TP DE COMPILATION

Flex et Bison

Dr. Hadjer MOULAI

INTRODUCTION

La compilation est un processus que les êtres humains utilisent dans la vie courante afin de communiquer entre eux.

En effet quand nous lisons ou entendons une phrase notre cerveau va:

1. **Reconnaitre les mots composant la phrase (analyse lexicale)**
2. **Déterminer si la phrase est grammaticalement correcte (analyse syntaxique)**
3. **Déterminer si la phrase a un sens (analyse sémantique)**

PROCESSUS DE COMPILATION

- **Analyse lexicale:** vérifie que les entités de l'instruction appartiennent au langage

Exemple: Je suis étudiant

Tous les mots appartiennent à la langue française

- **Analyse syntaxique:** vérifie que l'instruction appartient à la grammaire

Exemple: Je étudiant suis

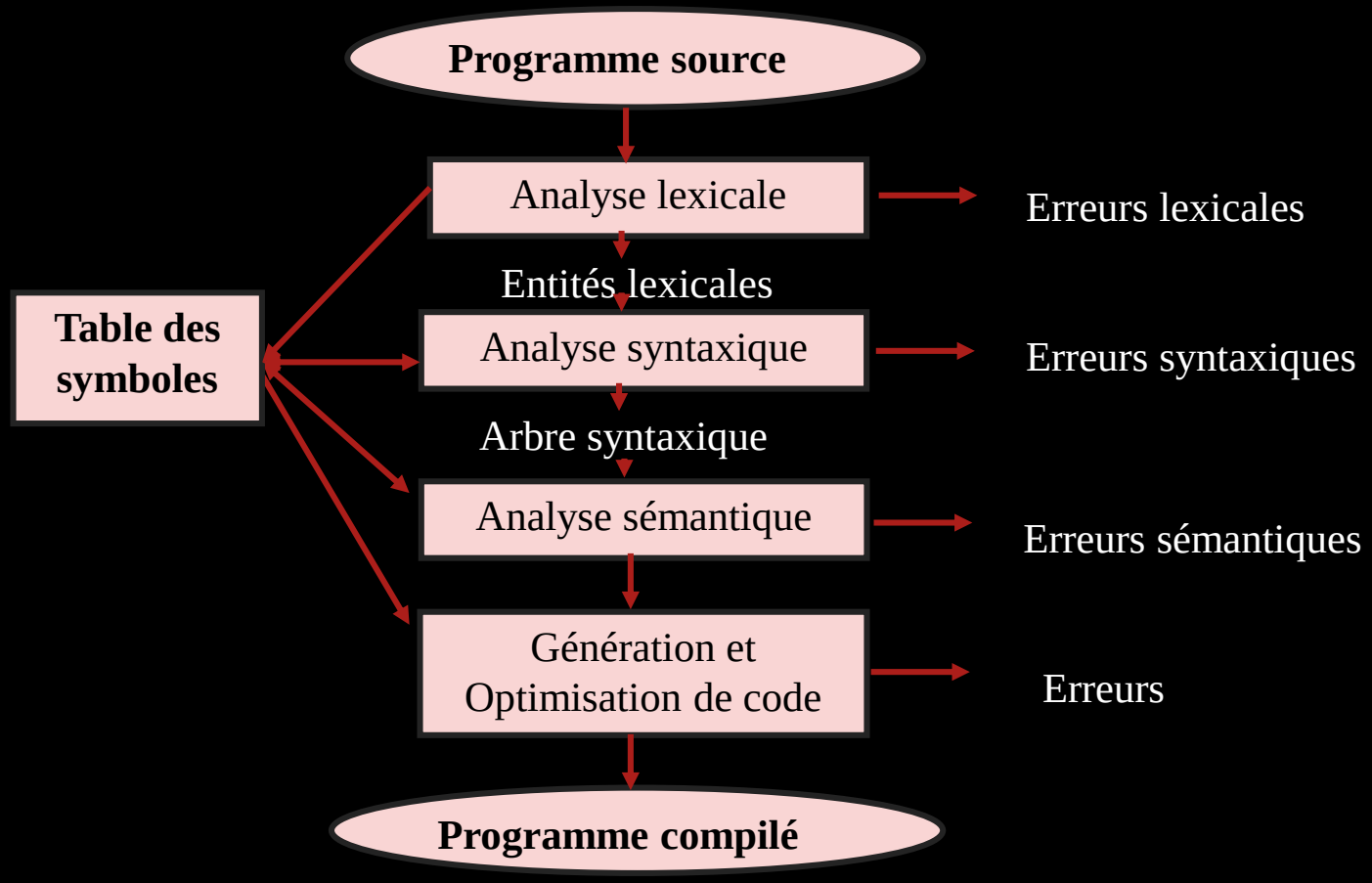
Ne respecte pas les règles de la grammaire mais lexicalement correcte

