Analyse sémantique

Table des symboles, variables, …

Introduction

- Le compilateur effectue un contrôle statique du programme source : syntaxique et sémantique
- Exemple : T tableau de 10 entiers de 1..10, I entier

Contrôle syntaxique :
correct incorrect

Contrôle sémantique :
correct incorrect

Aperçu de l'analyse sémantique

- Contrôle des types
  - Affectation d'une variable d'un certain type avec une expression de type différent
  - Référence à une variable inappropriés :
    - A[I] : erreur si A n'est pas un tableau
    - A + I : erreur si A n'est pas un entier ou un réel
    - I <= 10 : erreur si I n'est pas une variable numérique
- Contrôle d'existence et d'unicité
  - Un identificateur doit être déclaré avant d'être utilisé
  - Ne pas déclarer 2 fois le même identificateur dans le même contexte
  - Gérer la portée des identificateurs

Aperçu de l'analyse sémantique

- Contrôle du flux d'exécution
  - On peut redéclarer en local un identificateur déjà déclaré en global
  - L'instruction doit se trouver à l'intérieur d'une fonction et doit concerner une donnée de type attendu

Traduction dirigée par la syntaxe

A chaque symbole de la grammaire, on associe un ensemble d'attributs et, à chaque production, un ensemble de *règles sémantiques* pour calculer la valeur des attributs associés.

- Chaque symbole (terminal ou non) possède un ensemble d'attributs (i.e. des variables)
- Chaque règle possède un ensemble de règles sémantiques
- Une règle sémantique est une suite d'instructions algorithmiques

La grammaire et l'ensemble des règles sémantiques constituent la traduction dirigée par la syntaxe.

Définitions

On notera X.a l'attribut a du symbole X. S'il y a plusieurs symboles X dans une production, on notera $X^{(0)}$ s'il est en partie gauche, $X^{(1)}$ si c'est le plus à gauche de la partie droite, ..., $X^{(n)}$ si c'est le plus à droite de la partie droite

Exemple

Grammaire S → aSb | aS | cSacS | ε

attributs : nba (calcul du nombre a), nbc (calcul du nombre c)

règles sémantiques

$S \to aSb \quad S^{(0)}.nba := S^{(1)}.nba + 1 \quad S^{(0)}.nbc := S^{(1)}.nbc$

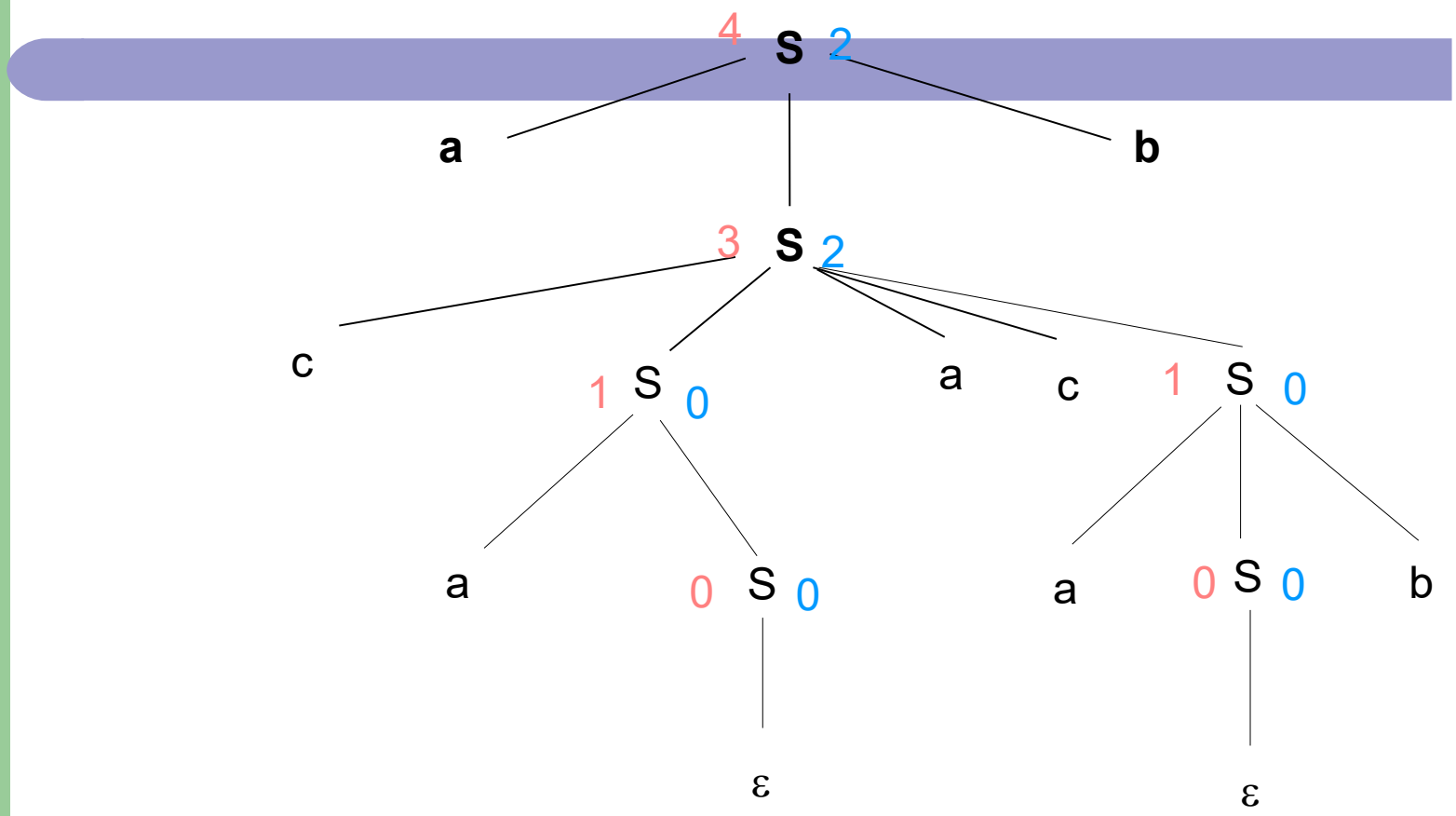
$S \to aS \quad S^{(0)}.nba := S^{(1)}.nba + 1 \quad S^{(0)}.nbc := S^{(1)}.nbc$

$S \to cSacS \quad S^{(0)}.nba := S^{(1)}.nba + 1 + S^{(2)}.nba + 1 \quad S^{(0)}.nbc := S^{(1)}.nbc + S^{(2)}.nbc + 2$

$S \to \epsilon \quad S^{(0)}.nba := 0 \quad S^{(0)}.nbc := 0$

$S' \to S \quad \leq résultat\ final\ est\ dans\ S.nba\ et\ S.nbc$

Calcul de **nba** / **nbc** pour le mot



Sur cet arbre syntaxique, on voit que l'on peut calculer les attributs d'un noeud dès que les attributs de tous ces fils ont été calculés

Remarques

- On appelle *arbre syntaxique décoré* un arbre syntaxique sur les nœuds duquel on rajoute la valeur de chaque attribut
- Dans une DDS (Définition Dirigée par la Syntaxe), il n'y a aucun ordre spécifique imposé pour l'évaluation des attributs. Tout ordre d'évaluation qui calcule sur l'arbre syntaxique l'attribut après tous les attributs dont dépend est possible. Plus généralement, il n'y a aucun ordre spécifique pour l'exécution des actions sémantiques

