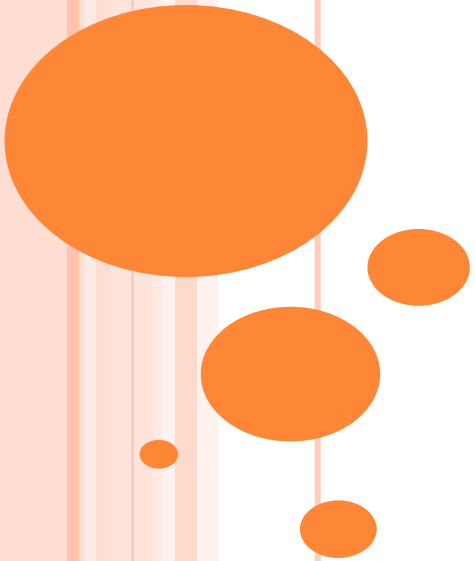


TP Compil
Analyse syntaxique & Table des symboles

L'ANALYSE SYNTAXIQUE

- L'analyse syntaxique permet de vérifier que les unités lexicales (le résultat de l'analyse lexicale) sont dans le bon ordre défini par le langage.

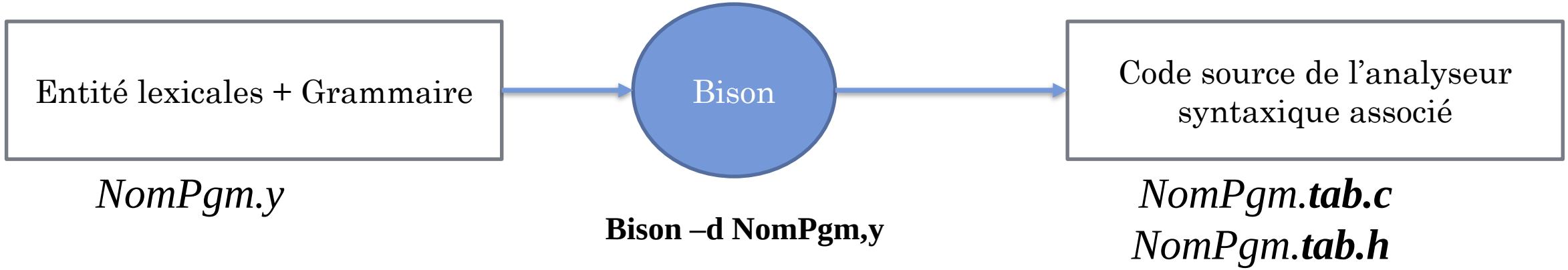
- **Exemple** : Dans le langage C, l'instruction If X==2) y=1; présente une erreur syntaxique → Dans une instrucion IF, la présence de la parenthèse ouvrante avant la condition est obligatoire

- Implémenter un analyseur syntaxique nécessite l'implémentation d'une méthode d'analyse syntaxique vue en cours LL(k), LR(k), SLR(k), LALR(k), ..etc. Ceci nécessite l'écriture de milliers de lignes de code.

- **D'où l'utilisation de l'outil Bison**

L'ANALYSE SYNTAXIQUE « BISON »

- Bison est un générateur de code d'analyseur syntaxique.
- Il accepte comme entrée la sortie de l'analyse lexicale (les entités lexicales), et la grammaire de langage à analyser (les fichiers bison portent l'extension « .y »).



- **Format d'un fichier Bison**
- Le format d'un fichier Bison est similaire à celui de Flex. Il est composé de trois parties séparées par '%%'.

LA STRUCTURE D'UN FICHIER BISON

%{

Définitions en langage C

%}

Les définitions des terminaux et d'axiome

%%

Partie 1

Les règles de la grammaire

%%

Partie 2

Redéfinitions des fonctions prédéfinies

Partie 3

LA STRUCTURE D'UN FICHIER BISON

La partie 1:

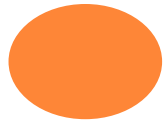
2.1. Déclaration C (pré-code)

Cette partie peut contenir les en-têtes, les macros et les autres déclarations C nécessaire pour le code de l'analyseur syntaxique.

2.2. Définitions et options

Cette partie contient toutes les déclarations nécessaires pour la grammaire à savoir:

a) **Déclaration des symboles terminaux**
b) **Définition des priorités et d'associativité**
c) **Autres déclarations**



LA STRUCTURE D'UN FICHIER BISON

A. Déclaration des symboles terminaux

- ceci est effectué en utilisant le mot clé %token.

