

COMPILATION

La compilation

- Un **compilateur** est un programme informatique qui traduit un langage, le *langage source*, en un autre, appelé le *langage cible*, en préservant la sémantique du texte source
- Traduction d'un langage en instructions machines
 - Ex : C / C++
- Traduction d'un langage de haut niveau vers un autre
 - Ex : Traduction de Pascal en C
- Traduction d'un langage quelconque vers un langage quelconque
 - Ex : Word vers html, pdf vers ps
- *Rq: Un langage décrit une information structurée*
 - Utile dans **tous** les domaines

Compilation : recontre GL / OS

- Langage de programmation
 - impératifs génie logiciel...
 - ...lisibilité
 - ...maintenabilité
 - ...orientation métier
- Langage d'exécution – impératifs système d'exploitation...
 - ...gestion des ressources

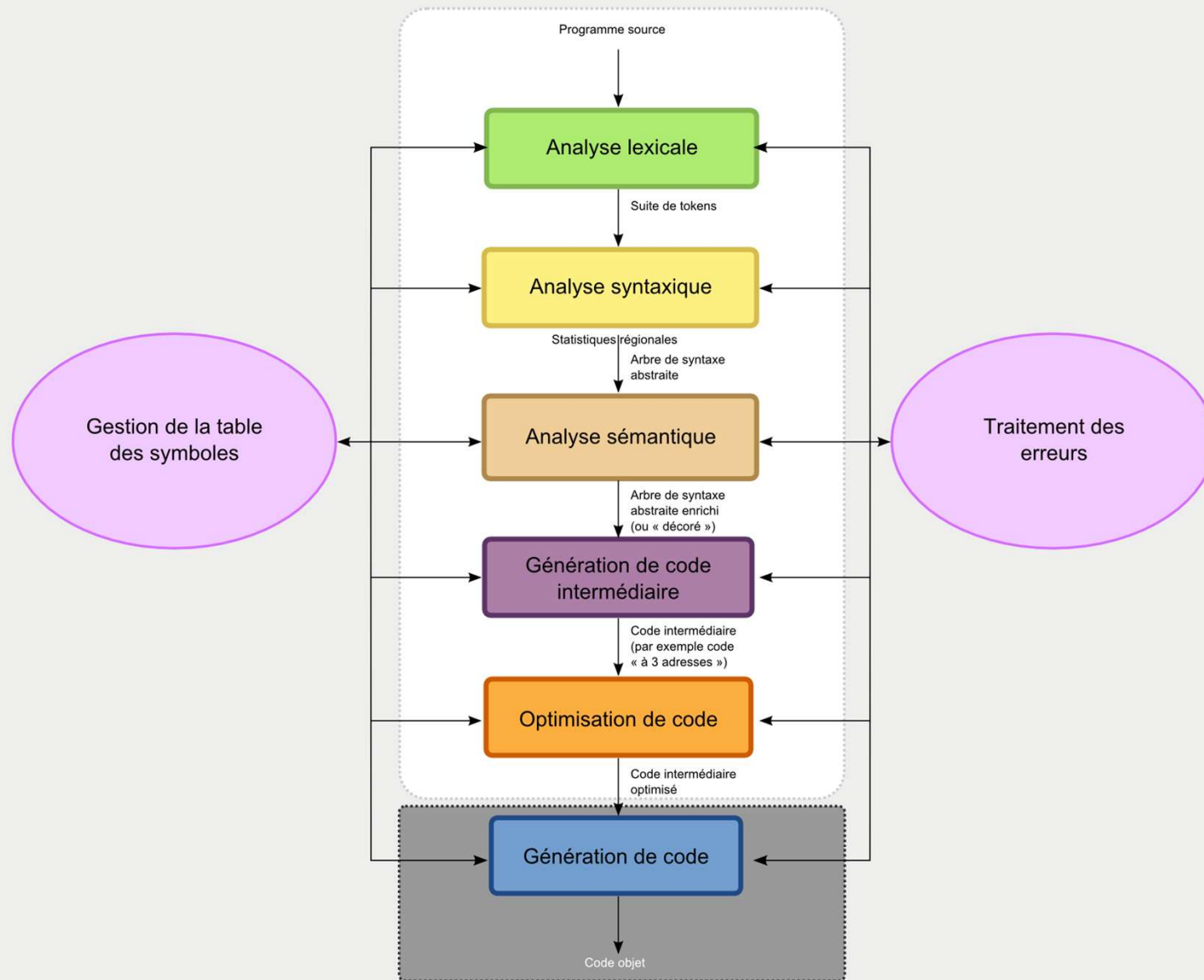
GL vs OS

GL	OS
Structures de données, tableaux, objets...	Mémoire plate, hiérarchie de mémoire (registres, caches, mémoire vive, swap)
Structures de contrôle, boucles, itérateurs...	goto L, ifz R goto L, pipe-line, branchement spéculatif...
Types de données, classes, héritage	1, 8, 16, 32, 64 bits...
Introspection du code	Programme ~ données

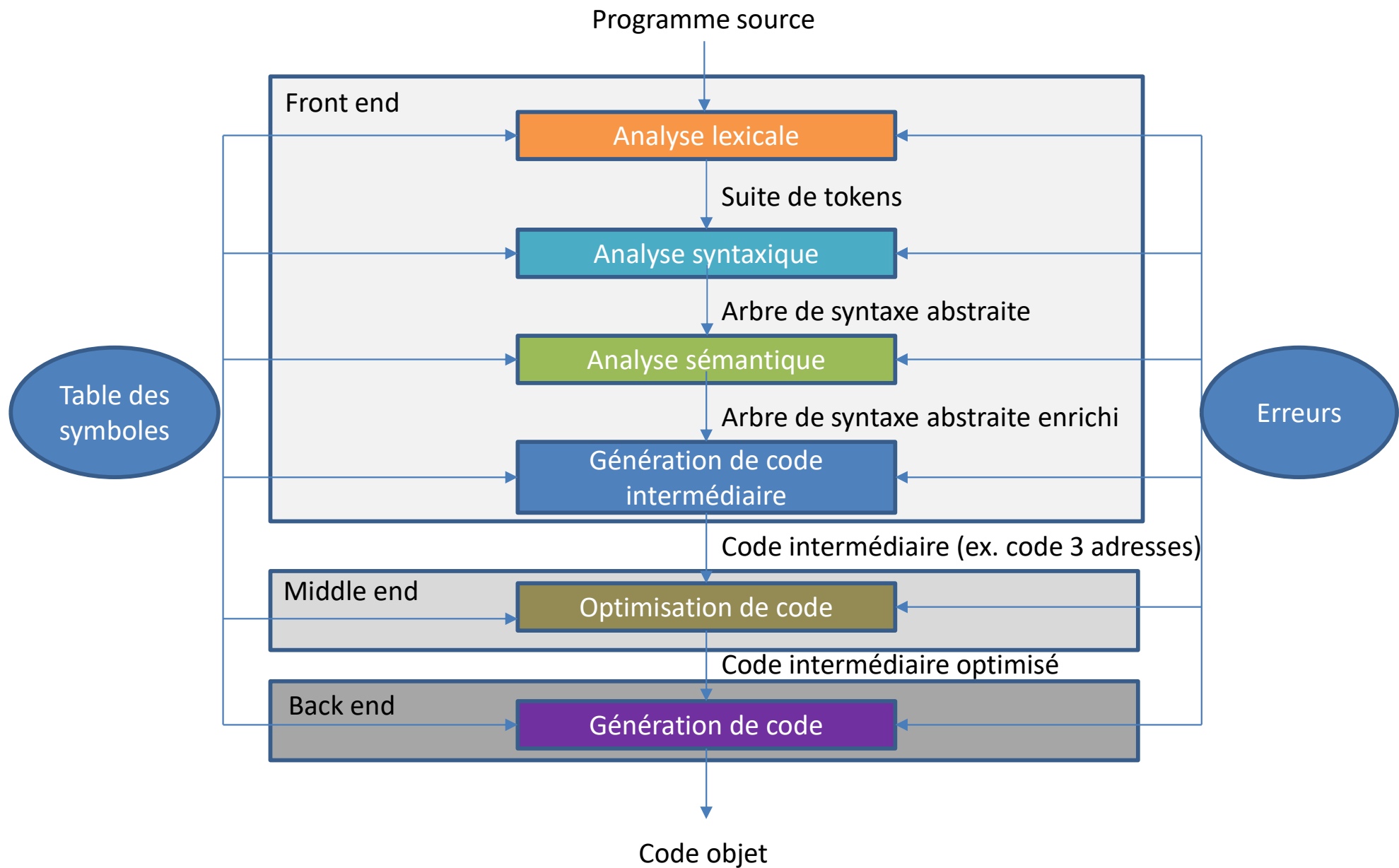
Les compilateurs

- Un logiciel parmi les plus sûrs
 - Composantes très automatisées
 - Analyse lexicale, syntaxique
 - Des composantes bien formalisées
 - Analyse sémantique, analyse statique
 - Une approche très régulière
 - Génération de code
- La compilation
 - Une théorie qui marche
 - NB : ce qui est théorisé marche le mieux...

Qu'est-ce qu'un compilateur ?



Chaîne de compilation



Chaine de compilation

- Analyse lexicale
 - Découpe du texte en petits morceaux appelés jetons (tokens)
 - Chaque jeton est une unité atomique du langage
 - Mots clés, identifiants, constantes numériques...
 - Les jetons sont décrits par un langage régulier
 - Détection via des automates à état finis
 - Description via des expression régulières
- Le logiciels effectuant l'analyse lexicale est appelé analyseur lexical ou scanner

Chaine de compilation

- Analyse syntaxique
 - Analyse de la séquence de jetons pour identifier la structure syntaxique du langage
 - S'appuie sur une grammaire formelle définissant la syntaxe du langage
 - Produit généralement un arbre syntaxique qui pourra être analysé et transformé par la suite
 - Détection des erreurs de syntaxe
 - Constructions ne respectant pas la grammaire

Chaine de compilation

- Analyse sémantique
 - Ajout d'information sémantique à l'arbre d'analyse
 - Construction de la table des symboles
 - Ex : noms de fonction, de variables etc...
 - Réalise des vérifications sémantiques
 - Vérification de type
 - Vérification de la déclaration des variables, des fonctions
 - Vérifie que les variables sont initialisées avant utilisation
 - ...
 - Emission d'erreurs et / ou d'avertissements
 - Rejet des programmes « incorrects »

Chaine de compilation

- Génération de code intermédiaire
 - Code indépendant de la machine cible
 - Généralement : utilisation du code 3 adresses
 - Nombre « infini » de registres

```
int main(void)
{
    int i;
    int b[10];
    for (i = 0; i < 10; ++i)
    { b[i] = i*i; }
}
```

```
                                i := 0 ; assignment
L1:                             if i >= 10 goto L2 ; conditional jump
                                t0 := i*i
                                t1 := &b ; address-of operation
                                t2 := t1 + i ; t2 holds the address of b[i]
                                *t2 := t0 ; store through pointer
                                i := i + 1
                                goto L1
L2:
```

Chaine de compilation

- Optimisation de code
 - Accélération du programme
 - Suppression des calculs inutiles
 - Elimination des sous expressions communes
 - Propagation des copies
 - Propagation des constantes
 - Extraction des calculs invariants
 - Elimination de code mort
 - Inlining
 - ...
- Transforme le code mais ne change pas sa sémantique
- La première optimisation est algorithmique !
 - Un compilateur ne changera pas votre algorithme...

