Université Gustave Eiffel

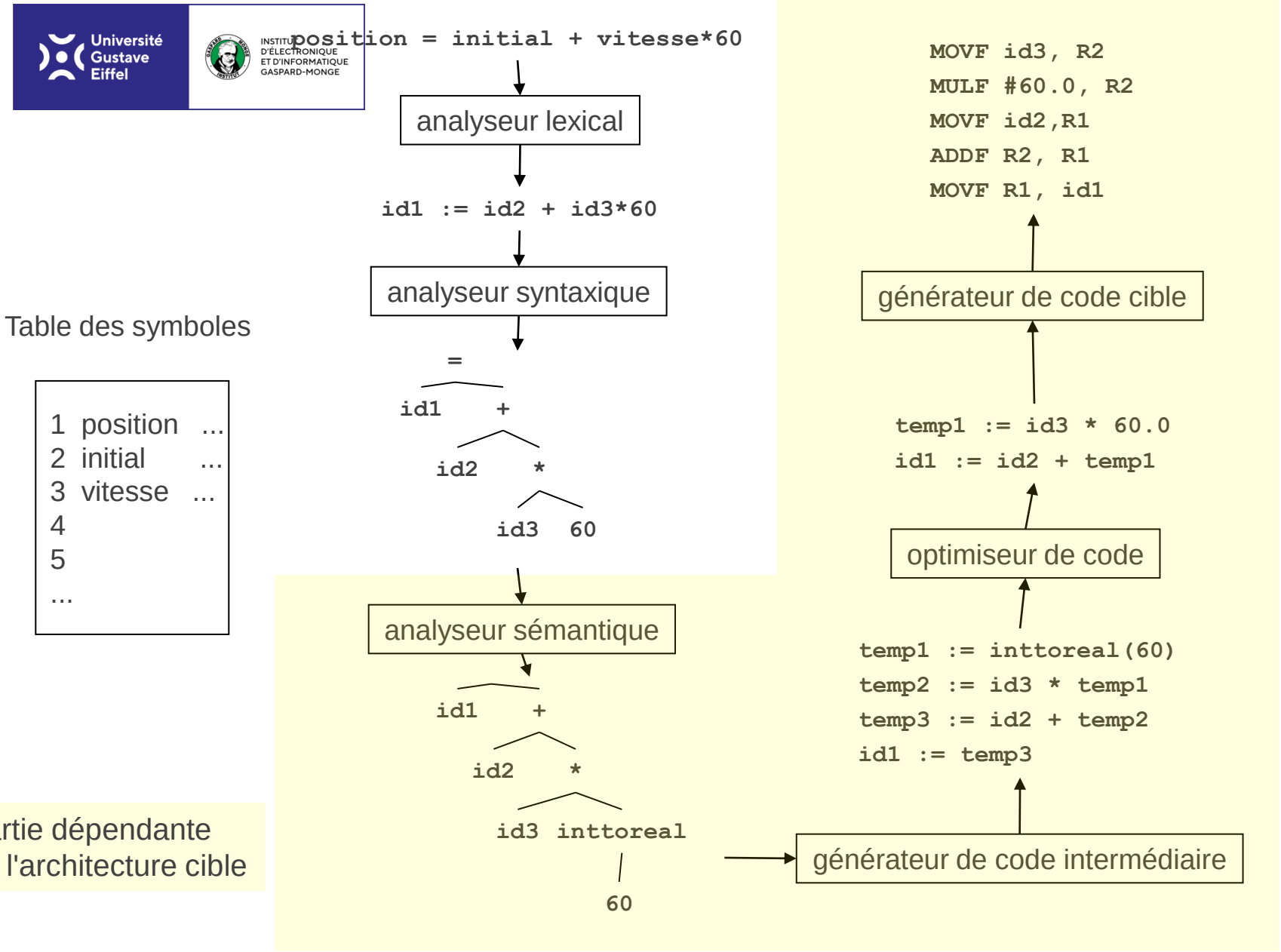
INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Traduction ascendante

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Sommaire


position = initial + vitesse*60
analyseur lexical
id1 := id2 + id3*60
analyseur syntaxique
=
id1 +
id2 *
id3 60
analyseur sémantique
id1 +
id2 *
id3 inttoreal
60
générateur de code intermédiaire
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
optimiseur de code
temp1 := id3 * 60.0
id1 := id2 + temp1
générateur de code cible
MOVF id3, R2
MULF #60.0, R2
MOVF id2,R1
ADDF R2, R1
MOVF R1, id1
Table des symboles
1 position ...
2 initial ...
3 vitesse ...
4
5
...
Partie dépendante
de l'architecture cible

Objectifs du cours

```
> gcc toto.c
toto.c:1: erreur d'analyse syntaxique before ',' token
/tmp/ccImHOOb.o(.text+0x27): In function 'main' : undefined reference to 'get'
toto.c: Dans la fonction "main" :
toto.c:5: attention : suggest explicit braces to avoid ambiguous 'else'
```

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

- Comprendre les messages d'erreur des compilateurs
- Savoir programmer en assembleur
- Connaitre le logiciel Bison
- Savoir écrire un compilateur
- Savoir quelles optimisations sont utiles

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Planning

12 cours, 12 TD sur machine

1 projet logiciel 40 %

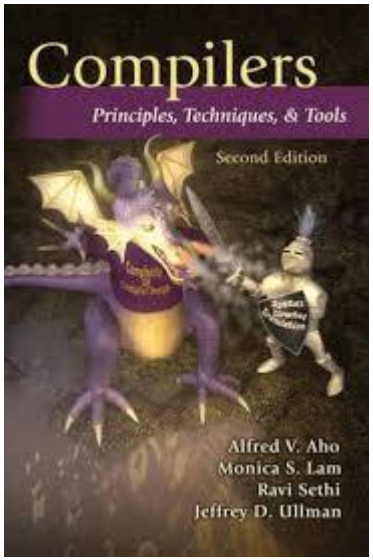
1 examen final 60 %

$nf=(2np+3nef)/5$

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Bibliographie



Aho, Sethi, Ullman, 1986/2007. *Compilateurs. Principes, techniques et outils*, Pearson Education France

Levine, Mason, Doug, 1990. *Lex & Yacc*, O'Reilly.

Ce support de cours est inspiré de Aho *et al.* (1986)

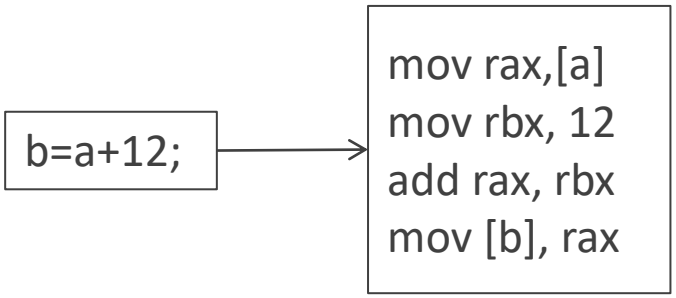
Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Sommaire

Le projet de compilation

TPC ⟶ assembleur nasm



Objectif
Écrire un compilateur en Flex et Bison
Langage source
TPC, presque un sous-ensemble du C
Langage cible
Assembleur nasm 64 bits
Analyse syntaxique
Partir de votre projet d'analyse syntaxique
Dates limites de rendu
intermédiaire : 26 avril
final : 6 juin
Dernier TP le 6 mai

