Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

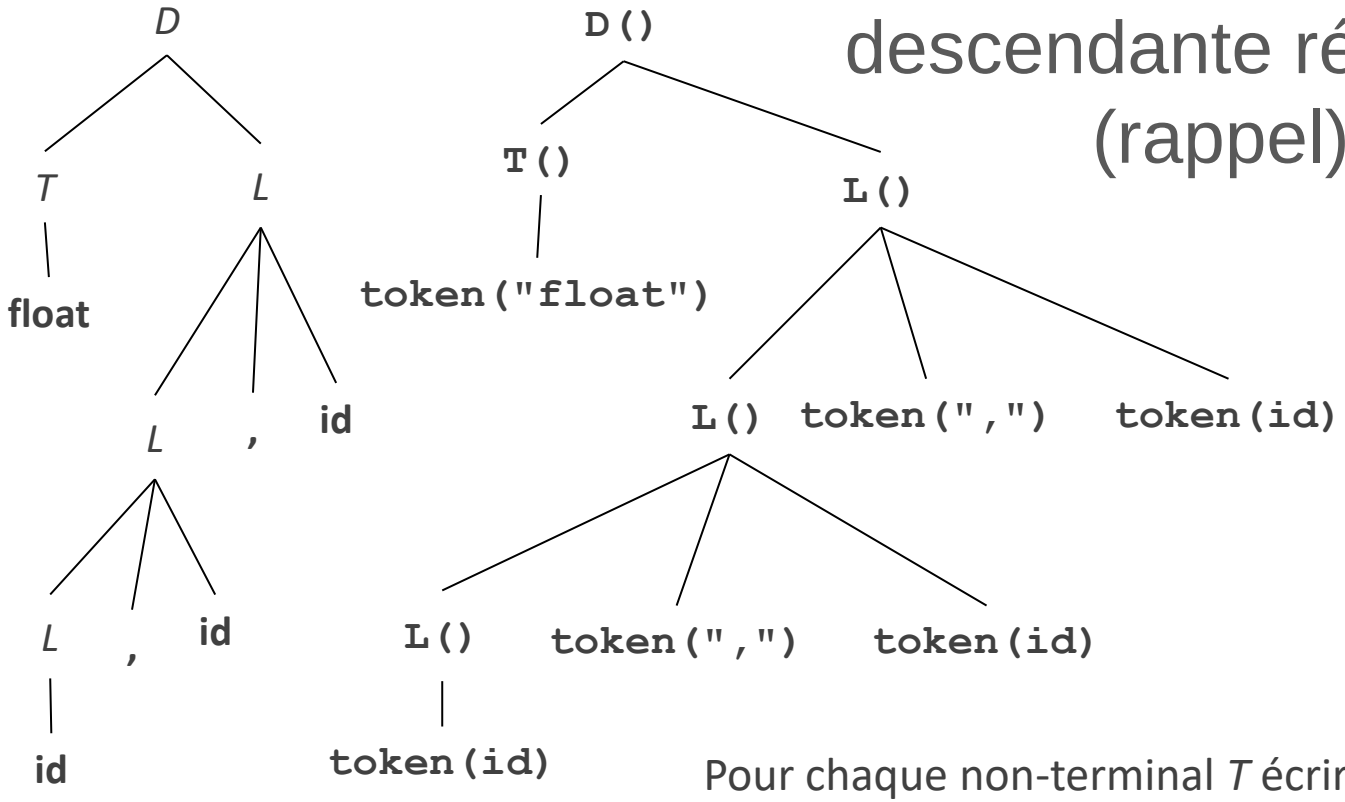
Traduction descendante récursive
Attributs hérités

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Sommaire

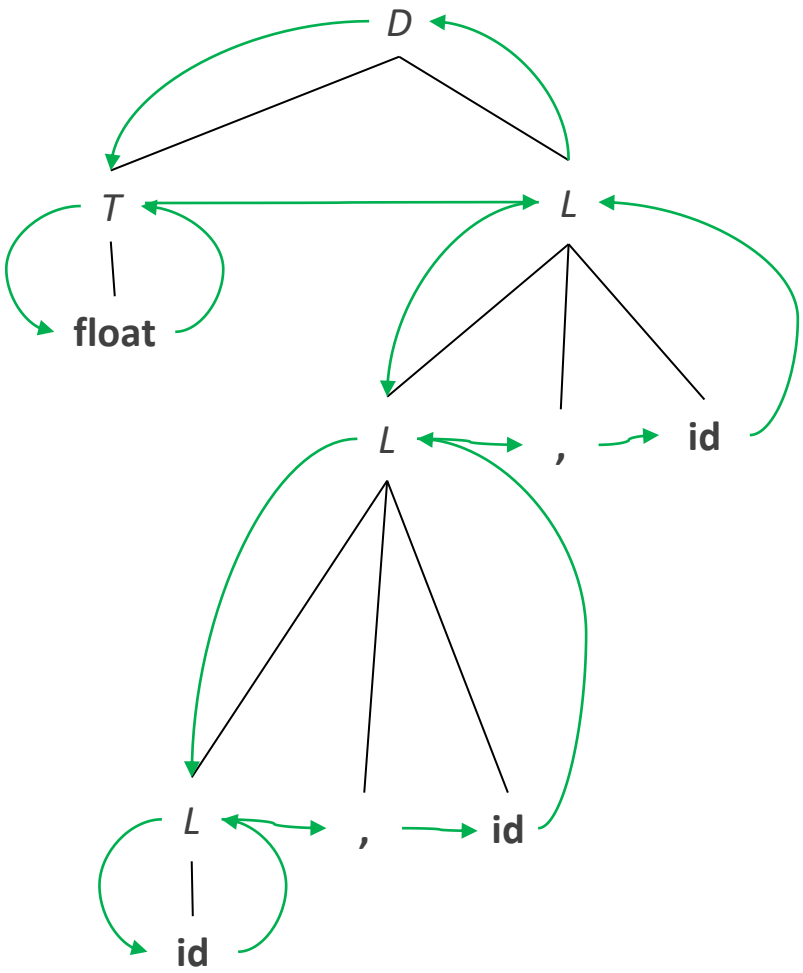
Analyse syntaxique descendante récursive (rappel)



Pour chaque non-terminal *T* écrire une fonction
`T()`

L'arbre des appels aux fonctions reproduit l'arbre de dérivation

Analyse syntaxique descendante récursive (rappel)