Chaine de compilation

- Génération de code
 - Transformation du code intermédiaire en code machine
 - Connaissance des particularités du processeur
 - Nombre de registres
 - Utilisation du jeu d'instruction du processeur
 - Ex : instructions vectorielles de type SSE
 - Ordonnancement des instructions
- La seule phase réellement dépendante du processeur
- Possibilité d'avoir des compilateurs multi-cible
 - La différence réside dans la génération de code

Avantage de la structuration front / middle / back end

- Front end
 - Dépendant du langage source
 - Un front end par langage
- Back end
 - Transforme le code intermédiaire (IR) en langage cible
 - Un back end par langage cible
 - Ex : IR -> assembleur x86-64, IR -> assembleur ARM etc...
- Middle end
 - Produit une IR optimisée à partir d'une IR
 - L'IR est indépendante du langage source *et* du langage cible !
- En conclusion :
 - Au tout départ, N langages sources et M langages cibles = $N \times M$ développements de compilateurs
 - Compilateurs modernes, N langages sources et M langages cibles = production de N front ends et M back ends i.e. $N+M$ développements et réutilisation du middle end !

Un compilateur moderne : LLVM

- Front ends pour Ada, C, C++, D, Delphi, Haskell, Julia, Objective-C, Rust, swift...
- Back ends pour IA-32, x86-64, ARM, Qualcomm hexagons (DSP), MIPS, Nvidia parallel thread execution (cartes graphiques), Power PC etc...
- IR : un assembleur portable de haut niveau (plus évolué que du code 3 adresses mais avec des propriétés similaires)
 - Plusieurs dizaines de passes d'analyse et de transformation déjà fournies
 - Ex : suppression de code mort, inlining, vectorisation...

Analyse sémantique

GRAMMAIRES À ATTRIBUTS

Introduction

- Analyse syntaxique
 - Permet de vérifier que les mots appartiennent à la grammaire
 - Ne présume en rien des propriétés sémantiques du langage
- Les grammaires à attributs
 - Prise en compte de la sémantique du langage
 - Réalisation de calculs dirigés par la syntaxe
 - Calculatrice : évaluer le résultat en cours d'analyse
 - Langage : est-ce que les variables sont déclarées avant d'être utilisées ?
 - Construction de l'arbre de syntaxe abstrait
 - ...

Notion d'attribut

- Pour tout symbole terminal ou non terminal de la grammaire
 - On associe zéro, un ou plusieurs attributs
 - Un attribut est une information jugée utile en cours du processus de compilation
 - Valeur, type de donnée, numéro de ligne, de colonne dans le fichier source...
 - L'attribut est défini par un nom et un type (le type de la donnée qu'il porte)
- En se plaçant dans le contexte de l'arbre de dérivation syntaxique, distinction de deux types d'attributs:
 - Les attributs synthétisés
 - Passés du nœud fils vers les nœuds parents
 - Les attributs hérités
 - Passés du nœud parent vers les nœuds fils
 - Rq : Un nœud et ses fils correspondent à une règle
 - Nœud père = partie gauche de la règle
 - Nœuds fils = partie droite de la règle

Règles sémantiques

- On associe à chaque *règle de production* une *règle sémantique*
- Une *règle sémantique* est une fonction associée à une production syntaxique
- Elle est exécutée lorsque la production est effectuée i.e. lorsque la règle est utilisée

Exemple de grammaire attribuée

Règles de production

1. $\text{Exp} \rightarrow (- \text{Exp} \text{Exp})$
2. $\text{Exp} \rightarrow (+ \text{Exp} \text{Exp})$
3. $\text{Exp} \rightarrow (* \text{Exp} \text{Exp})$
4. $\text{Exp} \rightarrow (/ \text{Exp} \text{Exp})$
5. $\text{Exp} \rightarrow nb$

Règles sémantiques

1. $\text{Exp}.v = \text{Exp1}.v - \text{Exp2}.v$
2. $\text{Exp}.v = \text{Exp1}.v + \text{Exp2}.v$
3. $\text{Exp}.v = \text{Exp1}.v * \text{Exp2}.v$
4. $\text{Exp}.v = \text{Exp1}.v / \text{Exp2}.v$
5. $\text{Exp}.v = \text{valeur}(nb)$

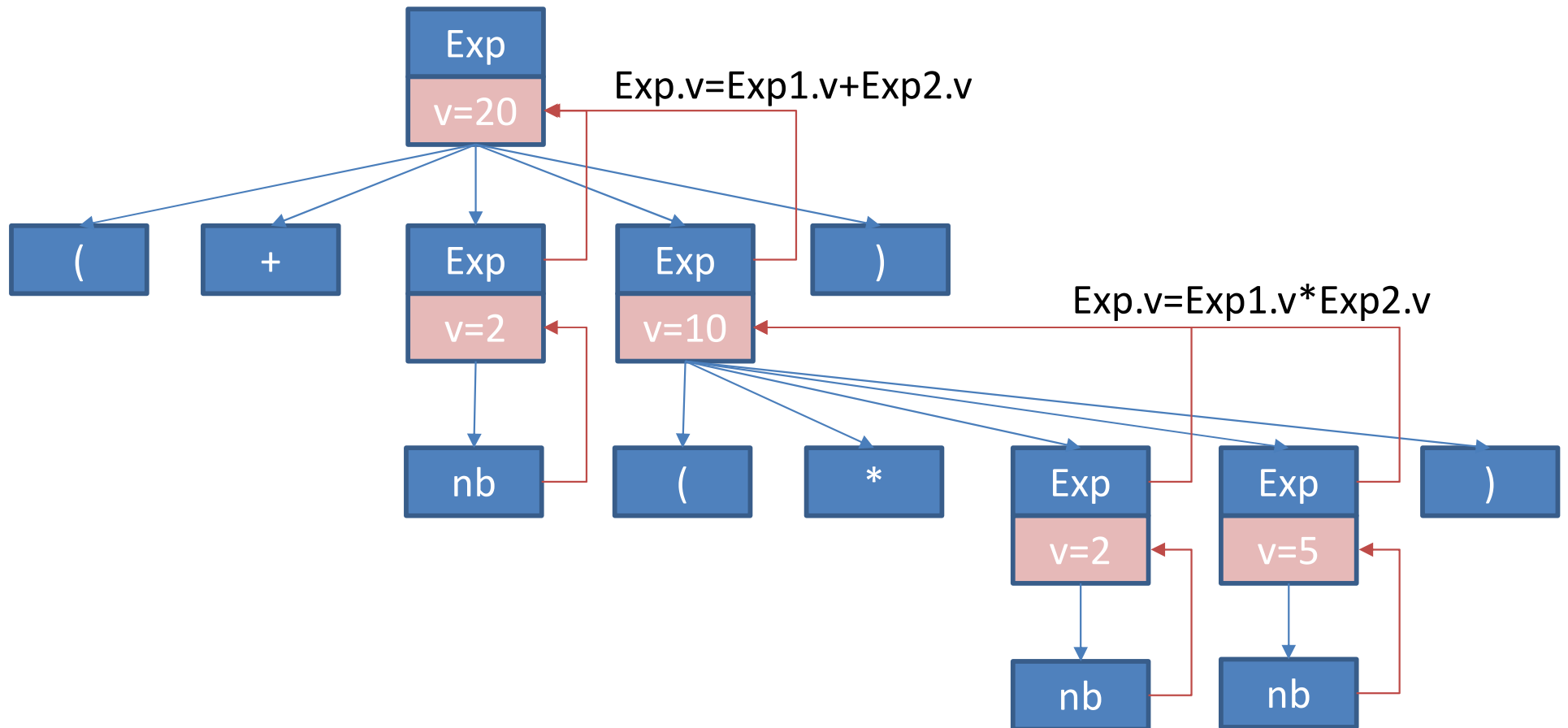
On associe un attribut numérique nommé v à Exp

➤ *Permet d'évaluer l'expression en cours d'analyse*

Note : Exp1 et Exp2 sont utilisés pour différencier les occurrences du non terminal Exp

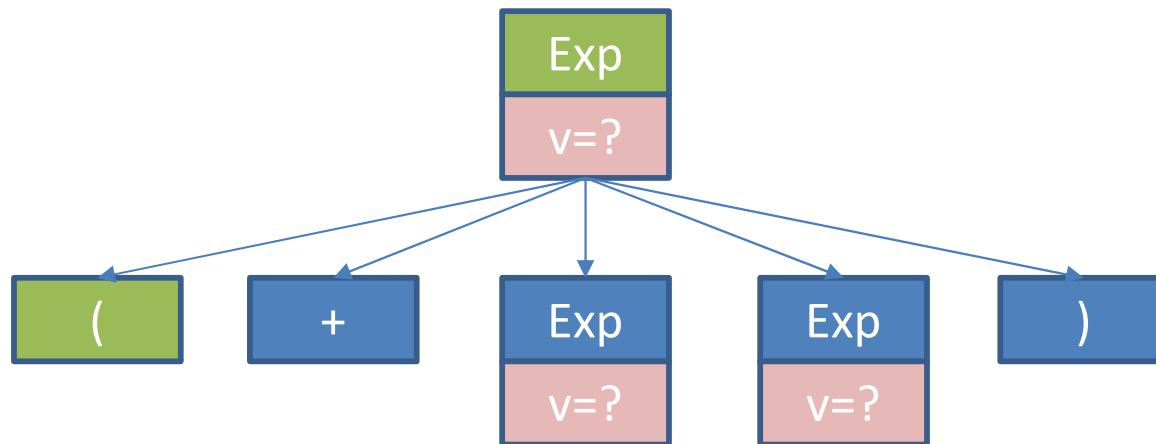
Exemple de grammaire attribuée

- Texte en entrée : $(+ 2 (* 2 5))$



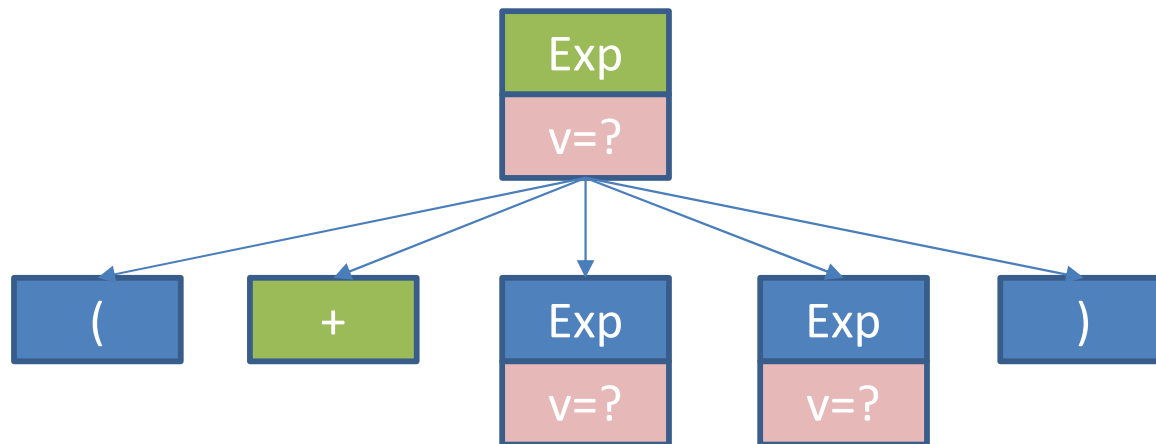
Exemple de grammaire attribuée

- Texte en entrée : (+ 2 (* 2 5))