Une différence avec le C
Expressions et instructions

```
Exp :  Exp OR TB
    | TB    ;
TB  :  TB AND FB
    | FB    ;
(...)
E   :  E ADDSUB T
    | T     ;
T   :  T DIVSTAR F
    | F     ;
F   :  ADDSUB F
    | '!' F
    | '(' Exp ')'
    | Lvalue
    | NUM
    | CHARACTER
    | IDENT '(' Arguments  ')'     ;
LValue:      IDENT     ;
```

```
Instr:     Lvalue '=' Exp ';'
    |  ';'
    |  RETURN Exp ';'
    |  RETURN ';'
    |  READE '(' IDENT ')' ';'
    |  READC '(' IDENT ')' ';'
    |  PRINT '(' Exp ')' ';'
    |  IF '(' Exp ')' Instr
    |  IF '(' Exp ')' Instr ELSE Instr
    |  WHILE '(' Exp ')' Instr
    |  IDENT '(' Arguments  ')'
    |  '{' SuiteInstr '}'     ;
```

Expressions	Instructions
`x*x+4`	`y=0;`
`x==y`	`return t;`
`sqrt(x*x+4)`	`printf("%s\n", msg);`

En C, toute expression suivie par un ";" est une instruction

```
y=0
1+2
```

```
y=0;
1+2;
```

Pas dans le projet

Paramètres et arguments

```
DeclFonct:
      EnTeteFonct Corps     ;
EnTeteFonct:
      TYPE IDENT '(' Parametres ')'
    |  VOID IDENT '(' Parametres ')'     ;
Parametres:
      VOID
    |  ListTypVar     ;
ListTypVar:
      ListTypVar ',' TYPE IDENT
    | TYPE IDENT     ;
 F   :  IDENT '(' Arguments  ')' ;
Arguments:
      ListExp
    | ;
ListExp:
      ListExp ',' Exp
    |  Exp     ;
```

Paramètre

Dans une définition de fonction :

```
int absolute_value( int i ) {
```

Argument

Dans un appel de fonction :

```
d=sqrt(absolute_value( delta ));
```

Autre terminologie

Paramètre formel

Paramètre effectif

Université Gustave Eiffel

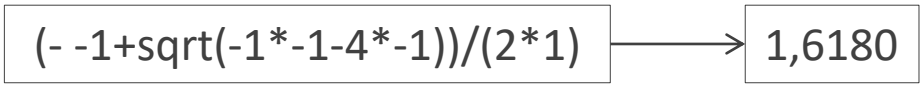
INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Sommaire

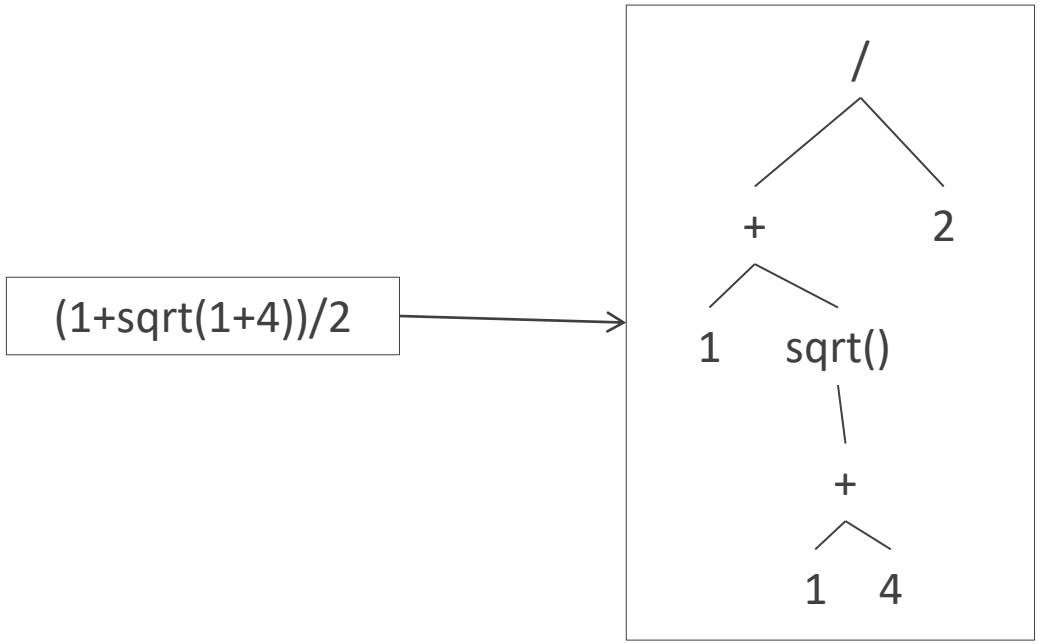
Qu'est-ce que la traduction ?

Traduire du code source en autre chose

Exemple 1 (tiré par les cheveux)

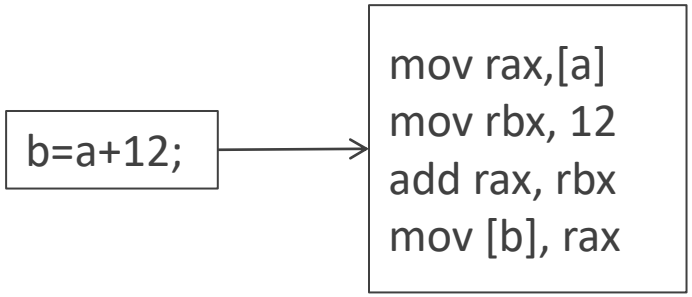


Qu'est-ce que la traduction ? Exemple 2



expression ⟶ arbre

Qu'est-ce que la traduction ?
Exemple 3



programme en TPC → programme en nasm 64 bits

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Calculette en Bison

```
L : E '\n'          { printf("%d", $1) ; }
E : E + T           { $$ = $1 + $3 ; }
  | T
  ;
T : T * F           { $$ = $1 * $3 ; }
  | F
  ;
F : ( E )           { $$ = $2 ; }
  | num
  ;
```

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Traduction et compilation

Traduire le programme source

- en un arbre abstrait
- ou dans un code intermédiaire ou final

On peut faire l'analyse syntaxique et la traduction

- soit en deux passes (plus facile à réaliser)
- soit en une seule passe (possible en Bison)

Université Gustave Eiffel

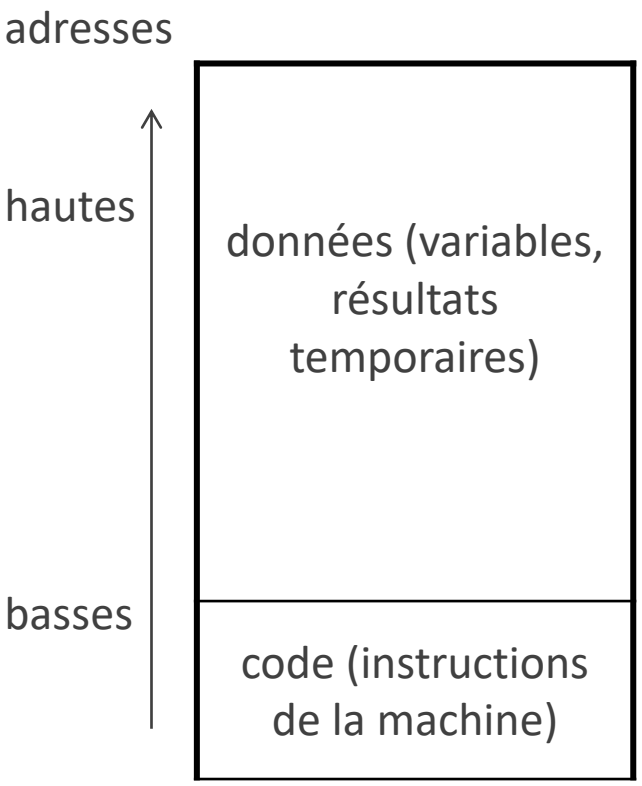
INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Comparaison entre les 3 exemples

Une calculette sert-elle comme module d'un compilateur ?

