## Compilation-ISTIC

# Introduction à Bison

Amira BELHEDI
belhedi.amira@yahoo.fr

Référence

Livre: Romain Legendre, François schwarzentruber, compilation analyse lexicale et syntaxique

support de cours: Alexis Nasr

support de cours:  Anne BERRY

# Programmer l'analyseur syntaxique

- Programmer directement l'analyse syntaxique: de la chaîne de caractères jusqu'à l'arbre de syntaxe abstraite
  - serait très fastidieux
  - sans doute peu efficace.
- On utilise des générateurs d'analyseurs qui permettent de ne spécifier que les parties "utiles":
  - les entités lexicales;
  - les règles de grammaire;
  - les arbres de syntaxe abstraite;

# Programmer l'analyseur syntaxique

- Les analyseurs obéissent à des règles syntaxiques spécifiques.
- Ils sont eux-mêmes compilés pour engendrer les fonctions d'analyse.
- Ils utilisent des méthodes puissantes à base d'automates.

# Analyse syntaxique avec bison

- L'outil bison permet de générer des analyseurs syntaxiques.
- Un fichier d'entrée décrit la grammaire à analyser ainsi que les attributs et actions sémantiques associés.
- Un fichier .c est généré à partir de cette description, celui-ci contient la mise en œuvre de l'automate à pile pour une analyse ascendante.
- L'appel de l'analyseur se fait par le biais de la fonction **int yyparse ()** créée par bison.

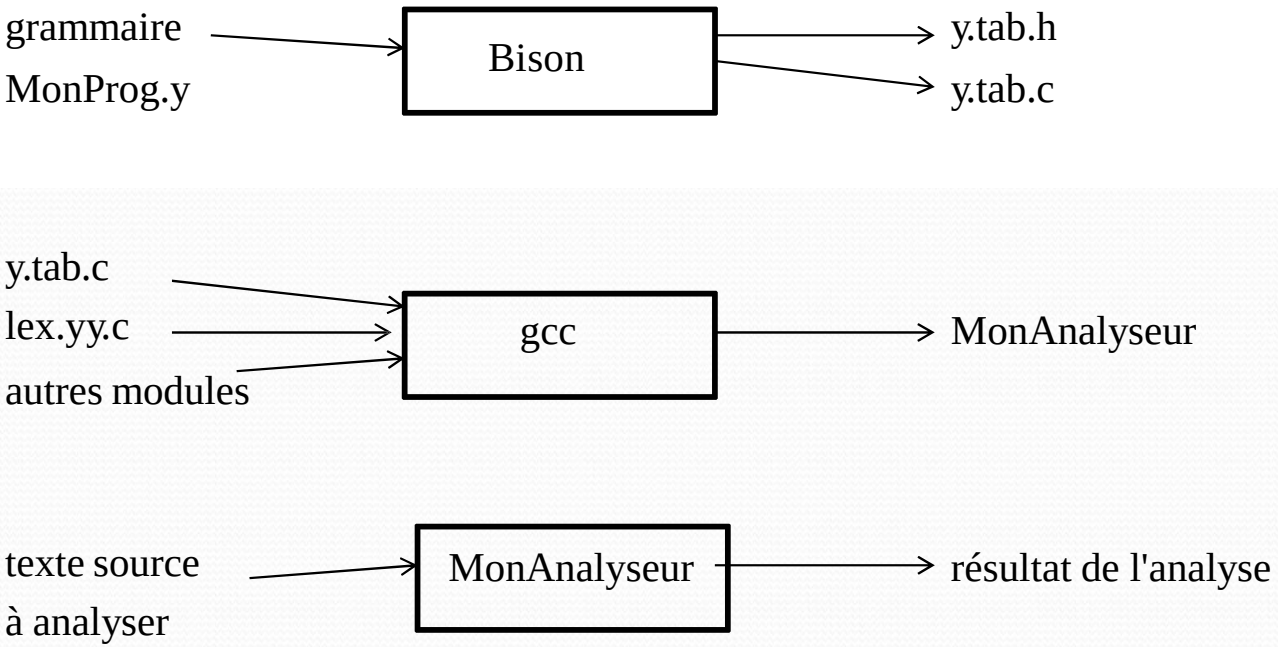
# Bison

- Bison: implémentation GNU de Yacc, développé en Juin 1985
- Yacc Conçu au début des années 70 par Johnson, le langage YACC veut dire, *Yet Another Compiler Compiler ( encore un autre compilateur de compilateurs).*

**1- <u>But</u>:**

- Construire à partir d'une grammaire une fonction C nommée ***yyparse()***, qui est un analyseur syntaxique qui reconnait les constructions du langage décrit par la grammaire.
- Un programme écrit en langage ***Yacc*** prend en entrée un fichier source constitué essentiellement des productions d'une grammaire ***G*** et produit un ***programme C*** qui, une fois compilé, est un analyseur syntaxique pour le langage ***L(G)***.