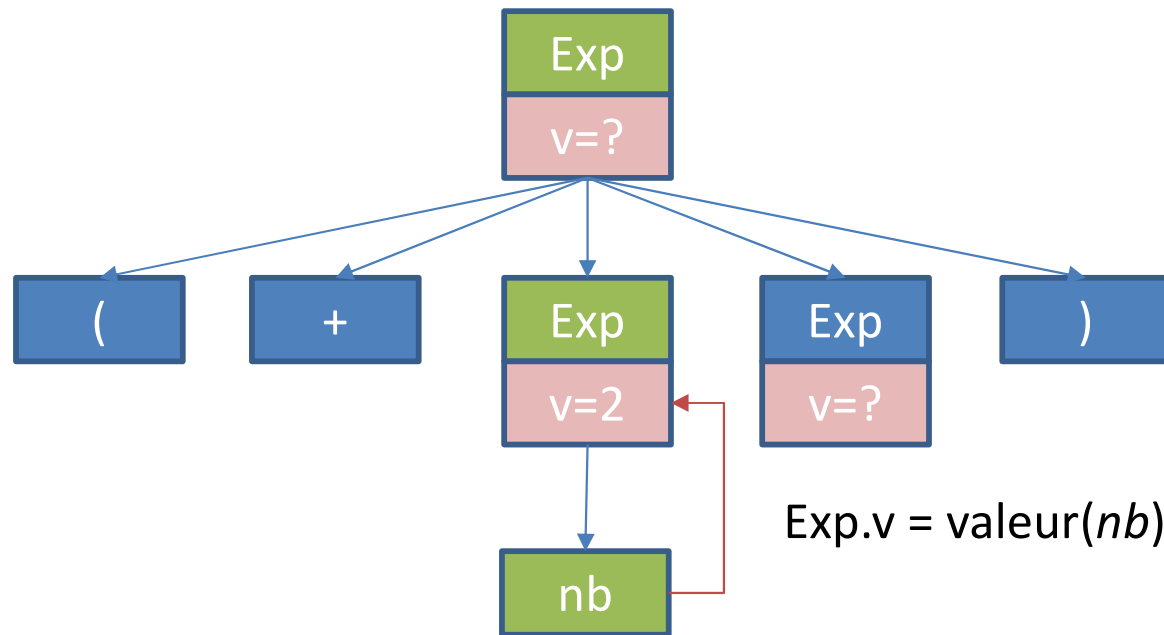
Exemple de grammaire attribuée

- Texte en entrée : (+ 2 (* 2 5))



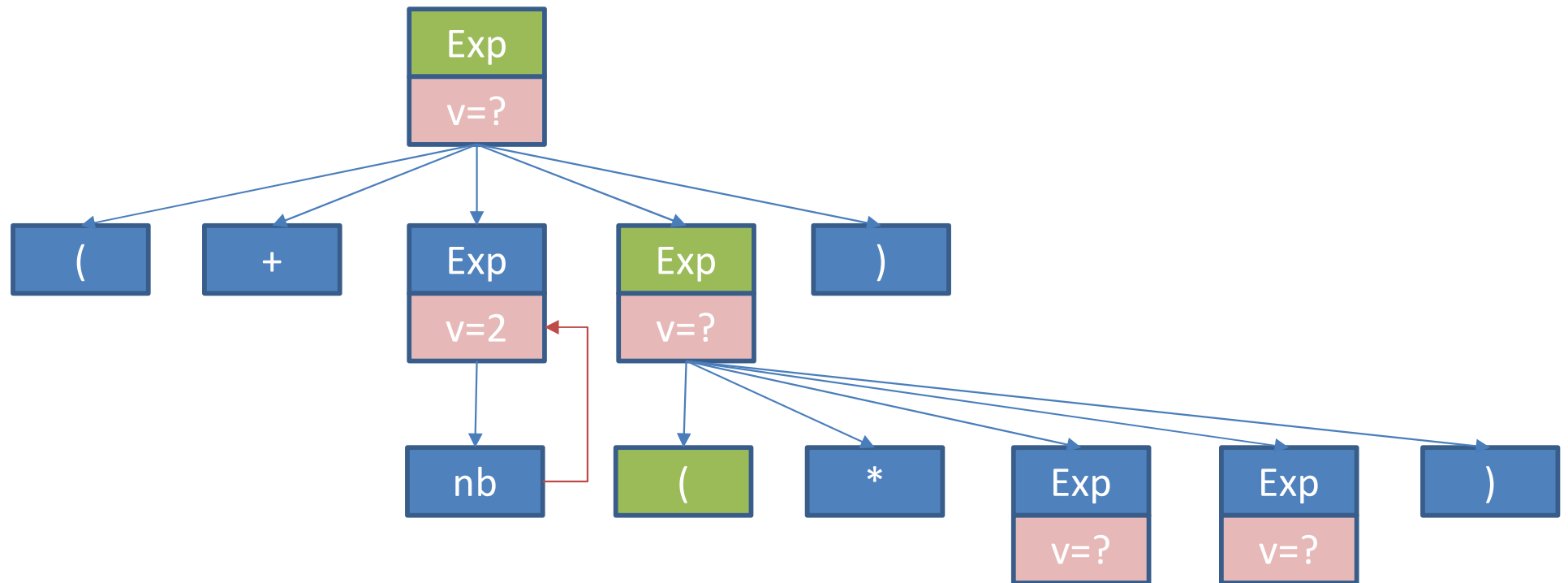
Exemple de grammaire attribuée

- Texte en entrée : (+ 2 (* 2 5))



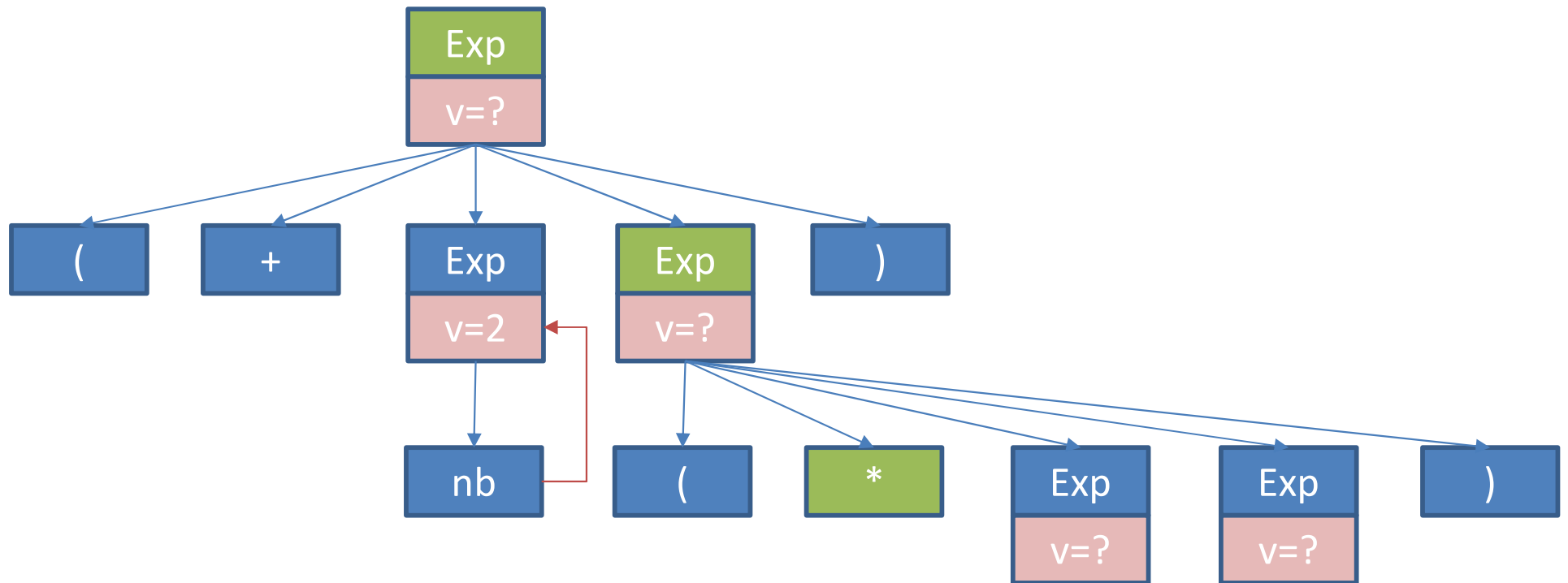
Exemple de grammaire attribuée

- Texte en entrée : (+ 2 (* 2 5))



