF_TOKEN;
    }
    break;
  
```

De meme pour l'affectation:

```
case '':
    SYM_COUR.CODE = PL_TOKEN;
    Lire_Car();
    if(Car_cour == '='){
        SYM_COUR.CODE = AFF_TOKEN;
        Lire_Car();}
    break;
```

Il nous reste encore un dernier symbole qui est celui des commentaires. Un commentaire doit commencer par {* et finir par *}. L'automate du commentaire est le suivant :



Le code est le suivant:

```
case '{': //S1
    Lire_Car();
    if(Car_cour == '*'){
        loop: //S3
        do{
            Lire_Car();
        }while(Car_cour != '*'); //S4
        Lire_Car();
        if(Car_cour == '}') //S5
        {
            Lire_Car();
            SYM_COUR.CODE = COMMENT_TOKEN;
        }
        else{goto loop;}
    }
else{
    Erreur(ERR_CAR_INC); //S2
}
```

L'état S5 indique la fin du commentaire. Les autres sont soit erreurs soit commentaire n'est pas encore fini.

On vous expliquera la fonction *Erreur()* dans le praragraphe suivant.

Il nous reste encore une fonction avant *main()*. Il s'agit de la fonction *Sym_Suiv* dans laquelle on va englober la totalité des fonctions définies et qu'on va introduire au cas le fichier est ouvert avec succès. La fonction modifie le code du symbole courant (*SYM_COUR.CODE*).

```
void Sym_Suiv(){
    while(est_separateur() == 1){ //Ignorer les separateurs
        Lire_Car();
    }
    if(est_lettre() == 1){
        Lire_mot();
        mot_cle_nom();
    }
    else if(isdigit(Car_cour)){
        Lire_nombre();
    }
    else if((est_symbole_special() == 1) || (Car_cour == EOF)){
        Lire_sym();
    }
    if(SYM_COUR.CODE == COMMENT_TOKEN)
        Sym_Suiv() //Passage au symbole suivant au
        cas ou il s'agit d'un commentaire.
}
```

Et finalement voila la fonction *main()*

```
int main()
{
    char chemin [32];
    printf("\nENTRER LE CHEMIN DU FICHIER :");
    scanf("%s", chemin);
    f=fopen(chemin, "r");
    if (f==NULL){
        SYM_COUR.CODE = ERREUR_TOKEN;
        Affichier_TOKEN();
    }
    else
    {
        Lire_Car();
        while(Car_cour != EOF){
            Sym_Suiv();
            Affichier_TOKEN();
        }
        Sym_Suiv();
        Affichier_TOKEN();
    }
}
```

Essayons une première execution du programme. Soit le programme PASCAL enregistré dans le fichier PASCAL.txt:

```
*****
program test11;
const toto=21;
var x,y;
Begin
  x:=toto;
  read(y);
end.
*****
```

Voila le résultat :



```
ENTRER LE CHEMIN DU FICHIER :C:\Workplace\PASCAL.txt
PROGRAM_TOKEN
ID_TOKEN
PV_TOKEN
CONST_TOKEN
ID_TOKEN
EG_TOKEN
NUM_TOKEN
PV_TOKEN
VAR_TOKEN
ID_TOKEN
VIR_TOKEN
ID_TOKEN
PV_TOKEN
BEGIN_TOKEN
ID_TOKEN
AFF_TOKEN
ID_TOKEN
PV_TOKEN
READ_TOKEN
PO_TOKEN
ID_TOKEN
PF_TOKEN
PV_TOKEN
END_TOKEN
PT_TOKEN
FIN_TOKEN
```

1.2 Les erreurs

Passons maintenant à la partie des erreurs, on va s'intéresser dans cette partie de l'analyseur lexicale sur les erreurs suivantes:

- Fichier introuvable.
- Fichier vide.
- ID est longue (>20).
- Constantes numérique longue (>11)
- Caractère inconnu.

On va définir quelques nouveaux variables à savoir:

```
int compteur = 0;
int id_longueur = 0;
int const_longueur = 0;

typedef enum {ERR_CAR_INC, ERR_FICH_VID, ERR_ID_LONG,
             ERR_FICH_INTRO, ERR_CONST_LONG }Erreurs;

typedef struct {Erreurs CODE_ERR; char mes[40]}Erreurs2;

Erreurs2 MES_ERR[]={{{ERR_CAR_INC, "Caractere inconnu"}, {
    ERR_FICH_VID, "Fichier vide"}, {ERR_ID_LONG, "IDF tres long"}, {
    ERR_FICH_INTRO, "Fichier introuvable"}, {ERR_CONST_LONG, "
    CONSTF tres long"}}};
```

MES_ERR[] est une liste des erreurs avec les messages qui les accompagnent.

On aura besoin de la fonction *Erreur* qui affiche l'erreur quand elle existe puis la fonction arrête l'analyse syntaxique.