Exemple : **%token** MAIN IDF Accolade PointVirgule

- On peut préciser le type d'un terminal par **%token<type> nom_terminal**

Exemple :

%token <int> entier

%token <chaine> chaine_cara

LA STRUCTURE D'UN FICHIER BISON

B. Définition des priorités et d'associativité

- l'associativité est définie par les mots clé :

%left, %right et %nonassoc.

la priorité est définie selon l'ordre de déclaration des unités lexicales.

Exemple :

%left A B /*associativité de gauche à droite*/

%right C D /* associativité de droite à gauche*/

Ordre de priorité

%nonassoc E F /* pas d'associativité*/

LA STRUCTURE D'UN FICHIER BISON

C. Autres déclarations

- **%start :** permet de définir l'axiome de la grammaire.

- En l'absence de cette déclaration, Bison considère le premier non-terminal de la grammaire en tant que son axiome.

- **%type :** définir un type à un symbole non-terminal.

- **%union :** permet de spécifier tous les types possibles pour les valeurs sémantiques.

LA STRUCTURE D'UN FICHIER BISON

- **La partie 2:** les règles de production

Ici, on décrit la grammaire LALR(1) du langage à compiler et les routines sémantiques à effectuées selon la syntaxe suivante :

<symbole NonTerminal> : <règle de dérivation 1> {action 1 en langage C }
| <règle de dérivation 2> {action 2 en langage C }
| ...
| <règle de dérivation N> {action N en langage C}

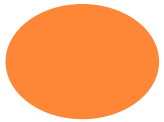
LA STRUCTURE D'UN FICHIER BISON

- **Partie 3:** Post-code C

C'est le code principal de l'analyseur syntaxique. Il contient le main ainsi que les définitions des fonctions.

→ Elle doit contenir au minimum les deux fonctions suivantes.

```
main ()
{ yyparse(); }
yywrap ()
{}
```



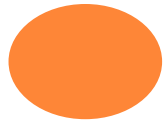
LA STRUCTURE D'UN FICHIER BISON

Important:

Bison vous affiche le message d'erreur " Syntax error " s'il recncontre une erreur lexicale dans le fichier source

Afin de pourvoir modifier le message d'erreur, on doit redéfinir la fonction yyerror :

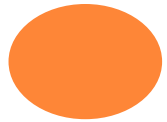
```
int yyerror(char *msg)        /* la signature de la fonction */
{
    printf(" Erreur syntaxique a la ligne %d ", nb_ligne);
   return 1;
}
```



Le lien entre Flex et Bison

- Afin de lier FLEX à BISON, On doit ajouter dans la partie C du FLEX l'instruction suivante:

```
%{
        # include " NomPgm.tab.h "
%}
```



Le lien entre Flex et Bison

- **Exemple**
- Création de compilateur lexical/syntaxique pour le langage: x=5;

