Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Grammaires algébriques

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Sommaire

Syntaxe

E --> **nombre**
E --> **(** *E* **)**
E --> *E* + *E*
E --> *E* **-** *E*
E --> *E* ***** *E*
E --> *E* **/** *E*

Syntaxe
Contraintes sur l'écriture du code dans les langages de programmation
Règles de grammaire
Servent à spécifier la syntaxe
symbole --> expression
Dans l'expression on peut avoir deux sortes de symboles :
- ceux du langage final : les **terminaux** (les lexèmes de l'analyse lexicale)
- des symboles intermédiaires, les **non-terminaux** ou **variables**

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Exemple

E --> **nombre**

E --> **(** *E* **)**

E --> *E* + *E*

E --> *E* **-** *E*

E --> *E* ***** *E*

E --> *E* **/** *E*

Grammaire pour les expressions arithmétiques simples

Le non-terminal *E* désigne les expressions

Le symbole terminal **nombre** représente les chaînes de caractères qui sont des nombres

Les autres terminaux sont **( ) + - * /**

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

E --> **nombre**

E --> **(** *E* **)**

E --> *E* + *E*

E --> *E* **-** *E*

E --> *E* ***** *E*

E --> *E* **/** *E*

Définition formelle

Une grammaire algébrique (*context-free grammar*) est définie par

- un alphabet *A* de terminaux
- un ensemble *V* de non-terminaux
- un ensemble fini de règles

$(x, w) \in V \times (A \mid V)^*$

notées $x \dashrightarrow w$

- un non-terminal *S* appelé axiome

Un mot sur *A* est une suite d'éléments de *A*

A^* est l'ensemble des mots sur *A*

Un langage formel est une partie de A^*

Dérivations

- *E* --> **nombre**
- *E* --> **(** *E* **)**
- *E* --> *E* + *E*
- *E* --> *E* **-** *E*
- *E* --> *E* ***** *E*
- *E* --> *E* **/** *E*

E ∗ *(E)* --> *E* ∗ *(E+E)*

E $\xrightarrow{*}$ **nombre ∗ (nombre + nombre)**

Si *x* --> *w* est une règle de la grammaire, en **remplaçant *x* par *w*** dans un mot on obtient une étape de dérivation

Dérivations

Si u_0 --> u_1 ... --> u_n on écrit :

$$u_0 \xrightarrow{*} u_n$$

Cette définition autorise *n*=0 et *n*=1

Langage engendré

E --> **nombre**
E --> E + E

$E \xrightarrow{*}$ **nombre** (**'+'** **nombre**) *

E --> **nombre**
E --> **(** E **)**
E --> E + E
E --> E **-** E
E --> E * E
E --> E **/** E

Si L est un langage, $\boldsymbol{u_0} \xrightarrow{*} \boldsymbol{L}$ veut dire :

$$\forall u \in A^* \ (u_0 \xrightarrow{*} u \Leftrightarrow u \in L)$$

Langage engendré

On s'intéresse aux dérivations qui vont de S à des mots de A^*

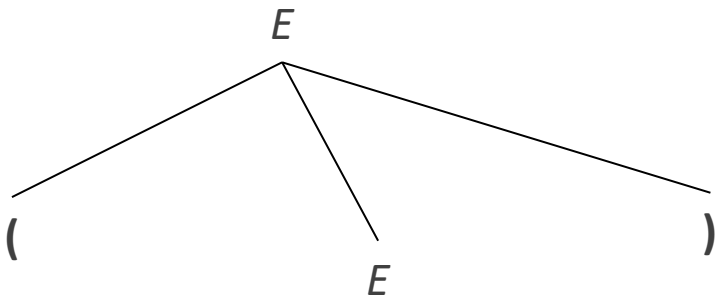
$$L = \{u \in A^* \mid S \xrightarrow{*} u\}$$

Exemple

Pour cette grammaire, le langage engendré est l'ensemble des expressions arithmétiques formées à l'aide des opérateurs **+**, **-**, *, /, de **nombre** et des parenthèses

Arbres

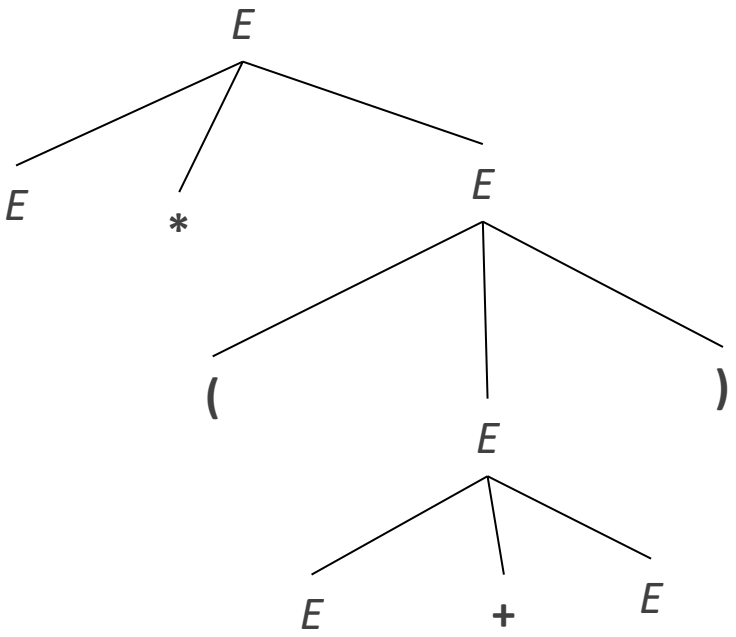
E --> (*E*)



On représente les règles sous forme d'arbres

Arbres

$E \rightarrow E * E \rightarrow E * (E) \rightarrow E * (E + E)$

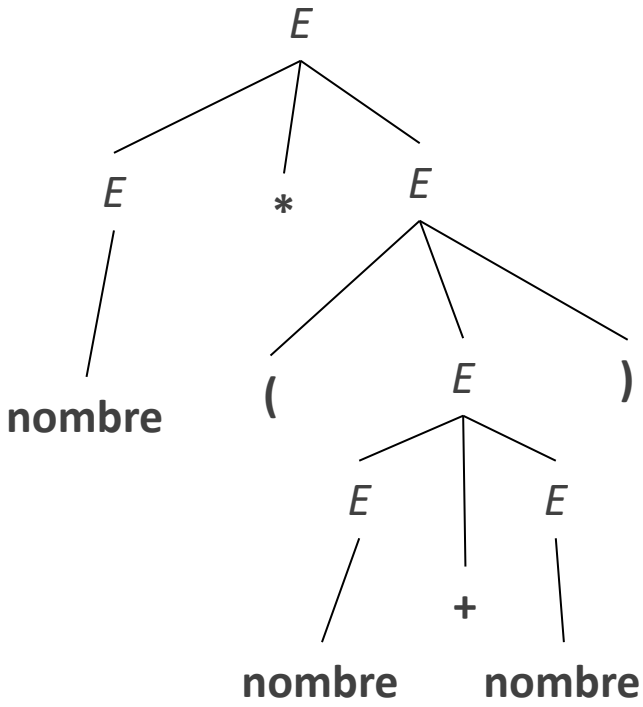


En partant d'une variable et enchaînant plusieurs étapes de dérivation, on a un arbre de hauteur supérieure à 1

