كلـيـــــــــــة العلوم
السملالية - مراكش
FACULTE DES SCIENCES
SEMLALIA-MARRAKECH

MARRAKECH
جامعة القاضي عياض
UNIVERSITÉ CADI AYYAD

Département Informatique

Licence SMI – S6

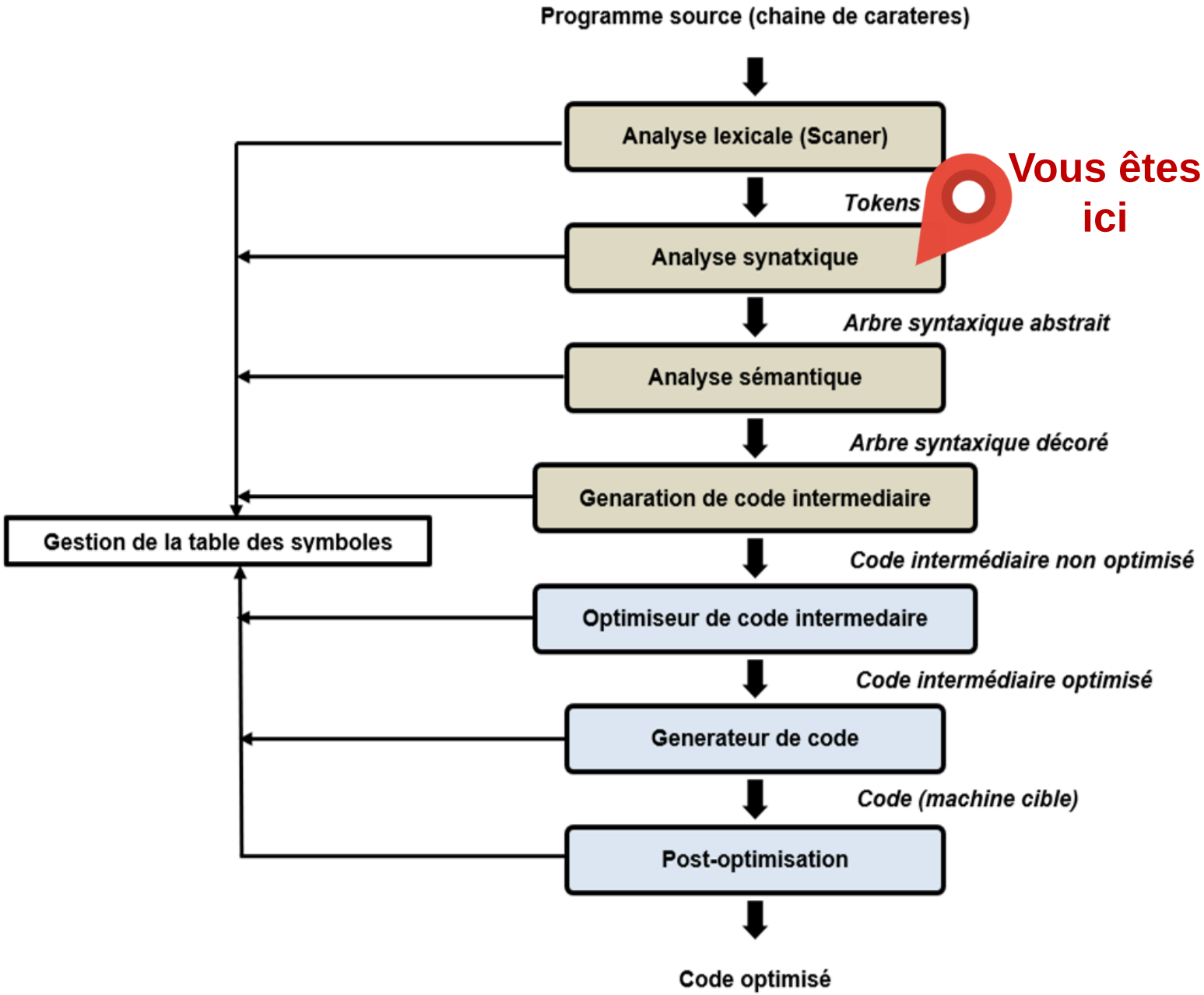
Compilation

Pr. Aimad QAZDAR

aimad.qazdar@uca.ac.ma

14 avril 2021