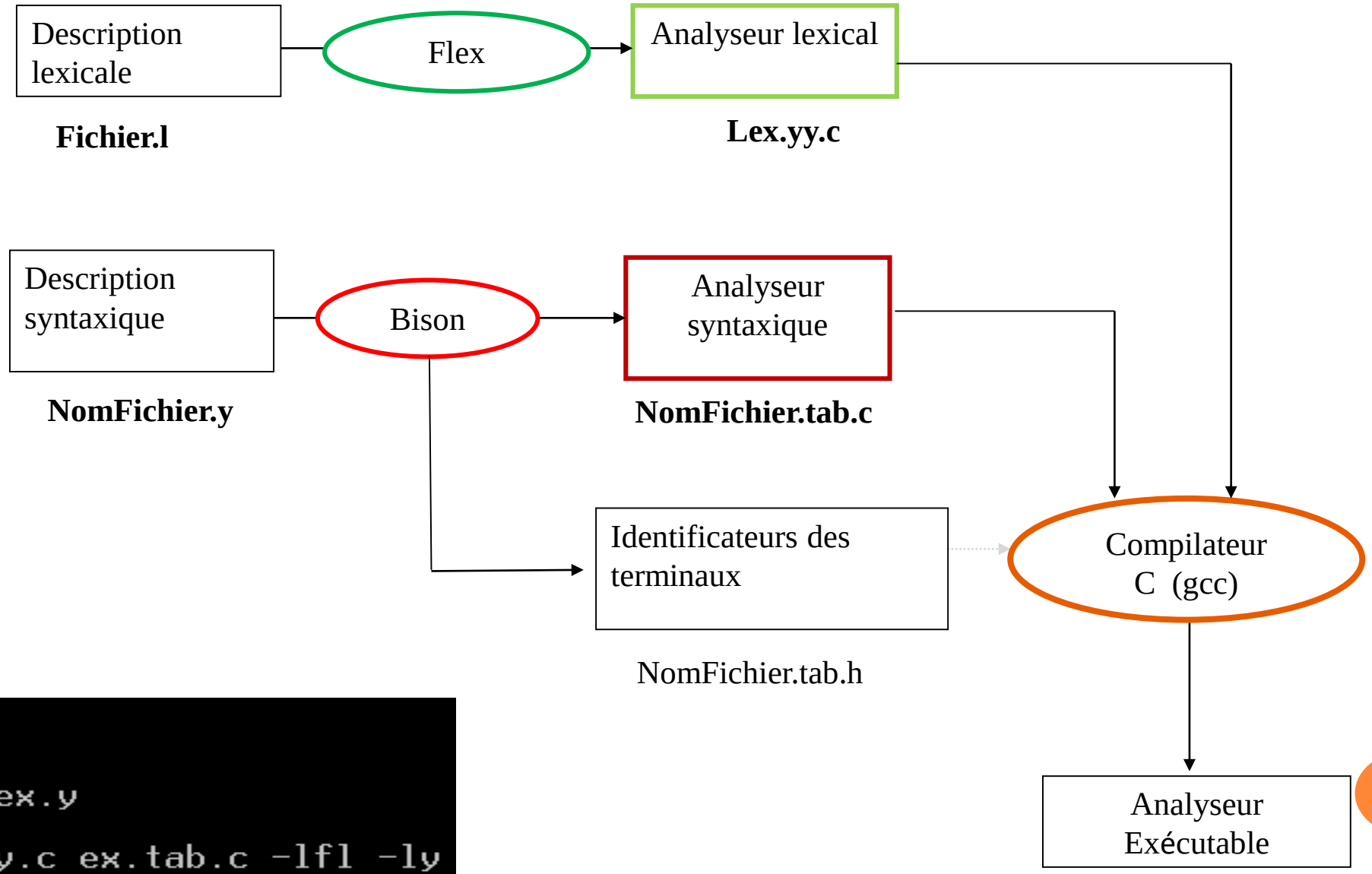
Lexical.l

```
%{
#include “Syntaxique.tab.h"
extern   nb_ligne;
%}
lettre [a-zA-Z]
chiffre [0-9]
IDF {lettre}({lettre}|{chiffre})*
cst {chiffre}+
%%
{IDF}  return idf;
{cst}    return cst;
=        return aff;
;        return pvg;
[ \t]
\n   {nb_ligne++; }
. printf("erreur lexicale à la ligne %d \n",nb_ligne) ;
```

Syntaxique.y

```
%{
      int nb_ligne=1;
%}
%token idf   cst   aff  pvg
%%
S: idf aff cst pvg {printf("syntaxe correcte");
                    YYACCEPT;}
;
%%
main ()
{
yyparse();
}
yywrap()
{}
int yyerror(char *msg)
{ printf(" Erreur syntaxique");
   return 1;
}
```

COMMANDE DE COMPILATION ET D'EXÉCUTION FLEX/ BISON




```
D:\Exp>flex ex.l
D:\Exp>bison -d ex.y
D:\Exp>gcc lex.yy.c ex.tab.c -lfl -ly
```

VARIABLES ET FONCTIONS PRÉDÉFINIES DE BISON

- **YYACCEPT :** instruction qui permet de stopper l'analyseur syntaxique en cas de succès.

- **main () :** elle doit appeler la fonction yyparse (). L'utilisateur doit écrire son propre main dans la partie du bloc principal.

- **yyparse () :** c'est la fonction principale de l'analyseur syntaxique. On doit faire appelle à cette fonction dans la fonction main().

- **int yyerror (char* msg)** : lorsque une erreur syntaxique est rencontrée, *yyparse* fait appelle à cette fonction. On peut la redéfinir pour donner plus de détails dans le message d'erreur. Par défaut elle est définie comme suit:

```
int yyerror ( char*  msg ) {
printf ( " Erreur Syntaxique rencontrée\n " );
 }
```