Université Gustave Eiffel — INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Arbres de dérivation

E --> **nombre**
E --> **(** *E* **)**
E --> *E* + *E*
E --> *E* **-** *E*
E --> *E* ***** *E*
E --> *E* **/** *E*



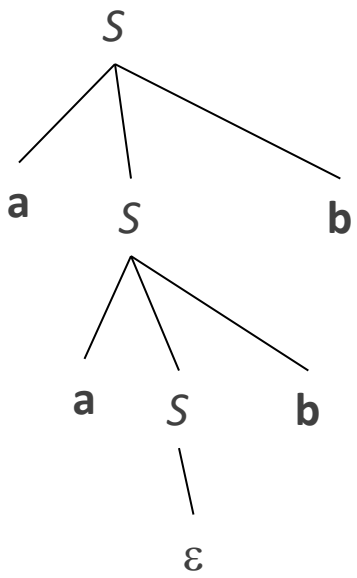
On s'intéresse aux arbres dont la racine est l'axiome et dont toutes les feuilles sont des terminaux ou ε

Le langage engendré par la grammaire est l'ensemble des frontières des arbres de dérivation

Arbres de dérivation

- *S* --> **a** *S* **b**
- *S* --> ε

Quels sont tous les arbres de dérivation pour cette grammaire ?

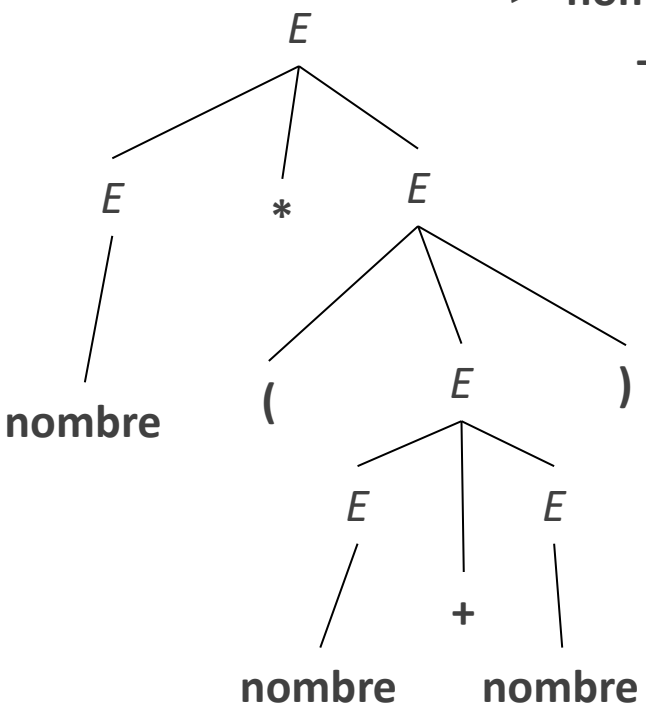


Les arbres de dérivation de hauteur $n > 0$ obtenus en utilisant $n - 1$ fois la première règle et 1 fois la deuxième

La frontière d'un tel arbre est $\mathbf{a}^{n-1}\mathbf{b}^{n-1}$

Arbres et dérivations

E --> *E* * *E* --> nombre * *E* --> nombre * (*E*)
--> nombre * (*E* + *E*) --> nombre * (nombre + *E*)
--> nombre * (nombre + nombre)



À toute dérivation (si on a spécifié à chaque étape quel non-terminal est dérivé) correspond un et un seul arbre de dérivation

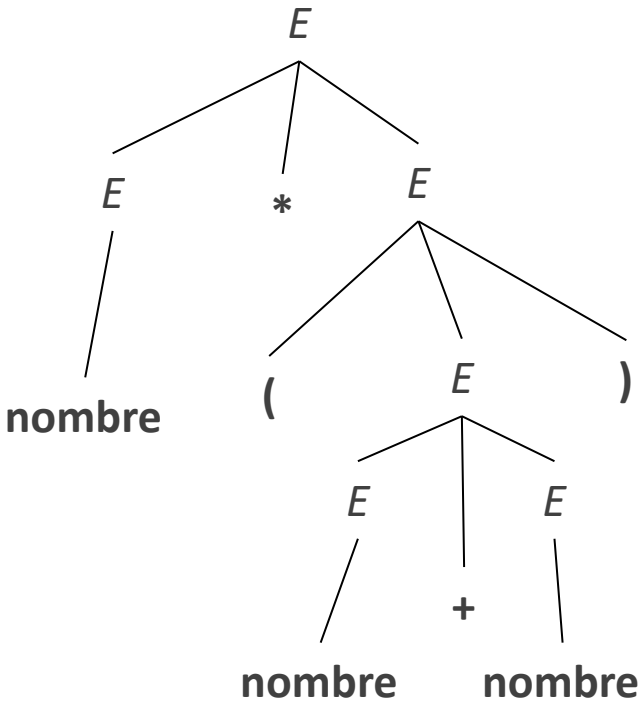
L'inverse n'est pas vrai : on peut dériver dans un autre ordre

Dérivations gauches et droites

E --> *E* * *E* --> nombre * *E* --> nombre * (*E*)
--> nombre * (*E* + *E*) --> nombre * (nombre + *E*)
--> nombre * (nombre + nombre)

Dérivation gauche : on remplace toujours le non-terminal le plus à gauche

E --> *E* * *E* --> *E* * (*E*) --> *E* * (*E* + *E*)
--> *E* * (*E* + nombre) --> *E* * (nombre + nombre)
--> nombre * (nombre + nombre)



Arbres et dérivations gauches

À tout arbre de dérivation correspond une et une seule dérivation gauche

Algorithme de construction : on fait un parcours préfixe de l'arbre ; quand on visite un nœud contenant un non-terminal, on remplace le non-terminal correspondant dans la dérivation

E --> *E* * *E* --> nombre * *E* --> nombre * (*E*)
--> nombre * (*E* + *E*) --> nombre * (nombre + *E*)
--> nombre * (nombre + nombre)

Langues naturelles

<phrase> --> <sujet> <verbe>
<phrase> --> <sujet> <verbe> <complement>
<phrase> --> <sujet> <verbe> <complement> <complement>
<sujet> --> <det> <nom>
<sujet> --> <det> <adj> <nom>
...

On peut utiliser les grammaires pour décrire la syntaxe des langues naturelles

D'où leur nom

Conventions de notation

E --> *E* **+** *E*
E --> **N**

E --> *E* **+** *E* | **N**

On peut regrouper plusieurs règles qui ont le même membre gauche

Cela revient à dire que le membre droit d'une règle est une partie finie de (*A*|*V*)*

inst --> **si** *cond* **alors** *inst*
| **si** *cond* **alors** *inst* **sinon** *inst*
| ...
cond --> ...