```
void Erreur(Erreurs ERR){
    int ind_err = ERR;
    printf("Erreur numero %d : %s \n", ind_err, MES_ERR[ind_err].
        mes);
    getch();
    exit(1);
}
```

1.2.1 Fichier introuvable

Commençons par l'erreur où le fichier n'existe pas. On doit tout simplement effectuer quelques modifications au niveau du *main()*.

```
scanf("%s", chemin);
f=fopen(chemin, "r");
if (f==NULL)
{
    Erreur(ERR_FICH_INTRO);
}
```

Voila le résultat si l'on passe un fichier introuvable:

```
ENTRER LE CHEMIN DU FICHIER :C:\Workplace\fichierintrouvable.txt
Erreur numero 3 :fichier introuvable
```

1.2.2 Fichier vide

Pour s'avoir si le fichier vide ou contient seulement des séparateurs, on a introduit la variable compteur pour compter le nombre des symboles dans le fichier. Les modifications qu'ont doit avoir seront dans la fonction `Sym_Suiv()` et `main()`

```
void Sym_Suiv(){
    while(est_separateur() == 1){
        Lire_Car();
    }
    if(est_lettre() == 1){
        Lire_mot();
        mot_cle_nom();
        compteur++;
    }
    else if(isdigit(Car_cour)){
        Lire_nombre();
        compteur++;
    }
    else if((est_symbole_special() == 1) || (Car_cour == EOF)){
        Lire_sym();
        compteur++;
    }
    else{
        Erreur(ERR_CAR_INC);
    }
}
```

Le compteur s'incrémente lorsqu'on rencontre un mot, un nombre ou un symbole spécial. Puis dans `main()` et avant la dernière `Affichier_TOKEN` on introduit la condition du compteur

```
Sym_Suiv();
if(compteur == 1)
    Erreur(ERR_FICH_VID);
Affichier_TOKEN();
```

Voila le résultat si le fichier est vide:

```
ENTRER LE CHEMIN DU FICHIER :C:\Workplace\PASCAL.txt
Erreur numero 1 :fichier vide
```

1.2.3 Caractère inconnu

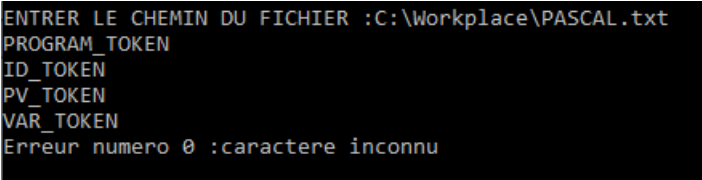
Cette erreur doit être reconnue lorsque l'analyseur rencontre un symbole qui est ni nombre ni mot ni symbole spécial. Elle serait donc introduite dans la fonction `Sym_Suiv()` comme indiqué ci-dessus.

un exemple de programme :

```
*****
program test;
var @pp;

*****
```

Résultat:



```
ENTRER LE CHEMIN DU FICHIER :C:\Workplace\PASCAL.txt
PROGRAM_TOKEN
ID_TOKEN
PV_TOKEN
VAR_TOKEN
Erreur numero 0 :caractere inconnu
```

1.2.4 Longueur de l'ID est supérieure à 20/ Longueur des constantes est supérieure à 11

La variable `id_longueur` va s'incrémenter lors de la lecture d'un mot. Pareil pour `const_longueur` lors de la lecture des constantes numériques. On ne s'intéresse à cette erreur que dans l'analyseur lexical, de ce fait on déclare la variable globale `Analyse_lexical` initialisée par la valeur 1.

```
|| int Analyse_lexical =1;
```

Puis on doit modifier `Lire_mot()` et `Lire_nombre()`.

```
void Lire_mot(){
    int i = 0;
    while((est_symbole_special() != 1) && (est_separateur() != 1)
        && (Car_cour != EOF)){
        SYM_COUR.NOM[i] = Car_cour;
        Lire_Car();
        i++;
        id_longueur++;
        if(id_longueur > 20 && Analyse_lexical == 1)
        {
            Erreur(ERR_ID_LONG);
        }
    }
    SYM_COUR.NOM[i] = '\0';
}
```

```

void Lire_nombre(){
    int i = 0;
    while((est_symbole_special() != 1) && (est_separateur() != 1)
        && (est_lettre() == 0) && (Car_cour != EOF) && (
            const_longueur < 12)){
        SYM_COUR.NOM[i] = Car_cour;
        Lire_Car();
        i++;
        const_longueur++;
        if(const_longueur > 11 && Analyse_lexical == 1)
        {
            Erreur(ERR_CONST_LONG);
        }
    }
    SYM_COUR.NOM[i] = '\0';
    SYM_COUR.CODE = NUM_TOKEN;
}

```

Finalement *main()* serait come suit

```

int main()
{
    char chemin [32];
    printf("\nENTRER LE CHEMIN DU FICHIER :");
    scanf("%s", chemin);
    f=fopen(chemin, "r");
    if (f==NULL)
    {
        Erreur(ERR_FICH_INTRO);
    }