Partie du fichier Flex du projet

```
%{
  #include "synt.tab.h"
  extern  nb_ligne;
  extern  col;
%}
lettre  [a-zA-Z]
chiffre [0-9]
IDF     {lettre}({lettre}|{chiffre})*
CST     {chiffre}+
CST_REAL {chiffre}+"."{chiffre}+
pt      "."
saut    \n
%%
```

```
%%
"DATA DIVISION."                { col = col + strlen(yytext); return mc_data;}
"IDENTIFICATION DIVISION."      { col = col + strlen(yytext); return mc_ident; }
PROGRAM-ID                      { col = col + strlen(yytext); return mc_pgm;}
"PROCEDURE DIVISION."           { col = col + strlen(yytext); return mc_proce;}
"WORKING-STORAGE SECTION."      { col = col + strlen(yytext); return mc_work;}
"STOP RUN."                     { col = col + strlen(yytext); return mc_stop;}
INTEGER                         { col = col + strlen(yytext); return mc_int;}
FLOAT                           { col = col + strlen(yytext); return mc_float;}
IF                              { col = col + strlen(yytext); return mc_if;}
{IDF}                           { col = col + strlen(yytext); return idf;}
{CST}                           { col = col + strlen(yytext); return cst;}
{CST_REAL}                      { col = col + strlen(yytext); return cst_reel;}
"."                             { col = col + strlen(yytext); return pt;}
":="                            { col = col + strlen(yytext); return aff;}
"|"                             { col = col + strlen(yytext); return sep;}
"("                             { col = col + strlen(yytext); return p_o;}
")"                             { col = col + strlen(yytext); return p_f;}
[ \t]                           {col = col + strlen(yytext);}
{saut}        {nb_ligne=nb_ligne+1; col=1;}
. printf("L entitie lexicale  %s au niveau de la ligne Num %d
            et le colonne %d \n", yytext,nb_ligne, col);
%%
```

Partie Bison

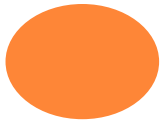
- Partie 1 du fichier synt.y

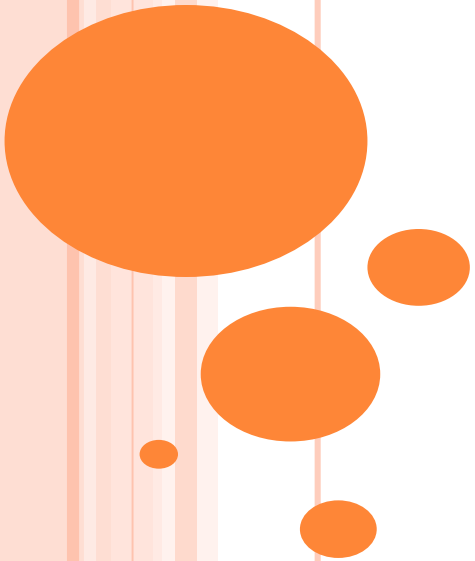
```
%{
 int nb_ligne=1;
 int col=1;
%}
%token    mc_pgm mc_data mc_proce mc_ident mc_work
%token  mc_stop mc_int mc_float mc_if idf cst
%token  cst_reel pt aff sep p_o p_f
%start  S
%%
```

```
return mc_data;}
return mc_ident; }
return mc_pgm;}
return mc_proce;}
return mc_work;}
return mc_stop;}
return mc_int;}
return mc_float;}
return mc_if;}
return idf;}
return cst;}
return cst_reel;}
return pt;}
return aff;}
return sep;}
return p_o;}
return p_f;}
```

- Partie 3 du fichier synt.y

```
%%
int yyerror(char *msg)
{
   printf ("Erreur syntaxique a la ligne %d et la colonne %d ", nb_ligne, col );
   return 1;
}
main()
{
yyparse();
}
yywrap()
{}
```



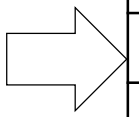


La table des symboles

La table des symboles

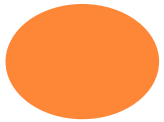
- La table de symbole doit être programmée manuellement dans flex et bison.
- La table de symbole doit contenir des information sur les entités

```
PROGRAM   TpCom
integer x;
Const  reel  y=5;
BEGIN
 x=x+1,5;
END ;
```



state	name	code/nature	Type	val	Taille
1	TpComp	IDF			
1	X	IDF			
1	Y	IDF			
1	5	const	int	5	
1	1,5	const	reel	1,5	

- Dans ce TP, nous distingions 3 types de TS
  - La TS des IDFs et des constantes
  - La TS des séparateurs
  - La TS des mots clés



- Chaque entité reconnu par le langage doit être insérée dans la table des symbole.

```
typedef  struct
{
  int    state;
  char name[20];
  char code [20];
  char type[20];
  float val;

} element;
```

```
typedef struct
{
  int    state;
  char name[20];
  char type[20];
} elt;
```

La table des symboles des constantes, variables

element tab[1000];

La table des symboles des séparateurs et des mots clés

elt tabs[40], tabm[40];