- pour calculer les valeurs des expressions ?
- pour calculer les valeurs des expressions constantes ?

Un compilateur peut-il garantir qu'une expression ne contient pas de division par zéro ?



Puisqu'on parle de constantes...

Organisation de la mémoire allouée à un programme en cours d'exécution

Faut-il réserver de la mémoire pour les constantes ?

Constantes littérales (**`#define`**) :

dans le code

Variables constantes (**`const`**) :

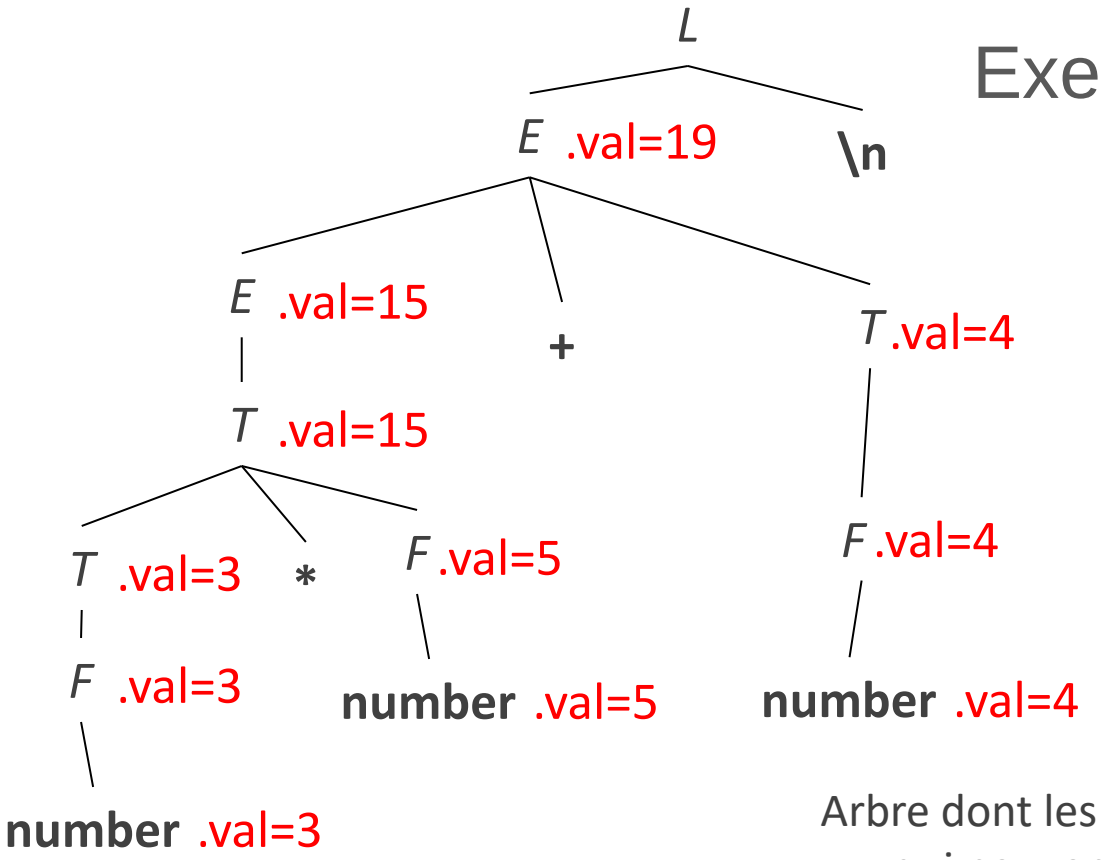
dans les données

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Sommaire

Arbre décoré
Exemple 1 : calculette



Arbre dont les nœuds peuvent avoir des **attributs** qui peuvent avoir une valeur

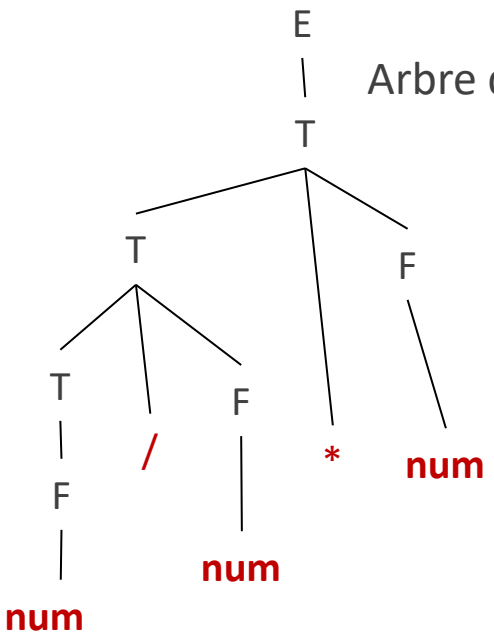
On sauvegarde des informations dans les attributs