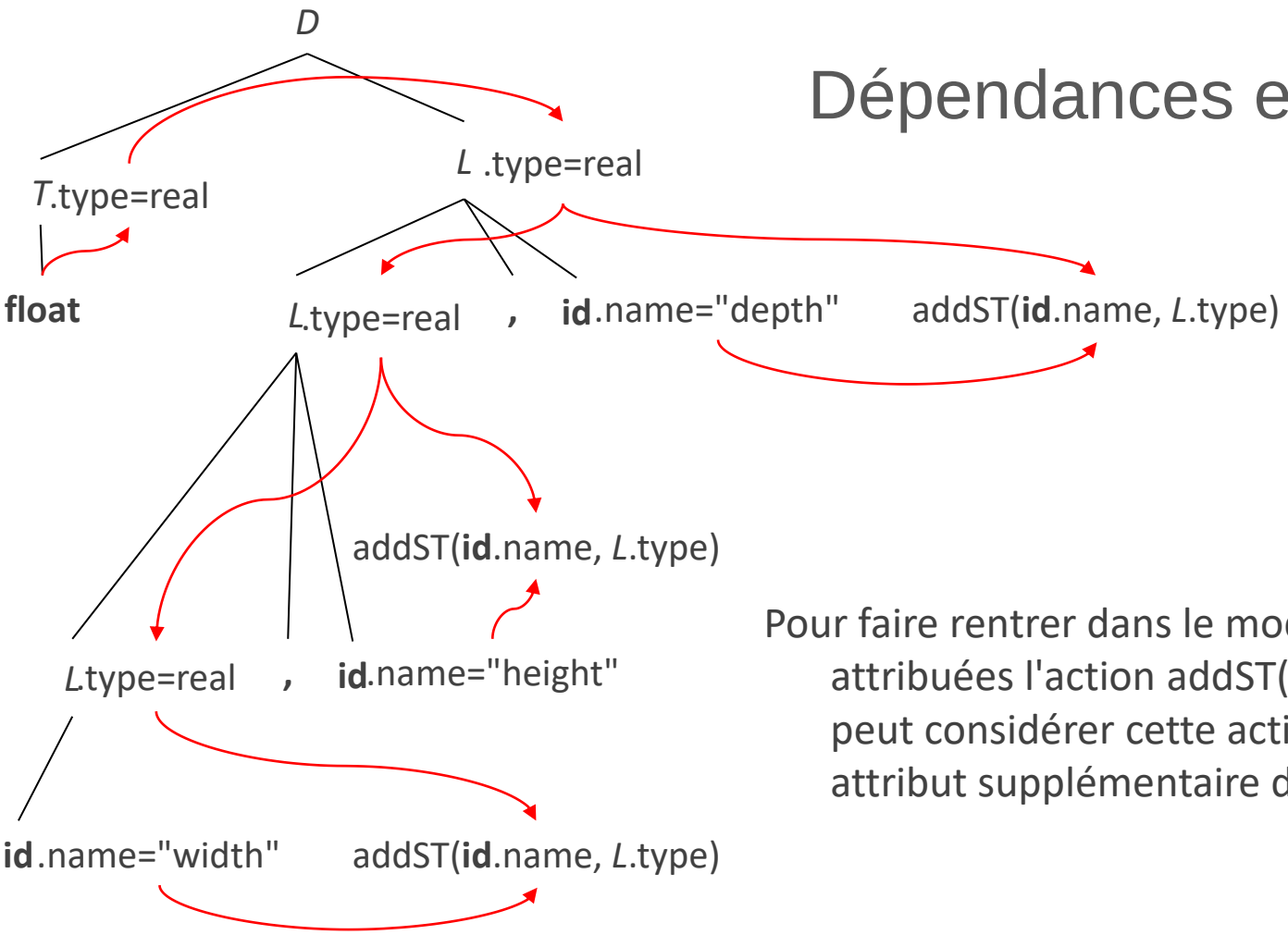


Parcours de l'arbre de dérivation

Parcours en profondeur

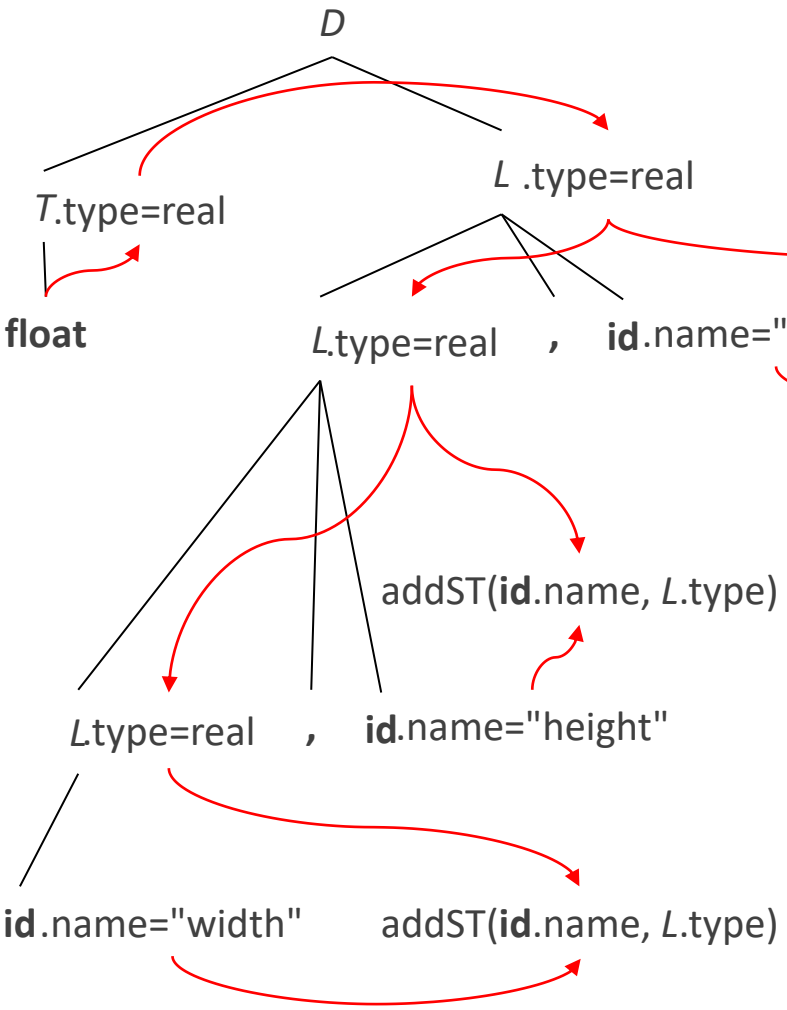
Chaque nœud est visité 2 fois

Dépendances entre attributs



Pour faire rentrer dans le modèle des grammaires attribuées l'action addST(**id**.name, *L*.type), on peut considérer cette action comme un attribut supplémentaire de *L*

Grammaires L-attribuées



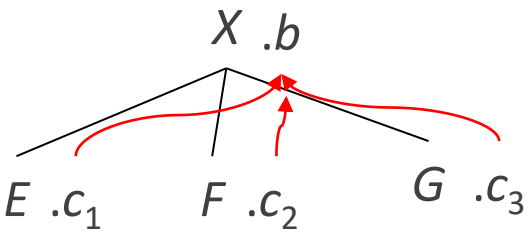
Grammaire attribuée dans laquelle toutes les dépendances entre attributs sont compatibles avec un parcours en profondeur

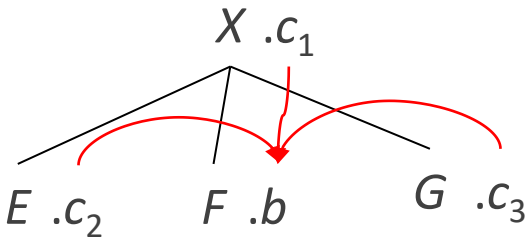
Déclaration de variables en C

D --> *T L*	*L*.type := *T*.type
T --> **int**	*T*.type := integer
T --> **float**	*T*.type := real
L --> *L* **, id**	L_1.type := *L*.type ; addST(**id**.name, *L*.type)
L --> **id**	addST(**id**.name, *L*.type)

Attributs synthétisés ou hérités (rappel)