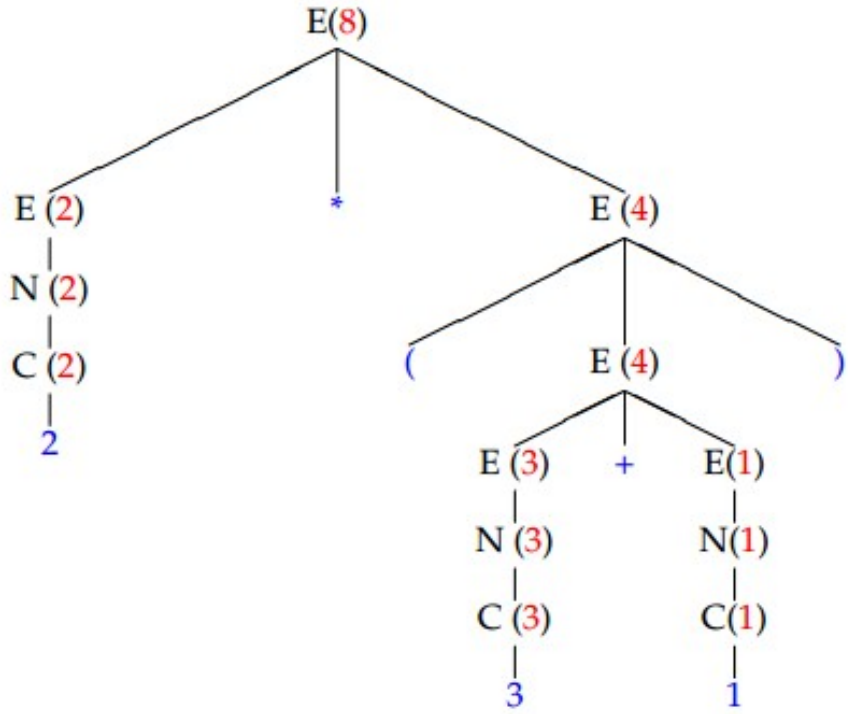
Traduction dirigée par la syntaxe

- Les schémas de traduction sont une notation permettant d'attacher des fragments de programme aux règles de la grammaire
- Les fragments sont exécutés quand la production est exécutées lors de l'analyse syntaxique
- Le résultat combiné des exécutions de ces fragments de code, dans l'ordre induit pas l'analyse syntaxique, produit une traduction dirigée par la syntaxe

Calculette

règle			action sémantique		
E	$\rightarrow$	E + E	$E.t$	=	$E_1.t + E_2.t$
E	$\rightarrow$	E * E	$E.t$	=	$E_1.t * E_2.t$
E	$\rightarrow$	(E)	$E.t$	=	$E_1.t$
E	$\rightarrow$	N	$E.t$	=	$N.t$
N	$\rightarrow$	C N	$N.t$	=	$10 * C.t + N_1.t$
N	$\rightarrow$	C	$N.t$	=	$C.t$
C	$\rightarrow$	0	$C.t$	=	0
C	$\rightarrow$	1	$C.t$	=	1
C	$\rightarrow$	2	$C.t$	=	2
	...			...	
C	$\rightarrow$	9	$C.t$	=	9

calculette



Attributs synthétisés

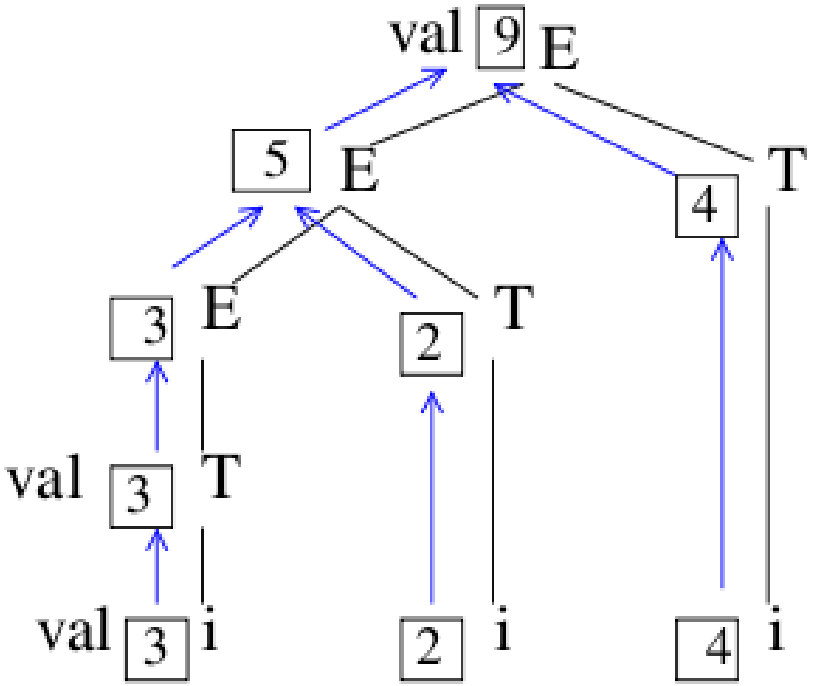
- Un attribut est *synthétisé* si sa valeur, à un nœud d'un arbre syntaxique, est déterminée à partir de valeurs d'attributs des fils de ce nœud. L'attribut de la partie gauche est calculé en fonction des attributs de la partie droite. Le calcul des attributs se fait des feuilles vers la racine.
- Les attributs synthétisés peuvent être donc évalués au cours d'un parcours ascendant de l'arbre de dérivation
- Dans l'exemple page suivante, les attributs sont synthétisés

Exemple

Règle	Action sémantique
$E \rightarrow E+T$	$E^{(0)}.val := E^{(1)}.val + T.val$
$E \rightarrow T$	$E.val := T.val$
$T \rightarrow (E)$	$T.val := E.val$
$T \rightarrow i$	$T.val := i.val$

On a affaire à des attributs synthétisés.

Visualisation



Attributs hérités

- Un attribut *hérité* est un attribut dont la valeur à un nœud d'un arbre syntaxique est définie en terme des attributs du père et/ou des frères de ce nœud C'est-à-dire le calcul est effectué à partir du non terminal de la partie gauche et éventuellement d'autres non terminaux de la partie droite
- Si les attributs d'un nœud donné ne dépendent pas des attributs de ses frères droits, alors les attributs hérités peuvent être facilement évalués lors d'une analyse descendante

Grammaire

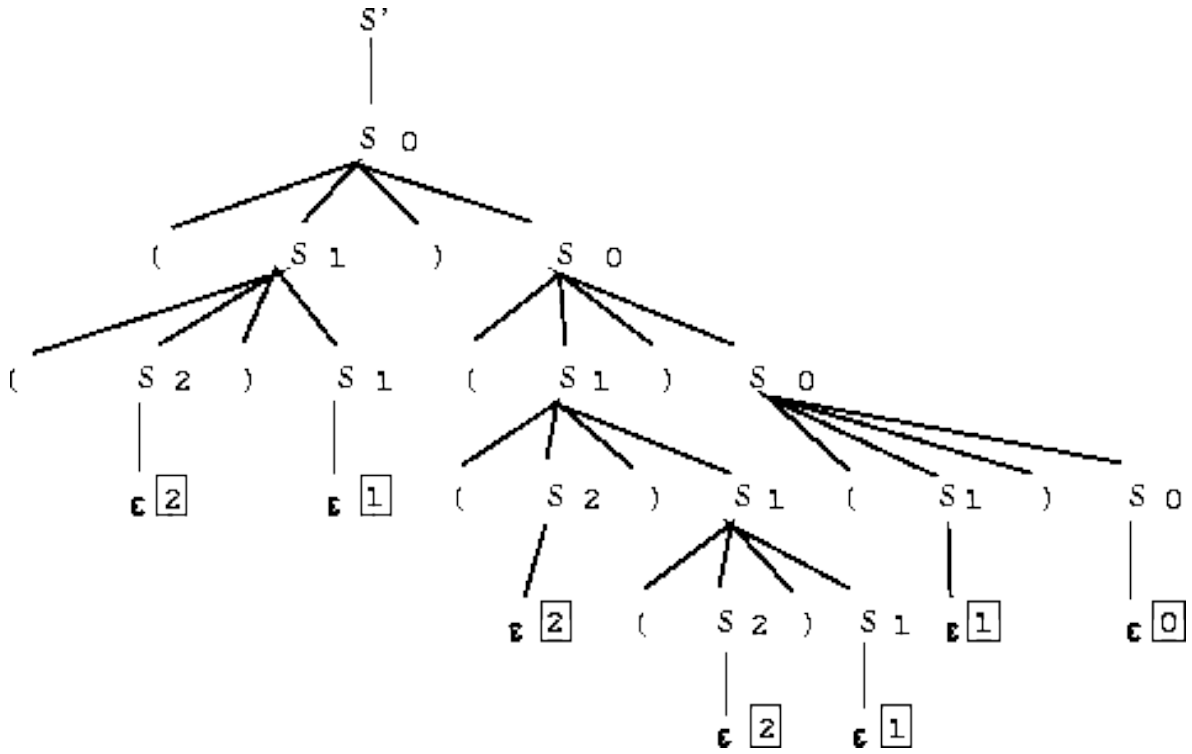
- Grammaire

$$\begin{cases} S' \rightarrow S \\ S \rightarrow (S)S \mid \epsilon \end{cases}$$