On donne la liste des règles en commençant par l'axiome

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Exemples

E --> *E* + *E* | **N**

Grammaire ambigüe

P --> **(** *P* **)** *P* | ε

Grammaire des expressions parenthésées (non ambigüe)

E --> *E E* **+** | **N**

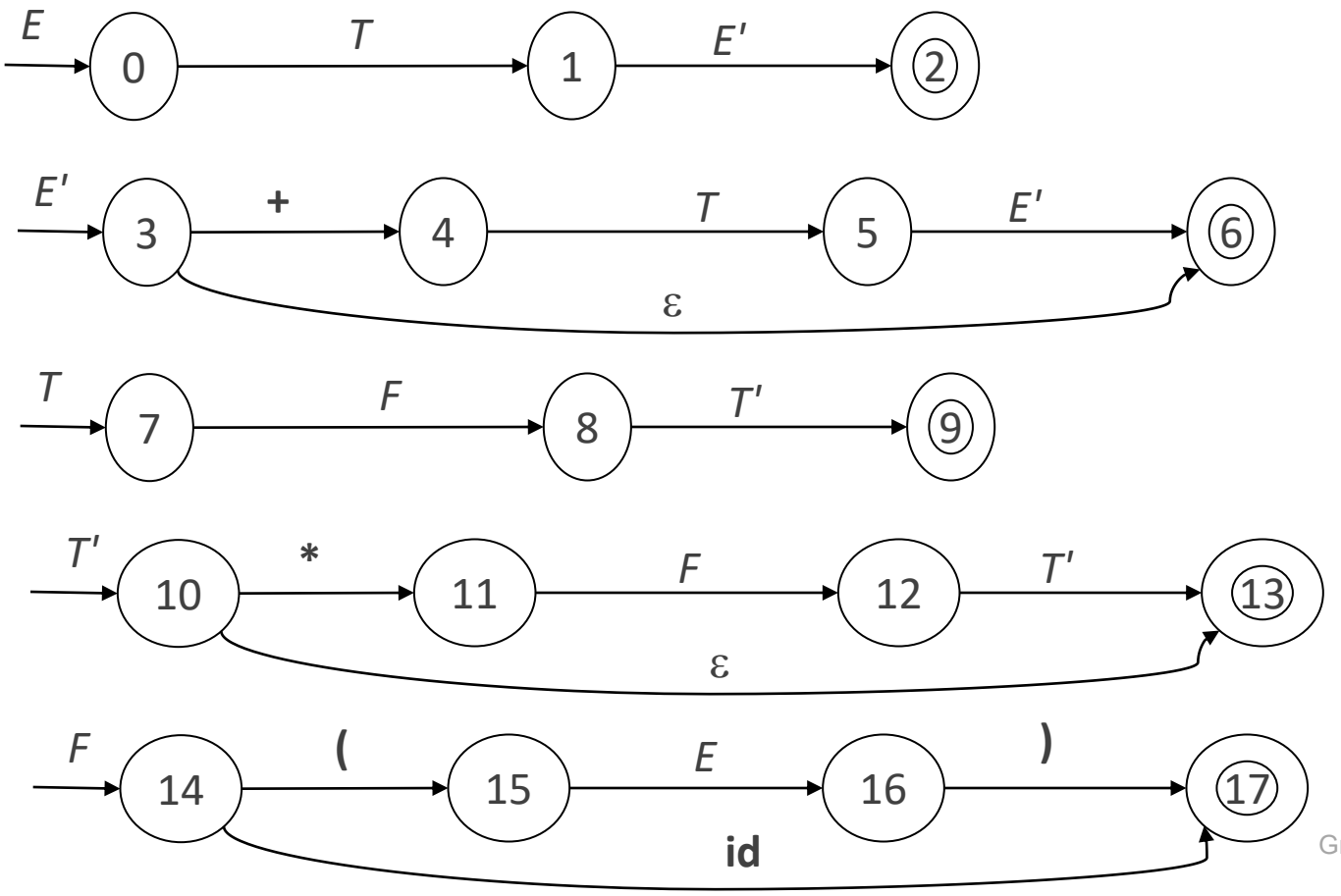
Expressions additives en notation postfixe (non ambigüe)

P --> **(** *P* **)** | ε

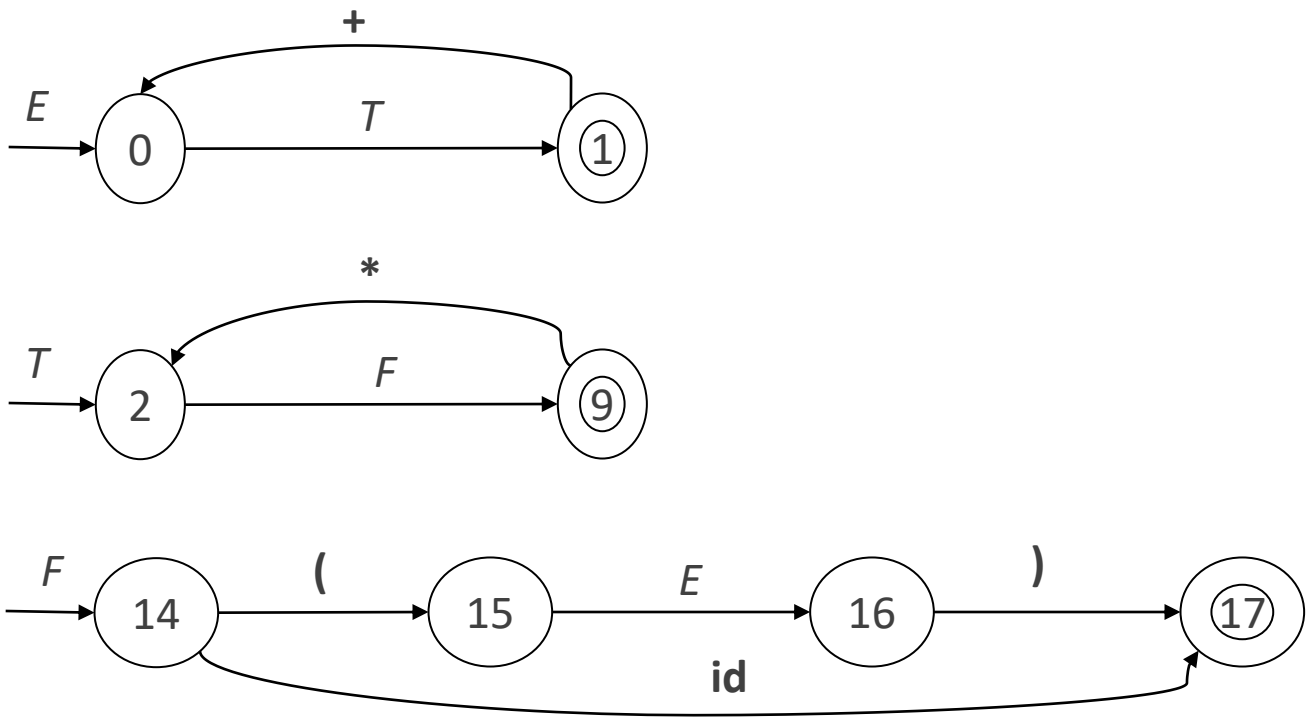
Compte des empilements et dépilements (non ambigüe)