Rappel : Phases d'un compilateur



Chapitre 3: Partie 2
Bison

Un générateur d'analyseurs syntaxiques

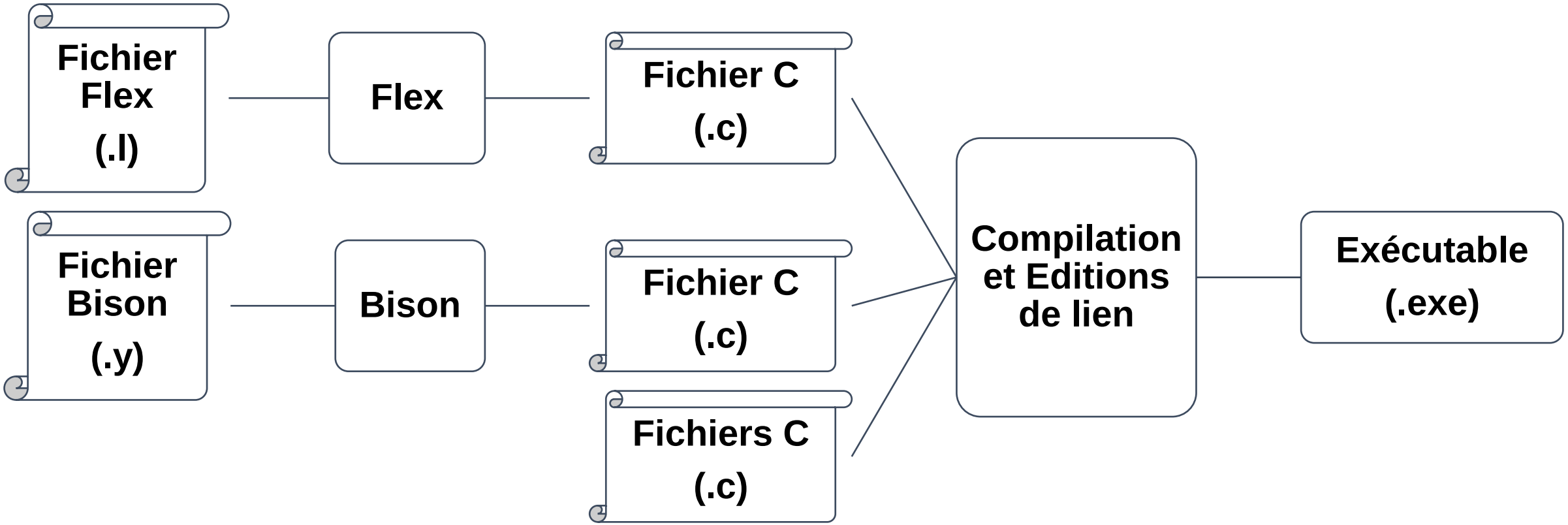
Plan

- Flex et Bison
- Structure d'un programme Bison
- Syntaxe Bison
- Association Flex/Bison
- Etapes de la construction
- Exemple pratique