- Règles — Action sémantique

Règles	Action sémantique
S'→S	S.nb := 0
S →(S)S	$S^{(1)}$.nb := $S^{(0)}$.nb+1
	$S^{(2)}$.nb := $S^{(0)}$.nb
S → ε	écrire S.nb

Attributs hérités



arbre décoré pour le mot (( )) (( ) ( )) ()

Niveau d'imbrication des) dans un système de parenthèses

Attributs synthétisés

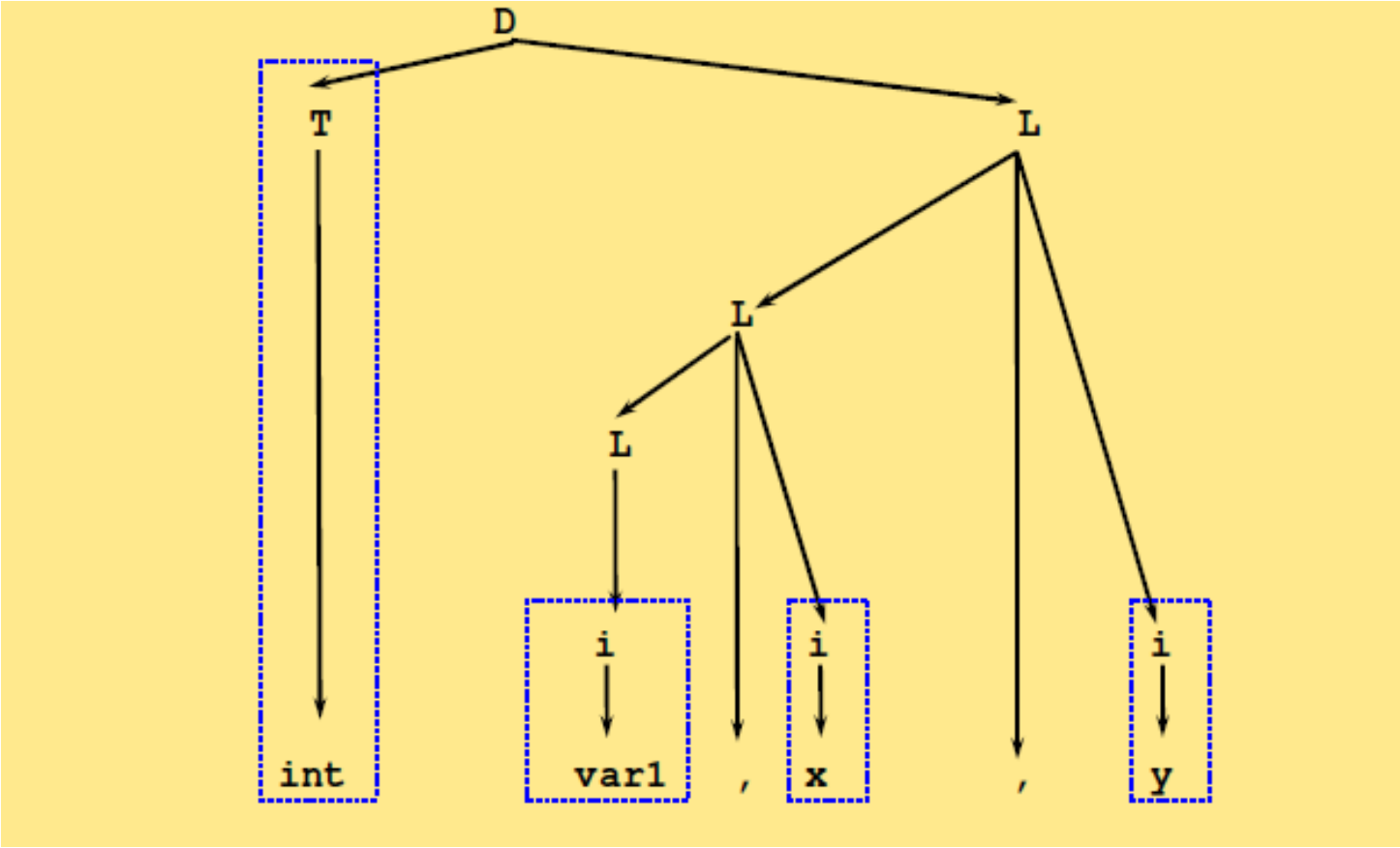
- Soit les règles suivantes :

$$
\begin{cases}
D \rightarrow T\ L \quad \{D.type = T.type \quad L.type = T.type\} \\
T \rightarrow \text{int} \quad \{T.type = \text{int}\} \\
T \rightarrow real \quad \{T.type = \text{real}\} \\
L \rightarrow L, i \quad \{L^1.Type = L^0.Type\} \\
\{addSymbTable(i.token, L^{(1)}.type)\} \\
L \rightarrow i \quad \{i.Type = L^0.Type\} \\
\{addSymbTable(i.token, L.type)\}
\end{cases}
$$

- exemple int A, B
  - D représente la début de la déclaration
  - i représente les identificateurs
  - T représente le type des variable
  - L la liste des variables séparées par une virgule