Diagrammes de transitions

Les grammaires peuvent être mises sous la forme de diagrammes de transitions



Diagrammes de transitions



Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Sommaire

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Expressivité des grammaires algébriques

Langages algébriques (*context-free languages*)

Un langage est algébrique ssi il est engendré par une grammaire algébrique

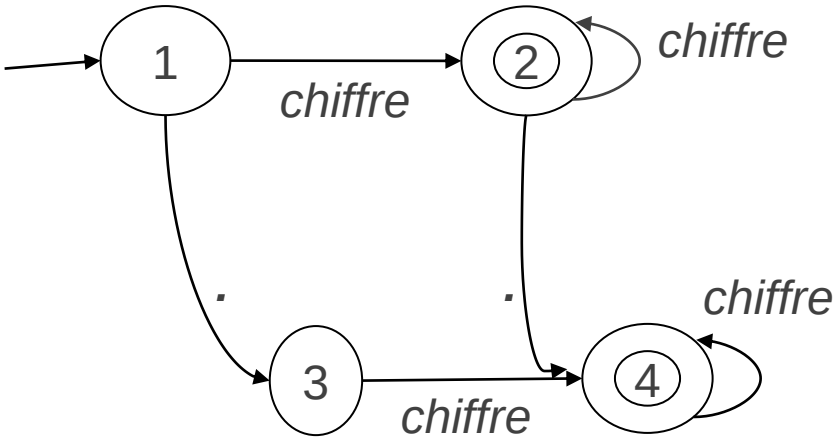
Langages réguliers

Un langage est régulier ssi il est reconnu par un automate fini

Tous les langages réguliers sont algébriques

L'inverse n'est pas vrai

Grammaires régulières à droite



V1 --> **chiffre** *V2* | **.** *V3*
V2 --> **chiffre** *V2* | **.** *V4* | ε
V3 --> **chiffre** *V4*
V4 --> **chiffre** *V4* | ε

Une grammaire algébrique est régulière à droite ssi chaque règle est de l'une des formes suivantes:

- $x \rightarrow ay$ avec $a \in A$ et $y \in V$
- $x \rightarrow a$ avec $a \in A$
- $x \rightarrow \varepsilon$

À tout automate fini correspond une grammaire régulière à droite qui engendre le langage reconnu par l'automate

Tous les langages réguliers sont algébriques

Expressivité des grammaires algébriques

Les grammaires algébriques sont plus puissantes que les automates finis

Ensemble des mots de la forme $\mathbf{a}^n\mathbf{b}^n$, avec $n \geq 0$

S --> **a** S **b** | ε

Ce langage est algébrique, mais pas régulier

Expressivité des grammaires algébriques

$\mathbf{a}^t\mathbf{b}^s$ appartiendrait au langage avec $t<s$

$\mathbf{a}^s\mathbf{b}^t$ appartiendrait au langage avec $s<t$

$\mathbf{a}^s\mathbf{b}^t\mathbf{a}^u\mathbf{b}^s$ appartiendrait au langage avec $t>0$ et $u>0$

L'ensemble des mots de la forme $\mathbf{a}^n\mathbf{b}^n$, avec $n\geq 0$ n'est pas régulier

Supposons qu'il soit reconnu par un automate fini déterministe à s états

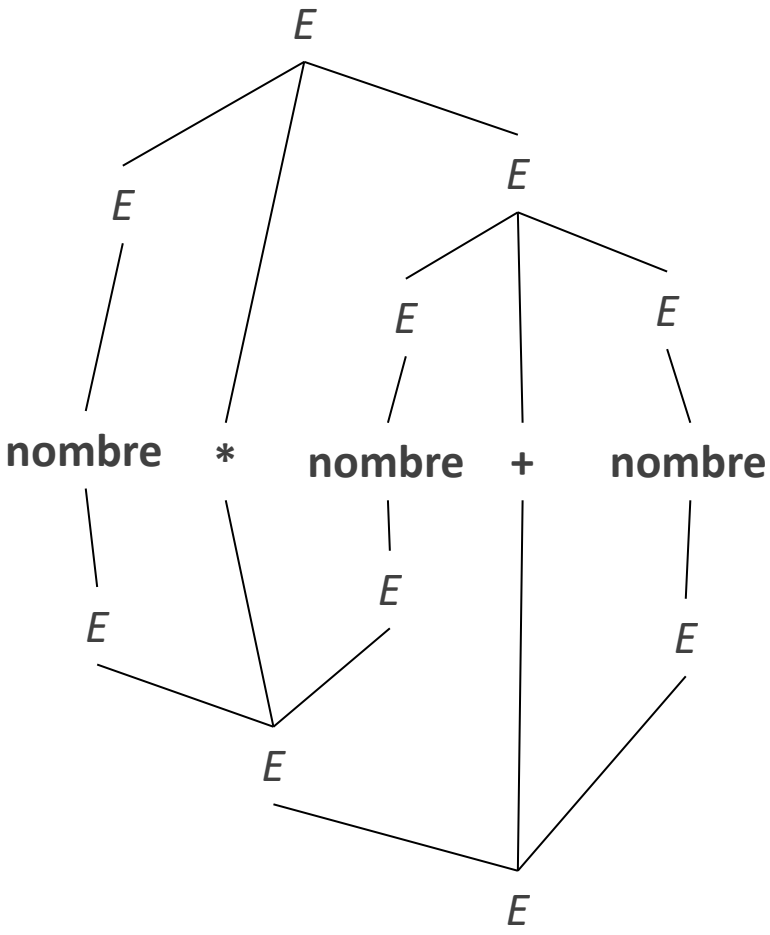
Soient q_0, q_1,... q_{2s} les états sur le chemin reconnaissant $\mathbf{a}^s\mathbf{b}^s$

Ce sont $2s+1$ noms d'éléments d'un ensemble à s éléments, donc $\exists\ i, j\ \ 0\leq i<j\leq 2s$ et $q_i=q_j$