# Utilisation de Bison



# Structure du fichier Bison

- Le générateur d'analyseur syntaxique Bison a une structure analogue à celle du LEX (FLEX).
- Le code source Bison, doit avoir un nom terminé par "**.y**". Il est composé de 3 sections délimitées par deux lignes "**%%**".

```
déclarations
%%
règles de traduction
%%
routines annexes en langage C
```

# Structure du fichier Bison

1. **La section des déclarations**:

   Elle est constituée de 2 parties:

   1. Une partie entre **%{** et **%}** des déclarations à la C, des variables, unions, structures, .....

   2. Une partie constituée de la déclaration de
      - l'axiome, avec **%start**
      - des terminaux, avec **%token**
      - des opérateurs, en précisant leur associativité, avec **%left**, **%right**, **%nonassoc**

Remarques:

- *Le symbole **%** doit être à la 1ère colonne*
- *Tout symbole non déclaré dans cette partie par **token** est considéré comme un **non terminal***

# Structure du fichier Bison

**2. La section des règles de traduction:**

Après **%%** nous avons dans cette partie une suite de règles de traduction.

Chaque règle est constituée par une production de la grammaire associé aux phrases du langage à analyser et éventuellement une action sous forme d'une règle sémantique.

La règle sémantique peut prendre la forme d'un schéma de traduction ou d'une définition dirigée par la syntaxe.

# Structure du fichier Bison

**3 La section des fonctions annexes en C:**

Après **%%** nous avons dans cette partie une suite de fonctions écrite en langage C et utilisables par la 2ème partie des règles de traduction.

Remarques : la 1ère et la 3ème parties sont facultatives.

# Exercice 1

- Soit la grammaire des expressions arithmétiques (addition et multiplication) entières :

E -> E + E

| E * E

| cste

- Ecrire le fichier Bison qui permet de décrire cette grammaire (sans utiliser Flex)

# Corrigé de l'exercice 1

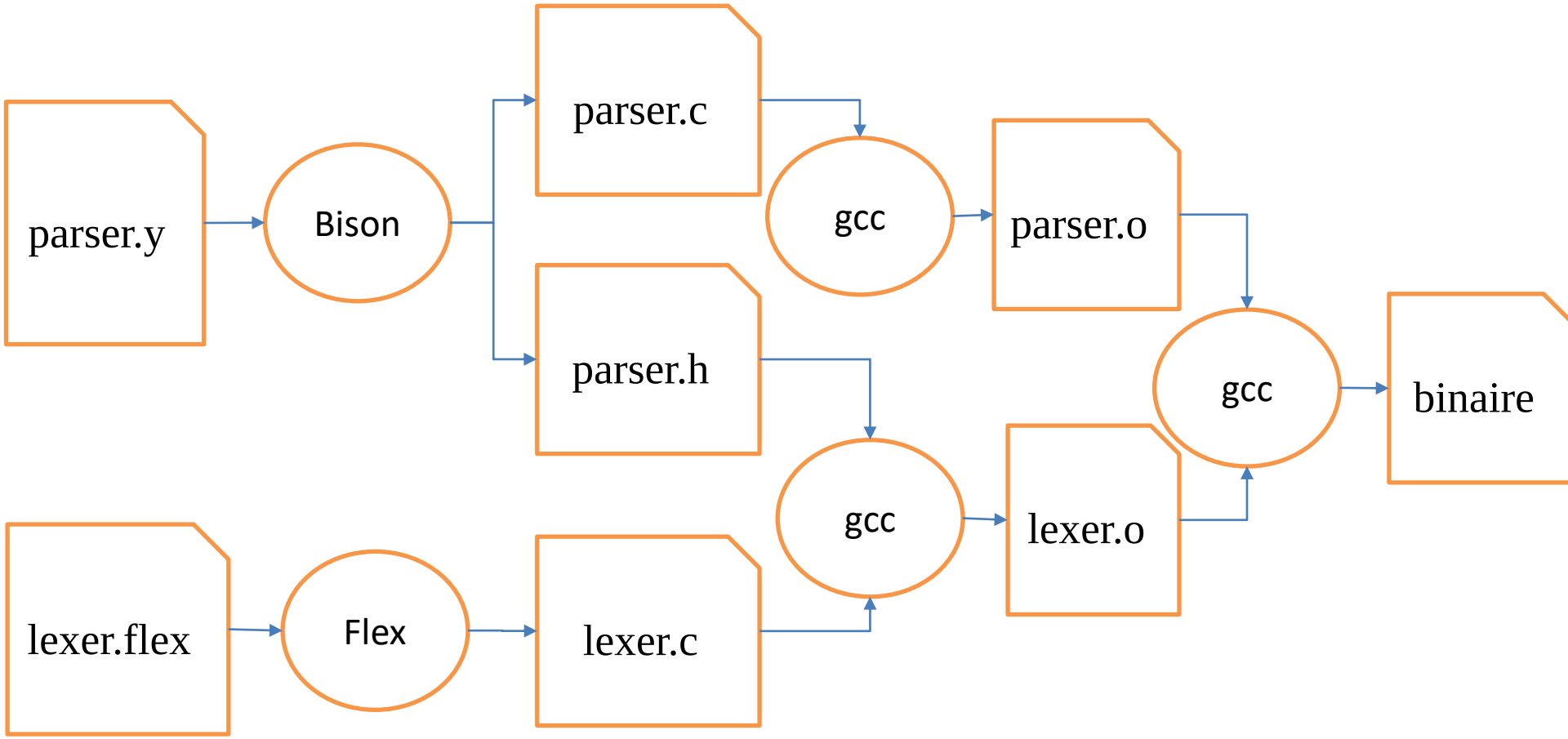
E -> E + E | E * E | cste

```
%start E
%token +
%token *
%token cste
%%
E : E + E
| E * E
| cste
;
%%
int main ( int argc, char** argv ) {yyparse ();}
```