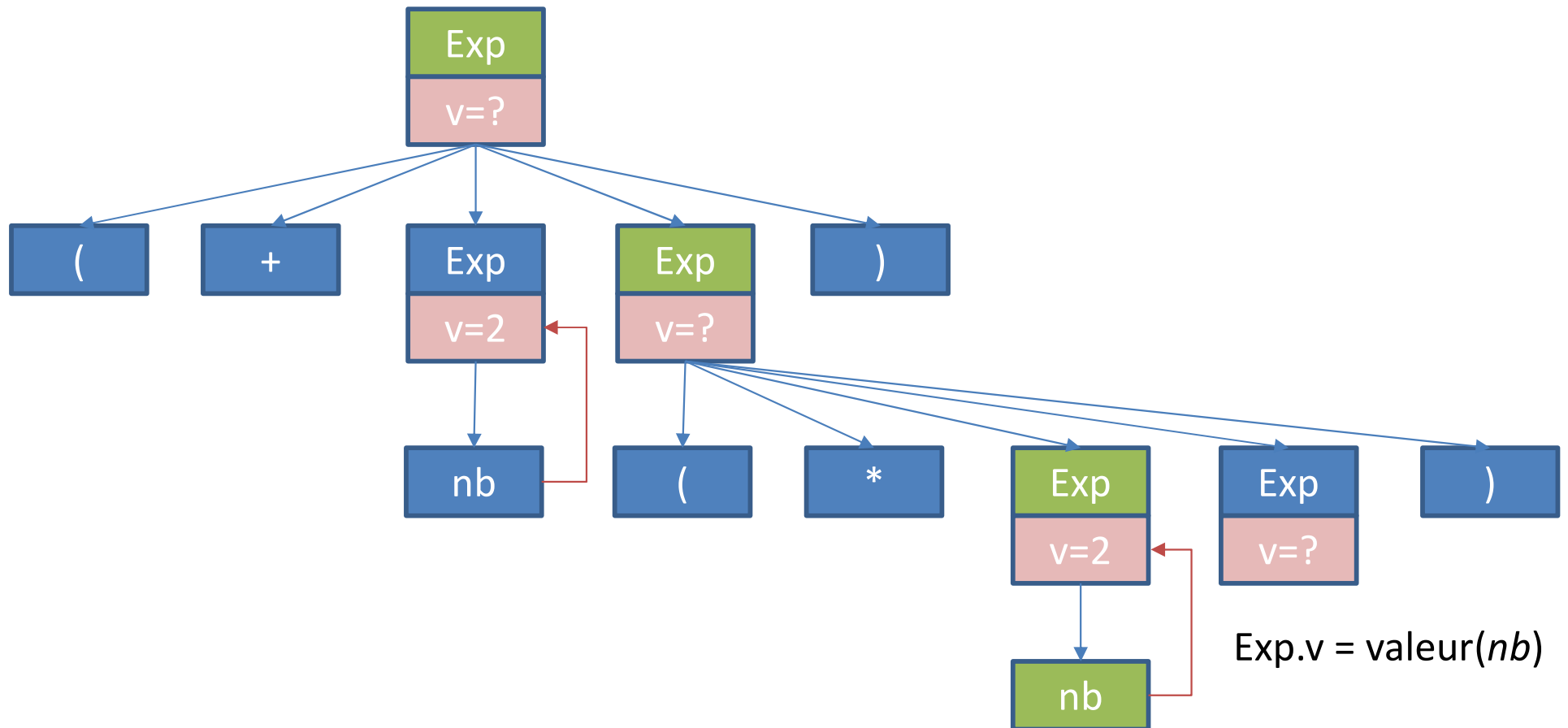
Exemple de grammaire attribuée

- Texte en entrée : (+ 2 (* 2 5))



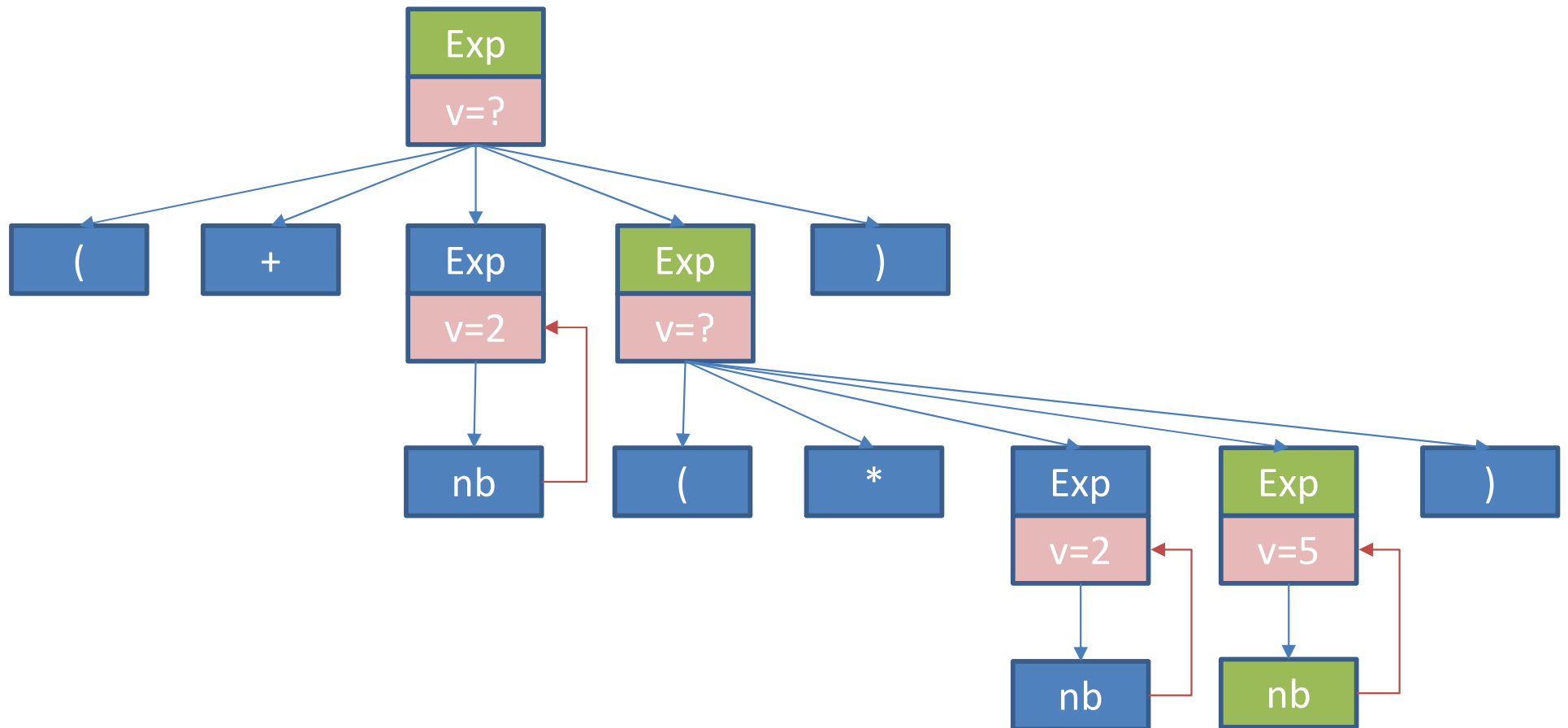
Exemple de grammaire attribuée

- Texte en entrée : (+ 2 (* 2 5))



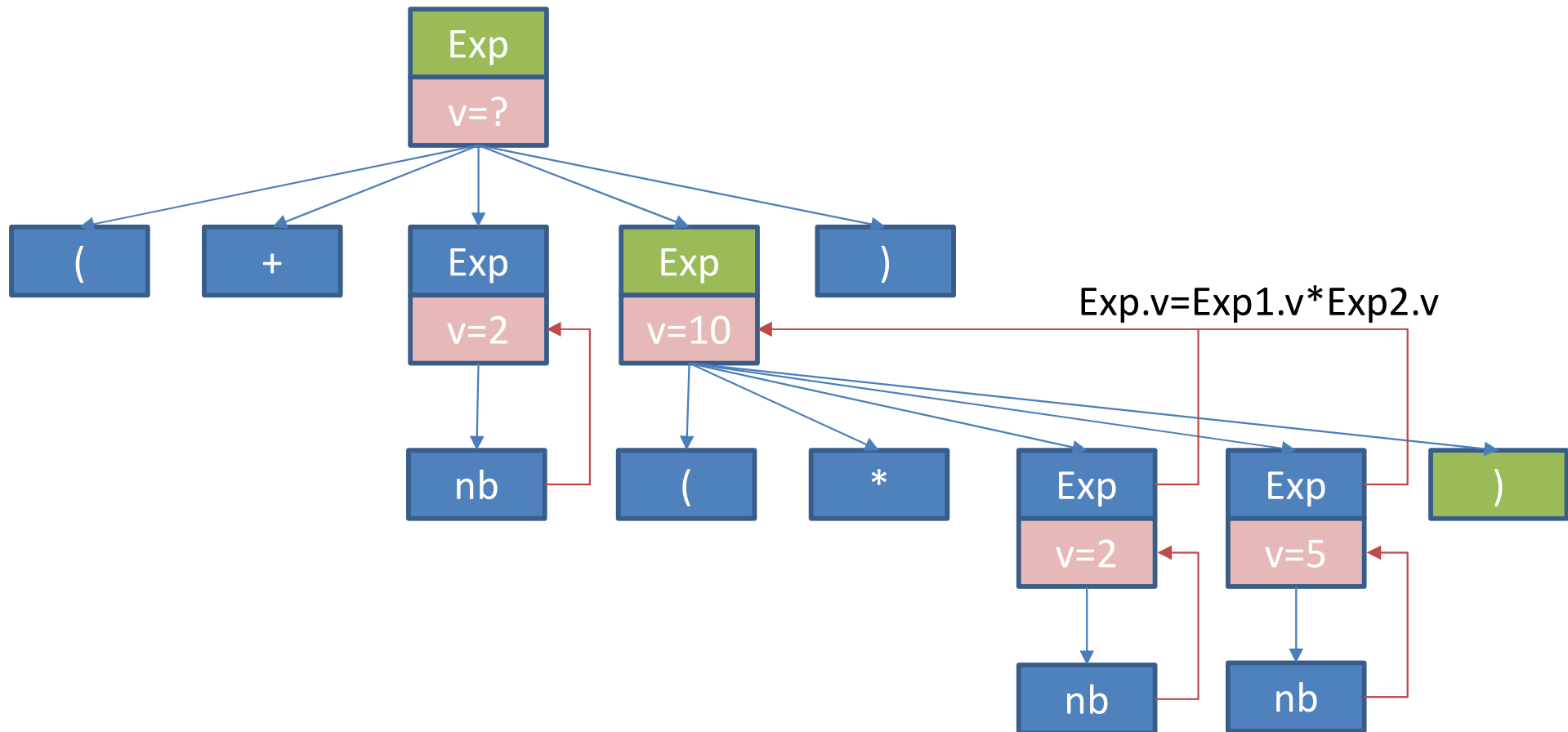
Exemple de grammaire attribuée

- Texte en entrée : (+ 2 (* 2 5))