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

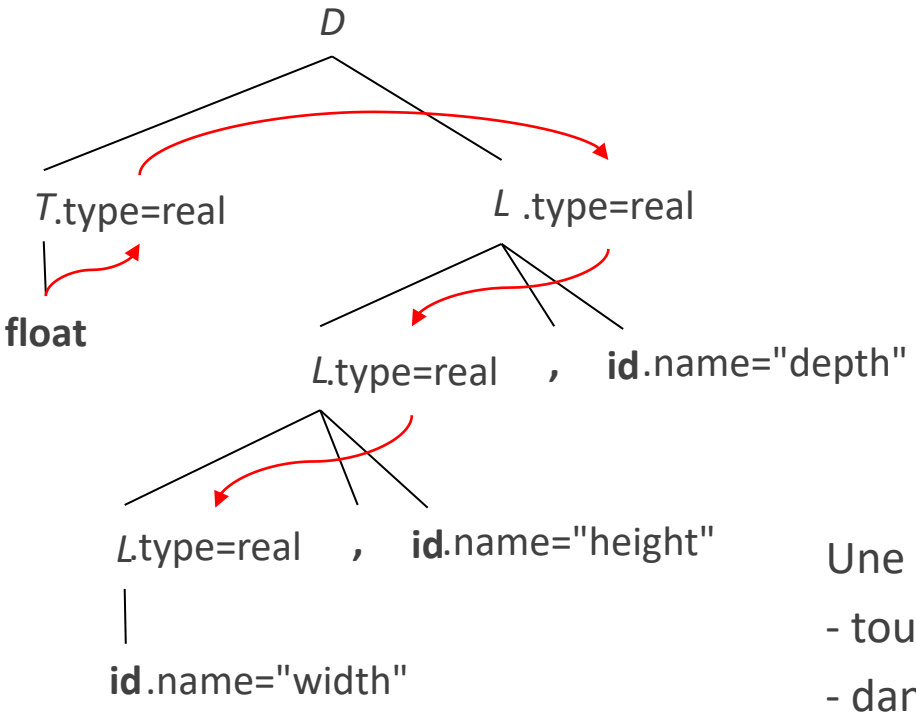


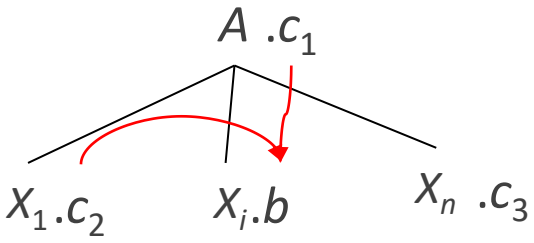


L'attribut *b* est un **attribut synthétisé** si
dans toutes les actions où il est calculé, c'est un attribut de *X*

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

Grammaires L-attribuées



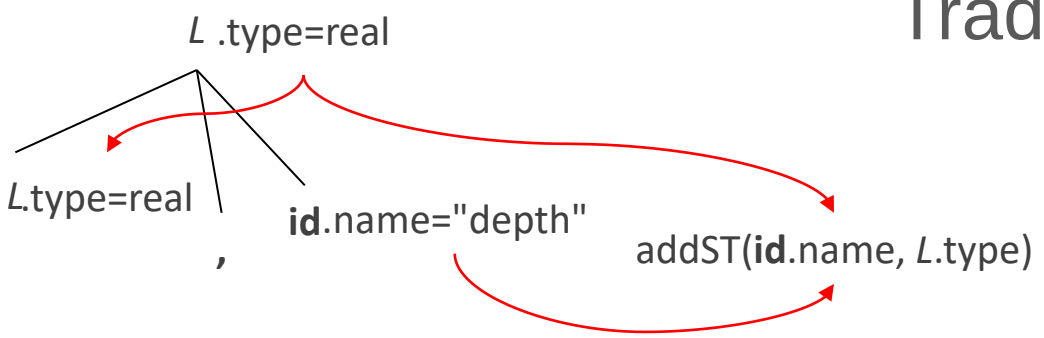


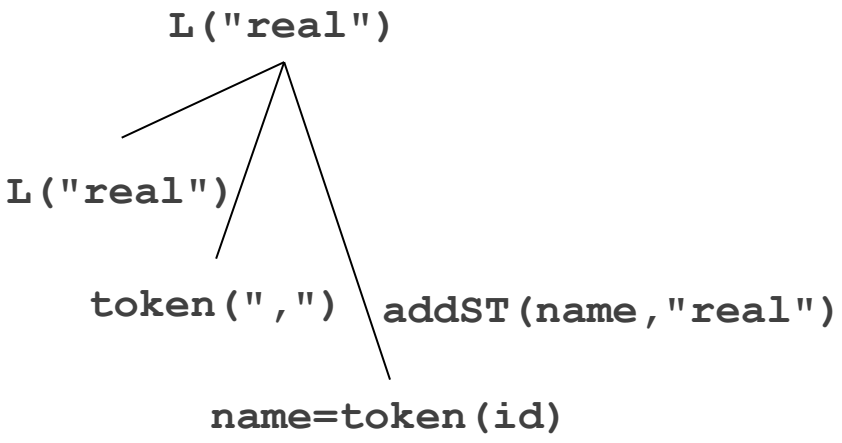
Une grammaire est L-attribuée si

- tout attribut est synthétisé ou hérité ;
- dans une règle $A \rightarrow X_1 X_2 \ldots X_n$, si un attribut X_i.b est calculé dans l'action associée, il ne dépend que des attributs des variables $X_1 X_2 \ldots X_{i-1}$ ou des attributs hérités de A

Toute grammaire S-attribuée est L-attribuée

Traduction descendante récursive





Transformer un analyseur syntaxique descendant récursif en traducteur

Donner aux fonctions `T()` des paramètres et des valeurs de retour

Sauvegarder les attributs dans les paramètres et les valeurs de retour

Exemple

gcc

Traduction descendante récursive

Cette méthode est applicable si
- la grammaire attribuée est L-attribuée
- et la grammaire algébrique est LL(1)

D --> *T L*	*L*.type := *T*.type
T --> **int**	*T*.type := integer
T --> **float**	*T*.type := real
L --> *L* **, id**	L_1.type := *L*.type ; addST(**id**.name, *L*.type)
L --> **id**	addST(**id**.name, *L*.type)

Avec cette grammaire attribuée, la traduction par descente récursive ne fonctionne pas

La grammaire algébrique n'est pas LL(1)

Traduction descendante récursive en pratique

```
int next_terminal;

void parse(void){
  next_terminal = lexan();
  start_symbol(); /* l'axiome */
  check(DONE);
}
```

Comment choisir entre deux règles *T* --> *u* et *T* --> *v* ?

En vérifiant si le prochain terminal appartient à Premier(*u*Suivant(*T*)) ou à Premier(*v*Suivant(*T*))

Interface avec l'analyseur lexical

Sauvegarder le prochain terminal dans une variable globale `next_terminal`

Initialiser `next_terminal` avant d'appeler la fonction de l'axiome

Quand la fonction de l'axiome s'est terminée, vérifier qu'on est à la fin de la donnée

En fin de fichier, `yylex()` renvoie 0

Traduction descendante récursive en pratique

E --> *E* + *T*	*E*.val := E_1.val + *T*.val
E --> *E* - *T*	*E*.val := E_1.val - *T*.val
E --> *T*	*E*.val := *T*.val
T --> **(** *E* **)**	*T*.val := *E*.val
T --> **number**	*T*.val := **number**.val