# Faire communiquer Flex et Bison

- L'analyseur syntaxique fait appel à l'analyseur lexical pour connaître le prochain symbole dans la chaîne à analyser.
- Pour Bison l'appel à l'analyseur lexical se fait par le biais de la fonction **int yylex ().**
- Dans la majeure partie des cas, cet analyseur lexical sera généré à l'aide de Flex.

# Chaine de compilation Flex/bison



# Faire communiquer Flex et Bison

- La génération de l'exécutable se fera à l'aide des commandes suivantes :

```
bison -d parser.y -o parser.c
gcc -c parser.c -o parser.o

flex lexer.flex -o lexer.c
gcc -c lexer.c -o lexer.o

gcc parser.o lexer.o -lfl -o main
```

# Faire communiquer Flex et Bison

- L'option -d de bison permet de dire à ce dernier de générer un fichier .h (exemple parser.h) contenant la définition des constantes associées aux différents symboles terminaux de la grammaire.

- Ce fichier est à inclure (#include<parser.h>) dans le fichier lexer.flex pour que l'analyseur lexical puisse renvoyer la valeur de ces constantes en résultat.

- Dans les actions de l'analyseur lexical, on aura un return «symbole» où «symbole» est l'une des valeurs définies dans les %token.

# Exercice 2

- Ecrire un couple de fichiers flex/bison qui permettent de reconnaître la grammaire $a^n b^n$

$a^n b^n$ ➔ $E \rightarrow aEb \mid ab$

# Corrigé de l'exercice 2

- Exemple: $a^n b^n$ en flex+bison ➔ Le fichier Flex retourne des tokens :

```
%{
#include "parser.h"
int yyerror (char*);
%}
%%
a return TOKEN_A;
b return TOKEN_B;
.      yyerror ( "symbole non reconnu" );
%%
int yyerror ( char* m ) {  printf ( "%s\n", m );   return 1;  }
```

# Corrigé de l'exercice 2

- Les tokens retournés par flex sont utilisés dans le fichier Bison

```
%token TOKEN_A
%token TOKEN_B
%start E
%%
E : TOKEN_A E TOKEN_B  |   TOKEN_A TOKEN_B;
%%
int main ( int argc, char** argv ) { yyparse ();  }
```

# Corrigé de l'exercice

- Comment utiliser **%left**, **%right** et **%nonassoc** ?
- Supposons la règle suivante: **x op y op z**
- **%left** spécifie une associativité à gauche (x op y sera évalué en premier)
- **%right** spécifie une associativité à droite (y op z sera évalué en premier)
- **%nonassoc** spécifie qu'il n'y a pas de règle d'associativité ➔'x op y op z' sera considéré comme syntaxiquement fausse

# Exercice 3

Nous allons écrire un code complet en Lex et Bison, d'une calculatrice scientifique, qui va lire une expression arithmétique, l'évaluer et afficher le résultat.