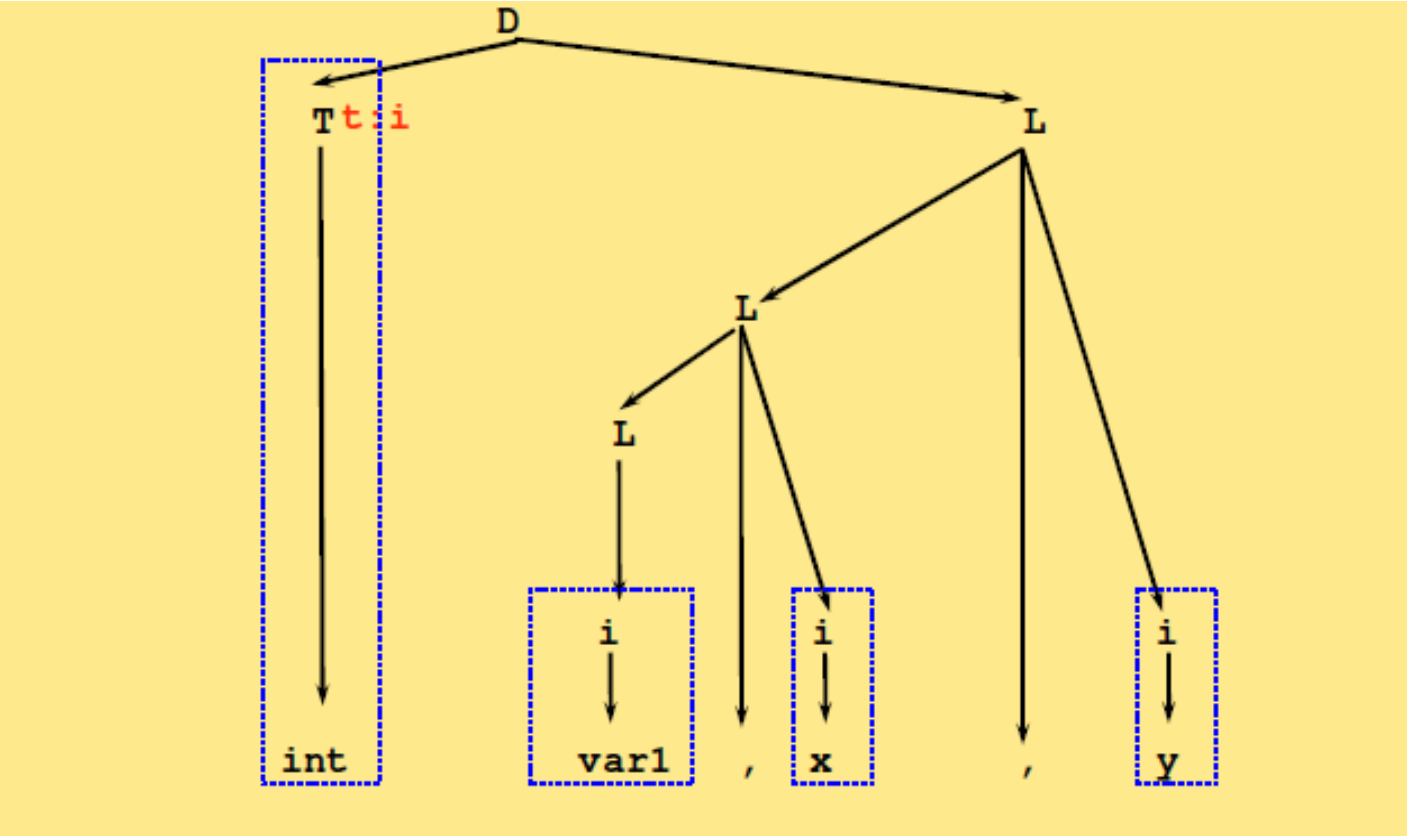
Exemple : analyse syntaxique

- int var1, x, y



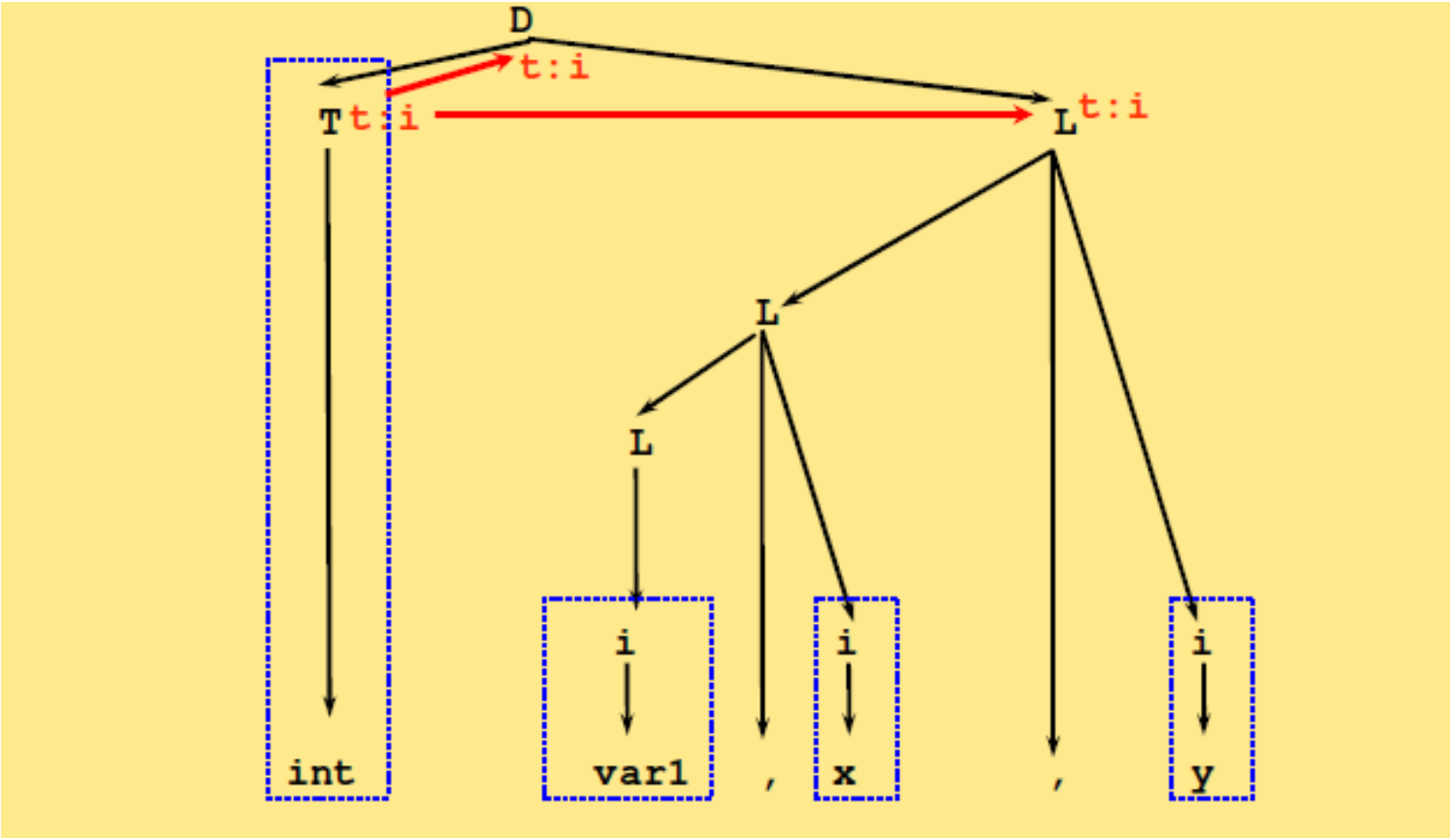
Exemple : analyse sémantique

- int var1, x, y



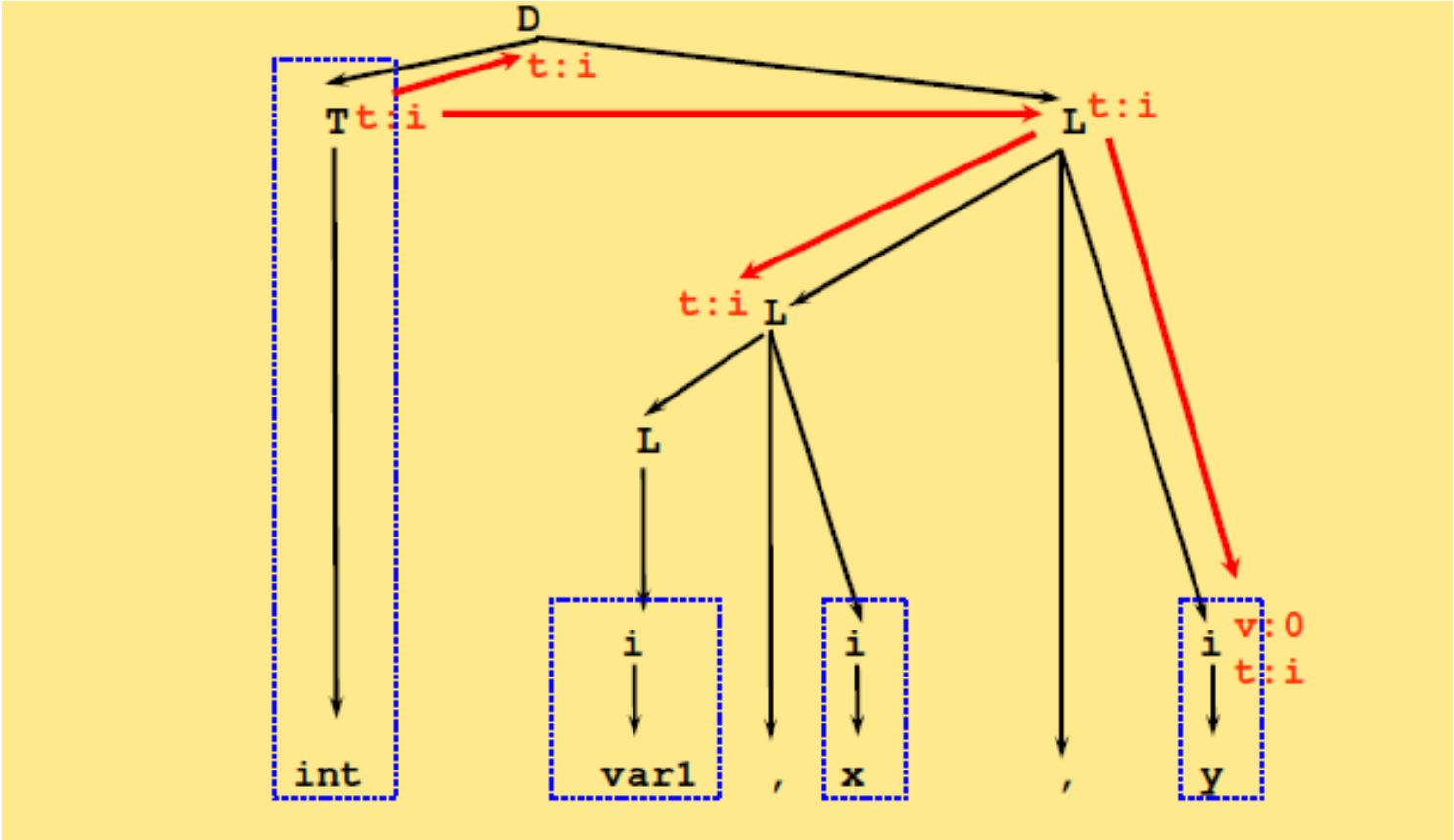
Exemple : analyse sémantique

- int var1, x, y



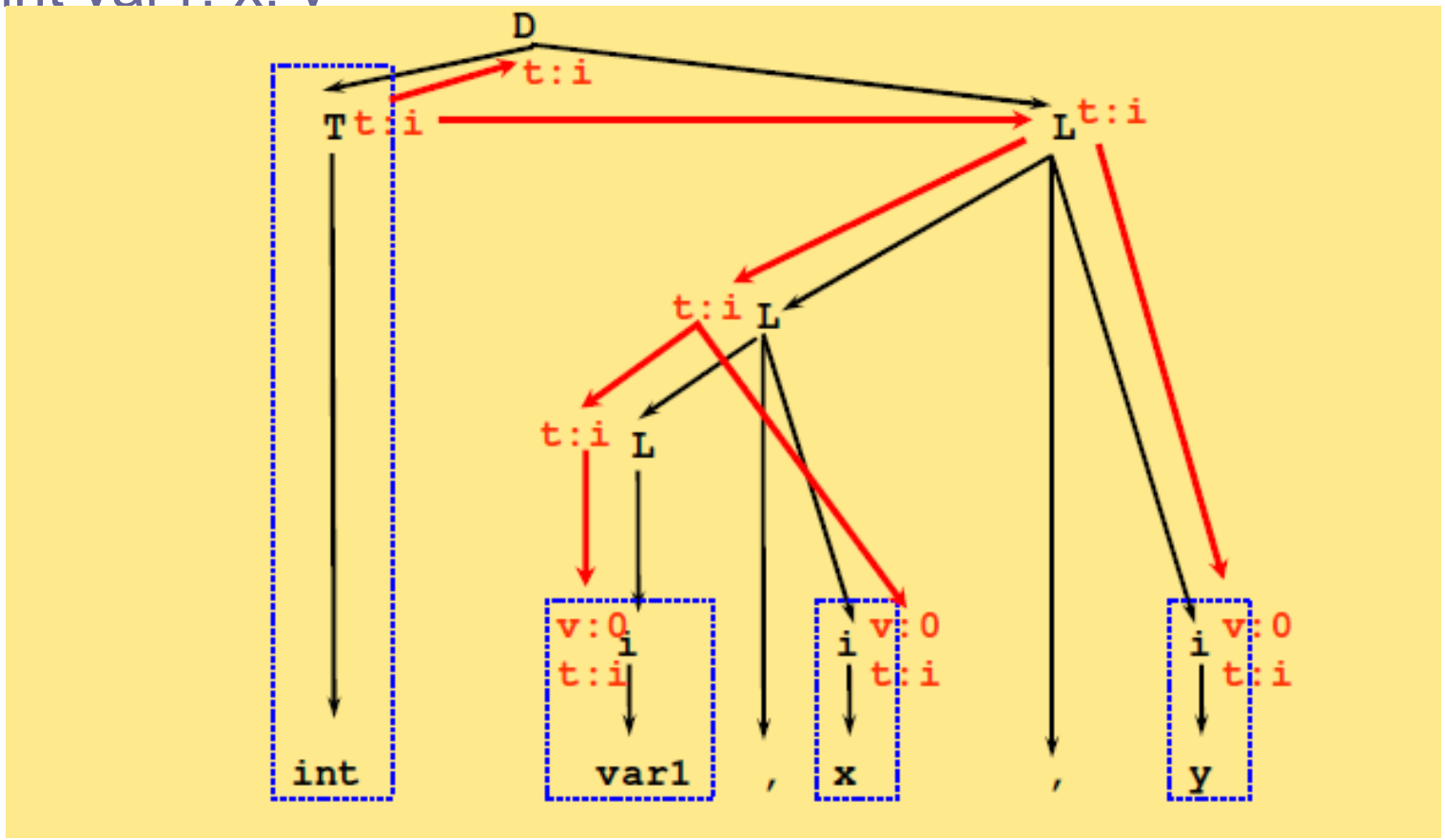
Exemple : analyse sémantique

- int var1, x, y



Exemple : analyse sémantique

- int var1, x, y

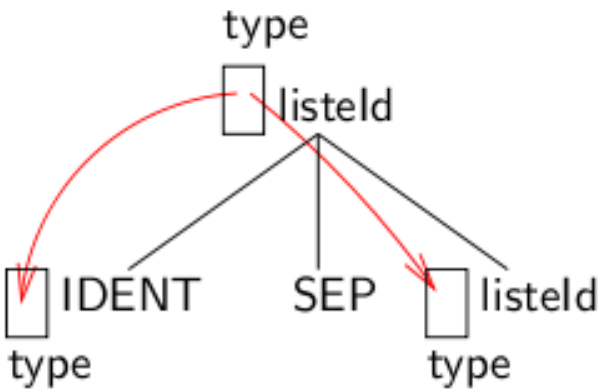


Ordre d'évaluation

- Dans quel ordre évaluer les attributs ?
  - les actions induisent des dépendances de données ;
  - construction d'un graphe de dépendances.