Exemple 2 : construction d'un arbre

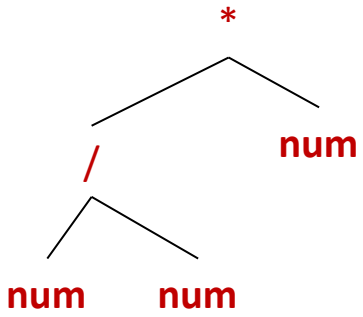
$E \rightarrow E + T \mid E - T \mid T$

$T \rightarrow T * F \mid T / F \mid F$

$F \rightarrow (E) \mid \mathbf{num}$



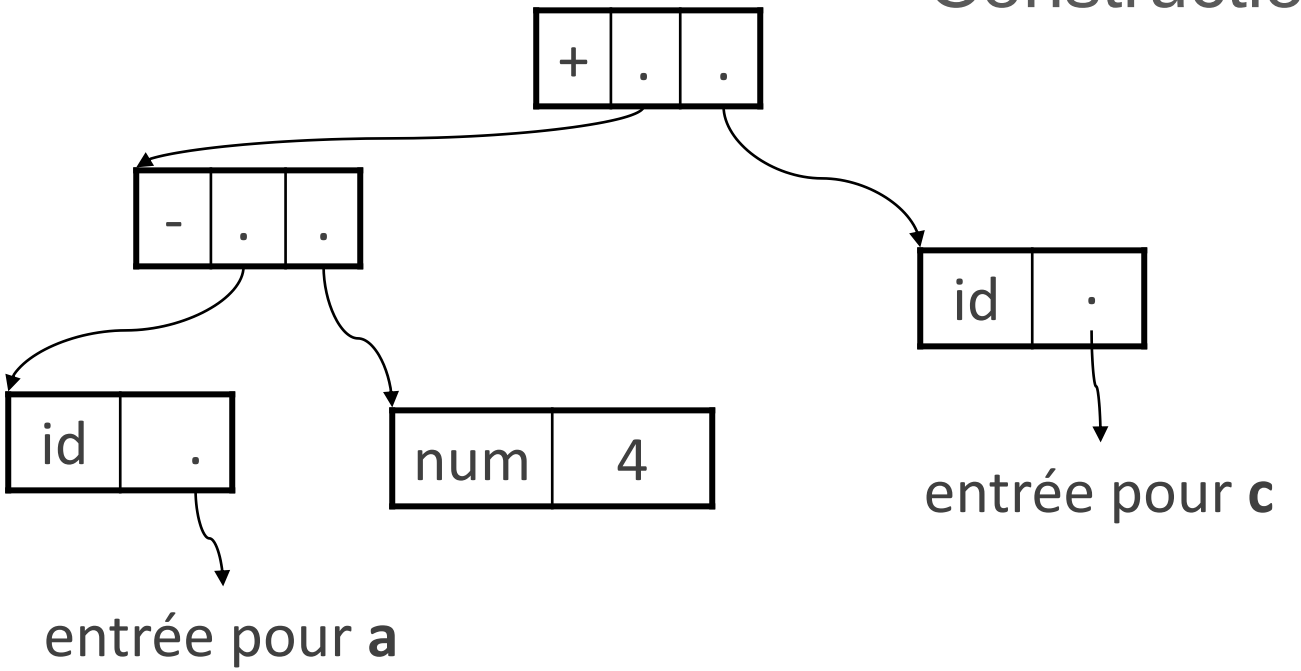
Arbre de dérivation



Arbre abstrait

On ne garde que les nœuds utiles

Construction d'un arbre



Arbre abstrait de `a-4+c`

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Actions dans Bison

```
L : E '\n'   { printf("%d", $1) ; }
E : E + T    { $$ = $1 + $3 ; }
  | T
  ;
T : T * F    { $$ = $1 * $3 ; }
  | F
  ;
F : ( E )    { $$ = $2 ; }
  | num
  ;
```

On peut attacher des actions aux règles

Numérotation

Chaque nœud fils, en commençant à 1

Ordre de réalisation des actions

L'analyseur syntaxique réalise l'action quand il réduit suivant la règle

Exploration de l'arbre de dérivation en profondeur (*depth-first*)

Grammaire attribuée
Exemple 1 : calculette

règle	action
L --> *E* '**\n**'	print(*E*.val)
E --> *E* + *T*	*E*.val := E_1.val + *T*.val
E --> *T*	*E*.val := *T*.val
T --> *T* * *F*	*T*.val := T_1.val * *F*.val
T --> *F*	*T*.val := *F*.val
F --> **(** *E* **)**	*F*.val := *E*.val
F --> **num**	*F*.val := **num**.val

Dans l'attribut d'un nœud de l'arbre de dérivation on sauvegarde la valeur de la sous-expression

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Numérotation des non-terminaux

E --> *E* + *T* **E.val := E_1.val** + *T*.val

Si la règle comporte plusieurs fois un même non-terminal

Une occurrence **avec un indice** dans l'action correspond à l'occurrence correspondante dans le **membre droit** de la règle

Une occurrence **sans indice** correspond au **membre gauche** de la règle

La convention Bison (`$1`, `$2`...) numérote autrement

Exemple 2 : construction d'un arbre binaire

Fonctions utilisées

`makeNode(op, left, right)`	crée un nœud avec une étiquette, l'opérateur `op`, plus deux autres champs pour les pointeurs `left` et `right`
`makeLeaf(id, attr)`	crée un nœud avec une étiquette `id` plus un autre champ pour un pointeur vers une entrée de la table des symboles
`makeLeaf(num, val)`	crée un nœud avec une étiquette `num` plus un autre champ pour la valeur de la constante

Grammaire attribuée Construction d'un arbre binaire

E --> *E* + *T*	*E*.ptr := makeNode('+', E_1.ptr, *T*.ptr)
E --> *E* - *T*	*E*.ptr := makeNode('-', E_1.ptr, *T*.ptr)
E --> *T*	*E*.ptr := *T*.ptr
T --> **(** *E* **)**	*T*.ptr := *E*.ptr
T --> **id**	*T*.ptr := makeLeaf(**id**, **id**.symb_tab_ind)
T --> **num**	*T*.ptr := makeLeaf(**num**, **num**.val)

Les deux attributs *E*.ptr et *T*.ptr contiennent des pointeurs sur les arbres

Arbres d'arité libre

Fonctions utilisées

`Node *makeNode()`	crée un nœud
`void addSibling(Node *node, Node *sibling)`	ajoute à un nœud un frère `sibling`
`void addChild(Node *node, Node *child)`	ajoute à un nœud un fils `child`

Grammaires attribuées

X --> expr $\quad b := f(c_1, c_2, \ldots c_k)$

Une grammaire attribuée est définie par
- une grammaire algébrique
- des attributs associés à chaque symbole terminal ou non-terminal
- des actions associées à chaque règle

Une action peut avoir des entrées $c_1, c_2, \ldots c_k$ et des sorties b

Elle calcule la valeur d'un attribut

Autre utilisation des attributs : analyse sémantique

Vérifier les contraintes supplémentaires par rapport à l'analyse syntaxique

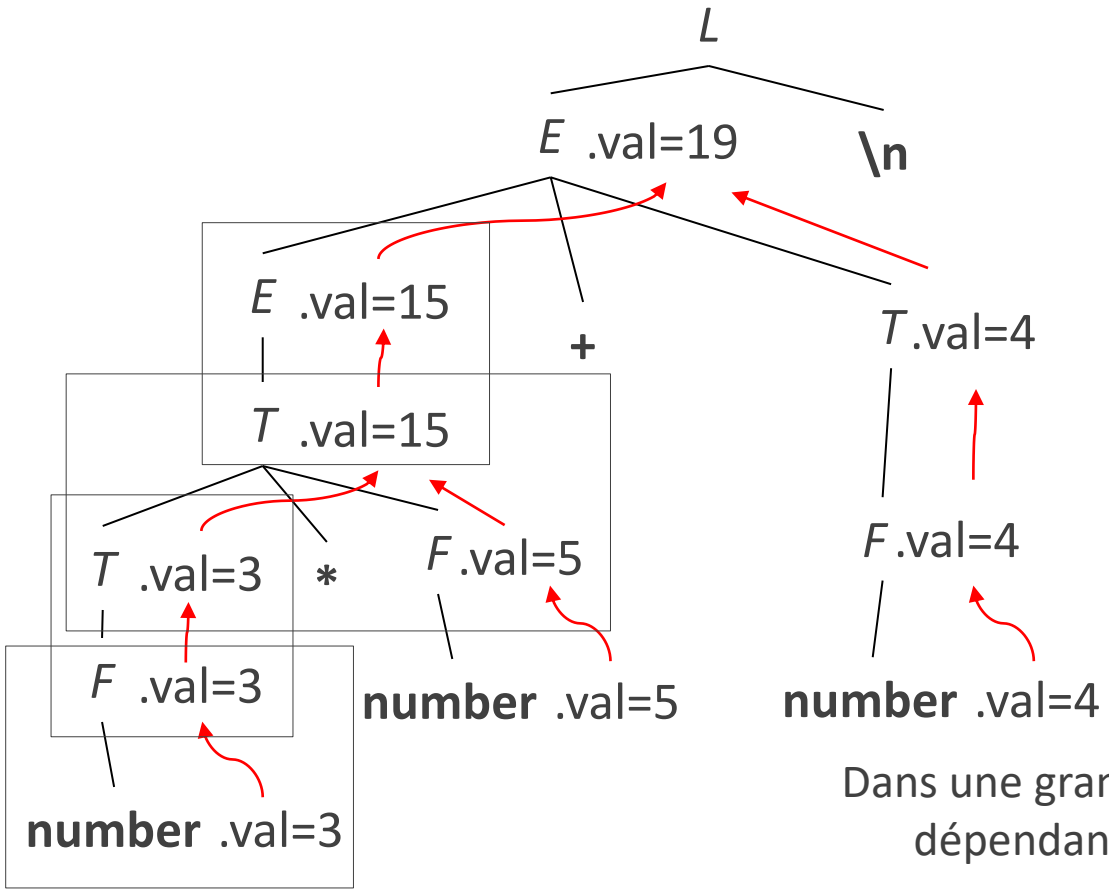
- on doit déclarer les identificateurs avant de les utiliser
- les opérateurs doivent être compatibles avec le type des opérandes
- ...