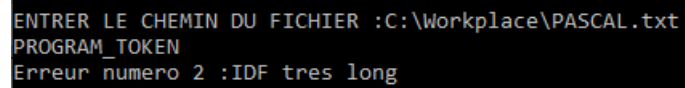
    else
    {
        Lire_Car();
        while((Car_cour != EOF) && (ERREUR_COUR == NO_ERREUR)){
            Sym_Suiv();
            id_longueur = 0;
            const_longueur = 0;
            Affichier_TOKEN();
        }
        Sym_Suiv();
        if(compteur == 1)
            Erreur(ERR_FICH_VID);
        Affichier_TOKEN();
    }
}

```

Essayons d'afficher ces deux erreurs en executant les deux programmes PASCAL suivants

Programme où ID est longue :

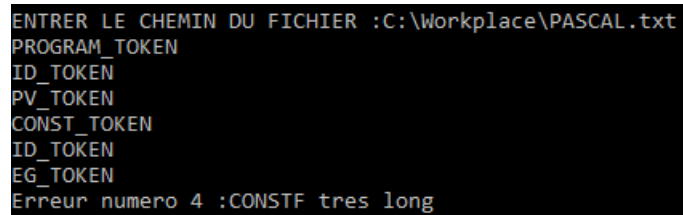
```
*****  
program testtesttesttesttesttest;  
  
*****
```



```
ENTRER LE CHEMIN DU FICHIER :C:\Workplace\PASCAL.txt  
PROGRAM_TOKEN  
Erreur numero 2 :IDF tres long
```

Programme où NUM est longue :

```
*****  
program test;  
const variable=22222222222222;  
  
*****
```



```
ENTRER LE CHEMIN DU FICHIER :C:\Workplace\PASCAL.txt  
PROGRAM_TOKEN  
ID_TOKEN  
PV_TOKEN  
CONST_TOKEN  
ID_TOKEN  
EG_TOKEN  
Erreur numero 4 :CONSTF tres long
```

Voila notre analyseur lexicale complet. Passons à l'analyse syntaxique.

2 ANALYSEUR SYNTAXIQUE

Définissons des nouvelles variables globales qu'on aura besoin dans cette partie.

```
int Ligne_erreur = 0;

typedef enum {ERR_CAR_INC, ERR_FICH_VID, ERR_ID_LONG,
    ERR_FICH_INTRO, ERR_CONST_LONG, PROGRAM_ERR, ID_ERR, PV_ERR,
    PT_ERR, EG_ERR, NUM_ERR, CONST_VAR_BEGIN_ERR, AFF_ERR, PF_ERR,
    BEGIN_ERR, END_ERR, COND_ERR, THEN_ERR, SYNT_ERR, DO_ERR,
    PO_ERR, UNTIL_ERR, INTO_DOWNTOW_ERR, OF_ERR, PL_ERR} Erreurs;

Erreurs2 MES_ERR[] = {
    {ERR_CAR_INC, "caractere inconnu"},
    {ERR_FICH_VID, "fichier vide"},
    {ERR_ID_LONG, "IDF tres long"},
    {ERR_FICH_INTRO, "fichier introuvable"},
    {ERR_CONST_LONG, "CONSTF tres long"},
    {PROGRAM_ERR, "Erreur de programme"},
    {ID_ERR, "Erreur en ID"}, {PV_ERR, "Ajoutez ;"},
    {PT_ERR, "Ajoutez point a la fin du programme"},
    {EG_ERR, "Ajoutez une valeur aux constantes"},
    {NUM_ERR, "Ajoutez un nombre"},
    {CONST_VAR_BEGIN_ERR, "La syntaxe non valide"},
    {AFF_ERR, "Ajoutez := dans l'affectation"},
    {PF_ERR, "Une ')' est attendue"},
    {BEGIN_ERR, "Pas de 'BEGIN'"},
    {END_ERR, "Il manque 'end' a la fin du programme"},
    {COND_ERR, "Condition incorrecte"},
    {THEN_ERR, "'then' attendue apres if"},
    {SYNT_ERR, "Erreur de la syntaxe"},
    {DO_ERR, "'do' attendue apres la boucle"},
    {PO_ERR, "Ajoutez les ()"},
    {UNTIL_ERR, "'until' attendue apres 'repeat'"},
    {INTO_DOWNTOW_ERR, "'into' ou 'downto' manquante"},
    {OF_ERR, "'of' attendue dans 'case'"},
    {PL_ERR, "':' attendue dans 'case'"}
};
```


Avant de passer à l'analyse syntaxique on a modifié la fonction *Erreur()* pour qu'elle puisse afficher le type d'erreur et aussi la ligne où l'erreur s'est intervenue (Ce qui justifie la déclaration de *Ligne_erreur*)