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Sommaire



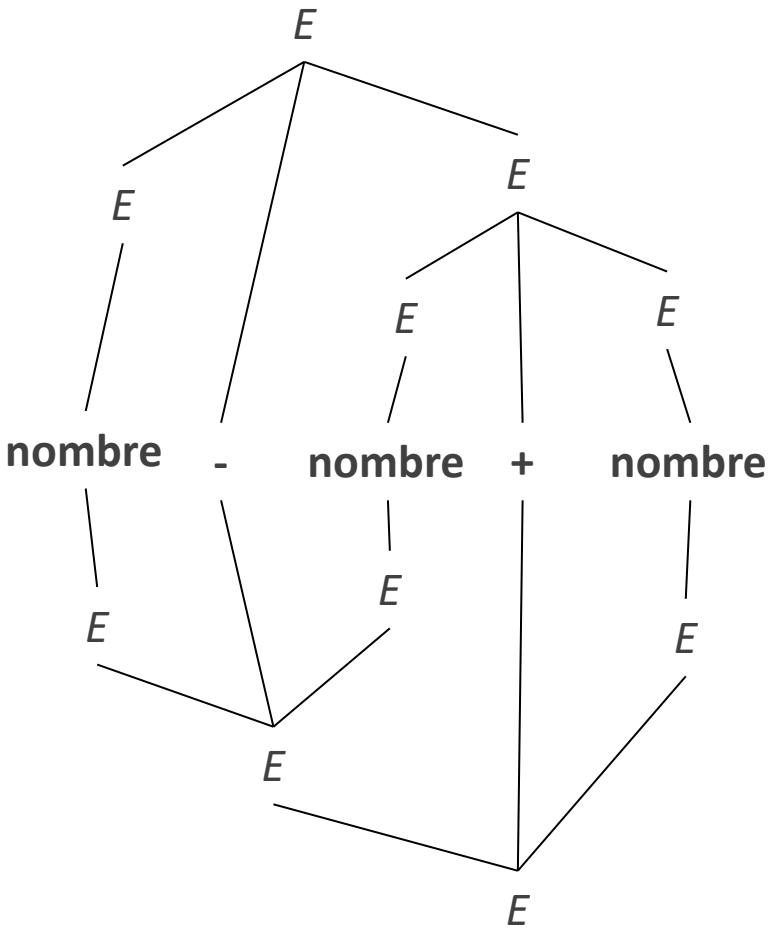
Ambiguïté

E --> *E* + *E* | *E* - *E* | *E* * *E* | *E* / *E* | **nombre** | (*E*)

Grammaire ambigüe : un même mot possède plusieurs arbres de dérivation

Ambiguïté sur l'ordre d'application des opérations

Quelle est la bonne interprétation en C ?



Ambiguïté

E --> *E* + *E* | *E* - *E* | *E* * *E* | *E* / *E* | **nombre** | (*E*)

Quelle est la bonne interprétation en C ?

Ambiguïté

inst --> **si** *cond* **alors** *inst*

inst --> **si** *cond* **alors** *inst* **sinon** *inst*

inst --> ...

cond --> ...

Grammaire ambigüe

Exercice : quels sont les arbres pour

si *cond* **alors si** *cond* **alors** *inst* **sinon** *inst*

Une grammaire non ambigüe pour les expressions

E --> *E* **+** *T* | *E* **-** *T* | *T*
T --> *T* ***** *F* | *T* **/** *F* | *F*
F --> **(** *E* **)** | **N**

Cette grammaire force les choix suivants :
- associativité à gauche (c'est-à-dire de gauche à droite) pour + et -
- associativité à gauche pour * et /
- priorité de * et / sur + et -.

Pour cela, elle utilise trois niveaux : *E, T, F,* au lieu d'un
- *F* (facteur) : n'est pas directement le résultat d'une opération
- *T* (terme) : fait de facteurs avec éventuellement * ou /
- *E* (expression) : fait de termes avec éventuellement **+** ou **-**

Grammaires équivalentes

E --> *E* **+** *E* | *E* **-** *E* | *E* ***** *E* | *E* **/** *E* | **(** *E* **)** | **N**

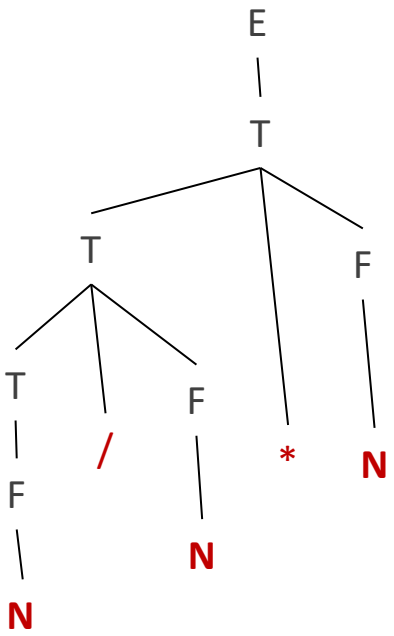
E --> *E* **+** *T* | *E* **-** *T* | *T*
T --> *T* ***** *F* | *T* **/** *F* | *F*
F --> **(** *E* **)** | **N**

Deux grammaires sont équivalentes si elles engendrent le même langage

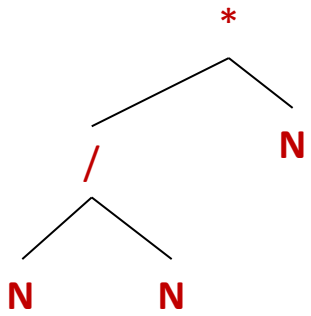
Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

$$
\begin{cases}
E \rightarrow E + T \mid E - T \mid T \\
T \rightarrow T * F \mid T / F \mid F \\
F \rightarrow (E) \mid \mathbf{N}
\end{cases}
$$



Arbre abstrait



Les arbres de dérivation peuvent être compliqués à cause du choix de la grammaire

Un arbre abstrait est une version simplifiée

On ne garde que les nœuds utiles

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Analyse sémantique

Contraintes de types

Faire le lien entre déclaration et utilisation

Calculer le type des expressions

Relations entre deux endroits du code

Si on codait ces contraintes dans la grammaire, elle deviendrait trop compliquée

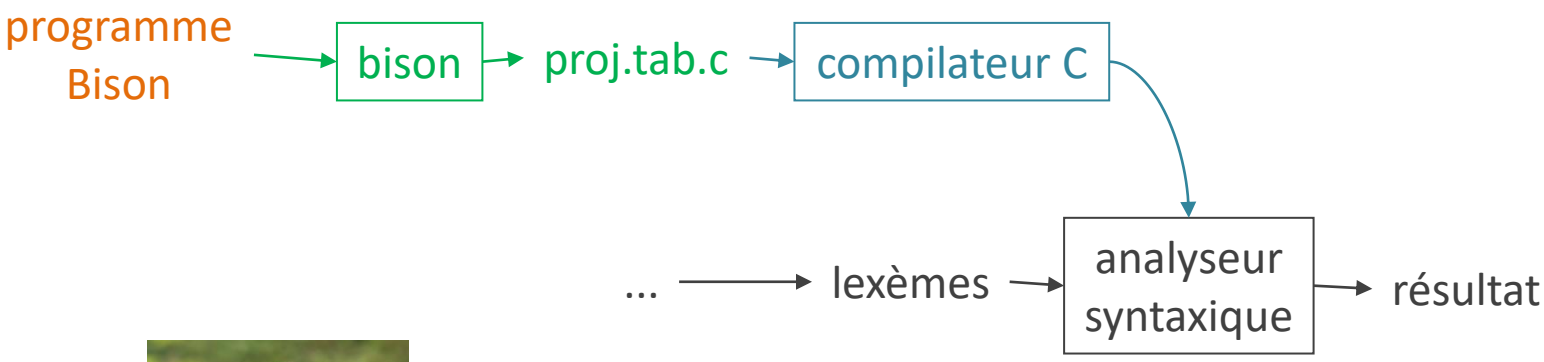
Si un compilateur fait des arbres abstraits, l'analyse sémantique peut se faire dessus