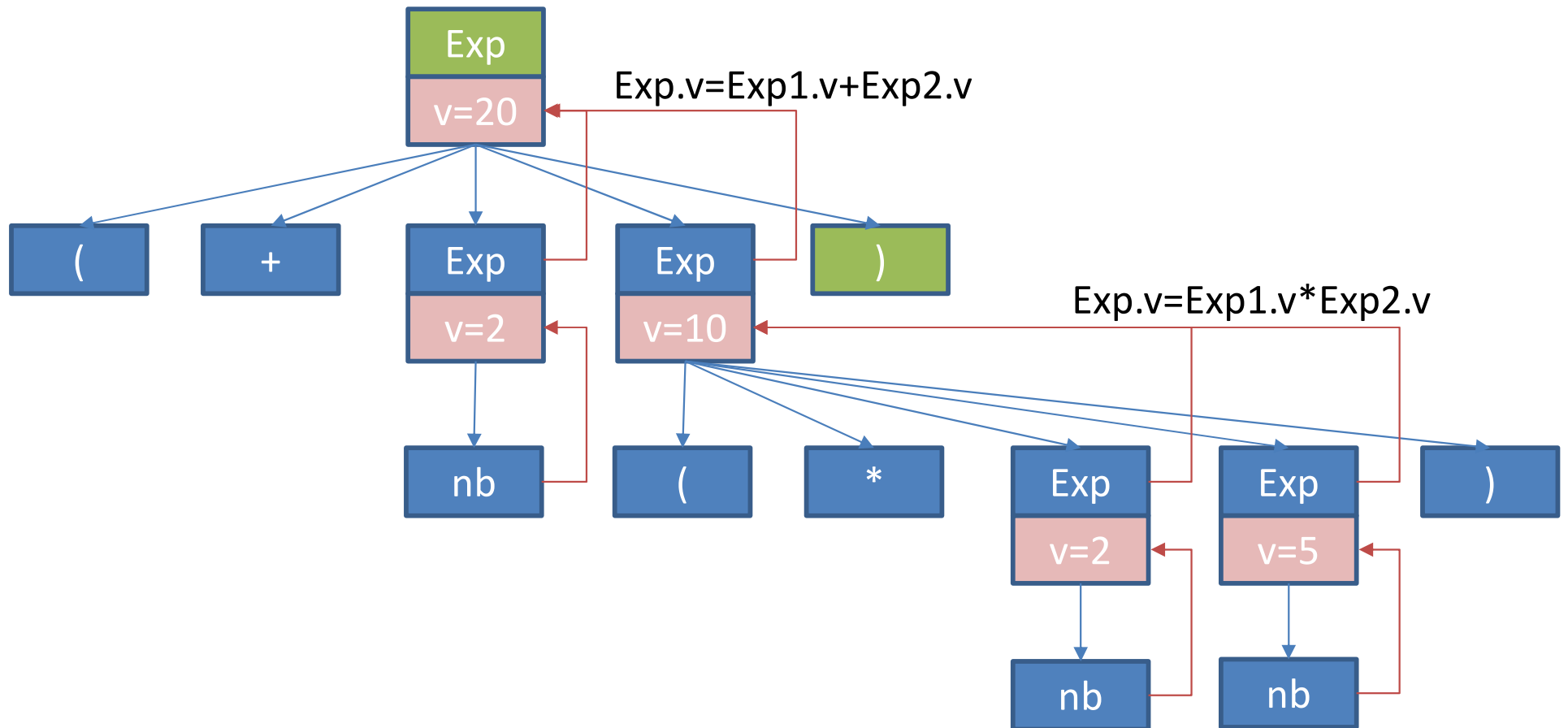
Exemple de grammaire attribuée

- Texte en entrée : (+ 2 (* 2 5))



Exemple de grammaire attribuée

- Texte en entrée : (+ 2 (* 2 5))



Grammaires attribuées

- Dans l'exemple précédent, les attributs sont synthétisés
 - Les attributs du nœud parent dépendent des attributs des nœuds fils
- Il est aussi possible de mixer attributs hérités et synthétisés
 - Ex : déclaration / utilisation de variable

Exemple déclaration /utilisation

Règles de production Règles sémantiques

- $S \rightarrow (Decl \mid Use)^*$
 - $Decl \rightarrow d ; Decl$
 - $Decl \rightarrow d$
 - $Use \rightarrow u ; Use$
 - $Use \rightarrow u$
- $\{Use.ts = Decl.ts ; S.ok = Use.ok\}$
 - $\{Decl.ts = \{d.decl\} \cup Decl1.ts\}$
 - $\{Decl.ts = \{d.decl\}\}$
 - $\{Use1.ts = Use.ts, \\ Use.ok = u.use \in Use.ts \wedge Use1.ok\}$
 - $\{Use.ok = u.use \in Use.ts\}$

Attribut ts : Ensemble des symboles déclarés

Attribut ok : Vrai si les symboles utilisés ont tous été déclarés

Attribut decl : Le symbole déclaré

Attribut use : Le symbole utilisé

Exemple déclaration /utilisation

Règles de production Règles sémantiques

- $S \rightarrow (Decl \mid Use)^*$
 - $Decl \rightarrow d ; Decl$
 - $Decl \rightarrow d$
 - $Use \rightarrow u ; Use$
 - $Use \rightarrow u$
- $\{Use.ts = Decl.ts ; S.ok = Use.ok\}$
 - $\{Decl.ts = \{d.decl\} \cup Decl1.ts\}$
 - $\{Decl.ts = \{d.decl\}\}$
 - $\{Use1.ts = Use.ts, \\ Use.ok = u.use \in Use.ts \wedge Use1.ok\}$
 - $\{Use.ok = u.use \in Use.ts\}$