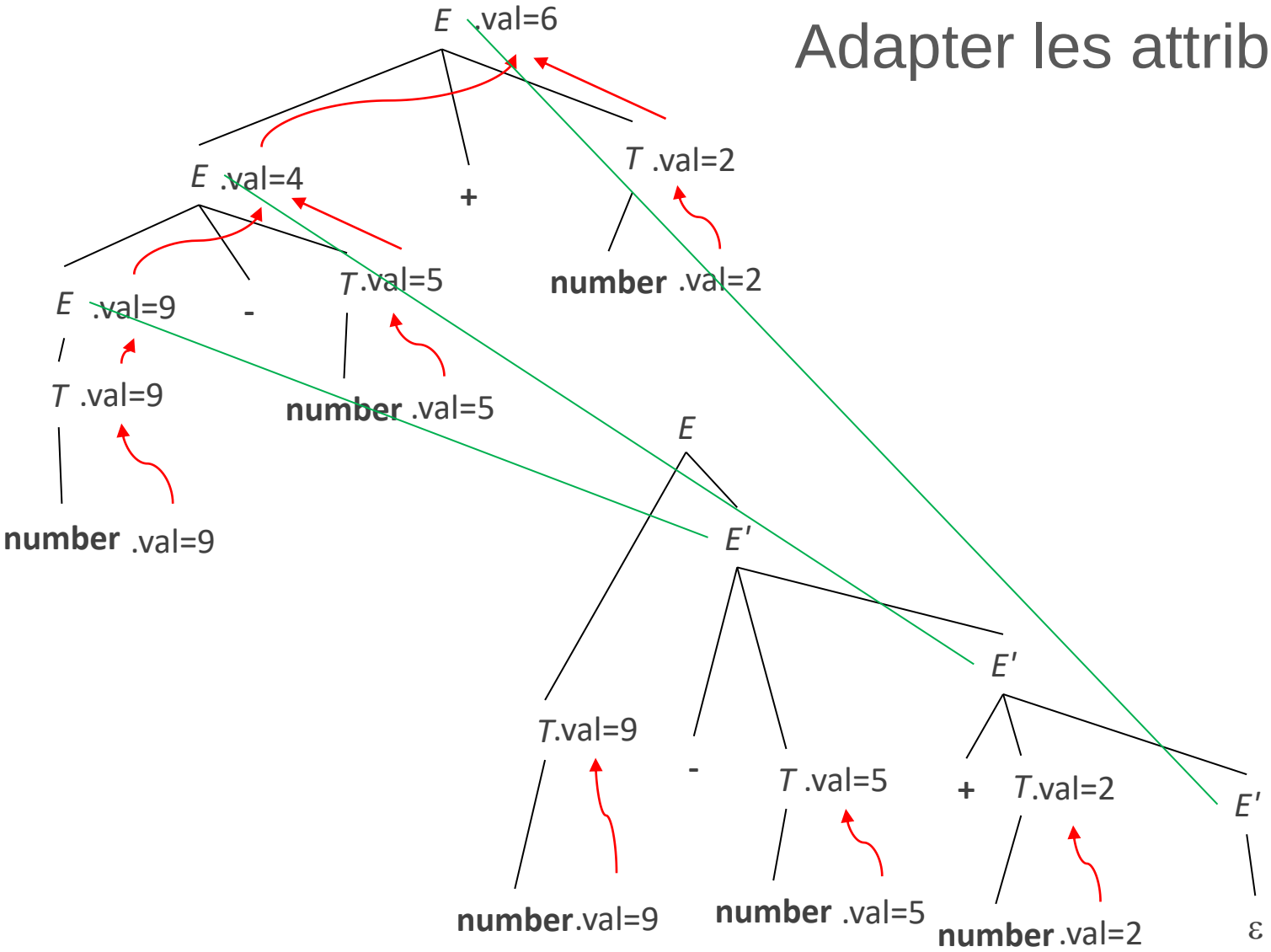
E --> *T E′*	?
E′ --> + *T E′*	?
E′ --> - *T E′*	?
E′ --> ε	?
T --> **(** *E* **)**	?
T --> **number**	*T*.val := **number**.val

Et si la grammaire algébrique n'est pas LL(1) ?

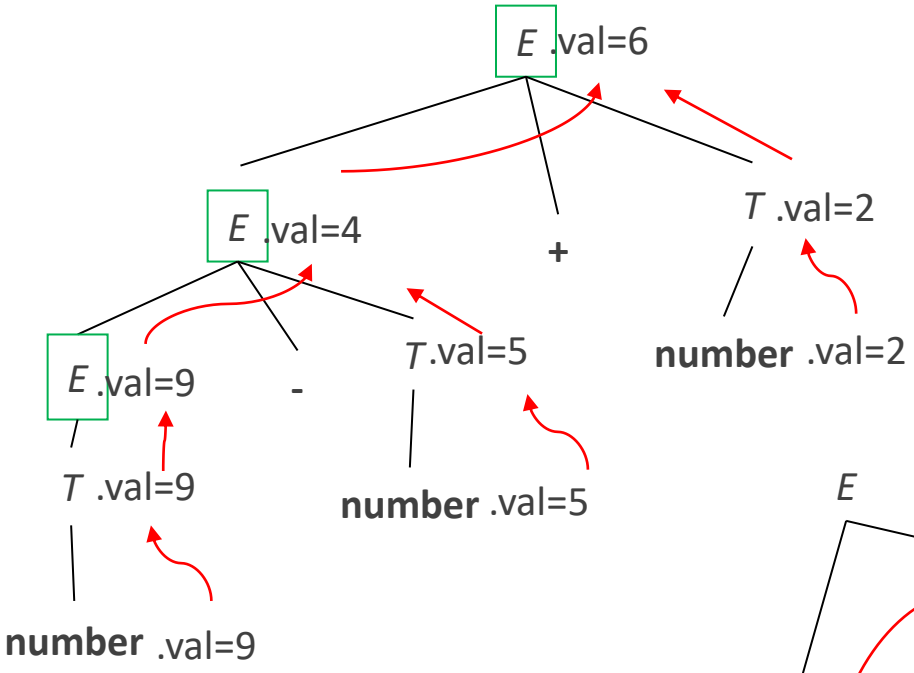
Modifier la grammaire algébrique

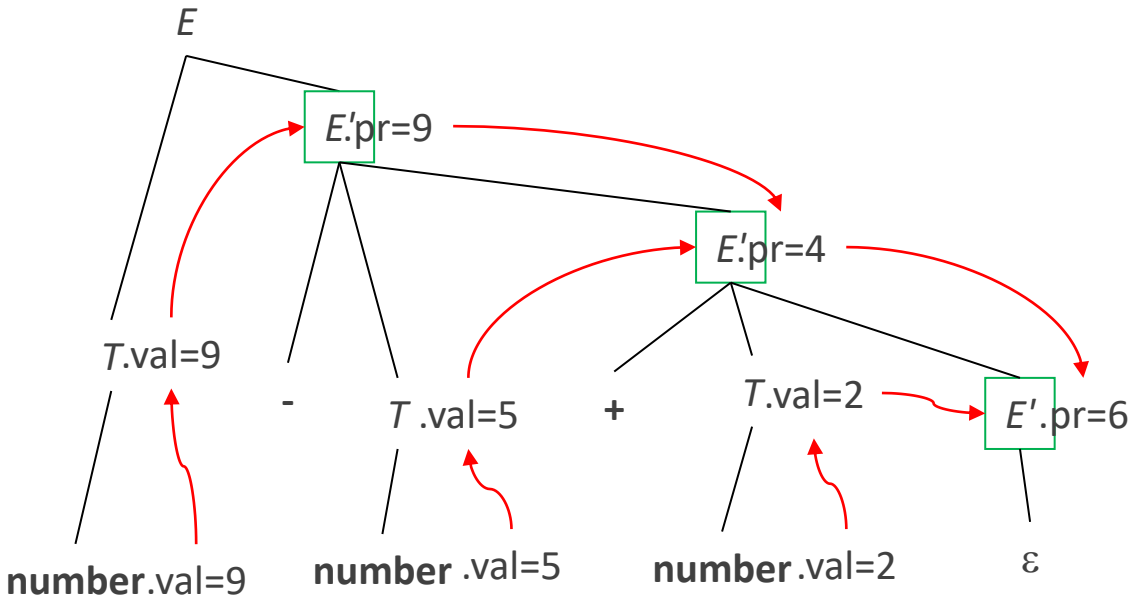
Adapter les attributs

Adapter les attributs



Adapter les attributs





L'attribut *E'*.pr (partial result) sert à sauvegarder le résultat des opérations à gauche