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Sommaire

Dépendances entre attributs



Dans une grammaire attribuée, toutes les dépendances sont locales

Les actions associées aux règles suffisent pour calculer tous les attributs

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Grammaires attribuées

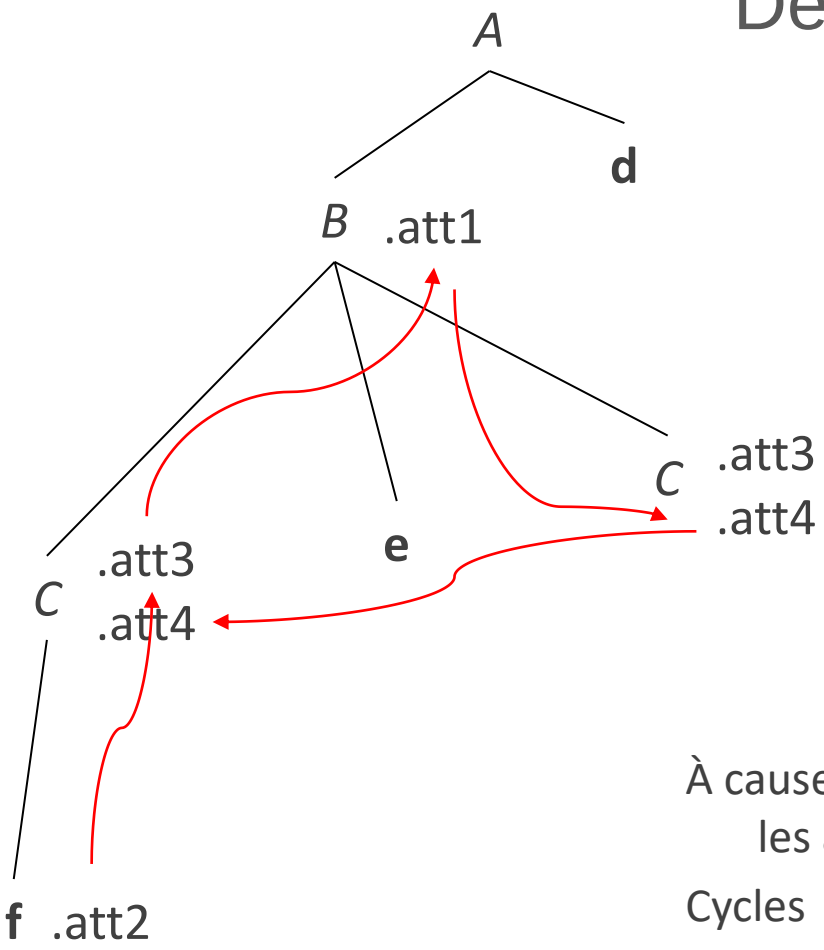
X --> *expr* $\qquad b := f(c_1, c_2, \ldots c_k)$

Une action associée à une règle peut avoir des entrées $c_1, c_2, \ldots c_k$ et des sorties *b*

Ce sont des valeurs des attributs

- de *X*
- et des symboles formant *expr*

Dépendances entre attributs

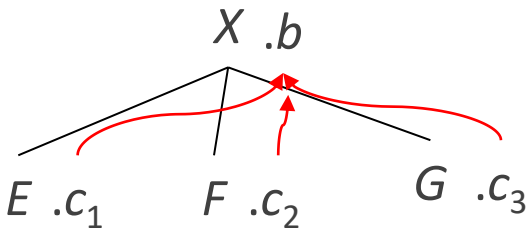


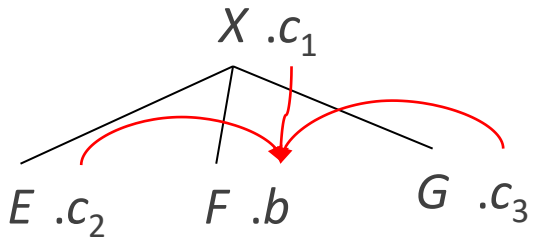
À cause des dépendances, on ne peut pas calculer les attributs dans n'importe quel ordre

Cycles

Attributs synthétisés ou hérités

X --> *expr* $\qquad\qquad b := f(c_1, c_2, \ldots c_k)$

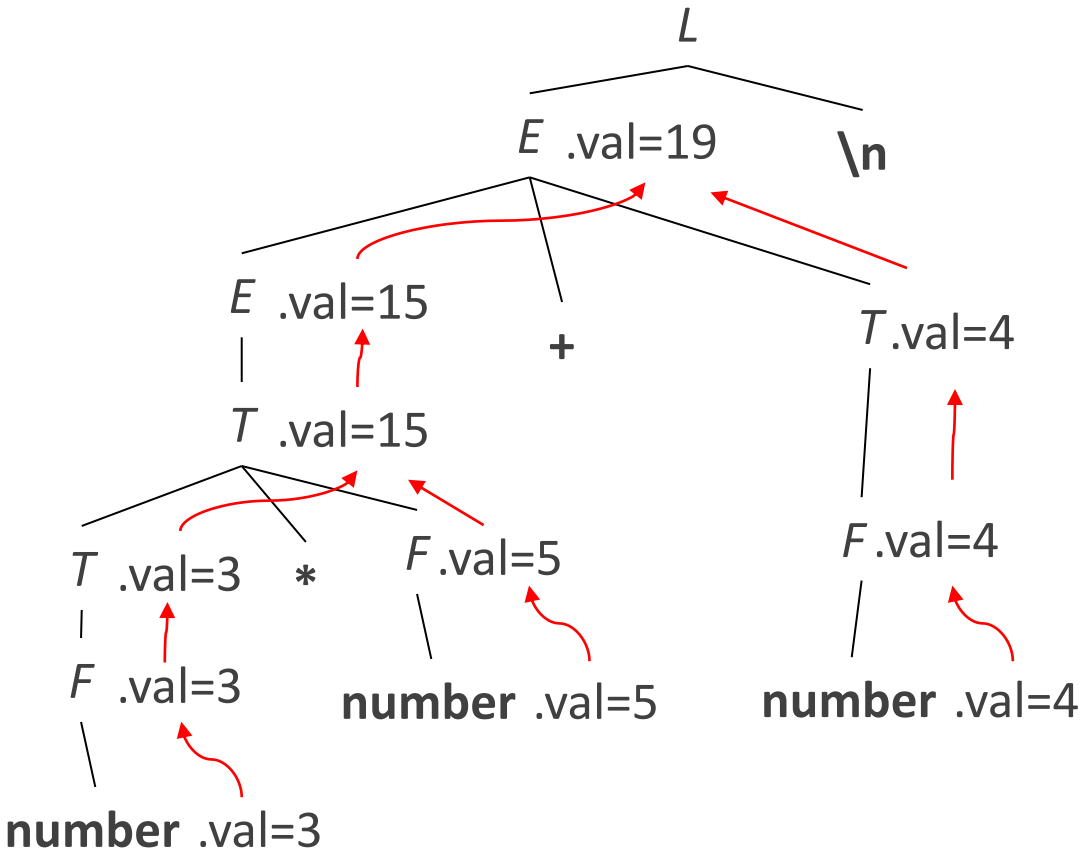




L'attribut *b* est un **attribut synthétisé** si
dans toutes les actions où il est calculé, c'est un attribut de *X*