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Sommaire

Utilisation de Bison



Par GrottesdeHan — Travail personnel, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=28004357

4 étapes :

- créer sous éditeur un programme Bison (grammaire)
- traiter ce programme par la commande **`bison`**
- compiler le programme C obtenu
- exécuter le programme exécutable obtenu (analyseur LALR(1))

Documentation en ligne :

https://www.gnu.org/software/bison/manual/html_node/index.html

Programmes Bison

Un programme Bison est fait de trois parties :

déclarations

%%

règles de traduction

%%

fonctions auxiliaires en C

Les règles de traduction sont de la forme

X : $expr_1$ { $action_1$ }

| $expr_2$ { $action_2$ }

…

| $expr_n$ { $action_n$ }

;

où chaque X --> $expr_i$ est une règle. Les actions sont facultatives et en C

Exemple

```
%{
#include <ctype.h>
int yyerror(char *);
%}
%token CHIFFRE
%%
ligne   : expr '\n'
        ;
expr    : expr '+' terme
        | terme
        ;
terme   : terme '*' fact
        | fact
        ;
fact    : '(' expr ')'
        | CHIFFRE
        ;
```

```
%%
int yylex() {
   int c ;
   c = getchar() ;
   if (isdigit(c)) {
        return CHIFFRE ; }
   return c ; }
```

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Programmes Bison

By Kalabaha1969 - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=32545345

Les commentaires /* ... */ peuvent être insérés n'importe où (pas dans une balise, quand même)

Bison est une version de Yacc (Yet another compiler compiler, 1975)

On utilise toujours l'extension .y

La commande bison


```
bison proj.y
```

Messages d'erreur

```
> my_parser
a * (b + c(
syntax error
>
```

```
%%
int yyerror(char * s){
  fprintf(stderr, "%s\n", s);
  return 0;
}
```

```
gcc lex.yy.o -Wall -ly -lfl
```

```
%{
#include <ctype.h>
int yyerror(char *);
%}
%token CHIFFRE
%%
```

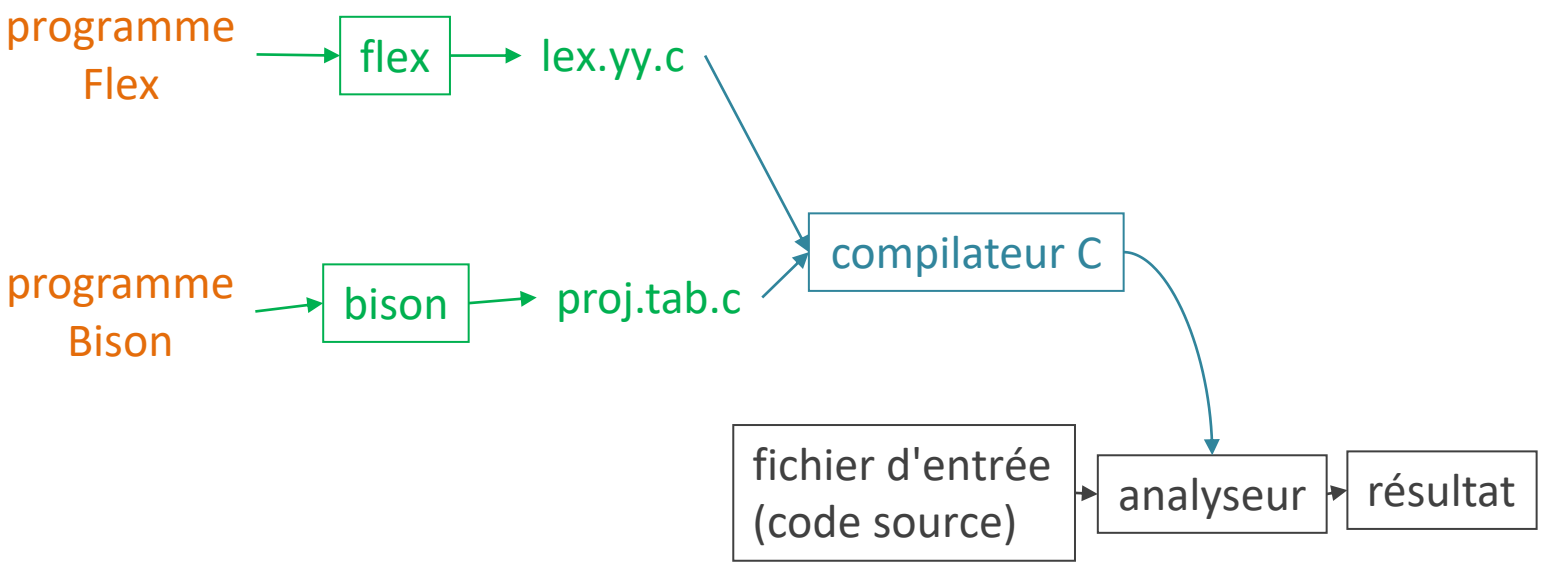
En cas d'erreur, l'analyseur syntaxique appelle yyerror() avec une chaine de caractères en paramètre

La **bibliothèque de Bison** (option **-ly**) fournit un yyerror() qui affiche simplement la chaine