Adapter les attributs

Il faut aussi un attribut pour transmettre à la racine le résultat final

E --> *T E'*

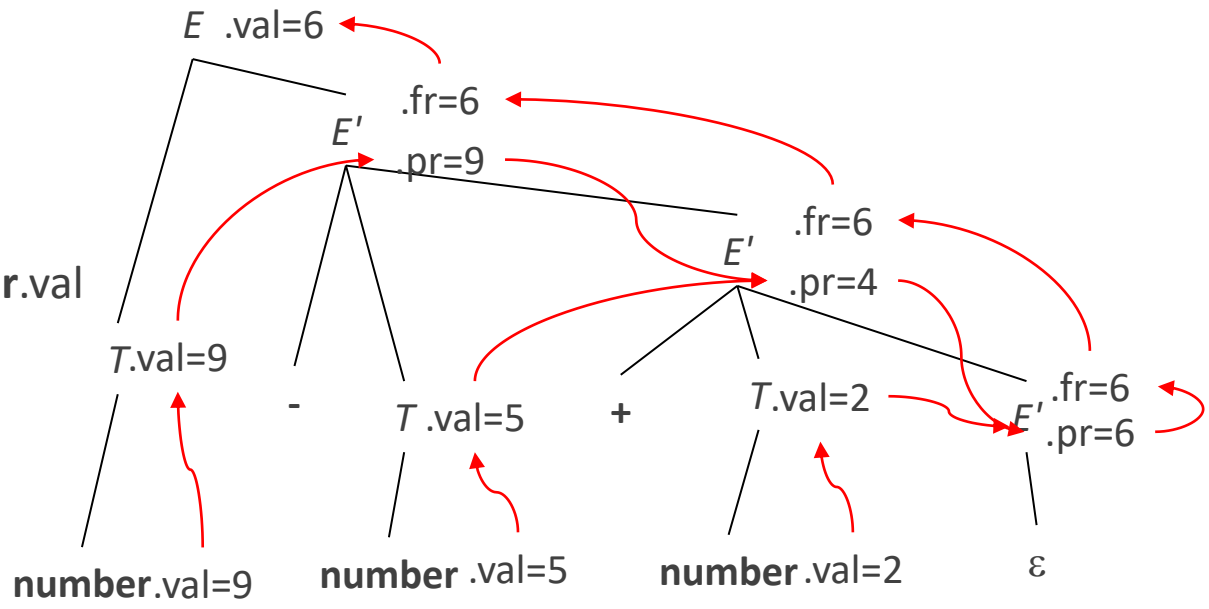
E' --> + *T E'*

E' --> - *T E'*

E' --> ε

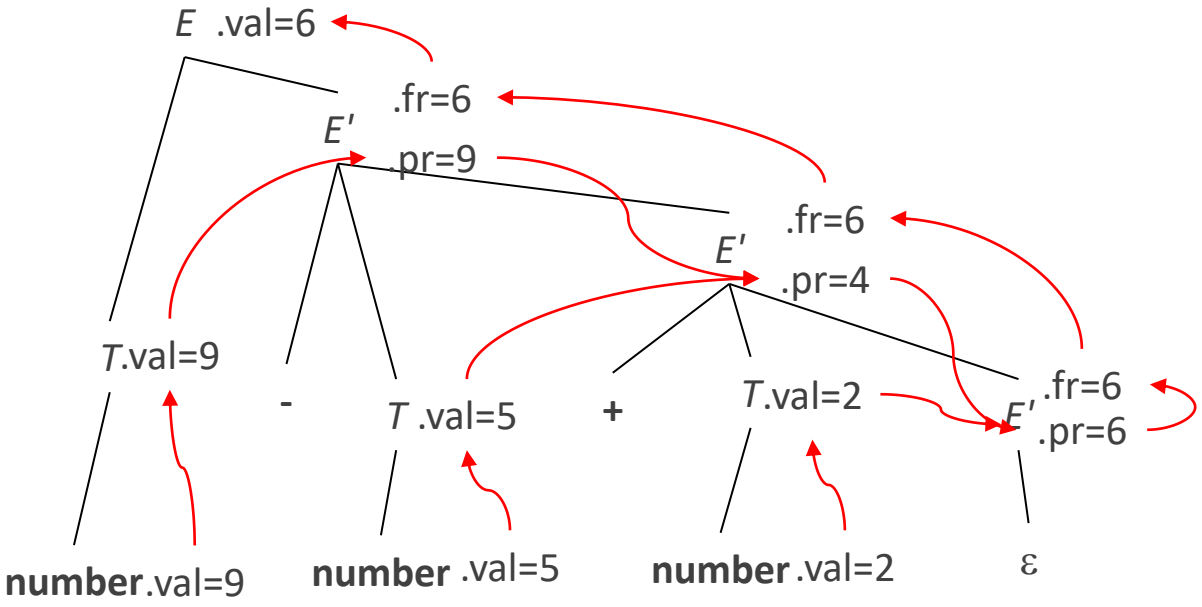
T --> **(*E*)**

T --> **number** *T*.val := **number**.val



Adapter les attributs

E --> *T E′*	*E′*.pr := *T*.val ; *E*.val := *E′*.fr
E′ --> + *T E′*	E'_1.pr := *E′*.pr + *T*.val ; *E′*.fr := E'_1.fr
E′ --> - *T E′*	E'_1.pr := *E′*.pr - *T*.val ; *E′*.fr := E'_1.fr
E′ --> ε	*E′*.fr := *E′*.pr
T --> **(** *E* **)**	*T*.val := *E*.val
T --> **number**	*T*.val := **number**.val



Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Traduction descendante avec une grammaire L-attribuée

Donnée : un schéma de traduction non récursif à gauche

Résultat : le code d'un traducteur descendant

Pour chaque non-terminal *A*, construire une fonction dont les paramètres sont les attributs hérités de *A* et qui renvoie comme valeur les attributs synthétisés de *A* (on suppose qu'il n'y en a qu'un)

Le code pour *A* décide quelle règle appliquer en fonction du symbole courant dans la donnée

Pour chaque attribut d'une variable du membre droit, déclarer une variable locale

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Traduction descendante avec une grammaire L-attribuée

Le code associé à une règle parcourt le membre droit et fait les actions suivantes :

- pour un symbole terminal **X** avec un attribut x, sauvegarder la valeur de x dans une variable locale et lire **X**
- pour un non-terminal *B*, faire c := *B*(b_1, b_2, ... b_k) où c est l'attribut synthétisé de B et b_1, b_2, ... b_k sont les attributs hérités de *B*
- pour une action, faire l'action