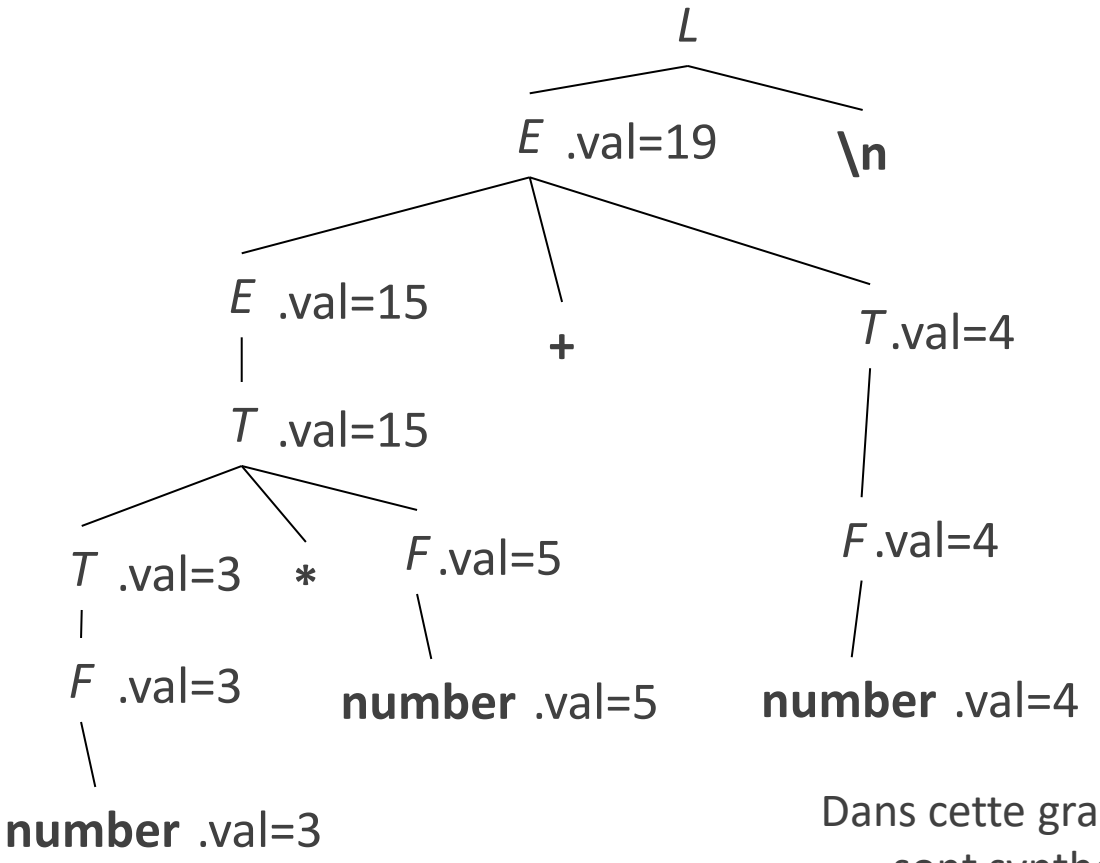
C'est un **attribut hérité** si
dans toutes les actions où il est calculé, c'est un attribut d'un des symboles formant *expr*

Dépendances entre attributs



Dans cet exemple, les attributs de chaque nœud ne dépendent que des attributs de ses fils

Grammaire S-attribuée



règle	action
L --> *E* '**\n**'	print(*E*.val)
E --> *E* + *T*	*E*.val := E_1.val + *T*.val
E --> *T*	*E*.val := *T*.val
T --> *T* * *F*	*T*.val := T_1.val * *F*.val
T --> *F*	*T*.val := *F*.val
F --> **(** *E* **)**	*F*.val := *E*.val
F --> **number**	*F*.val := **number**.val

Dans cette grammaire attribuée, tous les attributs sont synthétisés

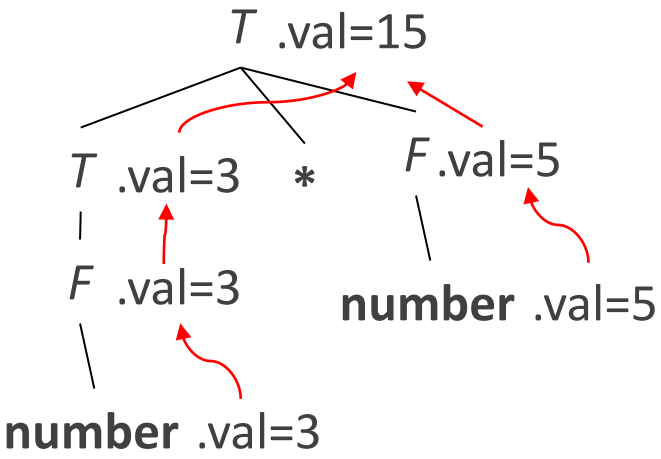
Grammaire S-attribuée

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Sommaire

Traduction ascendante avec une grammaire S-attribuée



Traduction ascendante

Analyse ascendante et traduction en une seule passe

Avec une grammaire S-attribuée

Un traducteur ascendant peut calculer tous les attributs au moment des réductions

C'est ce que font les traducteurs ascendants produits par Bison

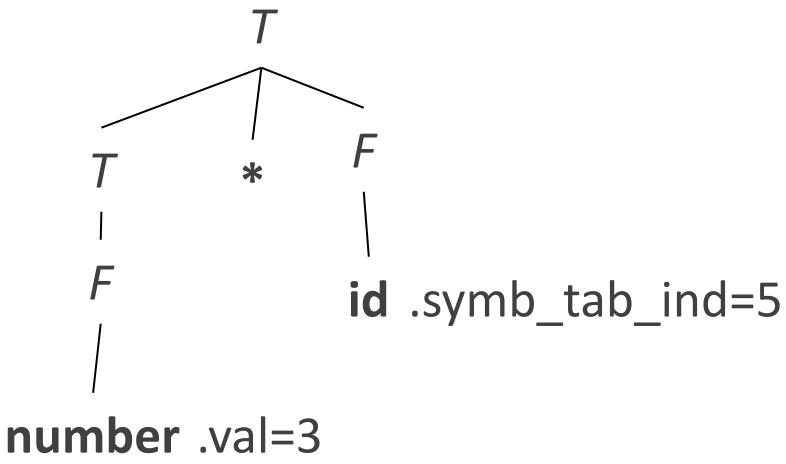
Traduction ascendante avec Bison

```
L : e '\n'        {printf("%d",$1);}
e   : e '+' t     {$$=$1+$3;}
    | t
    ;
t   : t '*' f     {$$=$1+$3;}
    | f
    ;
f   : '(' e ')'   {$$=$2;}
    | NUM
    ;
```