- **State: permet de vérifier si la ligne dans la TS est utilisée pour stockée une entité**
- **Name :** Nom de l'entité
- **Code :** variable simple, tableau, constante ...etc
- **Type :** type de l'entité (INT, Float ...etc)

```
void initialisation()
{
  int i;
  for (i=0;i<1000; i++)    Tab[i]. state=0;
  for (i=0;i<50;i++)
    {
      TabS[i].state=0;
      TabM[i].state=0;
    }
}
```

- Permet de vérifier si l'entité existe déjà dans la TS

y: permet de spécifier la Ts dans laquelle la recherche est faite
0: TS des IDFs
1: TS des mots clés
2: TS des séparateurs

```
void rechercher (char entite[ ], char type [ ], float val, int y)
{
  int j,i;
  Switch (y)
  {
    Case 0:  /*verifier si la case dans la tables des IDF et CONST est libre*/
              FOR (i=0;  (
                          (   i<1000)            &&     /* on a pas atteint la fin de la table*/
                          (Tab[i].state==1))    &&     /* la case est utilisée pour stockée une EL */
                          (strcmp (entite,   Tab[i].name)!=0) /* L'entité lexicale n'existe pas */
                    ; i++) ;
             IF (i<1000)
                 Inserer ( entite,  type,  val, i ,   0);
            ELSE
               printf("entité existe déjà\n");
     break;
```

Position

Case 1: /*verifier si la case dans la tables des mots clés est libre*/

```
FOR (i=0;((i<40)&&(TabM[i].state==1))&&(strcmp(entite,Tab[i].name)!=0);i++);

  IF (i<40)
          inserer(entite,type,val,i,1);
  ELSE
          Printf ("entité existe déjà\n");
   break;
```

```
Case 2: /*verifier si la case dans la tables des séparateurs est libre*/
        FOR (i=0;((i<40)&&(TabS[i].state==1))&&(strcmp(entite,TabS[i].name)!=0);i++);
        IF(i<40)
                inserer(entite,type,val,i,2);
        ELSE
                printf("entité existe déjà\n");
        break;
  } /* fin switch
} :* fin recherche
```

```
void inserer (char entite[], char type[], float val, int i, int y)
{
   switch (y)
 {
  Case 0: /*insertion dans la table des IDF et CONST*/
      Tab[i].state=1;
      Strcpy (Tab[i].name, entite);
      Strcpy (Tab[i].type, type);
      tab[i].val=val;
      break;

  Case 1: /*insertion dans la table des mots clés*/
      TabM[i].state=1;
      Strcpy (TabM[i].name,entite);
      Strcpy (TabM[i].type,type);
      break;

  Case 2: /*insertion dans la table des séparateurs*/
     TabS[i].state=1;
     Strcpy (TabS[i].name,entite);
     Strcpy (TabS[i].type,type);
      break;
 }
```

TS.H

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

typedef struct
{
   int state;
   char name[20];
   char type[20];
   float val;
   int sub;
} element;

typedef struct
{
   int state;
   char name[20];
   char type[20];
} elt;

element tab[1000];
elt tabs[40],tabm[40];
```

```
void initialisation();
{

}

void inserer (char entite[], char type[],float val,int i,int y)
{

}

void recherche (char entite[], char type[],float val,int y)
{

}

void afficher()
{

}
```

EXEMPLE

```
%{
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include "TS.h"
int yylineo=1;
int Col=1;
%}
lettre [a-zA-Z]
chiffre [0-9]
idf
{lettre}({lettre}|{chiffre})*
entier {chiffre}+
real {chiffre}+"."{chiffre}+
blanc [ \t]
Sl [ \n]
%%
```

```
"/" {
      rechercher (yytext,"sep",0,2);
      Col = Col + strlen(yytext);
    }

"=" {
      rechercher (yytext,"sep",0,2);
      Col = Col + strlen(yytext);
    }

{idf} {
        Rechercher (yytext," ",0,0);
        Col = Col + strlen(yytext);
     }

{real} {
      rechercher (yytext,"real",atof(yytext),0);
      Col = Col + strlen(yytext);
      }

{entier} {
          rechercher (yytext,"entier",atoi(yytext),0);
          Col = Col + strlen(yytext);
        }
{blanc} {Col = Col + strlen(yytext);}

{Sl} {yylineo++;Col=1;}
```

EXEMPLE (PARTIE 2)

```
. { printf  ("\n Erreur lexical:  Ligne: %d; Collone: %d; Entité << %s >>  non reconnu par le langage \n",yylineo, Col, yytext);
Col = Col + strlen(yytext);
}


%%
int main()
{
     initialisation();
     yyin = fopen( "input.txt", "r" );
    if (yyin==NULL) printf("ERROR  \n");
     else yylex();
    afficher();
    return 0;
 }
```