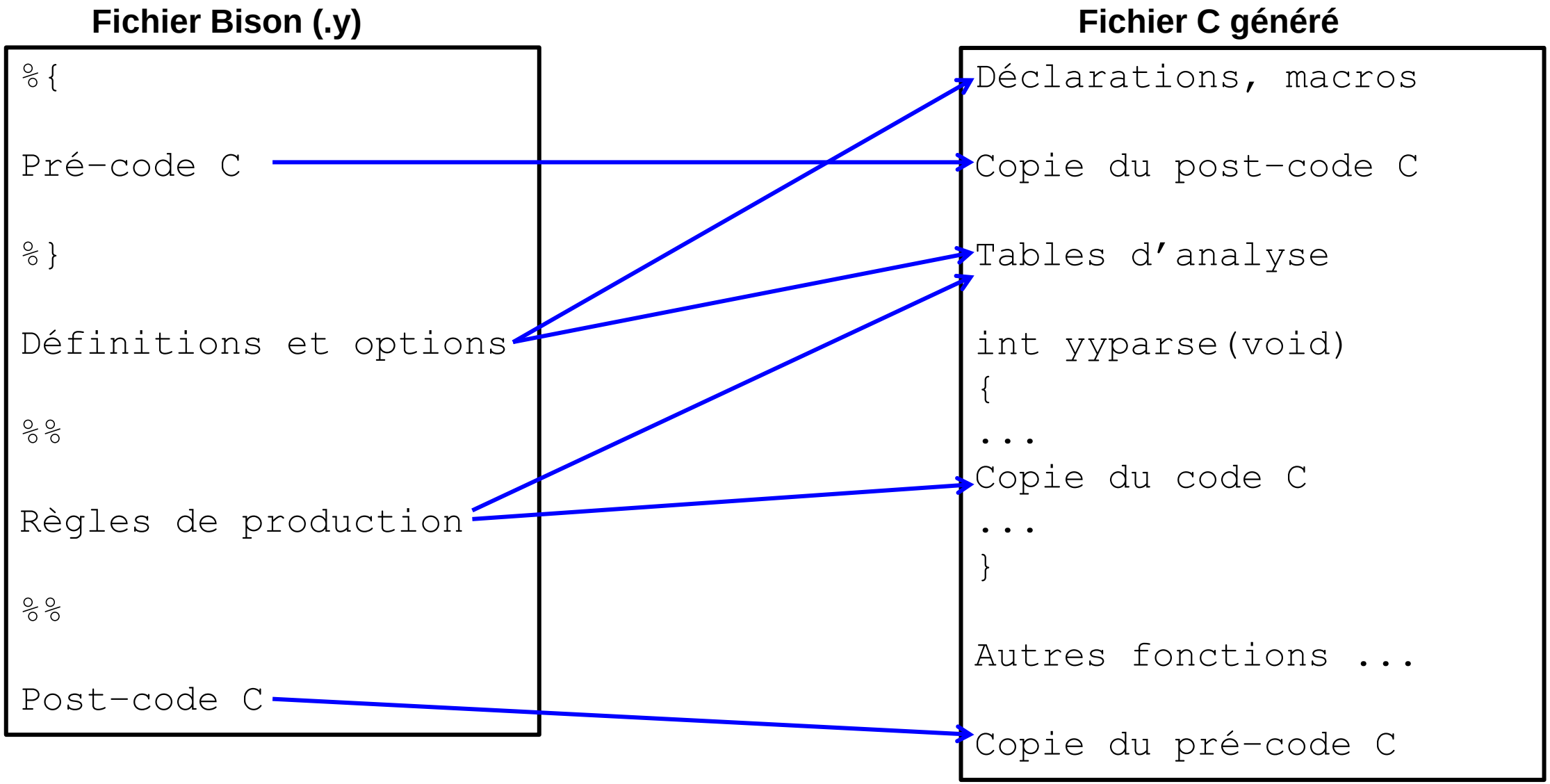
Flex et Bison

- **Lex/Flex : LEXical analyzer**
  - Reconnaisseur de langages réguliers
  - Expressions rationnelles → code C d'un analyseur lexical
  - Permet de reconnaitre les mots d'un langage
- **Yacc/Bison : Yet Another Compiler Compiler**
  - Reconnaisseur de langages non contextuels
  - Grammaire non contextuelle → code C d'un analyseur syntaxique
  - Permet de reconnaitre les phrases d'un langage
- Génération d'analyseurs statiques et non dynamiques
  - Le code C produit est spécifique au langage a reconnaitre → efficacité
  - Compilation du code génère comme le reste de l'application
  - Modification du langage → régénération du code des analyseurs

Flex et Bison



Structure d'un programme Bison



Syntaxe Bison

- Regles de production
  - non terminal : suite_de_symboles { /* code C */ }
    | autre suite { /* code C */ }
    | ...
    ;
- Les symboles sont représentés par des identificateurs
  - Terminaux : lexèmes provenant de l'analyse lexicale
  - Non-terminaux : symboles internes à l'analyseur syntaxique
- Le code C est optionnel et est exécute lorsque la règle est réduite

Syntaxe Bison

Mon langage

```
debut
a = 1;
c = a;
fin
```

Fichier Bison (.y)

```
%start program // l'axiome de notre grammaire
%%
program : DEBUT listInstr FIN {printf(" sqlt pgme\n");}
;

listInstr : listInstr inst
          | inst
;

inst : IDENTIF AFFECT expr PTVIRG {printf(" instr affect\n");}
;

expr : ENTIER      {printf(" expr entier \n");}
     | IDENTIF     {printf(" expr identif \n");}
;
%%
```

L'analyseur généré

- La fonction int yyparse(void)
  - Consomme les lexèmes obtenus par des appels à yylex() (à fournir) nt yylex(void);
  - Vérifier si la séquence de lexèmes permet de réduire l'axiome de la grammaire exprimée (%debut NT dans notre exemple)
  - Exécute les actions sémantiques (code C) associées aux règles réduites
  - Signale les erreurs a l'aide de la fonction yyerror() (à fournir)
- void yyerror(const char * msg);
  - Possibilité de récupération sur erreur pour poursuivre l'analyse
  - La valeur de retour
    - 0 si ok, 1 si erreur
    - Résultat de l'analyse complète → une seule invocation