$$listeId \rightarrow \text{IDENT SEP } listeId \quad \{\ \text{IDENT}.type = listeId_0.type \quad listeId_1.type = listeId_0.type\ \}$$

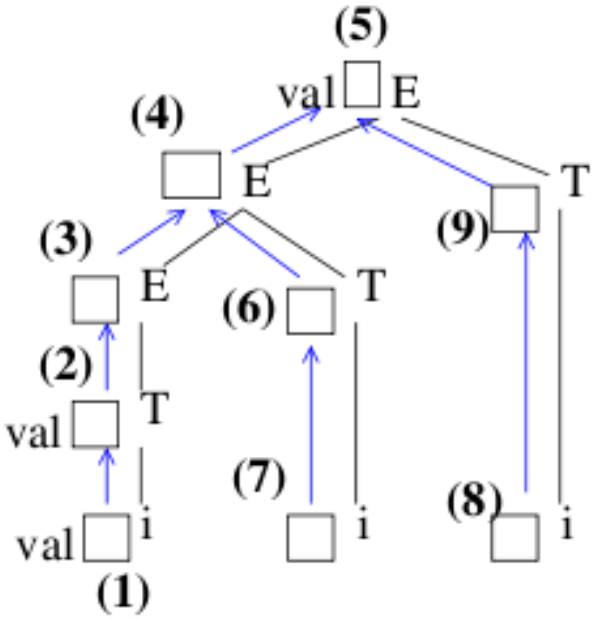
$$exprAffect \rightarrow \texttt{IDENT} \quad \{\ exprAffect.type = \texttt{IDENT.type}\ \}$$



Ordre d'évaluation

- Si ce graphe est cyclique : grammaire mal formée.
- sinon (grammaire bien formée) : on peut trouver un ordre d'évaluation, une numérotation des actions

Ordre d'évaluation et graphe de dépendances



$E \rightarrow E + T \quad \{ E_0.val = E_1.val + T.val \}$

$E \rightarrow T \quad \{ E.val = T.val \}$

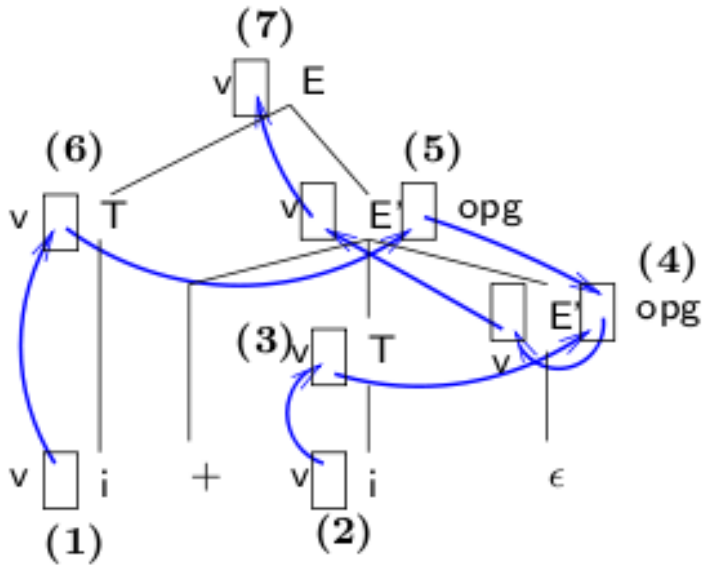
$T \rightarrow (E) \quad \{ T.val = E.val \}$