- **Analyse sémantique:** vérifie que l'instruction a un sens

Exemple: Je mange un bureau

Lexicalement et syntaxiquement correcte mais sémantiquement fausse

ÉTAPES DE COMPILATION



Exemple :

Soit le programme C suivant :

```
main ()
{
    int x, y  ;
    if x==2) y=1;
    x=3,14;
 }
```

- **Le programme est correcte lexicalement,**
- **Le programme est syntaxiquement incorrecte** : manque de la (dans l'instruction if
- **Le programme est sémantiquement incorrecte :** x = 3,14

ANALYSE LEXICALE

Dans l'analyse lexicale il faut faire en sorte qu'à chaque entité lexicale du langage correspond un automate d'états finies.

Comment programmer des automates ? ⟹ **Expressions régulières**

- Les expressions régulières sont une écriture formelle des automates.
- Dans une expression régulière tous les caractères sont significatifs, y compris les espaces.
- Les caractères dans le tableau suivant sont réservés : ils doivent être précédés par \ ou notés entre " " si l'on veut leur valeur littérale.

EXPRESSIONS RÉGULIÈRES

caractères	significations	exemples
+	Répéter 1 ou plusieurs fois	x+ x...x
*	Répéter 0 ou plusieurs fois	t* vide ou t...t
?	0 ou 1 fois	a? vide ou a
|	union	ab|bc ab ou bc
( )	factorisation	(a|b)c ac ou bc
"	valeur littérale des caractères	"+?"+ +?...+?
\	valeur littérale de caractère qui le précède même entre ""	\++ +...+ "\"+ erreur car la première " ne se ferme pas.
.	N'importe quel caractère sauf la fin de ligne (\n).	.|\n n'importe quel caractère

EXPRESSIONS RÉGULIÈRES

caractères	significations	exemples
[...]	ensemble de caractères.	[aeiou] la voyale :a ou e ou i ou o ou u.
-	Utilisé dans [] signifie un intervalle d'ensemble	[0-9] 0|1|2|3|4|5|6|7|8|9
^	Utilisé dans [] signifie le complément d'ensemble	[^0-9] tout caractère sauf chiffre

Attention :

[...] = ensemble de caractères, pas d'expressions régulières, ce qui veut dire:

[(0|1)+] un des caractères (,), 0, 1, |, +, pas une suite de 0 ou 1.

Mais les caractères ^, \, - restent des caractères spéciaux.

[^\]] tout caractère sauf]

EXPRESSIONS RÉGULIÈRES

Exercice:

Donner une expression régulière qui définit les entités suivantes :

1. **Une suite de caractères alphanumériques qui commence par un caractère alphabétique.**

   [a-zA-Z]([a-zA-Z]|[0-9])*

2. **Les constantes numérique signées par le (-/+).**

   [+-]?([1-9][0-9]*|0) ([+-][1-9][0-9]*)|0

3. **N'importe quel caractère (la fin de ligne (\n) est inclue).**

   .|\n

4. **Tous les caractère à part les espaces, les tabulations et les fins de lignes.**

   [^ \n\t]

ANALYSE LEXICALE

L'analyse lexicale est la première étape du processus de compilation. Ses tâches principales sont les suivantes:

1. Lire les caractères d'entrée et produire comme résultat une suite d'entités lexicales que l'analyseur syntaxique aura à traiter.
2. Eliminer les caractères superflus tels que les commentaires, les tabulations, fin de lignes, etc.
3. Gérer les numéros de ligne dans le programme source pour pouvoir associer à chaque erreur rencontrée par la suite la ligne lui correspondant.