Exercice

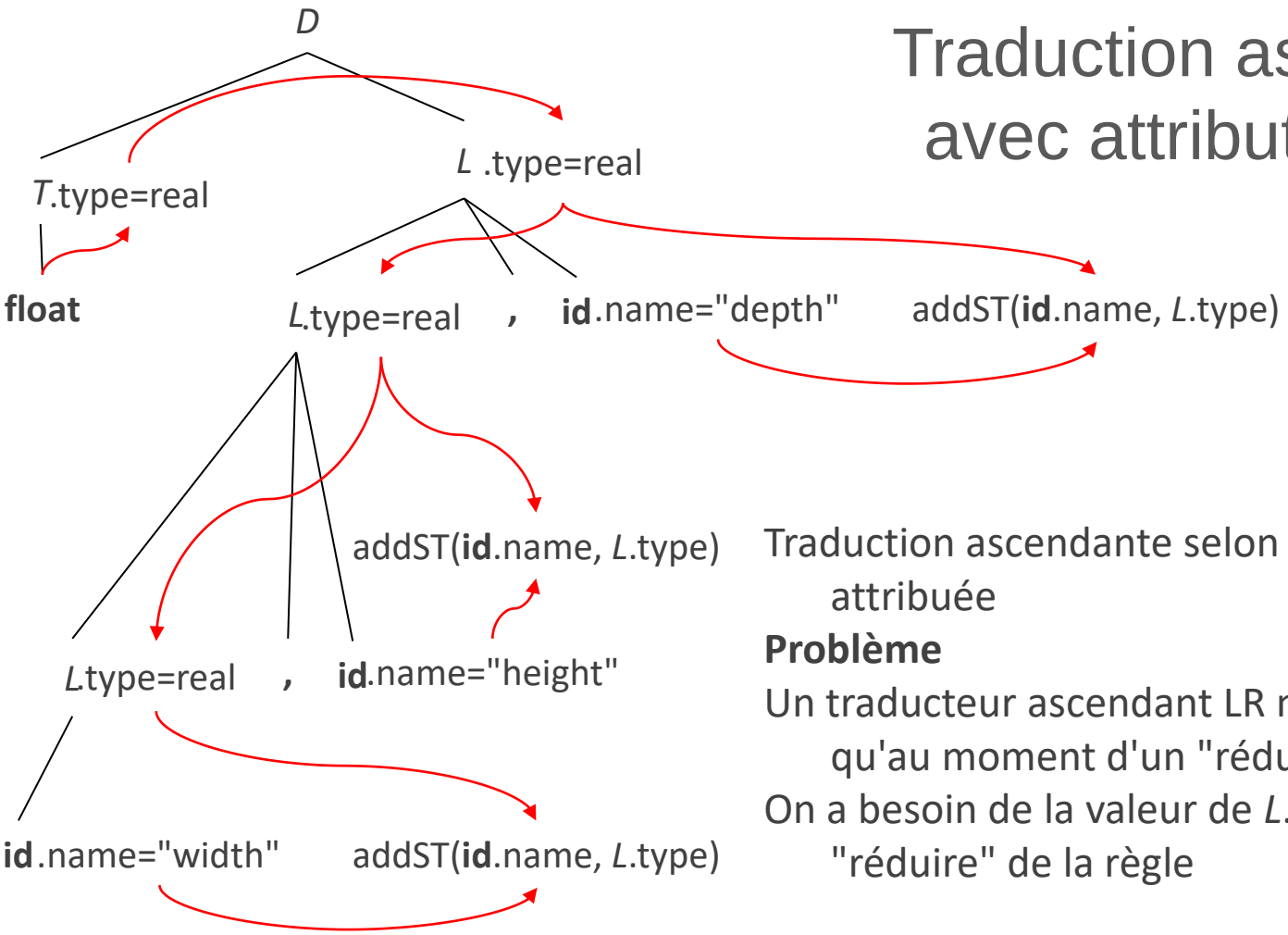
Dans les règles de la grammaire attribuée précédente, ranger dans l'ordre chronologique les actions et les symboles

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Sommaire

Traduction ascendante avec attributs hérités



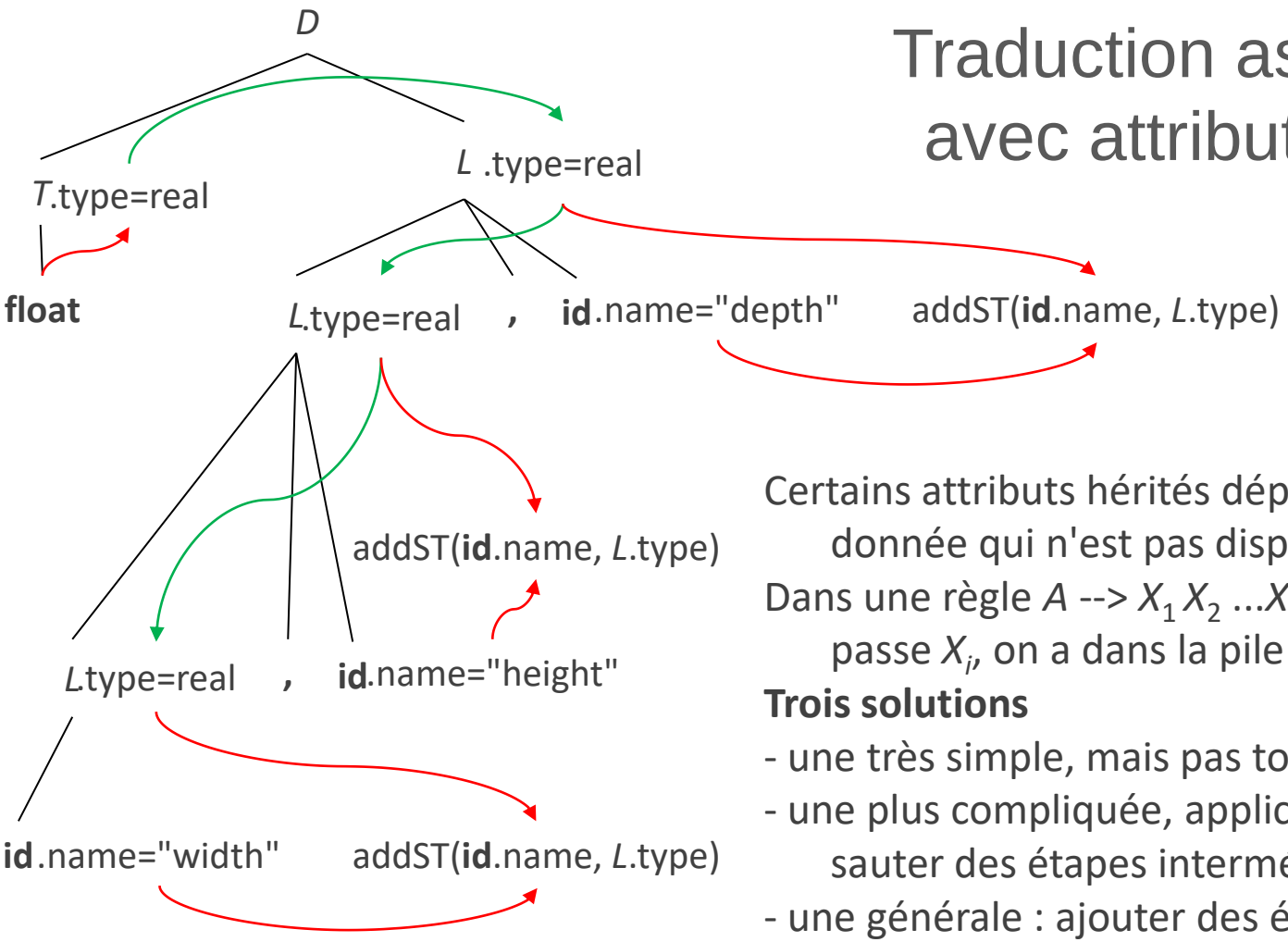
Traduction ascendante selon une grammaire L-attribuée

Problème

Un traducteur ascendant LR ne fait une action qu'au moment d'un "réduire"