$T \rightarrow \mathtt{i} \quad \{ T.val = \mathtt{i}.val \}$

Ex 1 : ((**1**), val) < ((**7**), val) < ((**8**), val) < ((**2**), val) < ((**6**), val) < ((**9**), val) < ((**3**), val) < ((**4**), val) < ((**5**), val)

Ex 2 : ((**1**), val) < ((**2**), val) < ((**3**), val) < ((**7**), val) < ((**6**), val) < ((**8**), val) < ((**9**), val) < ((**4**), val) < ((**5**), val)

Autre exemple



$E \to TE'$ { $E'.\text{opg} = \text{f}(T.\text{v})$ $E.\text{v} = \text{f}(E'.\text{v})$ }

$E' \to +TE'$ { $E'_1.\text{opg} = \text{f}(E'_0.\text{opg},\ T.\text{v})$ $E'_0.\text{v} = \text{f}(E'_1.\text{v})$ }

$E' \to \epsilon$ { $E'.\text{v} = \text{f}(E'.\text{opg})$ }

$T \to \texttt{i}$ { $T.\text{v} = \text{f}(\texttt{i}.\text{v})$ }

Ex : ((**1**), v) < ((**6**), v) < ((**5**), opg) < ((**2**), v) < ((**3**), v) < ((**4**), opg) < ((**4**), v) < ((**5**), v) < ((**7**), v)

Ordre d'évaluation

Une fois le graphe de dépendance donné, un tri topologique (si $m_i \rightarrow m_j$ est un arc de m_i à m_j, alors m_i doit apparaître avant m_j) du graphe permet d'obtenir l'ordre d'évaluation des règles sémantiques.

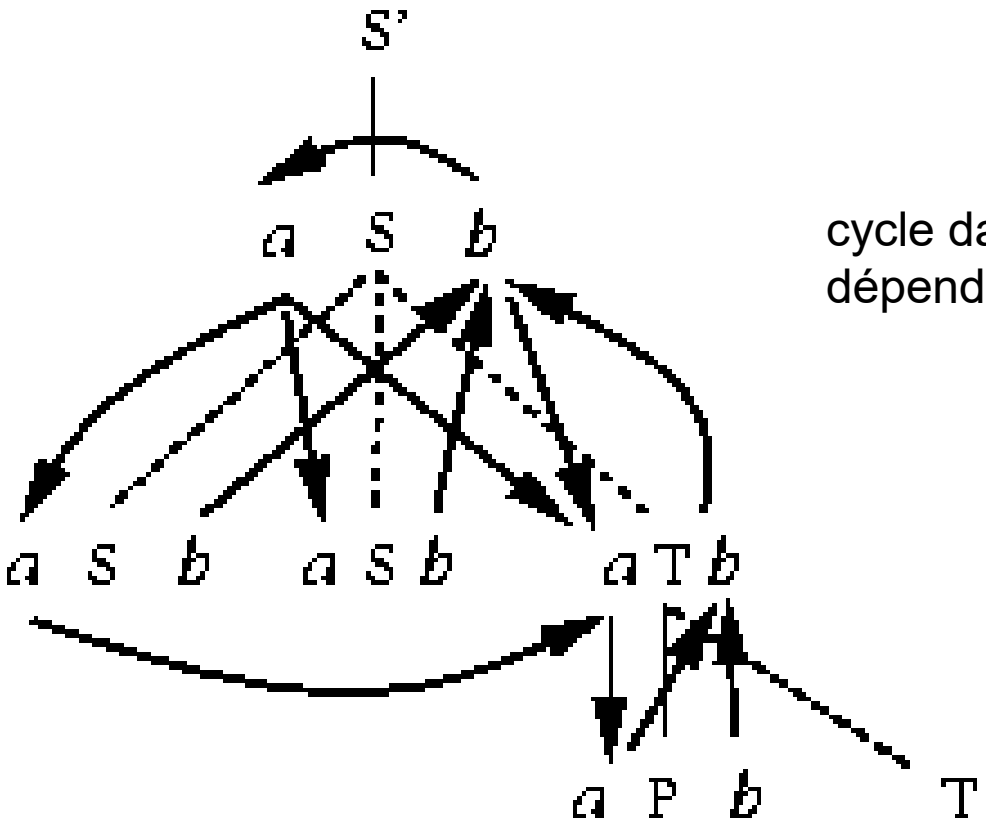
Des problèmes peuvent apparaître s'il existe des cycles dans le graphe de dépendance.

Ordre d'évaluation (suite)

Étudier les méthodes d'évaluation des graphes de dépendance des règle sémantiques dépassent le niveau de ce cours. Les techniques font appel à des techniques complexes :

- construction d'arbres abstraits
- les graphes orientés acycliques
- …

Graphe de dépendance

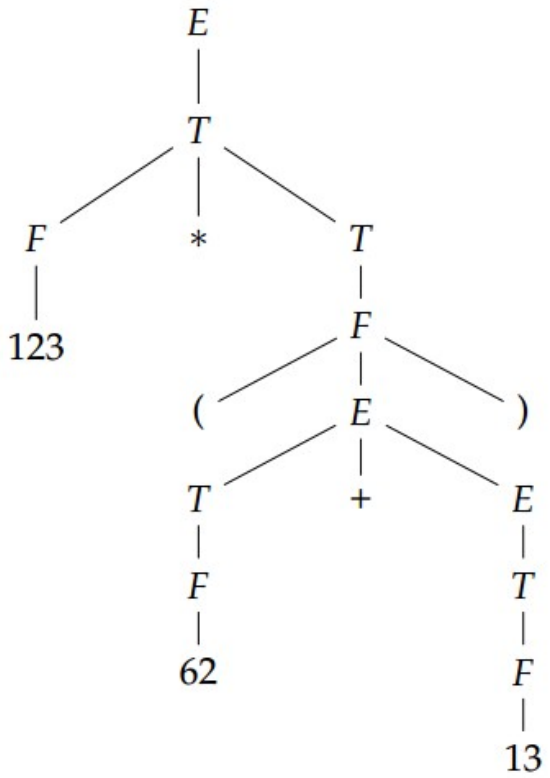


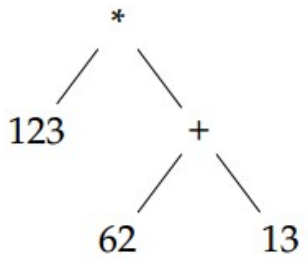
cycle dans le graphe de dépendance

Construction des arbres abstraits

- La construction d'arbres abstraits permet de séparer l'étape de traduction (génération de code) de l'analyse syntaxique
- L'ordre de construction de l'arbre syntaxique (pendant l'analyse syntaxique) peut être différent de l'ordre d'évaluation de l'arbre abstrait pour générer le code

Arbre de dérivation / arbre abstrait





Synthèse provisoire

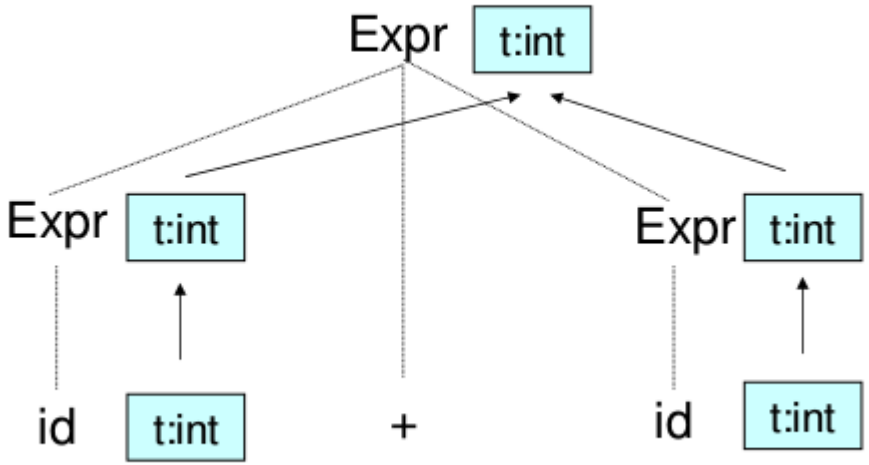
- Nous avons vu que le calcul des attributs peut être très complexe avec éventuellement plusieurs passes
- Nous allons nous intéresser à des attributs sémantiques et à des grammaires plus simples
  - les grammaires S-attribués
  - les grammaires L-attribués