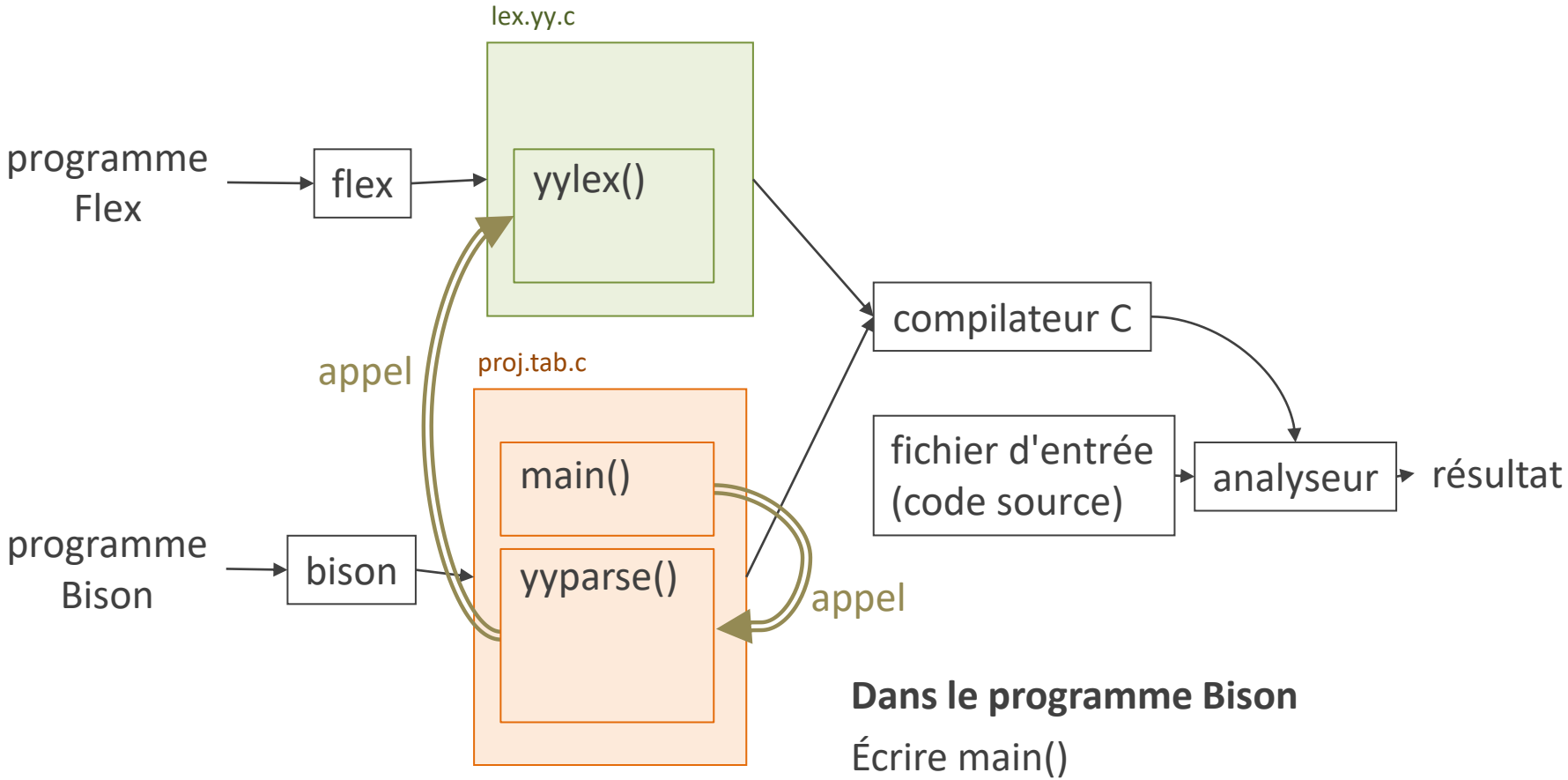
L'option -ly doit apparaitre vers la fin et non vers le début de la ligne de commande

Déclarer yyerror() dans le programme Bison, dans la partie déclarations

Utiliser Bison avec Flex



Interface Flex-Bison



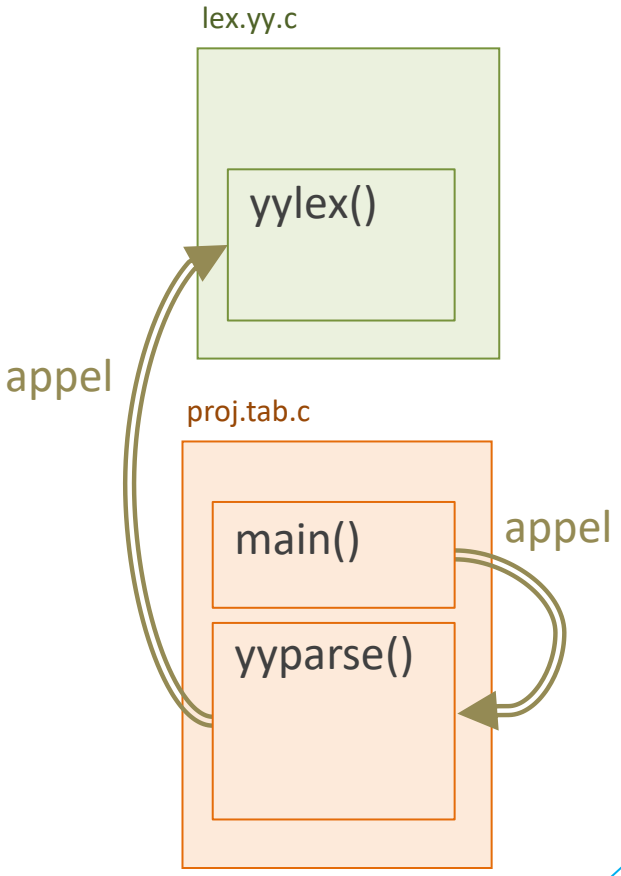
Dans le programme Bison

Écrire main()

Déclarer yylex()

Valeur de retour de yyparse(): 0 si succès, 1 si erreur

Interface Flex-Bison




```
[0-9]+   { yylval=atoi(yytext) ;
           return NUMBER; }
.          return yytext[0];
```

Les lexèmes sont représentés par la valeur de retour de yylex()

Type de retour de yylex() : int

- soit un caractère du code source : '(', '+'...
- soit une constante : NUMBER, ELSE...
- signal de fin de fichier : 0

Pas une chaine de caractères

Interface Flex-Bison

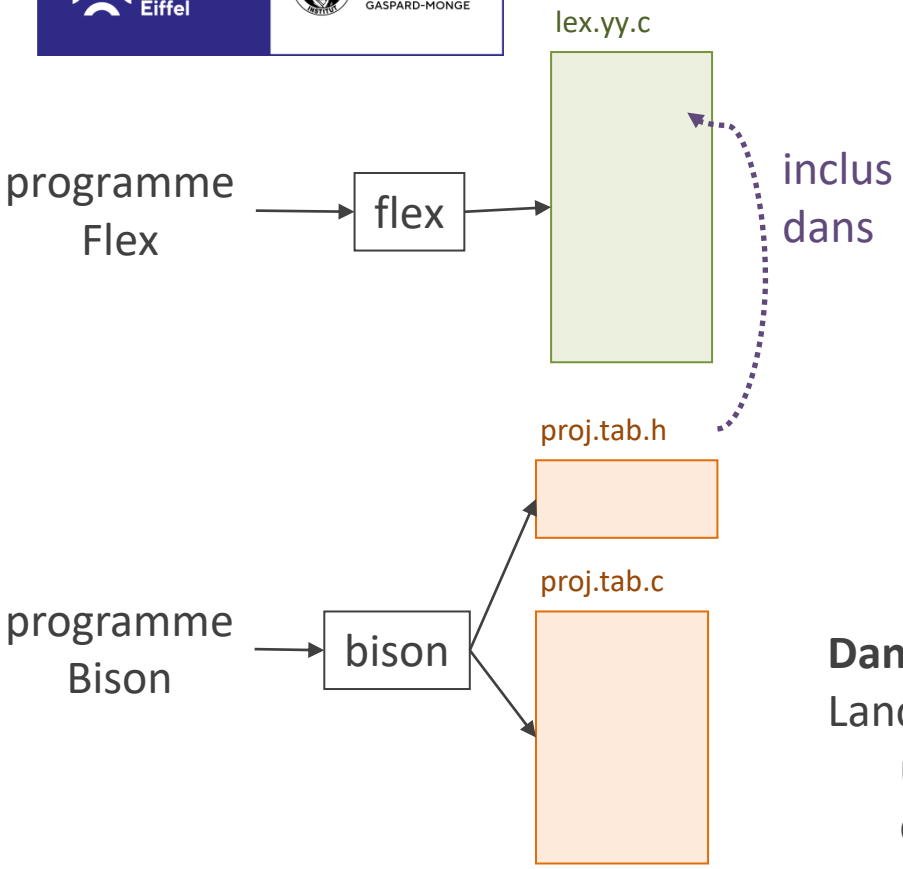
```
%{
#include <ctype.h>
int yylex();
int yyerror(char *);
%}
%token NUMBER
%%
ligne     : expr '\n'
          ;
expr      : expr '+' terme
          | terme
          ;
terme     : terme '*' fact
          | fact
          ;
fact      : '(' expr ')'
          | NUMBER
          ;
```