On a besoin de la valeur de *L*.type avant l'action "réduire" de la règle

Traduction ascendante avec attributs hérités



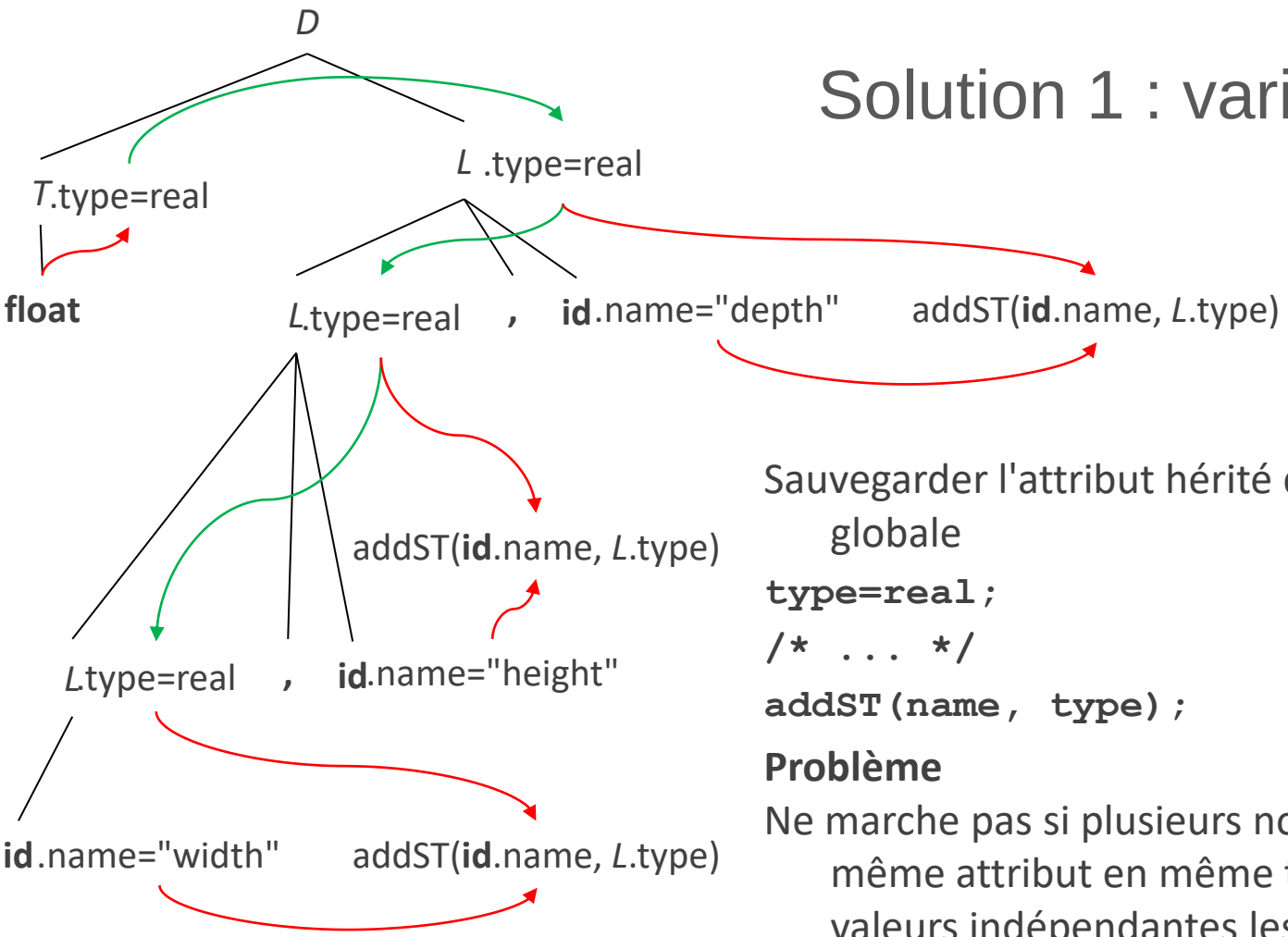
Certains attributs hérités dépendent d'une donnée qui n'est pas disponible à ce moment

Dans une règle $A \rightarrow X_1 X_2 \ldots X_n$, au moment où on passe X_i, on a dans la pile $X_1 X_2 \ldots X_{i-1}$ mais pas A

Trois solutions

- une très simple, mais pas toujours applicable
- une plus compliquée, applicable plus souvent : sauter des étapes intermédiaires
- une générale : ajouter des étapes intermédiaires

Solution 1 : variable globale



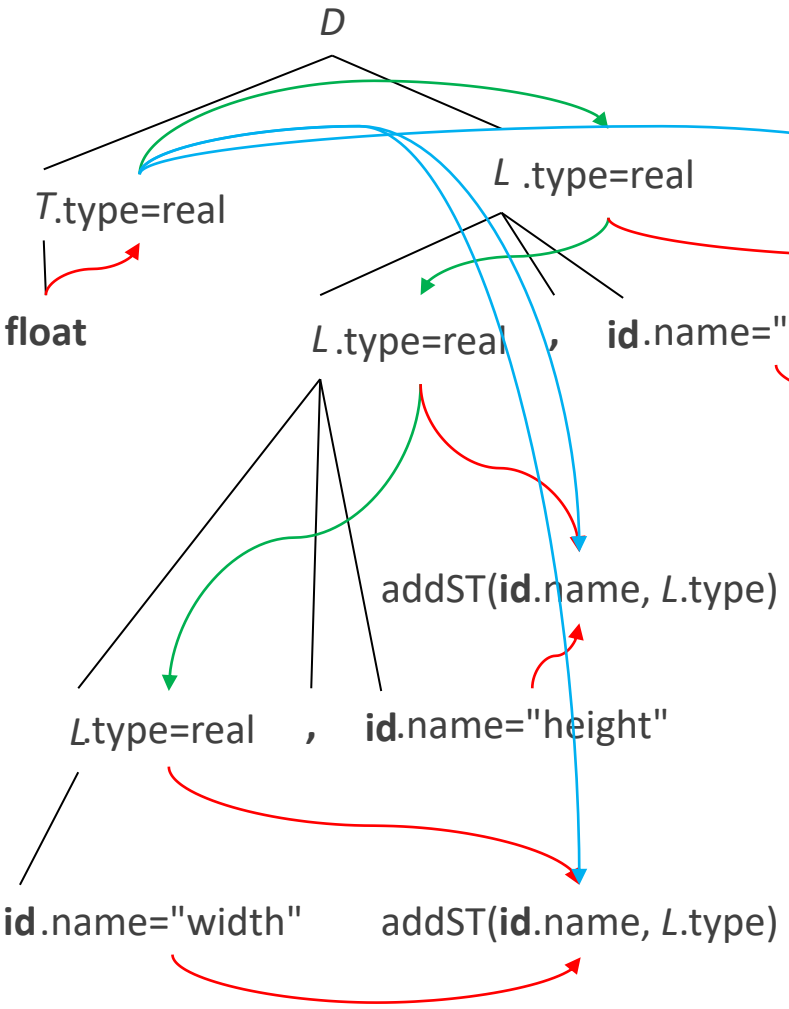
Sauvegarder l'attribut hérité dans une variable globale

```
type=real;
/* ... */
addST(name, type);
```

Problème

Ne marche pas si plusieurs nœuds de l'arbre ont le même attribut en même temps, avec des valeurs indépendantes les unes des autres

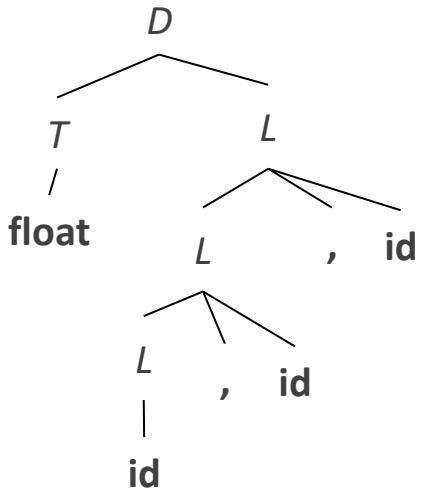
Solution 2 : court-circuiter une chaine de dépendances



- Calculer l'attribut hérité à partir d'un autre attribut qui est ailleurs dans l'arbre
- On ne passe plus par les dépendances intermédiaires
- Il faut savoir dans quelle case de la pile est représenté le nœud où se trouve l'attribut source
- *L*.type ne dépend que de *T*.type

donnée	pile	règle
float p , q , r $	ε	
p , q , r $	**float**	
p , q , r $	*T*	*T* --> **float**
, q , r $	*T* **id**	
, q , r $	*T L*	*L* --> **id**
, r $	*T L* **, id**	
, r $	*T L*	*L* --> *L* **, id**
r $	*T L* **,**	
$	*T L* **, id**	
$	*T L*	*L* --> *L* **, id**
$	*D*	*D* --> *T L*