```
void Erreur(Erreurs ERR){
    int ind_err = ERR;
    printf("Dans la ligne %d ,erreur numero %d : %s \n",
        Ligne_erreur, ind_err, MES_ERR[ind_err].mes);
    getch();
    exit(1);
}
```

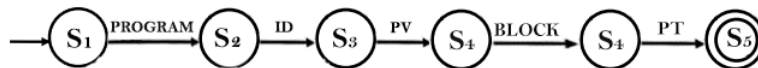
Cette variable doit s'incrémenter chaque fois que le *Car_Cour* est égal à '\n' chose qui nous a poussée à modifier *Sym_suiv()*

```
void Sym_Suiv(){
    while(est_separateur() == 1){
        if(Car_cour == '\n'){
            Ligne_erreur++;
        }
        Lire_Car();
    }
    [......]
}
```

2.1 Programme principal

La fonction *PROGRAM()* va nous aider à tester la syntaxe de programme en appelant une fonction *BLOCK()* qui va être définie par suite et la fonction *Test_Symbole(CODE_LEX Lc,Erreur COD_ERR)* qui va tester à chaque fois l'existence du symbole.

On va résumer tout cela par l'automate suivant:



On peut le traduire par le programme suivant:

```
void PROGRAM(){
    Test_Symbole(PROGRAM_TOKEN, PROGRAM_ERR);
    Test_Symbole(ID_TOKEN, ID_ERR);
    Test_Symbole(PV_TOKEN, PV_ERR);
    BLOCK();
    Test_Symbole(PT_TOKEN, PT_ERR);
}
```

2.2 Les fonctions secondaires

On va ensuite déclarer les fonctions qui vont être appelées par le programme principal.

- Fonction BLOCK()

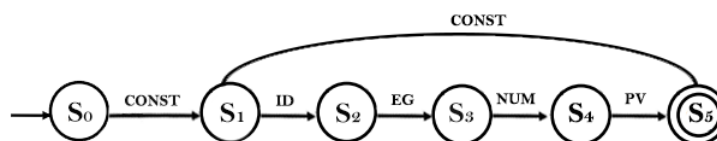
La fonction *BLOCK()* qui était appelée dans *PROGRAM()* et de son tour fait appel aux autres fonctions qui seront définies par suite.

```
void BLOCK(){
    CONSTS();
    VARS();
    INSTS();
}
```

- Fonction CONSTS()

la fonction *CONSTS()* permet une analyse syntaxique de déclaration des constantes.

L'automate des constantes est le suivant:

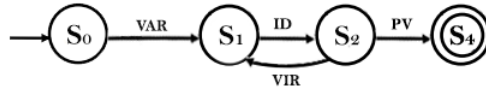


On obtient le programme de la fonction *CONSTS()* suivant:

```
void CONSTS(){
    switch (SYM_COUR.CODE) {
    case CONST_TOKEN:
        Sym_Suiv();
        Test_Symbole(ID_TOKEN, ID_ERR);
        loop1:
            Test_Symbole(EG_TOKEN, EG_ERR);
            Test_Symbole(NUM_TOKEN, NUM_ERR);
            Test_Symbole(PV_TOKEN, PV_ERR);
            if (SYM_COUR.CODE == ID_TOKEN)
            {
                Sym_Suiv();
                goto loop1;
            }
        break;
    case VAR_TOKEN:
        break;
    case BEGIN_TOKEN:
        break;
    default:
        Erreur(CONST_VAR_BEGIN_ERR);
        break;
    }
}
```

- Fonction VARS()

la fonction *VARS()* permet une analyse syntaxique de déclaration des variables.
l'automate des variables est le suivant:



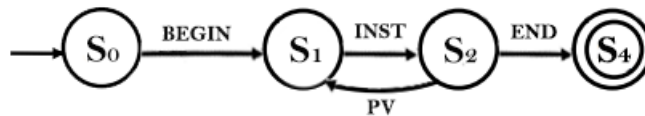
le code de la fonction *VARS()* est le suivant:

```

void VARS() {
    switch (SYM_COUR.CODE){
    case VAR_TOKEN:
        Sym_Suiv();
        Test_Symbole(ID_TOKEN, ID_ERR);
        while (SYM_COUR.CODE==VIR_TOKEN){
            Sym_Suiv();
            Test_Symbole(ID_TOKEN, ID_ERR);
        }
        Test_Symbole(PV_TOKEN, PV_ERR);
        break;
    case BEGIN_TOKEN:
        break;
    default:
        Erreur(CONST_VAR_BEGIN_ERR) ;
        break;
    }
}
  
```

- Fonction INSTS()

la fonction *INSTS()* permet une analyse syntaxique des instructions.
l'automate des instructions est le suivant:



Le code est le suivant:

```

void INSTS(){
    Test_Symbole(BEGIN_TOKEN, BEGIN_ERR);
    INST();
    while(SYM_COUR.CODE != END_TOKEN && SYM_COUR.CODE != FIN_TOKEN)
    {
        Test_Symbole(PV_TOKEN, PV_ERR);
        INST();
    }
    Ligne_erreur++;
    Test_Symbole(END_TOKEN, END_ERR);
}
  
```

NOTE: La dernière instruction n'est jamais suivie avec le symbole ';'.

- Fonction INST()

la fonction *INST()* permet une analyse syntaxique de déclaration d'une seule instruction.

l'automate d'une instruction est le suivant:



Le code est le suivant:

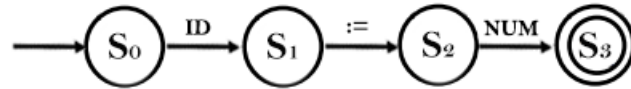
```

void INST(){
    switch(SYM_COUR.CODE){
        case BEGIN_TOKEN:
            INSTS();
            break;
        case ID_TOKEN:
            AFFEC();
            break;
        case IF_TOKEN:
            SI();
            break;
        case WHILE_TOKEN:
            TANTQUE();
            break;
        case WRITE_TOKEN:
            ECRIRE();
            break;
        case READ_TOKEN:
            LIRE();
            break;
        case REPEAT_TOKEN:
            REPETER();
            break;
        case FOR_TOKEN:
            POUR();
            break;
        case CASE_TOKEN:
            CAS();
            break;
        default:
            Erreur(SYNT_ERR);
    }
}

```

- Fonction AFF()

la fonction *AFF()* permet une analyse syntaxique d'une affectation.
l'automate d'affectation est le suivant:



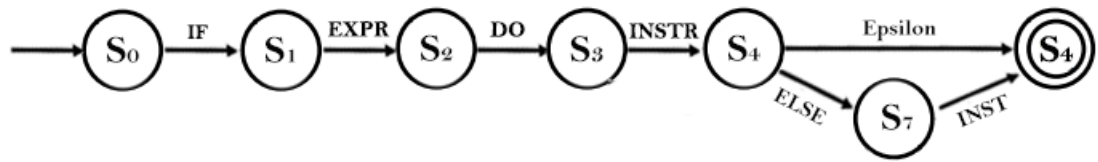
Le code est le suivant:

```

void AFPEC() {
    Test_Symbole(ID_TOKEN, ID_ERR);
    Test_Symbole(AFF_TOKEN, AFF_ERR);
    EXPR();
}
  
```

- Fonction SI()

la fonction *SI()* permet une analyse syntaxique de la condition *if..else*.
l'automate d'une condition *if .. else* est le suivant:



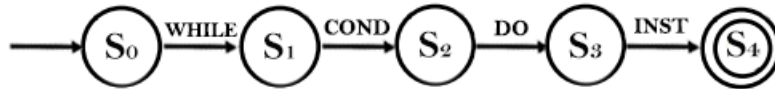
Le code est le suivant:

```

void SI() {
    Sym_Suiv();
    COND();
    Test_Symbole(THEN_TOKEN, THEN_ERR);
    INSTR();
    if (SYM_COUR.CODE == ELSE_TOKEN)
    {
        Sym_Suiv();
        INSTR();
    }
}
  
```

- Fonction TANTQUE()

la fonction *TANTQUE()* permet une analyse syntaxique de la condition *while*.
l'automate d'une condition *while* est le suivant:



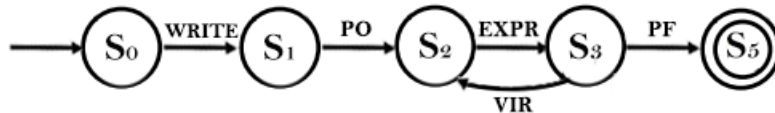
Le code est le suivant:

```

void TANTQUE() {
    Sym_Suiv();
    COND();
    Test_Symbole(DO_TOKEN, DO_ERR);
    INST();
}
  
```

- Fonction ECRIRE()

la fonction *ECRIRE()* permet une analyse syntaxique de la fonction *write()*.
l'automate de la fonction *write()* est le suivant:



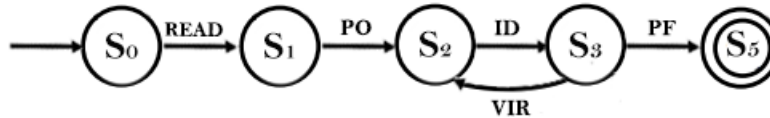
Le code est le suivant:

```

void ECRIRE() {
    Sym_Suiv();
    Test_Symbole(PO_TOKEN, PO_ERR);
    EXPR();
    while(SYM_COUR.CODE == VIR_TOKEN)
    {
        Sym_Suiv();
        EXPR();
    }
    Test_Symbole(PF_TOKEN, PF_ERR);
}
  
```

- Fonction `LIRE()`

la fonction `LIRE()` permet une analyse syntaxique de la fonction `read()`.
l'automate de la fonction `read()` est le suivant:



Le code est le suivant:

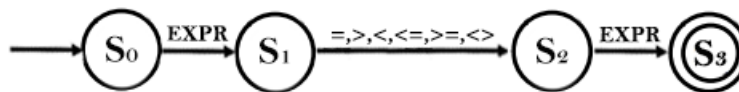
```

void LIRE() {
    Sym_Suiv();
    Test_Symbole(PO_TOKEN, PO_ERR);
    Test_Symbole(ID_TOKEN, ID_ERR);
    while (SYM_COUR.CODE == VIR_TOKEN)
    {
        Sym_Suiv();
        Test_Symbole(ID_TOKEN, ID_ERR);
    }
    Test_Symbole(PF_TOKEN, PF_ERR);
}

```

- Fonction `COND()`

la fonction `COND()` permet une analyse syntaxique d'une condition.
l'automate d'une condition est le suivant:



Le code est le suivant:

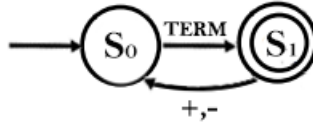
```

void COND() {
    EXPR();
    if ((SYM_COUR.CODE == EG_TOKEN) || (SYM_COUR.CODE == DIFF_TOKEN)
        || (SYM_COUR.CODE == INF_TOKEN) || (SYM_COUR.CODE ==
            SUP_TOKEN) || (SYM_COUR.CODE == INFEG_TOKEN) || (SYM_COUR
                .CODE == SUPEG_TOKEN)) {
        Sym_Suiv();
        EXPR();
    }
    else { Erreur(COND_ERR); }
}

```

- Fonction `EXPR()`

la fonction `EXPR()` permet une analyse syntaxique d'une expression.
l'automate d'une expression est le suivant:



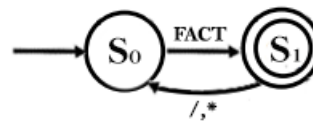
Le code est le suivant:

```

void EXPR() {
    TERM();
    while (SYM_COUR.CODE == MOINS_TOKEN || SYM_COUR.CODE ==
        PLUS_TOKEN) {
        Sym_Suiv();
        TERM();
    }
}
  
```

- Fonction `TERM()`

la fonction `TERM()` permet une analyse syntaxique d'un terme.
l'automate d'un terme est le suivant:



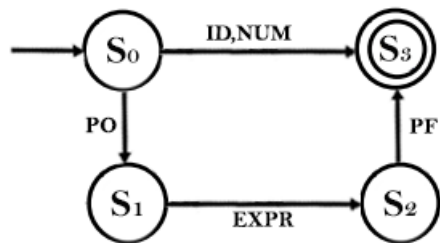
Le code est le suivant:

```

void TERM() {
    FACT();
    while (SYM_COUR.CODE == MULT_TOKEN || SYM_COUR.CODE == DIV_TOKEN) {
        Sym_Suiv();
        FACT();
    }
}
  
```


- Fonction `FACT()`

la fonction *FACT()* permet une analyse syntaxique d'un facteur.
l'automate d'un facteur est le suivant:



Le code est le suivant:

```

void FACT(){
    switch (SYM_COUR.CODE){
        case ID_TOKEN:
            Test_Symbole(ID_TOKEN, ID_ERR);
            break;
        case NUM_TOKEN:
            Test_Symbole(NUM_TOKEN, NUM_ERR);
            break;
        case PO_TOKEN:
            Sym_Suiv();
            EXPR();
            Test_Symbole(PF_TOKEN, PF_ERR);
            break;
        default:
            Erreur(SYNT_ERR);
    }
}

```

- Fonction REPETER()

la fonction *REPETER()* permet une analyse syntaxique d'une boucle *repeat*.
l'automate de la boucle *repeat* est le suivant:



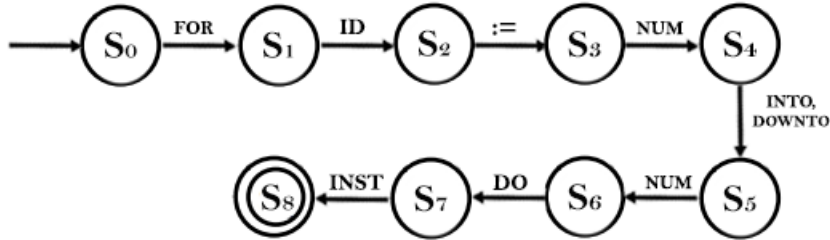
Le code est le suivant:

```

void REPETER() {
    Sym_Suiv();
    INST();
    Test_Symbole(UNTIL_TOKEN, UNTIL_ERR);
    COND();
}
  
```

- Fonction POUR()

la fonction *POUR()* permet une analyse syntaxique d'une boucle *for*.
l'automate d'une boucle *for* est le suivant:

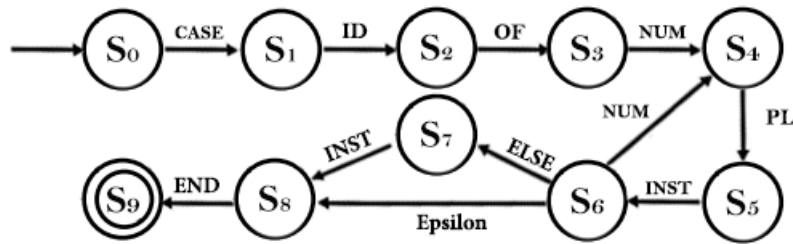


Le code est le suivant:

```
void POUR(){
    Sym_Suiv();
    Test_Symbole(ID_TOKEN, ID_ERR);
    Test_Symbole(AFF_TOKEN, AFF_ERR);
    Test_Symbole(NUM_TOKEN, NUM_ERR);
    if (SYM_COUR.CODE == INTO_TOKEN || SYM_COUR.CODE == DOWNTOKEN)
        Sym_Suiv();
    else { Erreur(INTO_DOWNTOKEN_ERR); }
    Test_Symbole(NUM_TOKEN, NUM_ERR);
    Test_Symbole(DO_TOKEN, DO_ERR);
    INST();
}
```

- Fonction CAS()

la fonction *CAS()* permet une analyse syntaxique d'une fonction *case*.
Son automate *case* est le suivant:



Le code est le suivant:

```
void CAS(){
    Sym_Suiv();
    Test_Symbole(ID_TOKEN, ID_ERR);
    Test_Symbole(OF_TOKEN, OF_ERR);
    Test_Symbole(NUM_TOKEN, NUM_ERR);
    Test_Symbole(PL_TOKEN, PL_ERR);
    INST();
    while (SYM_COUR.CODE == NUM_TOKEN)
    {
        Sym_Suiv();
        Test_Symbole(PL_TOKEN, PL_ERR);
        INST();
    }
    if (SYM_COUR.CODE == ELSE_TOKEN)
    {
        Sym_Suiv();
        INST();
    }
    Test_Symbole(END_TOKEN, END_ERR);
}
```

2.3 La fonction *main()*

On doit changer un peu dans le *main()* pour appeler la fonction *PROGRAM()*.
Et finalement le code de *main()* est le suivant:

```
int main(){
    char chemin [32];
    printf("\nENTRER LE CHEMIN DU FICHIER :");
    scanf("%s",chemin);
    f=fopen(chemin,"r");
    if (f==NULL){
        Erreur(ERR_FICH_INTRO);
    }

    else
    {
        Lire_Car();
        printf("\nANALYSEUR LEXICALn");
        while(Car_cour != EOF){
            Sym_Suiv();
            id_longeur = 0;
            const_longeur = 0;
            Affichier_TOKEN();
        }
        Sym_Suiv();
        if(compteur == 1)
            Erreur(ERR_FICH_VID);
        Affichier_TOKEN();
        printf("\nFin de l'analyse lexicale -- succes --\n");
        fclose(f);
        f=fopen(chemin,"r");
        printf("\nANALYSEUR SYNTAXIQUEn");
        Ligne_erreur = 0;
        Analyse_lexical = 0;
        Lire_Car();
        Sym_Suiv();
        PROGRAM();
        if(SYM_COUR.CODE == FIN_TOKEN)
            printf("\nFin de l'analyse syntaxique -- succes --\n");
        else printf("\nProgramme erronee");
    }
}
```

3 Test de quelques programmes PASCAL

3.0.1 Programmes avec des erreurs

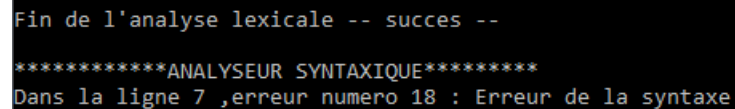
Dans cette section on va passer à notre analyseur des programmes PASCAL avec des erreurs pour les tester.

- **Programme 1:**

Erreur : contient ';' dans la dernière instruction

```
*****  
program test;  
const toto=21; titi=13;  
var x,y;  
begin  
{*Voila un commentaire*}  
x:=toto;  
y:=titi;  
end.  
*****
```

Résultat:



```
Fin de l'analyse lexicale -- succes --  
*****ANALYSEUR SYNTAXIQUE*****  
Dans la ligne 7 ,erreur numero 18 : Erreur de la syntaxe
```

- **Programme 2:**

Erreur :Parenthèses non fermée.

```
*****
program test;
var x,y;
begin
x:=(y+y*9
read y
end.
*****
```

Résultat:

```
Fin de l'analyse lexicale -- succes --
*****ANALYSEUR SYNTAXIQUE*****
Dans la ligne 4 ,erreur numero 13 : Une ')' est attendue
```

- **Programme 3:**

Erreur :Pas de 'begin'.