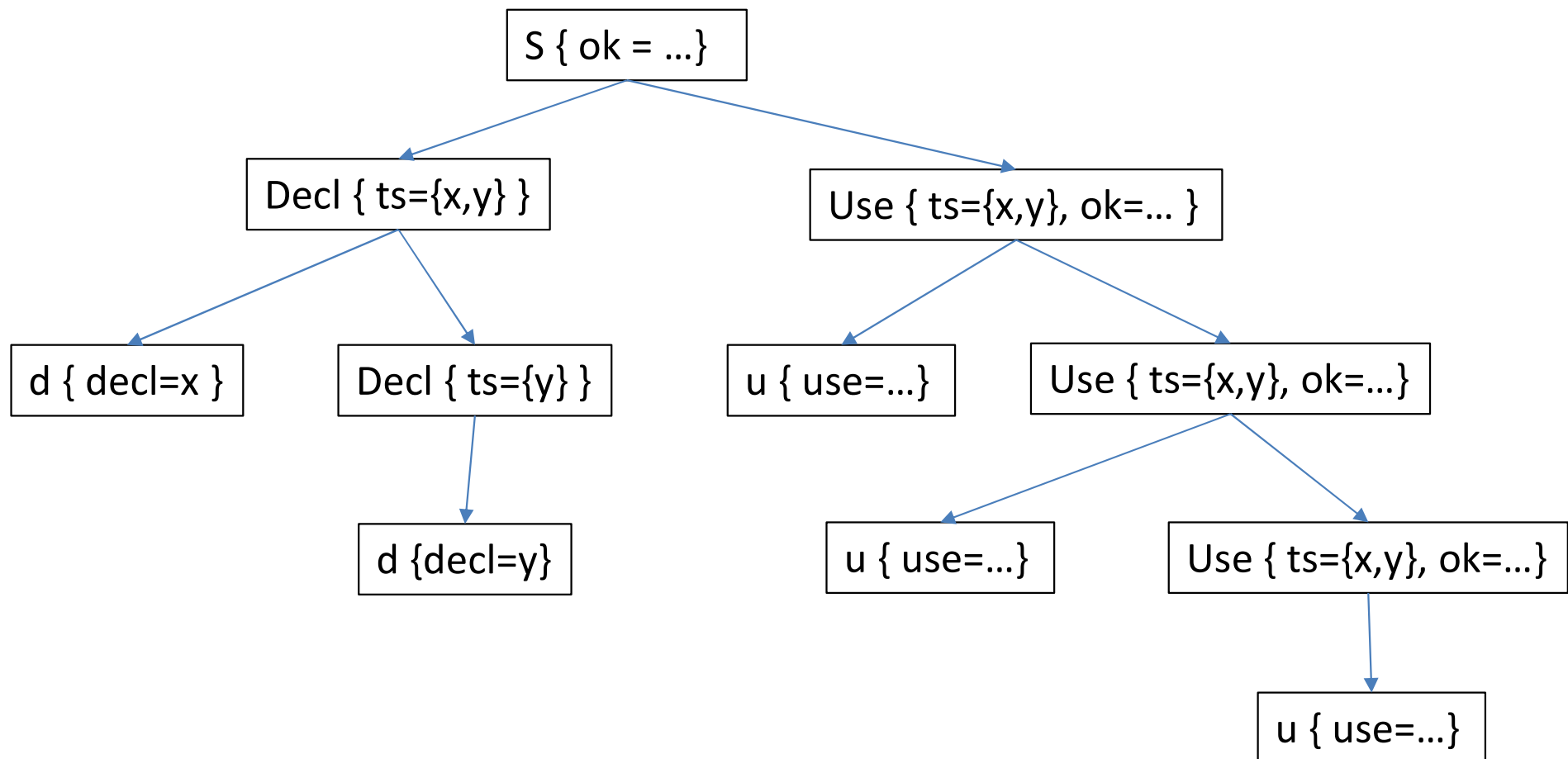
Attribut ts : Hérité et synthétisé

Attribut ok : Synthétisé

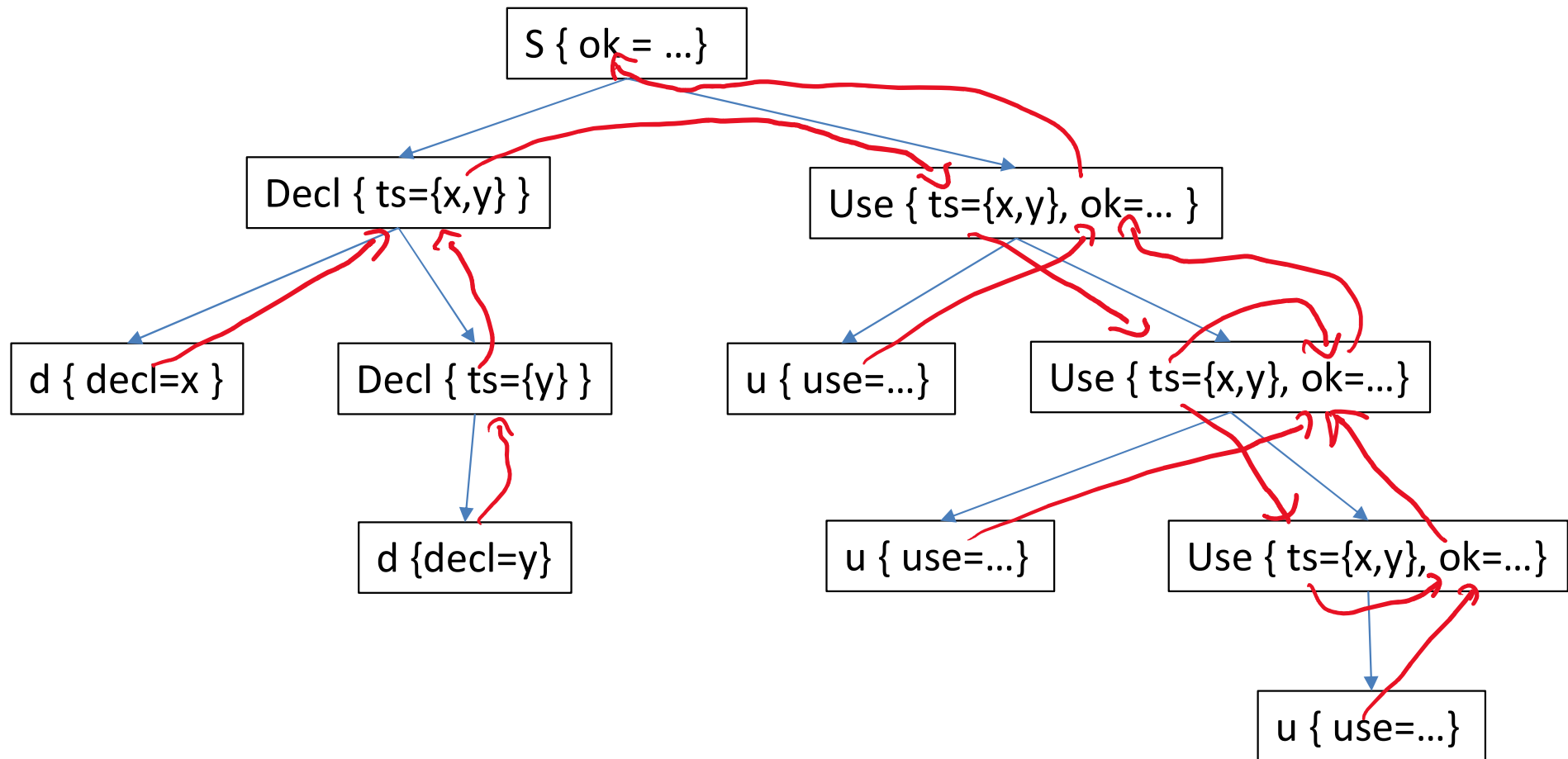
Attribut decl : Synthétisé

Attribut use : Synthétisé

Exemple déclaration /utilisation



Exemple déclaration /utilisation

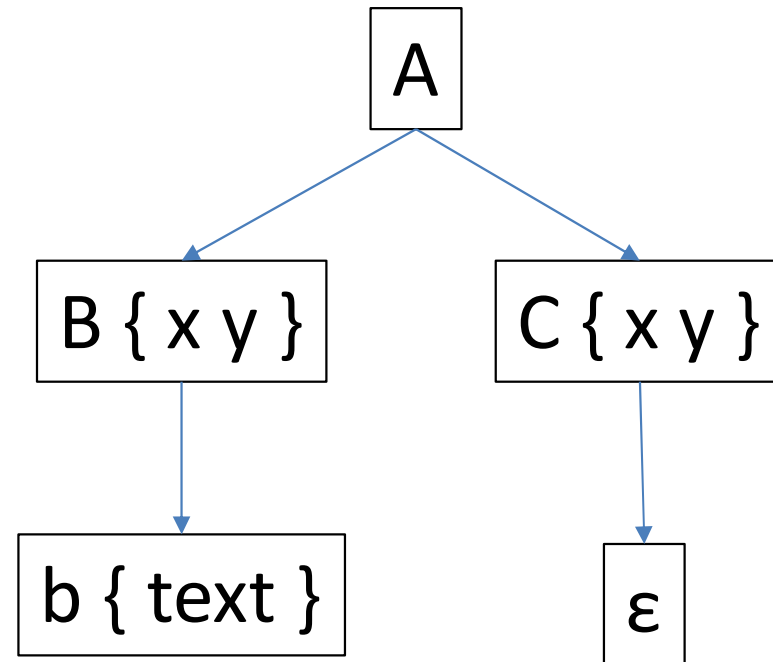


Evaluation des attributs

- Deux difficultés
 - Possibles dépendances circulaires entre règles sémantiques
 - Difficulté d'entrelacer (dans la mise en œuvre)
 - Construction de l'arbre
 - Création des instances d'attributs
 - Evaluation des règles sémantiques

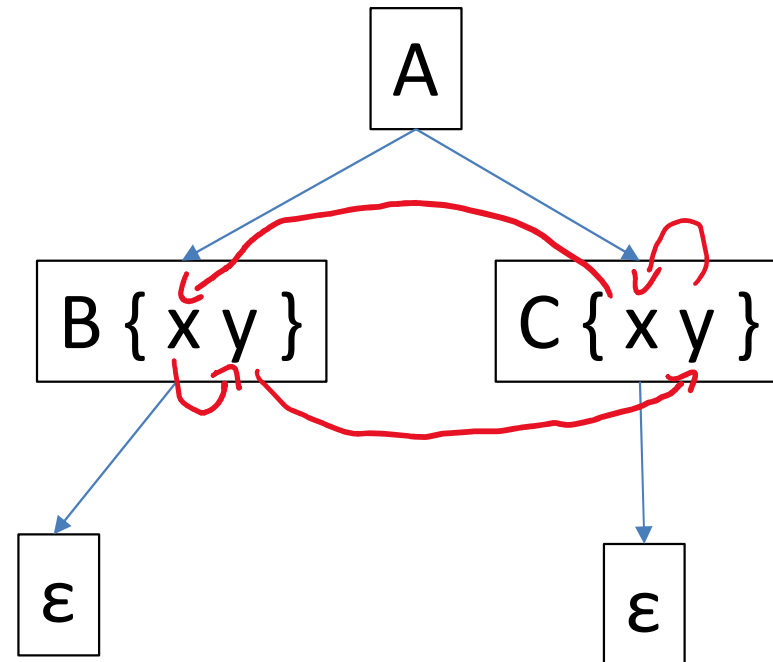
Dépendances circulaires

- $A \longrightarrow B \ C \ \{B.x = C.x ; C.y = B.y\}$
- $B \longrightarrow b \ \{B.y = b.text\}$
- $B \longrightarrow \varepsilon \ \{B.y = B.x\}$
- $C \longrightarrow \varepsilon \ \{C.x = C.y\}$



Dépendances circulaires

- $A \longrightarrow B \ C \ \{B.x = C.x ; C.y = B.y\}$
- $B \longrightarrow b \ \{B.y = b.text\}$
- $B \longrightarrow \varepsilon \ \{B.y = B.x\}$
- $C \longrightarrow \varepsilon \ \{C.x = C.y\}$



Détecter la circularité

Après construction de l'arbre

- Algorithme de tri topologique
 - Entrée : Toutes les instances d'attributs, ensemble A
 - Sortie : Un ordre total $<$ sur A , tel que
$$\forall (a, a') \in A^2, a' \rightarrow^d a \implies a' < a$$
 - $Rq : a' \rightarrow^d a$ se lit a est *dépendant* de a'
 - Si l'ordre total ne peut pas être produit : échec (lié à des dépendances circulaires)
 - *Rq : l'ordre total n'est pas forcément unique*
- Complexité : $O(\text{Card}(A))$

Détecter la circularité

Après construction de l'arbre

- Semble efficace
- Mais...
 - ... que faire en cas d'échec de l'analyse ?
 - Echec dû à une grammaire mal formée
 - Cela ne regarde pas l'utilisateur final !

➤ *Il faut détecter la circularité en amont*

Détecter la circularité

Avant la construction de l'arbre

- Ne doit pas dépendre du mot en entrée
- Il faut montrer que
 - $\forall m \in \mathcal{L}(G)$ le graphe de dépendance produit par l'analyse de m ne contient pas de cycle
- Algorithme de Knuth (1968) et variantes
 - Idée : représenter la fermeture transitive des dépendances connues de chaque non terminal
 - Complexité exponentielle : en pratique, non utilisable.

Critères de non-circularité

- Distinguer les attributs synthétisés et hérités
- Synthétisés
 - $corps \rightarrow^d tête$
- Hérités
 - $tête \rightarrow^d corps$
 - $corps \rightarrow^d corps$
- Tête = partie gauche de la règle, corps = partie droite

Critères de non circularité

- Si synthétisé seulement
 - *Pas de circularité*
 - Ex : YACC
- Si hérité et synthétisé
 - *Risque de circularité*
- Si hérité et synthétisé, mais héritage seulement *gauche* \rightarrow^d *droite*
 - *Pas de circularité*

Bilan des tests de non circularité

- Tri topologique
 - Grammaire attribuée et mot connus
- Algorithme de Knuth
 - Grammaire attribuée connue
 - Conclusion pour tout mot
- Critères de non-circularité
 - Sous-langage de Grammaire attribuée connu
 - Conclusion pour toute Grammaire attribuée du sous-langage et tout mot

Trop tardif

Trop complexe

Facile

Méthodologie des grammaires attribuées : étude

- **Identifier** ce qu'on veut calculer
- **Élaborer** des cas d'étude
- **Tracer** les arbres de syntaxe
- **Imaginer** la circulation de l'information nécessaire au calcul
- **Définir** les attributs qui opèrent la circulation
- Les **identifier** comme synthétisés ou hérités

Méthodologie des grammaires attribuées : réalisation

- **Lister** les attributs synthétisés
- **Lister** les attributs hérités
- **Définir** les règles sémantiques
- **Faire attention** aux initialisations
- **Tester** la non-circularité

Mise en œuvre en ANTLR

- Attributs synthétisés \equiv résultat d'appel de fonction
 - Toute règle peut produire un résultat \equiv attributs synthétisés
 - Exemple
expr **returns** [int v] : term '+' expr
{ \$v = \$term.v + \$expr.v } ;

Mise en œuvre en ANTLR

- Attributs hérités \equiv paramètres d'appel de procédure
 - Toute règle peut dépendre de paramètres \equiv attributs hérités
 - Exemple
expr [TS ts] **returns** [int v] :
term [ts] '+' expr [ts] {\$v = \$term.v + \$expr.v} ;

Note : ts représente une table des symboles contenant (entre autres) toutes les variables déclarées dans le contexte

Construction de l'arbre vs évaluation

- Analyse par descente récursive
 - Chaque règle \equiv procédure
 - Expansion \equiv appel de procédure
 - Héritage d'attributs
 - Réduction \equiv retour de procédure
 - Synthèse d'attributs
- Entrelacement possible, mais...

...Attention à la circularité !!!

Stratégie recommandée

- Limiter l'usage des attributs de la grammaire
 - Construction arbre de syntaxe abstraite (ASA)
 - Contrôles syntaxiques simples
- Programmer des passes d'analyse sur l'ASA
 - pour des calculs plus sémantiques*
 - Utilisation de visiteurs sur l'ASA
 - Possibilité d'ajouter des attributs
 - Eventuellement d'en supprimer si devenus inutiles
 - Transformations de l'ASA éventuellement possibles
 - Réécritures ne changeant pas la sémantique

Conclusion

- La grammaire attribuée est l'outil de base pour les calculs dirigés par la syntaxe
- Penser global – agir local
- Concevoir en termes de
 - Grammaire abstraite
 - Attribut synthétisé/hérité

*S'adapter dans un second temps
selon la mise en oeuvre*

ANALYSE SEMANTIQUE

Limites des grammaire algébriques

- Comment prévenir la définition multiple d'une même fonction ?
- Comment différencier des variables de type différent ?
- Comment s'assurer qu'une classe implémente toutes les méthodes d'une interface ?
- Etc...

➤ *Seule, une grammaire ne peut pas tout vérifier...*

Analyse sémantique

- Objectif : assurer que le programme a un sens
- Vérification de propriétés non captées par l'analyse lexicale et syntaxique
 - Les fonctions utilisées sont bien définies
 - Les variables sont déclarées avant leur utilisation
 - Les variables sont initialisées avant leur utilisation
 - Deux variables de même identifiant ne sont pas déclarées dans le même contexte
 - Les expressions ont un type correct
 - ...
- Une fois l'analyse sémantique terminée, nous savons que le programme est correct

Challenges

- Rejeter le plus grand nombre de programmes incorrects
- Accepter le plus grand nombre de programmes corrects
- Le faire rapidement...