Exemple :

Code source avant compilation :

```
If (x==2)          y=z ; /* affectation*/
```

Après l'analyse lexicale :

```
Mc_If  ( Idf  Egal  Const  )  Idf  Aff  Idf  pvg
```

ANALYSE LEXICALE

L'analyseur lexical est basé sur l'algorithme simple suivant :

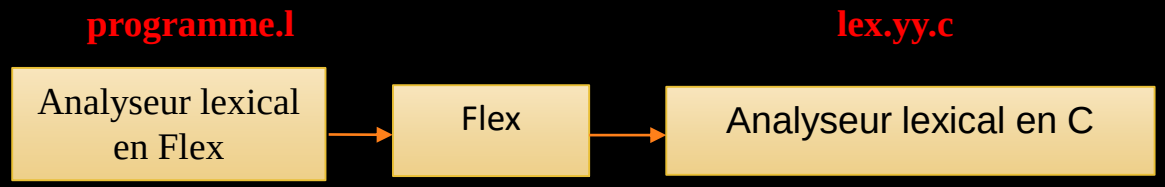
```
Lire (ChaineEntrée) ;
Switch (ChaineEntrée){
Case (ExpReg1) : coder(« Entité1 ») ;
Case (ExpReg2) : coder (« Entité2 ») ;
....
Case (ExpRegN) : coder (« EntitéN ») ;
Default : écrire (« Erreur lexicale ») ;
}
```

L'implémentation d'un tel algorithme nécessitera des milliers de lignes de code.

Heureusement que des outils logiciels, tel que **Flex**, existent pour nous faciliter la tâche.

FLEX

Flex est un traducteur qui traduit notre analyseur lexical écrit dans un langage simple (flex) vers le langage C.



Format d'un fichier Flex :

Un fichier Flex est composé de trois parties séparées par '%%'.

```
%{
Définitions en langage C
%}
Les définitions des expressions régulières
%%
ExpressionRégulière   { Action C}
%%
Redéfinitions des fonctions prédéfinies
```

Partie 1: les expressions régulières

Partie 2: les règles de traduction

Partie 3

FLEX

Partie 1:

- **But :** donner les ER des différentes entités lexicales du langage sous la forme suivante :

<identificateur_de_l'entité> <expression_régulière>

identificateur_de_l'entité : doit commencer par une lettre et ne comporte que des caractère alphanumériques, des _ ou bien des -

expression_régulière: doit être valide

Exemple: lettre [a-zA-Z]

FLEX

Partie 2:

- **But :** présenter l'ensemble des actions associées à chaque entité lexicale sous la forme suivante:

<pattern> <action>

pattern : doit commencer par une lettre et ne comporte que des caractère alphanumériques, des _ ou bien des -

action: c'est un code C qui sera exécuté à chaque fois qu'une entité lexicale appartenant au langage est reconnue

Exemple :

{entier } {printf (''L'entité reconnu est un entier'') ; }

FLEX

Exemple :

Un analyseur lexicale pour le langage

X=5; $

```
%{
#include<stdio.h>
 int nb_ligne=1;
%}
lettre [a-zA-Z]
chiffre [0-9]
IDF [a-zA-Z]([a-zA-Z] | [0-9])*
cst {chiffre}+
%%
{IDF}  printf ("IDF \n");
{cst}   printf (‘’%s’’, yytext) ;
"="      printf ("affectation\n");
";" printf("pvg\n");
[ \t]
\n   {nb_ligne++; }
. printf("erreur lexicale à la ligne %d \n",nb_ligne) ;
%%
int main ()
{  yylex () ;  return 0; }
```

FLEX

Commandes de compilation:

- Pour compiler le programme :

flex MonTP.l

cc lex.yy.c -o TP -lfl (Linux)

gcc lex.yy.c -o TP -lfl (Windows)

- Pour exécuter le programme :

.\TP

- Pour arrêter le programme :

Ctrl + c

MACRO-ACTION DE FLEX

Fonctions	yylex	Permet de lancer l'analyseur lexical
	yywrap	Elle est appelée par le Lexer quand il rencontre la fin du fichier. Elle doit soit obtenir un nouveau flux d'entrée et retourner simplement la valeur 0 soit renvoyer 1 signifiant que la totalité des flux a été consommée et que le lexer a fini sa tâche.
	yyterminate	Permet de provoquer la fin d'exécution du lexer
	ECHO	Affiche l'unité lexicale reconnue
Variables	yytext	Récupère le texte formant le lexème reconnu
	yyleng	Détermine la langueur du texte contennue dans yytext
	yylval	Est une variable globale utilisé par FLEX pour stocker la valeur correspondante au Token reconnu
	yylineno	Est le numéro de la ligne courante
	yyin	Fichier d'entrée
	yyout	Fichier de sortie