Grammaire S-attribuée

- Une grammaire est S-attribuée si elle utilise exclusivement des attributs synthétisés
- Très pratique lors d'une analyse ascendante, les enfants sont connus avant le père. Les valeurs des attributs sont passés pendant les réductions

Exemple

$$Expr \rightarrow id \quad \{Expr\,.\,Type = id\,.\,Type\}$$
$$Expr \rightarrow Expr_1 + Expr_2 \quad \{Expr^{(0)}\,.\,Type = Expr^{(1)}\,.\,Type\}$$



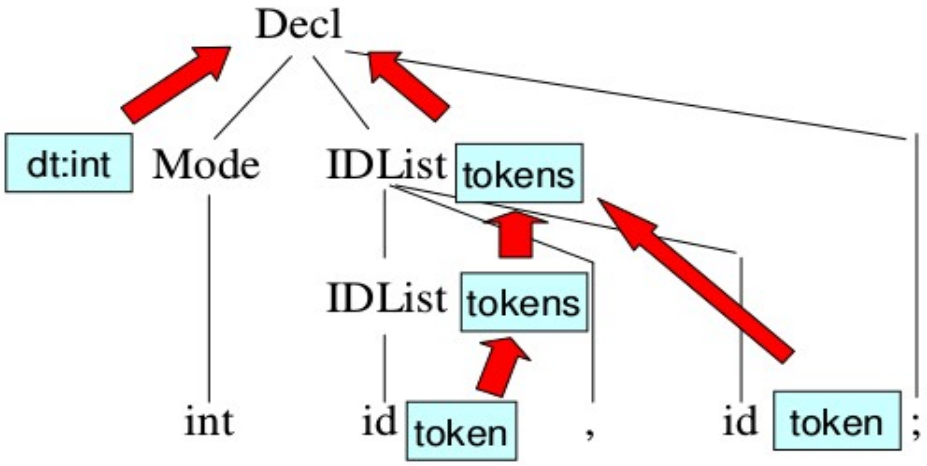
Contraintes des grammaires S-attribuées

- Il est possible d'écrire des grammaires de langage de programmation avec des attributs uniquement synthétisés
- Mais cela entraîne des réécritures de grammaire

$$
\begin{cases}
D \rightarrow T\ L\ ; \quad \{L.type = T.type\} \\
T \rightarrow \text{int} \quad \{T.type = \text{int}\} \\
T \rightarrow real \quad \{T.type = \text{real}\} \\
L \rightarrow L, i \quad \{L^1.Type = L^0.Type\} \\
\{addSymbTable(i.token, L^{(1)}.type)\} \\
L \rightarrow i \quad \{i.Type = L^0.Type\} \\
\{addSymbTable(i.token, L.type)\}
\end{cases}
$$

Réécriture plus lourde et moins naturelle

$$\begin{cases} D \rightarrow T\ L\ ; & \{addAllToSymTable(L.tokens, T.type)\} \\ T \rightarrow \text{int} & \{T.type = \text{int}\} \\ T \rightarrow real & \{T.type = \text{real}\} \\ L \rightarrow L, i & \{L^{(0)}.tokens = ajoute(L^{(1)}.tokens, i.token)\} \end{cases}$$



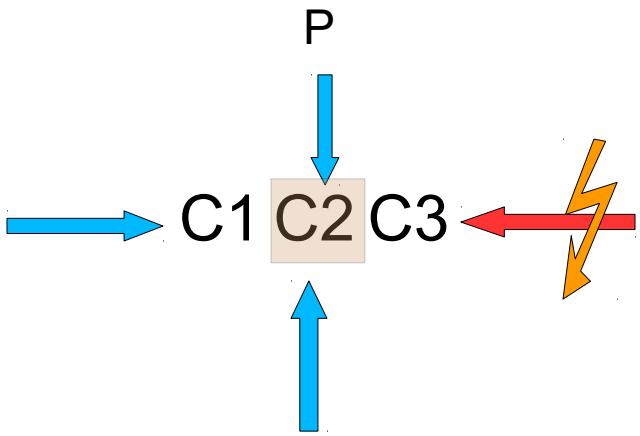
Synthèse

- Les grammaires S-attribuées permettent d'effectuer une analyse sémantique lors d'une analyse ascendante. On fait passer les attributs des enfants aux parents
- Elles nécessitent cependant souvent une réécriture des actions sémantiques pour éviter l'héritage

Les grammaires L-attribuées

- Classe plus large que les grammaires S-attribuées. Elle permet d'avoir des attributs synthétisés et sous certaines contraintes des attributs hérités
- Étant donné une règle $P \rightarrow C_1 C_2 ... C_n$
- Tout C_k peut avoir des attributs hérités
  - les attributs de ses frères à sa gauche $C_1 C_2 ... C_{k-1}$
  - ou hérités de son père P
- Le flux de l'information est dirigé de gauche à droite

Flux d'information dans une grammaire L-attribuée



Grammaire L-attribuée et descente récursive

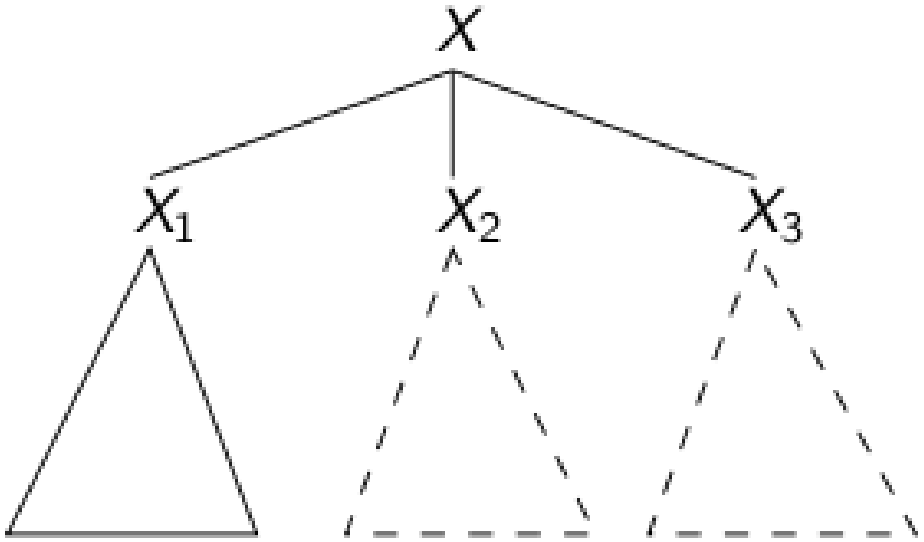
1) Démarrage au nœud sommet (axiome)
2) Lorsqu'on arrive à un nœud, on exécuté toutes les actions sémantiques demandant des attributs hérités
3) La procédure 2) est appelée récursivement sur chaque nœud fils
4) Au retour, on exécute toutes les actions sémantiques demandant des attributs synthétisés

- Une seule passe est donc nécessaire pour les grammaires L-attribués

Analyse descendante

- Analyse descendante : parcours de l'arbre en ordre préfixé en profondeur d'abord :
  père puis fils gauche à droite, récursivement

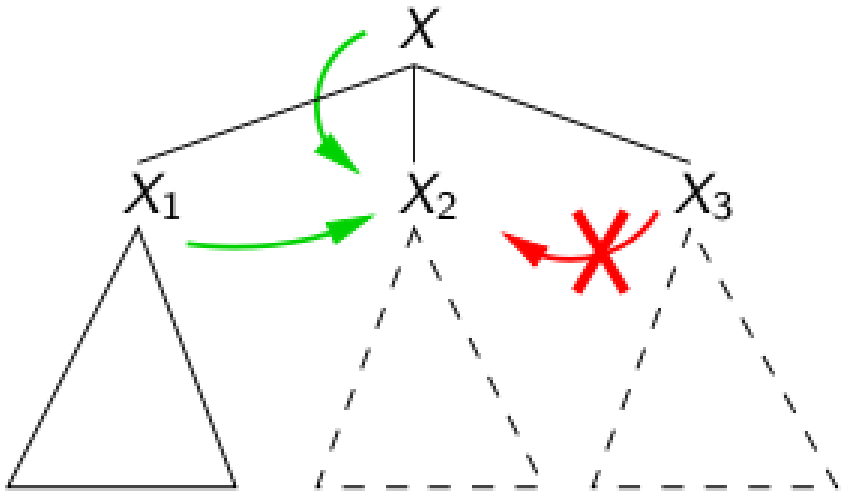
$X \rightarrow X_1 X_2 X_3$ et la pile $\begin{array}{|c|}\hline X_2 \\ X_3 \\ \hline \end{array}$



Restriction

- X_2 doit par définition :
  - avoir à sa disposition ses attributs hérités ;
  - calculer ses attributs synthétisés.