Trouver un attribut dans la pile

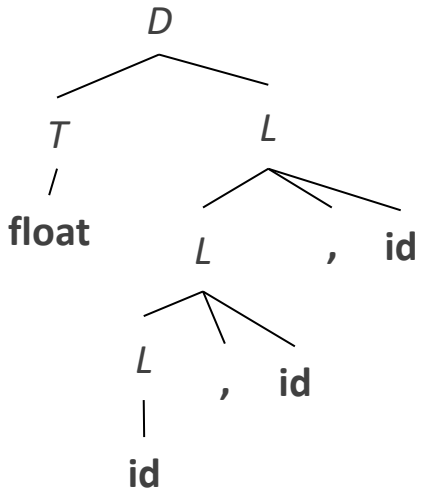


L.type ne dépend que de *T*.type

Le *T* est toujours juste au-dessous du *L* ou du **id** dans la pile

Trouver un attribut dans la pile

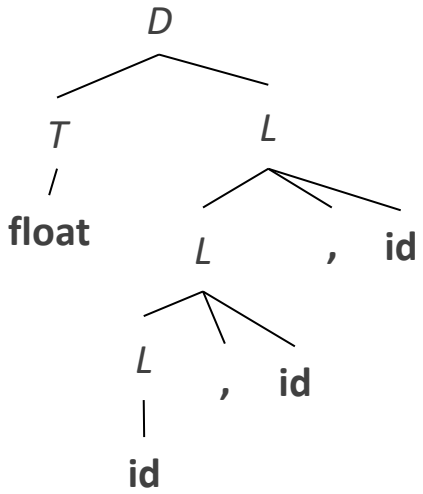
D --> *T L*	*L*.type := *T*.type
T --> **int**	*T*.type := integer
T --> **float**	*T*.type := real
L --> *L* **, id**	L_1.type := *L*.type ;
	addST(**id**.name, *L*.type)
L --> **id**	addST(**id**.name, *L*.type)



Quand on appelle addST(**id**.name, *L*.type), on peut calculer *L*.type à partir de *T*.type qui est déjà dans la pile

Trouver un attribut dans la pile

D --> *T L*	*L*.type := *T*.type
T --> **int**	*T*.type := integer
T --> **float**	*T*.type := real
L --> *L* **, id**	L_1.type := *L*.type ;
	addST(**id**.name, *L*.type)
L --> **id**	addST(**id**.name, *L*.type)



D --> *T L*	/* inutile de recopier ici */
T --> **int**	val[ntop] := integer
T --> **float**	val[ntop] := real
L --> *L* **, id**	addST(val[top], **val[top-3]**)
	/* dans la pile : *T L* **, id** */
L --> **id**	addST(val[top], **val[top-1]**)
	/* dans la pile : *T* **id** */

On accède à une case de la pile au-dessous de la règle en cours

On n'a plus besoin de passer par les dépendances intermédiaires

Trouver un attribut dans la pile

```
D    : T L
     ;
T    : int          { $$ = INTEGER ; }
     ;
T    : float        { $$ = REAL ; }
     ;
L    : L , id       { addST($3, $0) ;
                /* dans la pile : T L , id */ }
     | id           { addST($1, $0) ;
                /* dans la pile : T id */ }
     ;
```

D --> *T L*	*L*.type := *T*.type
T --> **int**	*T*.type := integer
T --> **float**	*T*.type := real
L --> *L* **, id**	L_1.type := *L*.type ; addST(**id**.name, *L*.type)
L --> **id**	addST(**id**.name, *L*.type)

Cette méthode est applicable avec Bison

Pour descendre dans la pile : **$0**, **$-1**, **$-2**...

Si on a déclaré une union pour les types des attributs, déclarer quel champ de l'union correspond à l'attribut : **$<name>0**

Trouver un attribut dans la pile

S --> **d** *A C*	*C*.he := *A*.sy
S --> **e** *A B C*	*C*.he := *A*.sy
C --> **f**	*C*.sy := g(*C*.he)

Problème

Avec certaines grammaires, on ne peut pas prévoir dans quelle case de la pile se trouve l'attribut nécessaire

Exemple

Calcul de *C*.he quand on réduit **f** vers *C*

A.sy peut se trouver en val[top-1] ou en val[top-2]

Solution 3 : insérer une action avant un non-terminal

S --> **d** *A C* *C*.he := *A*.sy
S --> **e** *A B C* *C*.he := *A*.sy
C --> **f** *C*.sy := g(*C*.he)

Insérer une action avant *C*

S --> **d** *A C* *C*.he := *A*.sy
S --> **e** *A B* {*M*.he:=*A*.sy} *C* *C*.he := *M*.he
C --> **f** *C*.sy := g(*C*.he)

Quand on réduit **f** vers *C*, *A*.sy est toujours en val[top-1]

C'est en fait une modification de la grammaire pour pouvoir appliquer la solution 2

S --> **d** *A C* *C*.he := *A*.sy
S --> **e** *A B M C* *M*.he := *A*.sy ; *C*.he := *M*.he
M --> ε
C --> **f** *C*.sy := g(*C*.he)

Solution 3 : algorithme général

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

état	...	*A*			
valeur	...	*A*.a			

Cette méthode est applicable à certaines grammaires L-attribuées dont la grammaire sous-jacente est LR(1)

Solution 3 : algorithme général

état	...	M_1	X	M_2	Y	M_3	Z	
valeur	...	X.he	X.x	Y.he	Y.y	Z.he	Z.z	

Donnée : une grammaire L-attribuée

Résultat : un traducteur ascendant

On suppose que chaque non-terminal A a un attribut hérité A.he et que chaque symbole X a un attribut synthétisé X.sy

Remplacer chaque règle $A \dashrightarrow X_1 X_2 \ldots X_n$ par

$A \dashrightarrow M_1 X_1 M_2 X_2 \ldots M_n X_n$

Mettre les X_i.he comme attributs des M_i

Quand on réduit ε vers M_i, la position de A.he, X_1.he, X_2.he... dans la pile se déduit de la nouvelle grammaire

Actions placées avant le dernier symbole avec Bison

E --> *T E'*

E' --> **+** *T* { print('+') } *E'* | **-** *T* { print('-') } *E'* | ε

T --> **num** { print(**num**.val) }

E --> *T E'*

E' --> **+** *T M E'* | **-** *T N E'* | ε

T --> **num** { print(**num**.val) }

M --> ε { print('+') }

N --> ε { print('-') }

Quand on insère ces actions, Bison

- ajoute de nouveaux nœuds dans l'arbre
- ajoute des non-terminaux ("marqueurs")
- change la grammaire

Cela peut créer des conflits d'analyse LR

Actions placées avant le dernier symbole avec Bison

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

est équivalent à :

```
R    : addop T M R
     |
     ;
M    : { printf($-1) ; }
     ;
```

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 placées avant le dernier symbole avec Bison

```
R    : addop T { printf($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`**

Actions placées avant le dernier symbole avec Bison

```
R    : addop T { $$=f($2) ; } R
     |
     ;
```

est équivalent à :

```
R    : addop T M R
     |
     ;
M    : { $$=f($0) ; }
     ;
```

Dans ces actions,

$1, $2, $3, $4 se réfèrent à la numérotation dans la règle écrite par l'utilisateur

- $1 : addop
- $2 : T
- $3 : M
- $4 : R

$$ se réfère à la règle ajoutée en interne par Bison

- $$: M

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Résumé

Les attributs hérités sont calculés

- en traduction descendante récursive : comme paramètres des fonctions
- en traduction ascendante : avec des variables globales, ou en descendant dans la pile, et s'il le faut en introduisant des actions équivalentes à des non-terminaux supplémentaires ("marqueurs")