Avec Bison,

- **chaque nœud de l'arbre a au plus 1 attribut**
- **tous les attributs ont le même type**
  - par défaut, int
  - en général, une union

Les attributs des lexèmes

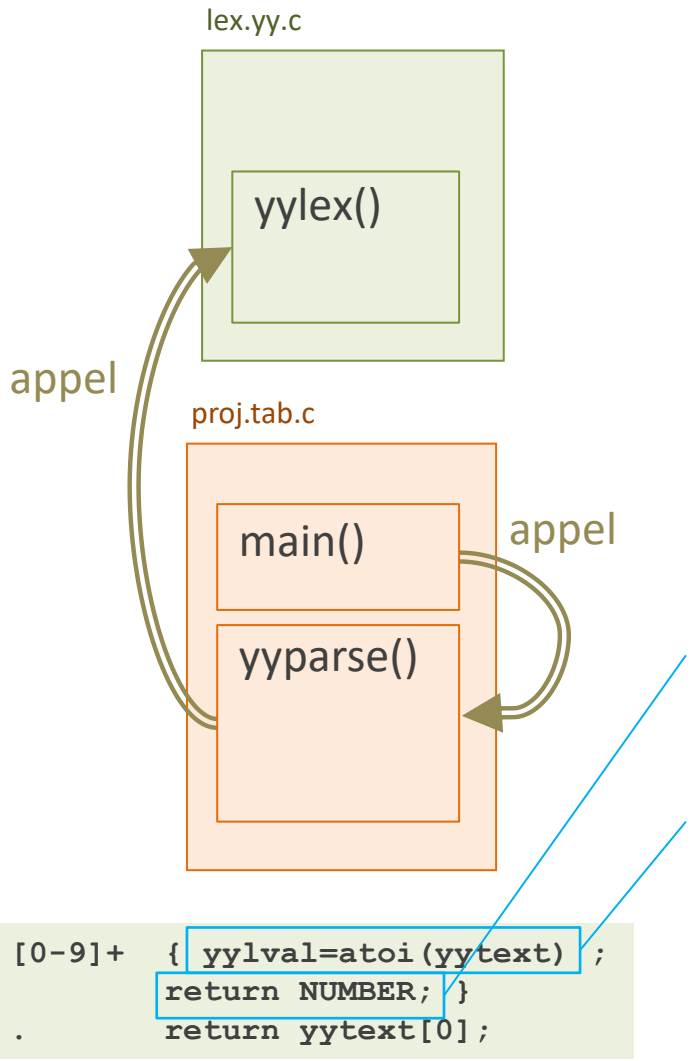


Point de départ de la traduction ascendante

Attributs fournis par l'analyseur lexical

Avec Flex, l'attribut est dans **`yylval`**

Représentation des lexèmes avec Flex et Bison



Les lexèmes sont représentés en 2 parties

1. Valeur de retour de `yylex()`

Elle correspond à un **terminal** de la grammaire

Type de retour de yylex() : **`int`**

2. Attribut du lexème

Regroupe toutes autres informations sur le lexème

Variable globale **`yylval`**

Bison déclare **`yylval`** dans le fichier d'en-tête .h

Type par défaut : **`int`**

Déclarer le type de yylval

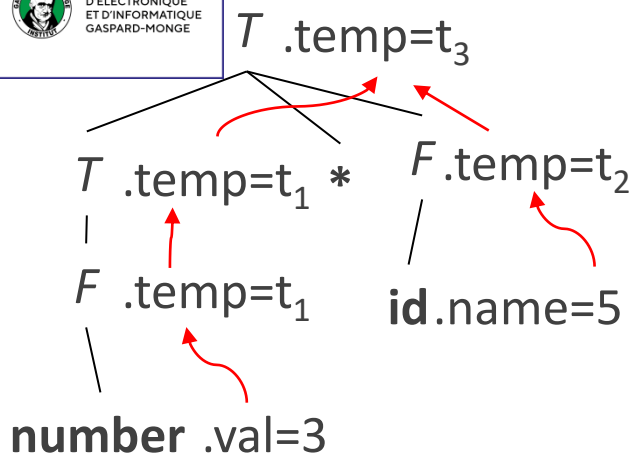
```
[a-zA-Z_][a-zA-Z0-9_]*   { strcpy(yylval.ident, yytext) ;
                          return IDENT; }
[0-9]+                   { sscanf(yytext, "%d", &(yylval.num) );
                          return NUMBER; }
```

```
%union {
    char ident[64];
    int num;
}
%token <ident> IDENT
%token <num> NUMBER
%%
```

On veut que **`yylval`** n'ait pas le même type pour tous les lexèmes

Dans le programme Bison

- mettre une déclaration **`%union`** avec les types de tous les attributs
- dans la déclaration **`%token`** des lexèmes qui ont un **`yylval`**, préciser le champ de l'union



Les attributs des non-terminaux avec Flex et Bison

```
typedef struct {
    type_descriptor type;
    int offset;
} temporary_descriptor;
```

```
%union {
    char name[64];
    int val;
    temporary_descriptor *temp;
}
%type <temp> E T F
%%
```