ANALYSE SYNTAXIQUE

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

- **Exemple :** Dans le langage C

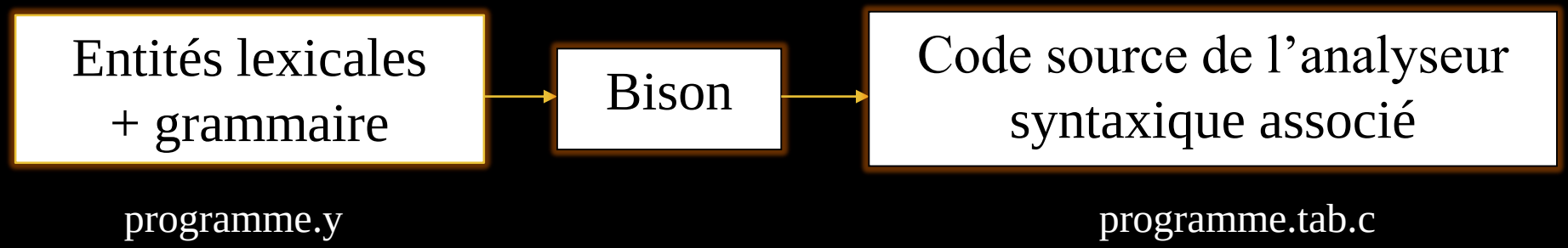
  if X==2) y=1;

est une erreur de syntaxe car la règle de if impose une parenthèse avant la condition

- 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.
- Heureusement que des outils logiciels, tel que **Bison**, existent pour nous faciliter la tâche.

BISON

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



BISON

Format d'un fichier Bison:

Un fichier Bison est composé de trois parties séparées par '%%'.

%{

Définitions en langage C — Partie1

%}

Les définitions des terminaux et d'axiome — alphabet du langage/entités lexicales

%%

Les règles de la grammaire — Partie2

%%

Redéfinitions des fonctions prédéfinies — Partie3

BISON

Partie1:

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

- **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**

BISON

Partie1:

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>.**

Exemple :

%token<int> entier

%token<chaine> chaine_cara

BISON

Partie1:

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*/

BISON

Partie1:

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.

BISON

Partie1: Exemple

Avec les déclarations:

%rigth '='

%left '+' '-'

%left '*' '/'

l'entrée :

a = b = c * d - e - f * g

est analysée comme:

a = (b = (((c*d) -e) - (f*g))).

Priorité croissante

```
%union{ int entier;}
%token idf  <entier>const
%type<entier> S
%%
S: const '/' const {$$= $1/$3; YYACCEPT;}
;
```

BISON

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}
;

Exemple: **%token a b**
%%
S:aAb {printf(''règle de dérivation '');

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 ()
{}
```

LIEN 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 "
%}
```