Les attributs hérités de X_2 ne peuvent venir de ses frères droits.



Analyseur récursif

Une méthode X( ) par non-terminal X.

Les valeurs des attributs :

- hérités de X sont disponibles pour X avant sa reconnaissance ;
- synthétisés de X sont calculées après que X a été reconnu.

- Donc :
  - les attributs hérités Her(X) dont des entrées de X ;
  - les attributs synthétisés Synth(X) sont des sorties de X.

En Java : SynthX x(h1x, ..., hnX)
avec SynthX type objet regroupant les attributs de Synth(X).

Implantation dans un analyseur récursif

- pour la règle X → AbC :

X → { calcul Her(A) } A { calcul Synth(A) }
b { calcul Synth(b) }
{ calcul Her(C) } C { calcul Synth(C) }
{ calcul Synth(X) }

grammaires L-attribuées et analyse ascendante

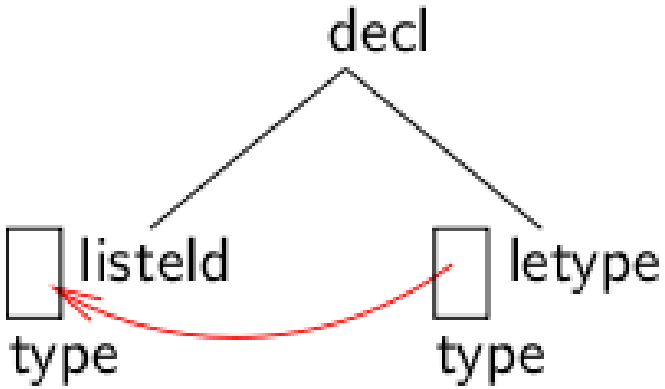
- Dans une analyse ascendante, les grammaires L-attribuées sont un problème
  - les attributs synthétisés ne sont par essence pas un problème
  - les attributs hérités des attributs synthétisés des frères ne sont pas un problème non plus parce que ces enfants ont déjà été construits
  - cependant les parents sont crées **après** tous les enfants, les attributs hérités du père ne sont donc pas disponibles pour ces enfants
  - On peut s'en sortir en général avec une réécriture
  - c'est ce que fait bison ou cup (grammaires LALR(1))

Exemple de grammaire L-attribuée

$decl \rightarrow$ INT $listeId$ PV	$\{\ listeId.type = \texttt{Type.ENTIER}\}$
$decl \rightarrow$ LIST $listeId$ PV	$\{\ listeId.type = \texttt{Type.LISTE}\}$
$listeId \rightarrow$ IDENT	$\{\ \text{IDENT}.type = listeId.type\ \}$
$listeId \rightarrow$ IDENT SEP $listeId$	$\{\ \text{IDENT}.type = listeId_0.type$
	$listeId_1.type = listeId_0.type\ \}$
...	
$affect \rightarrow$ IDENT AFF $exprAffect$	$\{$ verif type($exprAffect.type$,
	IDENT.$type$) $\}$
$exprAffect \rightarrow liste$	$\{\ exprAffect.type = \texttt{Type.LISTE}\ \}$
$exprAffect \rightarrow$ IDENT	$\{\ exprAffect.type = \text{IDENT}.type\ \}$

Exemple de grammaire non L-attribuée

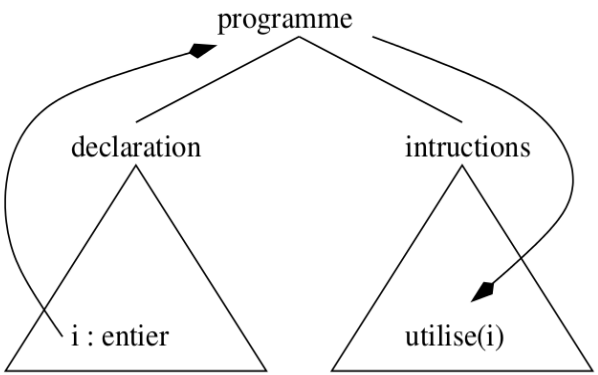
$$
\begin{array}{ll}
\mathit{decl} \rightarrow \mathit{listeId}\ \mathit{letype} & \{\ \mathit{listeId.type} = \mathit{letype.type}\ \} \\
\mathit{letype} \rightarrow \mathrm{INT} & \{\ \mathit{letype.type} = \mathrm{Type.ENTIER}\ \} \\
\mathit{letype} \rightarrow \mathrm{LIST} & \{\ \mathit{letype.type} = \mathrm{Type.LISTE}\ \} \\
\mathit{listeId} \rightarrow \ldots &
\end{array}
$$



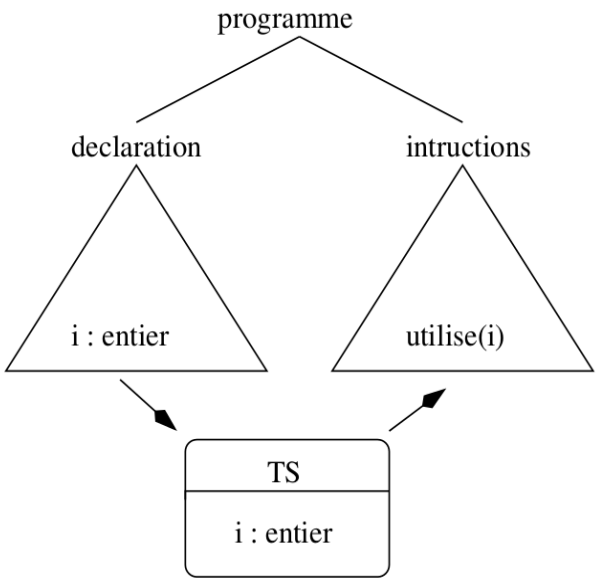
En pratique

- l'utilisation de la table des symboles permet de résoudre la plupart des problèmes rencontrés ;
- les attributs synthétisés sont très utilisés ;
- les attributs hérités sont utiles pour les déclarations de variables.

Héritage :



Utilisation d'une table des symboles :



Résumé

- Attributs synthétisés bien adaptés pour une analyse ascendante et descendante récursive
- Attributs hérités bien adaptés pour une analyse descendante
- Question : comment faire avec des attributs synthétisés **et** hérités ?
  - on peut souvent utiliser des attributs synthétisés qui font la même chose que les attributs hérités qu'on voulait

Attribut synthétisé

- But : calculer la valeur de chaque expression
- Calcul d'un attribut synthétisé val sur la grammaire
- Définition de val :
  - Règle expr ::= INT
    val(expr) = val(INT)
  - Règle expr ::= expr(1) PLUS expr(2)
    val(expr) = val(expr1) + val(expr2)
- Val est synthétisé : la valeur sur le membre gauche d'une règle est fonction des valeurs sur les membres de droite

Exemple de grammaire

```
/* attribut.cup */

import java_cup.runtime.*;

   ;

terminal Integer INT;
terminal PLUS, MOINS, FOIS, DIV, PARENG,
PAREND;

non terminal list_expr;
non terminal Integer expr;

expr        ::= expr:e1 PLUS expr:e2
   {: RESULT = new Integer(e1.intValue() +
e2.intValue()); :}
   |
   INT:n
   {: RESULT = new Integer(n.intValue()); :}
   |expr:e1 FOIS expr:e2
   {: RESULT = new
Integer(e1.intValue()*e2.intValue()); :}
   ;
```

Attention – fichier attribut.flex

```
%{
// fonction pour manipuler des tokens avec ligne, colonne, nombre

%}

// modèles
chiffre = [0-9]
entier = {chiffre}+
espace = [ \t\n]

%%
// Règles
{entier}        { return new Symbol (sym.INT, new Integer(yytext())); }
\+              {
                return new Symbol (sym.PLUS); }
```