Soit la grammaire:

| | | |
|---|---|---|
| R | ⟶ | E fin |
| E | ⟶ | E + T \| E - T \|T |
| T | ⟶ | T * F \| T / F \|F |
| F | ⟶ | (E) \| nombre |

# Corrigé de l'exercice 3
# Fichier Bison

Un programme en Bison: *calc.y*

1ère partie

```
%{
#include<stdio.h>
%}
%token nombre fin // nombre et fin dont des terminaux
%left plus moins  // '+' et le '-' associatif à gauche
%left fois div
%start R // R est l'axiome
%%
```

# Corrigé de l'exercice 3
# Fichier Bison

Un programme Bison (suite): 2 ème partie *sans l'évaluation des expressions*

```
R   : E fin
    ;
E : E plus T
    | E moins T
    | T
    ;
T : T fois F
    | T div F
    | F
    ;
F : '(' E ')'
    | nombre
    ;
%%
```

# Corrigé de l'exercice 3
# Fichier Bison

2 ème partie *avec l'évaluation des expressions*

```
R   : E fin                 { printf("le résultat est: %d", $1);}
    ;
E : E plus T                { $$ = $1 + $3;}
  | E moins T               { $$ = $1 - $3;}
  | T                       { $$ = $1; // facultatif
                              }
    ;
T : T fois F                { $$ = $1 * $3;}
  | T div F                 { if ($3==0) printf ("division par zéro  interdite)
                            else $$ = $1 / $3;}
  | F                       { $$ = $1;}
    ;
F : '(' E ')'               { $$ = $2;}
  | nombre                  { $$ = $1;}
    ;
%%
```

# Corrigé de l'exercice 3
# Fichier Bison

- 3 ème partie:

```
%%
  void yyerror(char *message) {
  printf("<< %s", message);
}
int main(void) {
        printf("début de l'analyse\n");  yyparse();
        printf("fin de l'analyse\n");
}
```

- L'analyseur syntaxique se présente comme une fonction `int yyparse(void)` qui rend 0 si la chaîne est acceptée, non nulle dans le cas

# Corrigé de l'exercice 3
# Fichier Flex

Un programme en Flex: *analex.l ou analex.flex*

```
%{
#include<stdlib.h>
%include<calc.tab.h>
%}
nombre [0-9]+
%%

[ \t]+        {/* ne rien faire */}
{nombre}      {yylval = atoi(yytext); return(nombre);
"\n"          {return (fin);}
"+"           {return (plus);}
"-"           {return (moins);}
"/"           {return (div);}
"*"           {return (fois);}
%%
```

# Corrigé de l'exercice 3
# Etapes de compilation

1 étape: > ***bison -d calc.y***

en sortie on a, ***calc.tab.c*** et ***calc.tab.h***

2ème étape: > ***flex analex.l***

en sortie on a: ***lex.yy.c***

3ème étape: > ***gcc -c lex.yy.c -o cal.l.o***

***gcc -c calc.tab.c -o calc.y.o***

***gcc -o calc calc.l.o calc.y.o -lfl -lm***

- ***lfl***: ***Library Fast Lex*** *(la librairie du Flex)*
- ***lm***: la librairie de *Bison*

# Corrigé de l'exercice 3
# Remarque

- L'analyseur lexical produit par flex transmet les attributs des unités lexicales à l'analyseur syntaxique produit par *yacc* via une variable *yylval* qui par défaut est de type `int`.

- Si nous voulons manipuler des réels, nous devons modifier ce type dans le fichier *calc.tab.h*:

```
#define YYTYPE int
...
extern YYSTYPE yylval;
```

A remplacer `int` par `double`

# Gestion des conflits par Bison

- Bison adopte une *analyse ascendante par décalage-réduction*, il peut rencontrer lors de l'analyse par une grammaire 2 types de conflits:

1 Conflit entre un décalage et une réduction *(shift/reduce conflict)*, ce type correspond à choisir entre 2 actions possibles.

➔ Bison favorise par défaut le décalage.

2 Conflit entre une réduction et une autre réduction *(reduce/ reduce conflict)*, ce conflit se produit quand 2 membres droits peuvent se réduire par le symbole gauche d'une production au sommet de la pile.

➔ Bison choisit la 1ère production écrite dans le fichier Yacc.