LIEN FLEX ET BISON

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

Lexical.l

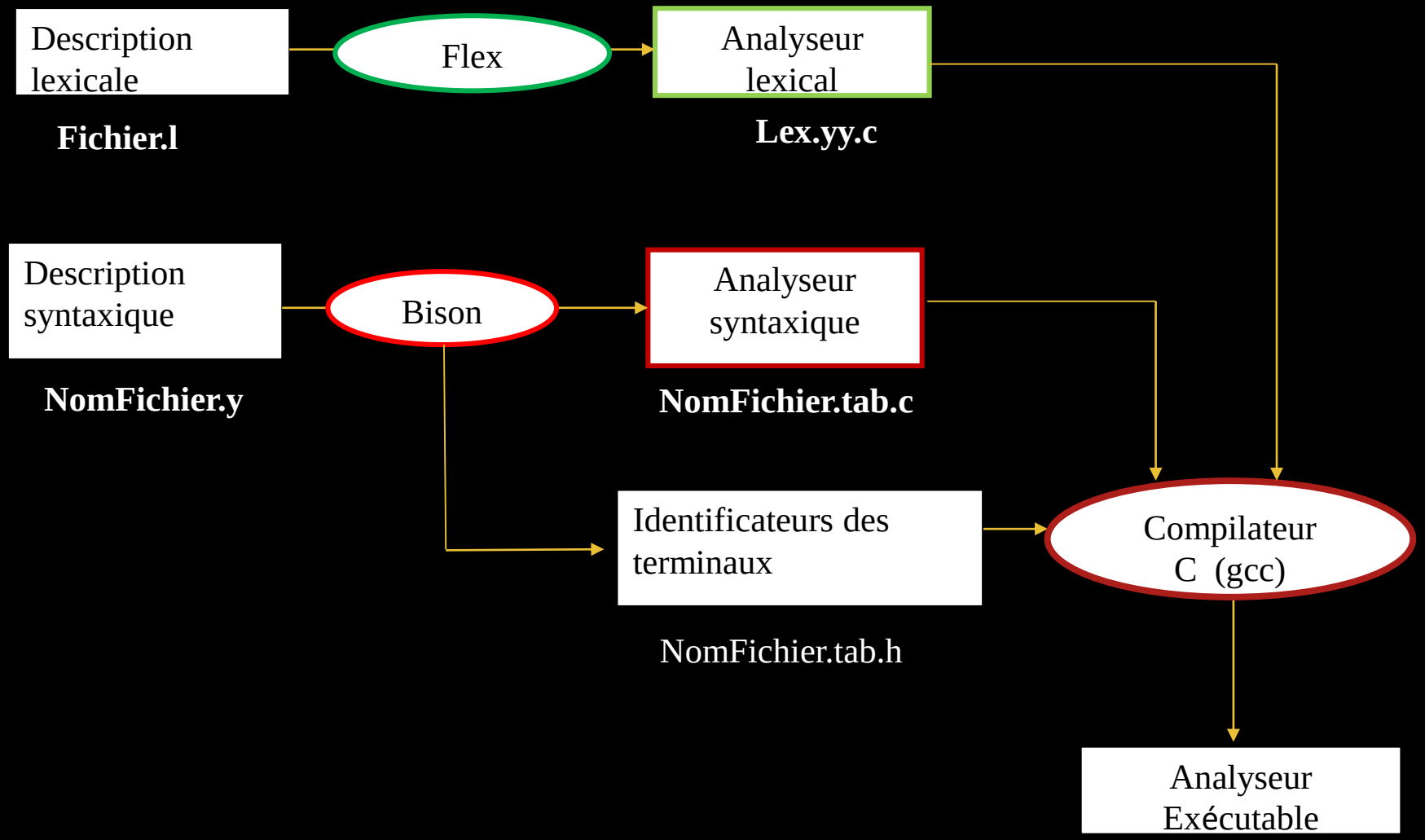
```
%{
#include “Syntaxique.tab.h"   //fichier partagé
int nb_ligne=1;
%}
lettre [a-zA-Z]
chiffre [0-9]
IDF {lettre}({lettre}|{chiffre})*
cst {chiffre}+
%%
{IDF}  return idf;
{cst}    return cst;
=        return aff;
;        return pvg;
[ \t]
\n   {nb_ligne++; }
. printf("erreur lexicale à la ligne %d \n",nb_ligne) ;
```

Syntaxique.y

```
%start S //optionnel car un seul axiome
%token idf   cst   aff  pvg //entités lexicales
%%
S: idf aff cst pvg {printf("syntaxe correcte");
                    YYACCEPT; //arrêter l’analyseur
}; //règle

%%
int yyerror(char *msg)
{ printf("%s error syntaxique");
   return 1;  }
main ()
{yyparse();}
yywrap(){}
```

COMMANDE DE COMPILATION ET D'EXÉCUTION



BISON

Commandes de compilation:

- Pour compiler le programme :

```
flex lexical.l
bison –d synt.y
gcc lex.yy.c synt.tab.c –lfl –ly -o tp
```

- Pour exécuter le programme :

```
tp<exemple.txt
```

VARIABLES ET FONCTIONS PRÉDÉFINIES DE BISON

- **YYACCEPT :** instruction qui permet de stopper l'analyseur syntaxique en cas de sucées.
- **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 " );
 }
```