Dans le programme Bison

Déclarer les constantes qui représentent des lexèmes : NUMBER, ELSE...

Bison les déclare comme constantes entières

Interface Flex-Bison



Dans la ligne de commande pour Bison

Lancer Bison avec l'option -d pour qu'il produise un fichier d'en-tête .h avec la déclaration des constantes :

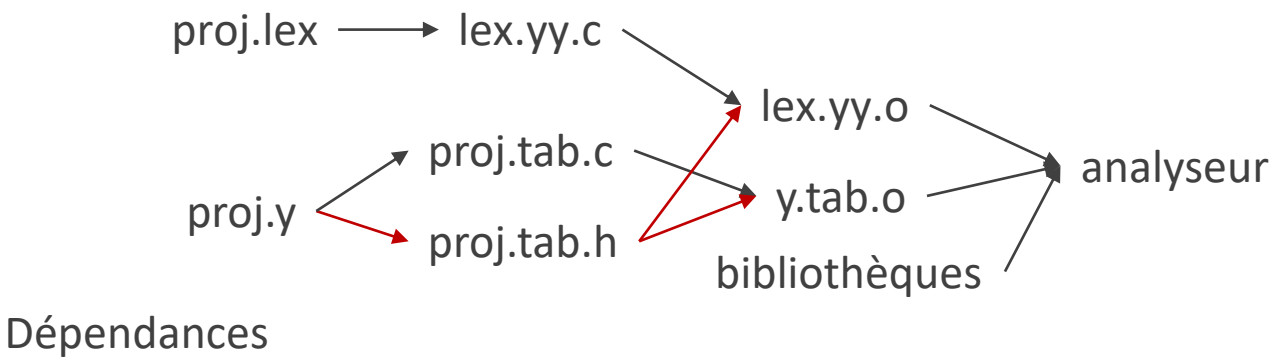
```
bison -d proj.y
```

```
%{
#include "proj.tab.h"
%}
%option nounput
%option noinput
%%
[0-9]+   return NUMBER;
.        return yytext[0];
```

Dans le programme Flex

Inclure le fichier .h

Makefile pour Bison et Flex



Dépendances

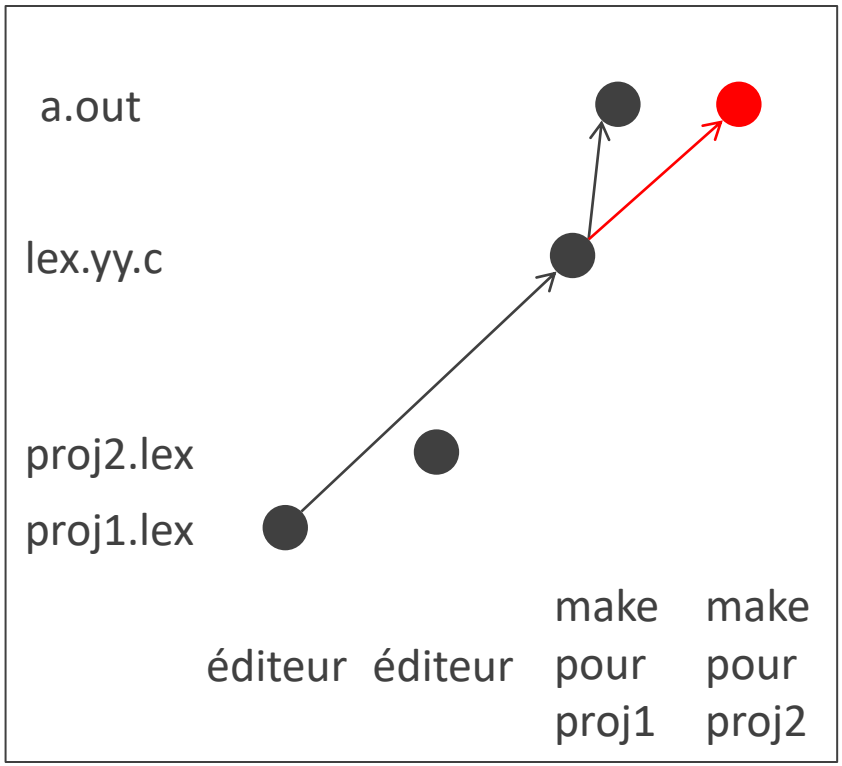
Différences par rapport à un makefile pour Flex

Nouvelles dépendances

Exemple : le fichier .h doit être engendré avant de compiler lex.yy.c

Bibliothèque de Bison (-ly)

Deux projets dans le même répertoire




```
all: $(EXEC) clean
```

```
$(ANL).c: $(ANL).lex
  flex -o $(ANL).c $(ANL).lex
```

Trois solutions

Un seul projet et un seul makefile par répertoire

Inclure un nettoyage automatique

Donner à lex.yy.c un nom différent pour chaque projet

Tests fonctionnels

```
int
main
(
void
)
{
return
0
;
}
```

```
int main(void){
  turner 0;
}
```

Jeux d'essai

Entrées correctes variées

Entrées incorrectes variées

Script de déploiement

```
for file in test/* ; do
cat $file
done
```

Produit un rapport donnant les résultats des tests

Boucler sur les fichiers d'un répertoire

Code de retour de la commande précédente : `$?`

```
echo "fin" >> report.txt
```

Ajouter une ligne à un fichier

Utilisation de grammaires ambigües

```
expr    : expr '+' expr
        | expr '*' expr
        | '(' expr ')'
        | CHIFFRE
        ;
```

- Bison ne peut pas traiter n'importe quelle grammaire
- Pour certaines grammaires, il fait des avertissements mais produit quand même un analyseur syntaxique qui reconnait le langage
- Pour d'autres, il fait des avertissements et produit un analyseur qui reconnait seulement une partie du langage
- Pour d'autres, il ne produit pas d'analyseur syntaxique
- Chaque grammaire ambigüe fait partie d'un de ces trois cas