Association Flex/Bison

- Flex fourni les lexemes à Bison
  - Bison invoque la fonction yylex() produite par Flex
  - Bison génère un .h de
  - yylex() doit renvoyer des constantes connues de Bison

  → %token IDENTIF ENTIER... dans les définitions de Bison
  - Bison génère un .h définissant ces constantes (et d'autres choses)
  - Le pré-code C de Flex inclue .h

Etapes de la construction

1. **bison -d –o syntaxeY.c syntaxe.y**
   - Produit le code C syntaxeY.c depuis le fichier Bison syntaxe.y
   - Option -d pour générer le .h syntaxeY.h
2. **flex –o lexiqueL.c lexique.l**
   - Produit le code C lexiqueL.c depuis le chier Flex lexique.l
   - Le pré-code C doit inclure syntaxeY.h
3. **gcc –o prog lexiqueL.c syntaxeY.c**
   - Créer l’exécutable (prog.exe)
4. **Prog < code.txt**
   - Analyser la syntaxe du fichier code.txt

Exemple pratique : objectif

- Créer un analyseur syntaxique pour ce code :

```
debut
Var = 1;
c = Var;
a=4;
fin
```

- Définir la lexique : **lexique.l**
- Définir la syntaxe: **syntaxe.y**
- Générer l'exécutable : **prog.exe**

Exemple pratique : définir la lexique

```
%{ /*----- lexique.l -------*/
extern int lineNumber; // definie dans syntaxe.y, utilise pour compter \n
#include "syntaxeY.h"  //fichier genere par syntaxe.y

%}
%option noyywrap
nbr [0-9]
entier {nbr}+
identif [a-zA-Z_][0-9a-zA-Z_]*
%%
debut        { return DEBUT; }
fin          { return FIN; }
[" "\t]      { /* rien */ }
{entier}     { return ENTIER; }
{identif}    { return IDENTIF; }
"="          { return AFFECT; }
";"          { return PTVIRG; }
"\n"         { ++lineNumber; }
.            { return yytext[0]; }
%%
```

Exemple pratique : définir la syntaxe (1)

```
%{/*-------- syntaxe.y ----------*/
#include <stdio.h>

extern FILE* yyin;    //file pointer by default points to terminal

int yylex(void); // defini dans lexiqueL.c, utilise par yyparse()
void yyerror(const char * msg); // definie dans syntaxe.y, utilise par
notre code pour .

int lineNumber; // notre compteur de lignes
%}

%token DEBUT FIN // les lexemes que doit fournir yylex()
%token IDENTIF ENTIER AFFECT PTVIRG

%start program // l'axiome de notre grammaire
%%
program : DEBUT listInstr FIN {printf(" squelette programme \n");}
;
```

Exemple pratique : définir la syntaxe (2)

```
listInstr : listInstr inst
          | inst
;

inst : IDENTIF AFFECT expr PTVIRG {printf(" instr affectation \n");}
;

expr : ENTIER      {printf(" expr ➔ entier \n");}
     | IDENTIF     {printf(" expr ➔ identif \n");}
;
%%
void yyerror( const char * msg){
      printf("line %d : %s", lineNumber, msg);
}
```

Exemple pratique : définir la syntaxe (3)

```
int main(int argc,char ** argv){
	if(argc>1) yyin=fopen(argv[1],"r"); // vérifier résultat !!!
	lineNumber=1;
	if(!yyparse())
		printf("Expression correct\n");

	return(0);
}
```

Exemple pratique : générer l'executable

1. bison -d –o syntaxeY.c syntaxe.y
2. flex –o lexiqueL.c lexique.l
3. gcc –o prog lexiqueL.c syntaxeY.c
4. Prog < code.txt

Exemple pratique : analyser le code

Code1.txt

```
debut
Var = 1;
c = Var;
a=4;
fin
```

Code2.txt

```
debut
Var = -1;
c = Var;
a=4;
fin
```

Code3.txt

```
debut
Var = 1;
c = Var;
a=4
fin
```

Résultat d'analyse syntaxique

```
>prog.exe < code.txt
 expr entier
 instr affect
 expr identif
 instr affect
 sqlt pgme
Expression correct
```

```
>prog.exe < code2.txt
line 2 : syntax error
```

```
>prog.exe < code3.txt
 expr entier
 instr affect
 expr identif
 instr affect
 expr entier
line 5 : syntax error
```