Analyse sémantique : implémentation

- Grammaires à attributs
 - Augmenter les règles pour faire la vérification durant le parsing
 - Complexe et peu lisible
 - Peu modulaire et extensible
- Parcours récursif de l'arbre de syntaxe abstraite (ASA)
 - Construire l'ASA avec une grammaire à attributs
 - Ecrire des passes parcourant l'ASA
 - Synthèse d'information
 - Vérifications (déclarations, existence, typage...)
 - Transformations préservant la sémantique
 - ...
 - Approche plus modulaire, extensible... A privilégier...

Overview du chapitre

- Vérification de portée
 - Comment dire à quoi réfère un identifiant ?
 - Comment stocke-t-on cette information ?
- Vérification de type
 - Comment dire si des expressions ont le bon type ?
 - Comment dire si les fonctions ont des paramètres valides ?

Analyse sémantique

TABLE DES SYMBOLES

Qu'est qu'un symbole ?

- Le même symbole dans un programme peut faire référence à des choses totalement différentes.
 - Ex en java:

```
public class A
{
    char A;
    A A(A A)
    {
        A.A = 'A'
        return A( (A) A ) ;
    }
}
```

Qu'est qu'un symbole ?

- Le même symbole dans un programme peut faire référence à des choses totalement différentes.
 - Ex en java:

```
public class A
{
    char A;
    A A(A A)
    {
        A.A = 'A'
        return A( (A) A );
    }
}
```

Notion de portée

- La portée d'une entité est l'ensemble des endroits dans un programme où le symbole associé à l'entité réfère à cette entité
- L'introduction d'une nouvelle entité dans un contexte peut cacher des entités préalablement visibles
 - Ex : une variable locale peut cacher un paramètre de fonction
- Comment savoir à tout instance ce qui est visible et à quoi un symbole réfère ?

Table des symboles

- La table des symboles est une association entre un symbole et l'entité à laquelle il réfère
- En cours d'analyse sémantique la table des symboles est continuellement mise à jour pour savoir ce qui est à portée
- Questions :
 - A quoi cela ressemble-t-il ?
 - Quelle opération doit-elle définir ?
 - Comment l'implémenter ?

Table des symboles : intuition

```
01: int x = 137;
02: int z = 42;
03: int function(int x, int y)
04: {
05:     std::cout<<x<<" "<<y<<" "<<z<<std::endl;
06:     {
07:         int x, z;
08:         z=y;
09:         x=z;
10:         {
11:             int y=x;
12:             {
13:                 std::cout<<x<<" "<<y<<" "<<z<<std::endl;
14:             }
15:             std::cout<<x<<" "<<y<<" "<<z<<std::endl;
16:         }
17:         std::cout<<x<<" "<<y<<" "<<z<<std::endl;
18:     }
19: }
```


Table des symboles : intuition

```
01: int x = 137;  
02: int z = 42;  
03: int function(int x, int y)  
04: {  
05:     std::cout<<x<<" "<<y<<" "<<z<<std::endl;  
06:     {  
07:         int x, z;  
08:         z=y;  
09:         x=z;  
10:         {  
11:             int y=x;  
12:             {  
13:                 std::cout<<x<<" "<<y<<" "<<z<<std::endl;  
14:             }  
15:             std::cout<<x<<" "<<y<<" "<<z<<std::endl;  
16:         }  
17:         std::cout<<x<<" "<<y<<" "<<z<<std::endl;  
18:     }  
19: }
```

Symbole	Ligne
x	01

Table des symboles : intuition

```
01: int x = 137;  
02: int z = 42;  
03: int function(int x, int y)  
04: {  
05:     std::cout<<x<<" "<<y<<" "<<z<<std::endl;  
06:     {  
07:         int x, z;  
08:         z=y;  
09:         x=z;  
10:         {  
11:             int y=x;  
12:             {  
13:                 std::cout<<x<<" "<<y<<" "<<z<<std::endl;  
14:             }  
15:             std::cout<<x<<" "<<y<<" "<<z<<std::endl;  
16:         }  
17:         std::cout<<x<<" "<<y<<" "<<z<<std::endl;  
18:     }  
19: }
```

Symbole	Ligne
x	01
z	02

Table des symboles : intuition

```

01: int x = 137;
02: int z = 42;
03: int function(int x, int y)
04: {
05:     std::cout<<x<<" "<<y<<" "<<z<<std::endl;
06:     {
07:         int x, z;
08:         z=y;
09:         x=z;
10:         {
11:             int y=x;
12:             {
13:                 std::cout<<x<<" "<<y<<" "<<z<<std::endl;
14:             }
15:             std::cout<<x<<" "<<y<<" "<<z<<std::endl;
16:         }
17:         std::cout<<x<<" "<<y<<" "<<z<<std::endl;
18:     }
19: }
    
```

Symbole	Ligne
x	01
z	02
x	03
y	03

Table des symboles : intuition

```

01: int x = 137;
02: int z = 42;
03: int function(int x, int y)
04: {
05:     std::cout<<x<<" "<<y<<" "<<z<<std::endl;
06:     {
07:         int x, z;
08:         z=y;
09:         x=z;
10:         {
11:             int y=x;
12:             {
13:                 std::cout<<x<<" "<<y<<" "<<z<<std::endl;
14:             }
15:             std::cout<<x<<" "<<y<<" "<<z<<std::endl;
16:         }
17:         std::cout<<x<<" "<<y<<" "<<z<<std::endl;
18:     }
19: }

```

Symbole	Ligne
x	01
z	02
x	03
y	03

Table des symboles : intuition

```

01: int x = 137;
02: int z = 42;
03: int function(int x, int y)
04: {
05:  std::cout<<x@03<<"<<y@03<<"<<z@02<<std::endl;
06:  {
07:   int x, z;
08:   z=y;
09:   x=z;
10:   {
11:    int y=x;
12:    {
13:     std::cout<<x<<"<<y<<"<<z<<std::endl;
14:    }
15:    std::cout<<x<<"<<y<<"<<z<<std::endl;
16:   }
17:   std::cout<<x<<"<<y<<"<<z<<std::endl;
18:  }
19: }

```

Symbole	Ligne
x	01
z	02
x	03
y	03

Table des symboles : intuition

```

01: int x = 137;
02: int z = 42;
03: int function(int x, int y)
04: {
05:   std::cout<<x@03<<"<<y@03<<"<<z@02<<std::endl;
06: {
07:   int x, z;
08:   z=y;
09:   x=z;
10:   {
11:     int y=x;
12:     {
13:       std::cout<<x<<"<<y<<"<<z<<std::endl;
14:     }
15:     std::cout<<x<<"<<y<<"<<z<<std::endl;
16:   }
17:   std::cout<<x<<"<<y<<"<<z<<std::endl;
18: }
19: }

```

Symbole	Ligne
x	01
z	02
x	03
y	03

Table des symboles : intuition

```

01: int x = 137;
02: int z = 42;
03: int function(int x, int y)
04: {
05:     std::cout<<x@03<<"<<y@03<<"<<z@02<<std::endl;
06:     {
07:         int x, z;
08:         z=y;
09:         x=z;
10:         {
11:             int y=x;
12:             {
13:                 std::cout<<x<<"<<y<<"<<z<<std::endl;
14:             }
15:             std::cout<<x<<"<<y<<"<<z<<std::endl;
16:         }
17:         std::cout<<x<<"<<y<<"<<z<<std::endl;
18:     }
19: }

```

Symbole	Ligne
x	01
z	02
x	03
y	03
x	07
z	07

Table des symboles : intuition

```

01: int x = 137;
02: int z = 42;
03: int function(int x, int y)
04: {
05:   std::cout<<x@03<<"<<y@03<<"<<z@02<<std::endl;
06:   {
07:     int x, z;
08:     z@07=y@03;
09:     x=z;
10:     {
11:       int y=x;
12:       {
13:         std::cout<<x<<"<<y<<"<<z<<std::endl;
14:       }
15:       std::cout<<x<<"<<y<<"<<z<<std::endl;
16:     }
17:     std::cout<<x<<"<<y<<"<<z<<std::endl;
18:   }
19: }

```

Symbole	Ligne
x	01
z	02
x	03
y	03
x	07
z	07

Table des symboles : intuition

```

01: int x = 137;
02: int z = 42;
03: int function(int x, int y)
04: {
05:   std::cout<<x@03<<"<<y@03<<"<<z@02<<std::endl;
06:   {
07:     int x, z;
08:     z@07=y@03;
09:     x@07=z@07;
10:   {
11:     int y=x;
12:     {
13:       std::cout<<x<<"<<y<<"<<z<<std::endl;
14:     }
15:     std::cout<<x<<"<<y<<"<<z<<std::endl;
16:   }
17:   std::cout<<x<<"<<y<<"<<z<<std::endl;
18: }
19: }

```

Symbole	Ligne
x	01
z	02
x	03
y	03
x	07
z	07

Table des symboles : intuition

```

01: int x = 137;
02: int z = 42;
03: int function(int x, int y)
04: {
05:   std::cout<<x@03<<"<<y@03<<"<<z@02<<std::endl;
06:   {
07:     int x, z;
08:     z@07=y@03;
09:     x@07=z@07;
10:   {
11:     int y=x;
12:     {
13:       std::cout<<x<<"<<y<<"<<z<<std::endl;
14:     }
15:     std::cout<<x<<"<<y<<"<<z<<std::endl;
16:   }
17:   std::cout<<x<<"<<y<<"<<z<<std::endl;
18: }
19: }

```

Symbole	Ligne
x	01
z	02
x	03
y	03
x	07
z	07

Table des symboles : intuition

```

01: int x = 137;
02: int z = 42;
03: int function(int x, int y)
04: {
05:     std::cout<<x@03<<"<<y@03<<"<<z@02<<std::endl;
06:     {
07:         int x, z;
08:         z@07=y@03;
09:         x@07=z@07;
10:         {
11:             int y=x@07;
12:             {
13:                 std::cout<<x<<"<<y<<"<<z<<std::endl;
14:             }
15:             std::cout<<x<<"<<y<<"<<z<<std::endl;
16:         }
17:         std::cout<<x<<"<<y<<"<<z<<std::endl;
18:     }
19: }

```

Symbole	Ligne
x	01
z	02
x	03
y	03
x	07
z	07
y	11

Table des symboles : intuition

```

01: int x = 137;
02: int z = 42;
03: int function(int x, int y)
04: {
05:     std::cout<<x@03<<"<<y@03<<"<<z@02<<std::endl;
06:     {
07:         int x, z;
08:         z@07=y@03;
09:         x@07=z@07;
10:         {
11:             int y=x@07;
12:             {
13:                 std::cout<<x<<"<<y<<"<<z<<std::endl;
14:             }
15:             std::cout<<x<<"<<y<<"<<z<<std::endl;
16:         }
17:         std::cout<<x<<"<<y<<"<<z<<std::endl;
18:     }
19: }