```
*****
program test;
var x,y;
y:=9;
x:=5
end.
*****
```

Résultat:

```
Fin de l'analyse lexicale -- succes --
*****ANALYSEUR SYNTAXIQUE*****
Dans la ligne 2 ,erreur numero 14 : Pas de 'BEGIN'
```

- **Programme 4:**

Erreur : Pas de 'end'.

```
*****  
program test;  
var x,y;  
begin  
y:=9  
*****
```

Résultat:

```
Fin de l'analyse lexicale -- succes --  
  
*****ANALYSEUR SYNTAXIQUE*****  
Dans la ligne 4 ,erreur numero 15 : Il manque 'end' a la fin du programme
```

- **Programme 5:**

Erreur : Pas de point à la fin.

```
*****  
program test;  
var x,y;  
begin  
y:=9  
end  
*****
```

Résultat:

```
Fin de l'analyse lexicale -- succes --  
  
*****ANALYSEUR SYNTAXIQUE*****  
Dans la ligne 5 ,erreur numero 8 : Ajoutez point a la fin du programme
```

- **Programme 6:**

Erreur : 'Do' maquante.

```
*****
program test;
var x,y;
begin
    begin
        for x:=3 into 10
            write(x)
        end
    end
end.
*****
```

Résultat:

```
Fin de l'analyse lexicale -- succes --
*****ANALYSEUR SYNTAXIQUE*****
Dans la ligne 5 ,erreur numero 19 : 'do' attendue apres la boucle
```

- **Programme 7:**

Erreur : 'then' maquante.

```
*****
program test;
var x,y;
begin
    begin
        for x:=3 into 10 do
            if x <> 6
                write(x)
            end
        end
    end
end.
*****
```

Résultat:

```
Fin de l'analyse lexicale -- succes --
*****ANALYSEUR SYNTAXIQUE*****
Dans la ligne 6 ,erreur numero 17 : 'then' attendue apres if
```


- **Programme 8:**

Erreur : Condition incorrecte.

```
*****
program test;
var x,y;
begin
    begin
        for x:=3 into 10 do
            if x:= 6 then
                write(x)
            end
        end
    end
end.
*****
```

Résultat:

```
Fin de l'analyse lexicale -- succes --
*****ANALYSEUR SYNTAXIQUE*****
Dans la ligne 5 ,erreur numero 16 : Condition incorrecte
```

- **Programme 9:**

Erreur : 'until' manquante après 'repeat'.

```
*****
program test;
var x,y;
begin
    begin
        repeat x:= x+1
        end
    end
end.
*****
```

Résultat:

```
Fin de l'analyse lexicale -- succes --
*****ANALYSEUR SYNTAXIQUE*****
Dans la ligne 5 ,erreur numero 21 : 'until' attendue apres 'repeat'
```

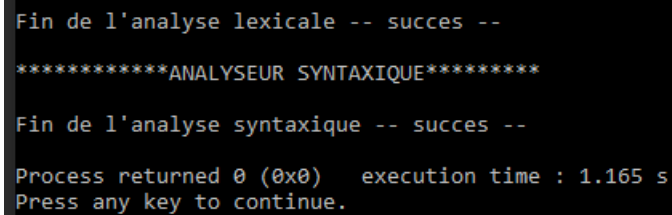
3.0.2 Programmes sans erreurs

On va conclure avec 2 exemples de programme PASCAL qui sont reconnus par les deux analyseurs.

- **Programme 1:**

```
*****
program test;
var x1,x2,x3,x4,x5;
begin
  { *Demander a l'utilisateur d'entrer les variables* }
  read(x1,x2,x3,x4);
  x5:=x1+x2+x3+x4;
  { *afficher a l'utilisateur la somme* }
  write(x5)
end.
*****
```

Résultat:

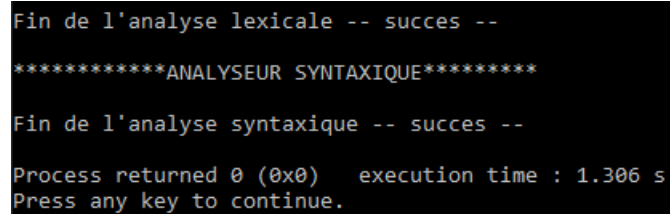


```
Fin de l'analyse lexicale -- succes --
*****ANALYSEUR SYNTAXIQUE*****
Fin de l'analyse syntaxique -- succes --
Process returned 0 (0x0)   execution time : 1.165 s
Press any key to continue.
```

- **Programme 2:**

```
*****  
program factoriel;  
var n,i;  
begin  
n:=1;  
    begin  
        for i:=0 into 100 do  
            n:=(n*(n+1))  
        end;  
    write(n)  
end.  
*****
```

Résultat:



```
Fin de l'analyse lexicale -- succes --  
*****ANALYSEUR SYNTAXIQUE*****  
Fin de l'analyse syntaxique -- succes --  
Process returned 0 (0x0) execution time : 1.306 s  
Press any key to continue.
```