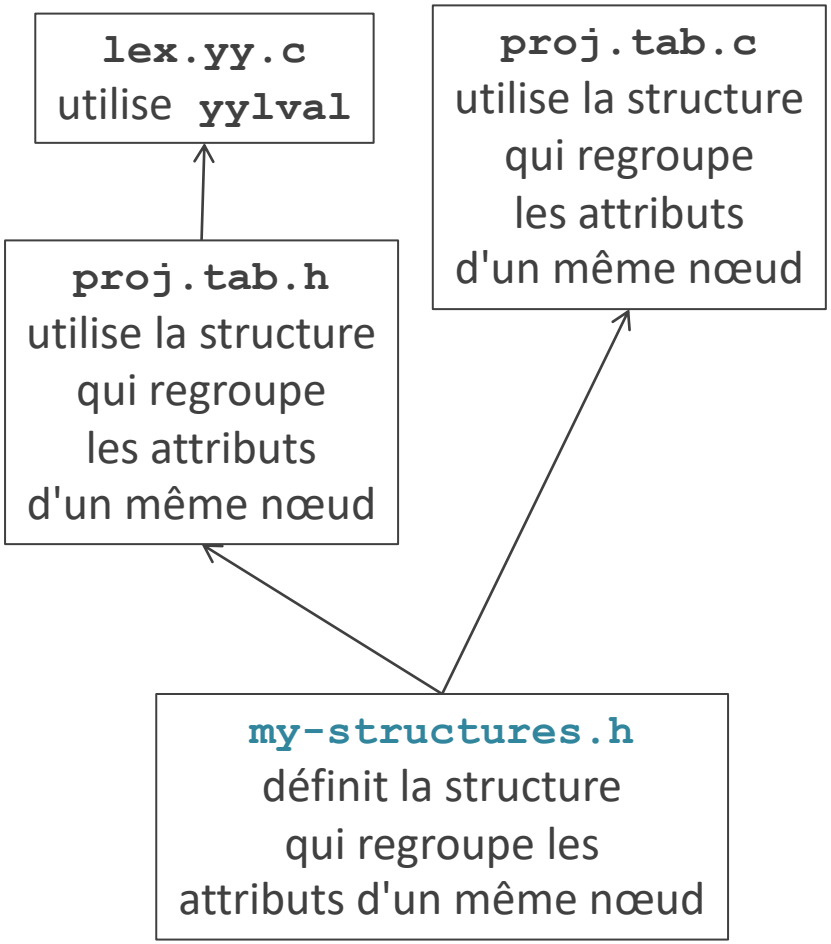
Un seul attribut par non-terminal

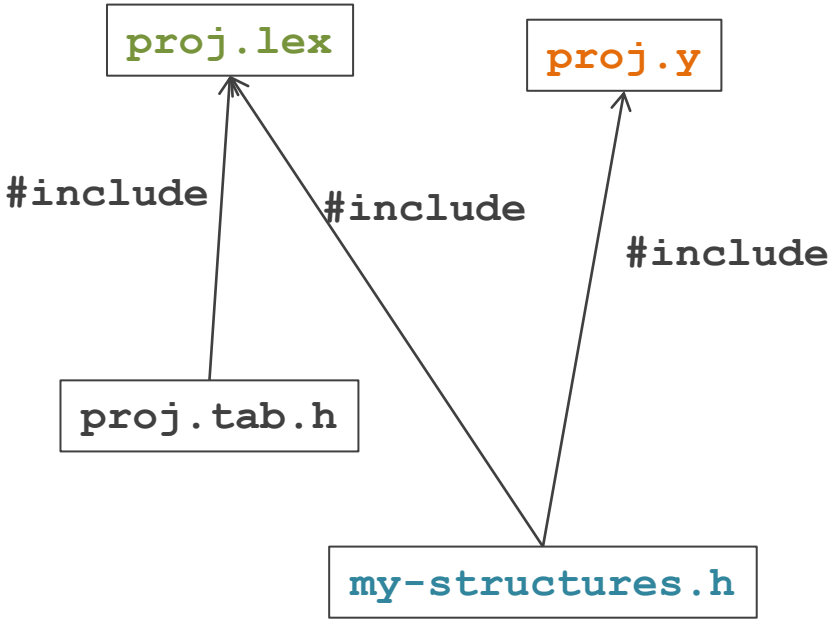
Recenser les attributs du non-terminal

Les regrouper dans une structure s'il y en a plusieurs

Où déclarer cette structure ?

Les attributs des non-terminaux avec Flex et Bison





Traduction avec Bison

état	…	*X*	*Y*	*Z*	
attribut	…	*X*.x	*Y*.y	*Z*.z	

Le traducteur sauvegarde les attributs dans la pile de l'analyseur

S --> *E* **$**

E --> *E* **+** *T*

E --> *T*

T --> *T* * *F*

T --> *F*

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

F --> **num**

Donnée	Pile	Attributs	Règle
3*5+4$	ε	-	
*5+4$	**num**	3	*F* --> **num**
*5+4$	*F*	3	*T* --> *F*
*5+4$	*T*	3	
5+4$	*T* *	3 -	
+4$	*T* * **num**	3 - 5	*F* --> **num**
+4$	*T* * *F*	3 - 5	*T* --> *T* * *F*
+4$	*T*	15	

Traduction avec Bison

avant réduction

état	...	*X*	*Y*	*Z*	
attribut	...	*X*.x	*Y*.y	*Z*.z	

après réduction

état	...	*A*			
attribut	...	*A*.a			

Calcul d'un attribut pendant une réduction

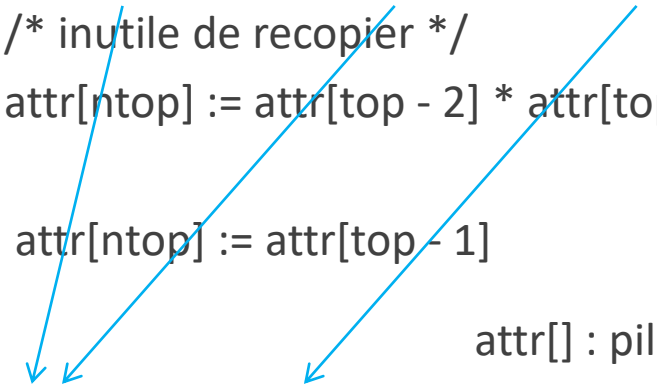
A --> *X Y Z* *A*.a := *f*(*X*.x, *Y*.y, *Z*.z)

Réduction :

- le traducteur calcule *A*.a en fonction des valeurs contenues dans la pile
- il dépile *X Y Z*
- il empile *A*
- il sauvegarde *A*.a dans la pile

Implémentation en C

L --> *E* '**\n**'	print(attr[top-1])
E --> *E* + *T*	attr[ntop] := attr[top - 2] + attr[top]
E --> *T*	/* inutile de recopier */
T --> *T* * *F*	attr[ntop] := attr[top - 2] * attr[top]
T --> *F*	
F --> **(** *E* **)**	attr[ntop] := attr[top - 1]
F --> **num**	



état	...	*E*	+	*T*	
attribut	...	*E*.val		*T*.val	

état	...	*E*			
attribut	...	*E*.val			

attr[] : pile des attributs

top : taille actuelle de la pile

ntop : taille de la pile après la réduction en cours
(se déduit de top et de la longueur de la règle)

Traduction avec Bison

```
L : E '\n' { printf("%d", $1) ; }
E : E + T  { $$ = $1 + $3 ; }
  | T
  ;
T : T * F   { $$ = $1 * $3 ; }
  | F
  ;
F : ( E )   { $$ = $2 ; }
  | num
  ;
```

Certaines règles n'ont pas d'action

On aurait pu mettre `{ $$ = $1 ; }`

Traduction avec Bison

état	...	*X*	*Y*	*Z*	
attribut	...	*X*.x	*Y*.y	*Z*.z	

état	...	*A*			
attribut	...	*A*.a			

Le traducteur effectue les actions seulement au moment où il réduit

Dans une règle $A \dashrightarrow X_1 X_2 \ldots X_n$, comment faire si une action doit être placée entre X_i et X_{i+1} ?

Actions avant la fin de la règle

```
E    : T R
     ;
R    : addop T { print($1) ; } R
     |
     ;
T    : num { print($1) ; }
     ;
```

On peut mettre des actions avant le dernier symbole des règles

Le traducteur fera les actions au moment correspondant

Actions avant la fin de la règle

```
R    : addop T { printf("\n") ; } R
     |
     ;
```

devient en interne :

```
R    : addop T M R
     |
     ;
M    : { printf("\n") ; }
     ;
```

En interne, Bison remplace ces actions par des non-terminaux supplémentaires

L'automate LR(0) et la table LALR construits par Bison sont faits à partir d'une grammaire qui contient ces non-terminaux

Actions avant la fin de la règle

```
R    : addop T { print($1) ; } R
     |
     ;
```

La numérotation des symboles dans le membre droit des règles tient compte des actions

Chaque action compte comme un symbole

L'attribut de *R* est dans **$4**

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Résumé

Les grammaires attribuées permettent d'incorporer la traduction à l'analyse syntaxique pour obtenir un traducteur en une passe

Quand tous les attributs sont synthétisés, un traducteur ascendant peut les sauvegarder dans la pile