```

Symbole	Ligne
x	01
z	02
x	03
y	03
x	07
z	07
y	11

Table des symboles : intuition

```

01: int x = 137;
02: int z = 42;
03: int function(int x, int y)
04: {
05:   std::cout<<x@03<<"<<y@03<<"<<z@02<<std::endl;
06:   {
07:     int x, z;
08:     z@07=y@03;
09:     x@07=z@07;
10:     {
11:       int y=x@07;
12:       {
13:         std::cout<<x@07<<"<<y@11<<"<<z@07<<std::endl;
14:       }
15:       std::cout<<x<<"<<y<<"<<z<<std::endl;
16:     }
17:     std::cout<<x<<"<<y<<"<<z<<std::endl;
18:   }
19: }

```

Symbole	Ligne
x	01
z	02
x	03
y	03
x	07
z	07
y	11

Table des symboles : intuition

```

01: int x = 137;
02: int z = 42;
03: int function(int x, int y)
04: {
05:     std::cout<<x@03<<"<<y@03<<"<<z@02<<std::endl;
06:     {
07:         int x, z;
08:         z@07=y@03;
09:         x@07=z@07;
10:         {
11:             int y=x@07;
12:             {
13:                 std::cout<<x@07<<"<<y@11<<"<<z@07<<std::endl;
14:             }
15:             std::cout<<x<<"<<y<<"<<z<<std::endl;
16:         }
17:         std::cout<<x<<"<<y<<"<<z<<std::endl;
18:     }
19: }

```

Symbole	Ligne
x	01
z	02
x	03
y	03
x	07
z	07
y	11

Table des symboles : intuition

```

01: int x = 137;
02: int z = 42;
03: int function(int x, int y)
04: {
05:   std::cout<<x@03<<"<<y@03<<"<<z@02<<std::endl;
06:   {
07:     int x, z;
08:     z@07=y@03;
09:     x@07=z@07;
10:     {
11:       int y=x@07;
12:       {
13:         std::cout<<x@07<<"<<y@11<<"<<z@07<<std::endl;
14:       }
15:       std::cout<<x@07<<"<<y@11<<"<<z@07<<std::endl;
16:     }
17:     std::cout<<x<<"<<y<<"<<z<<std::endl;
18:   }
19: }

```

Symbole	Ligne
x	01
z	02
x	03
y	03
x	07
z	07
y	11

Table des symboles : intuition

```

01: int x = 137;
02: int z = 42;
03: int function(int x, int y)
04: {
05:     std::cout<<x@03<<"<<y@03<<"<<z@02<<std::endl;
06:     {
07:         int x, z;
08:         z@07=y@03;
09:         x@07=z@07;
10:         {
11:             int y=x@07;
12:             {
13:                 std::cout<<x@07<<"<<y@11<<"<<z@07<<std::endl;
14:             }
15:             std::cout<<x@07<<"<<y@11<<"<<z@07<<std::endl;
16:         }
17:         std::cout<<x<<"<<y<<"<<z<<std::endl;
18:     }
19: }

```

Symbole	Ligne
x	01
z	02
x	03
y	03
x	07
z	07

Table des symboles : intuition

```

01: int x = 137;
02: int z = 42;
03: int function(int x, int y)
04: {
05:   std::cout<<x@03<<"<<y@03<<"<<z@02<<std::endl;
06:   {
07:     int x, z;
08:     z@07=y@03;
09:     x@07=z@07;
10:     {
11:       int y=x@07;
12:       {
13:         std::cout<<x@07<<"<<y@11<<"<<z@07<<std::endl;
14:       }
15:       std::cout<<x@07<<"<<y@11<<"<<z@07<<std::endl;
16:     }
17:     std::cout<<x@07<<"<<y@03<<"<<z@07<<std::endl;
18:   }
19: }

```

Symbole	Ligne
x	01
z	02
x	03
y	03
x	07
z	07

Table des symboles : intuition

```

01: int x = 137;
02: int z = 42;
03: int function(int x, int y)
04: {
05:     std::cout<<x@03<<"<<y@03<<"<<z@02<<std::endl;
06:     {
07:         int x, z;
08:         z@07=y@03;
09:         x@07=z@07;
10:         {
11:             int y=x@07;
12:             {
13:                 std::cout<<x@07<<"<<y@11<<"<<z@07<<std::endl;
14:             }
15:             std::cout<<x@07<<"<<y@11<<"<<z@07<<std::endl;
16:         }
17:         std::cout<<x@07<<"<<y@03<<"<<z@07<<std::endl;
18:     }
19: }

```

Symbole	Ligne
x	01
z	02
x	03
y	03

Table des symboles : intuition

```
01: int x = 137;
02: int z = 42;
03: int function(int x, int y)
04: {
05:     std::cout<<x@03<<"<<y@03<<"<<z@02<<std::endl;
06:     {
07:         int x, z;
08:         z@07=y@03;
09:         x@07=z@07;
10:         {
11:             int y=x@07;
12:             {
13:                 std::cout<<x@07<<"<<y@11<<"<<z@07<<std::endl;
14:             }
15:             std::cout<<x@07<<"<<y@11<<"<<z@07<<std::endl;
16:         }
17:         std::cout<<x@07<<"<<y@03<<"<<z@07<<std::endl;
18:     }
19: }
```

[illegible]

Remarque

- Dans l'exemple précédent
 - La table contient les déclarations de variables
 - Les numéros de ligne de déclaration
- De façon usuelle
 - La table contient tous les symboles (variables, fonctions...)
 - Le type de l'entité associée au symbole
 - D'autres informations utiles à la compilation...

Implémentation 1 : pile

- Une implémentation typique : une pile de tables de correspondance
- Chaque table correspond à un contexte (bloc, fonction...)
- La pile permet de facilement gérer l'entrée et la sortie d'un contexte
- Les opérations :
 - Entrée dans un contexte : push d'une nouvelle table
 - Sortie d'un contexte : pop de la table en sommet de pile
 - Insertion de symbole : ajout d'une entrée dans la table en sommet de pile
 - Recherche d'un symbole : parcours des tables depuis le sommet de la pile (du plus récent au plus ancien) pour rechercher la première occurrence du symbole

Utilisation de la table des symboles

- Pour traiter une portion de programme qui crée un contexte (bloc, déclaration de fonction, déclaration de classe...)
 - Entrer dans un nouveau contexte
 - Ajouter toutes les déclarations de variables dans la table
 - Traiter le corps du block / de la fonction / de la classe
 - Sortir du contexte
- Avec cette structure de données, la table n'est généralement pas conservée mais construite/détruite en cours de parcours de l'ASA
- *La plupart des analyses sémantiques sont définies comme une passe récursive de ce type sur l'ASA*

Une autre vision

```
01: int x;  
02: int y;  
03: int function(int x, int y)  
04: {  
05:   int w,z;  
06:   {  
07:     int y;  
08:   }  
09:   {  
10:     int w;  
11:   }  
12: }
```

Une autre vision

01: int x;

02: int y;

03: int function(int x, int y)

04: {

05: int w,z;

06: {

07: int y;

08: }

09: {

10: int w;

11: }


12: }

Global	
x	01
y	02

Une autre vision

```
01: int x;  
02: int y;  
03: int function(int x, int y)  
04: {  
05:   int w,z;  
06:   {  
07:     int y;  
08:   }  
09:   {  
10:     int w;  
11:   }  
12: }
```

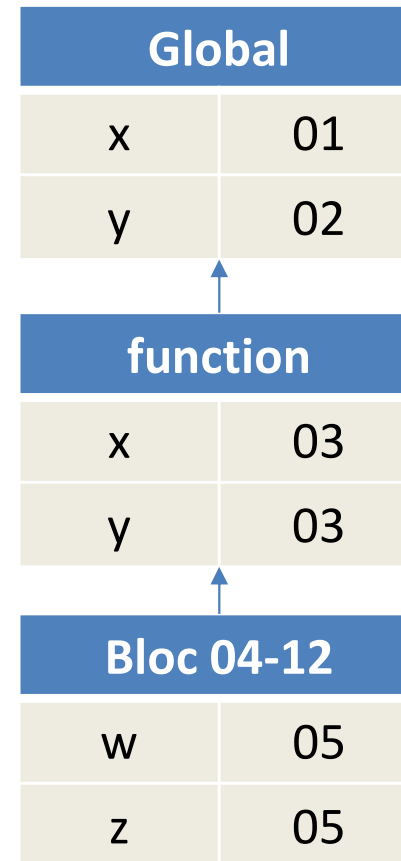
Global	
x	01
y	02



function	
x	03
y	03

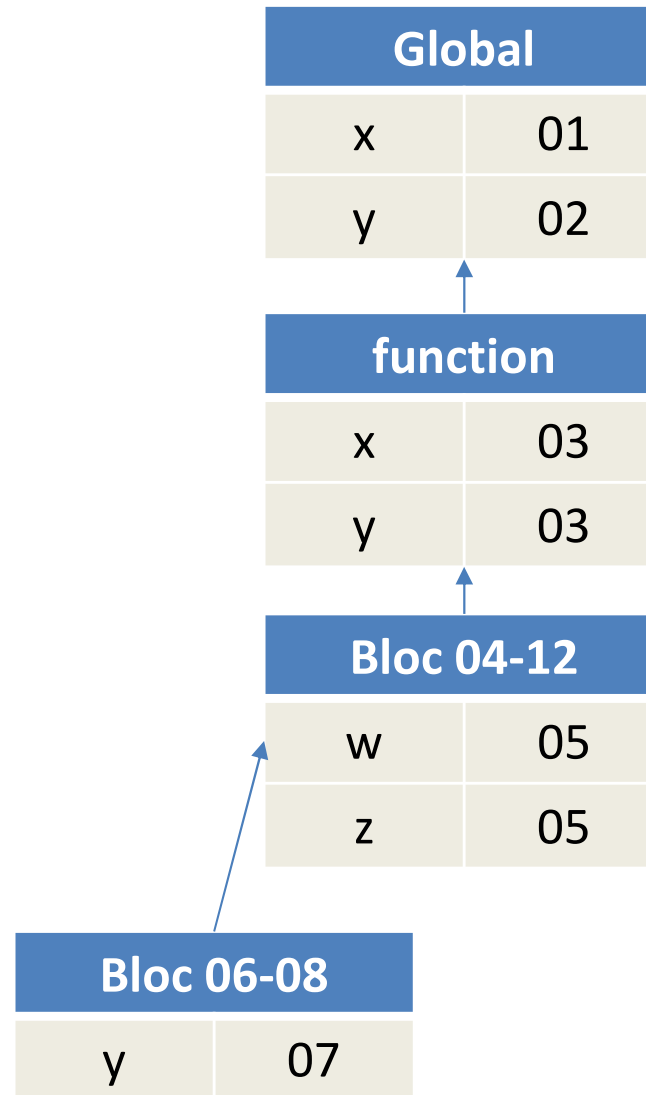
Une autre vision

```
01: int x;  
02: int y;  
03: int function(int x, int y)  
04: {  
05:   int w,z;  
06:   {  
07:     int y;  
08:   }  
09:   {  
10:     int w;  
11:   }  
12: }
```



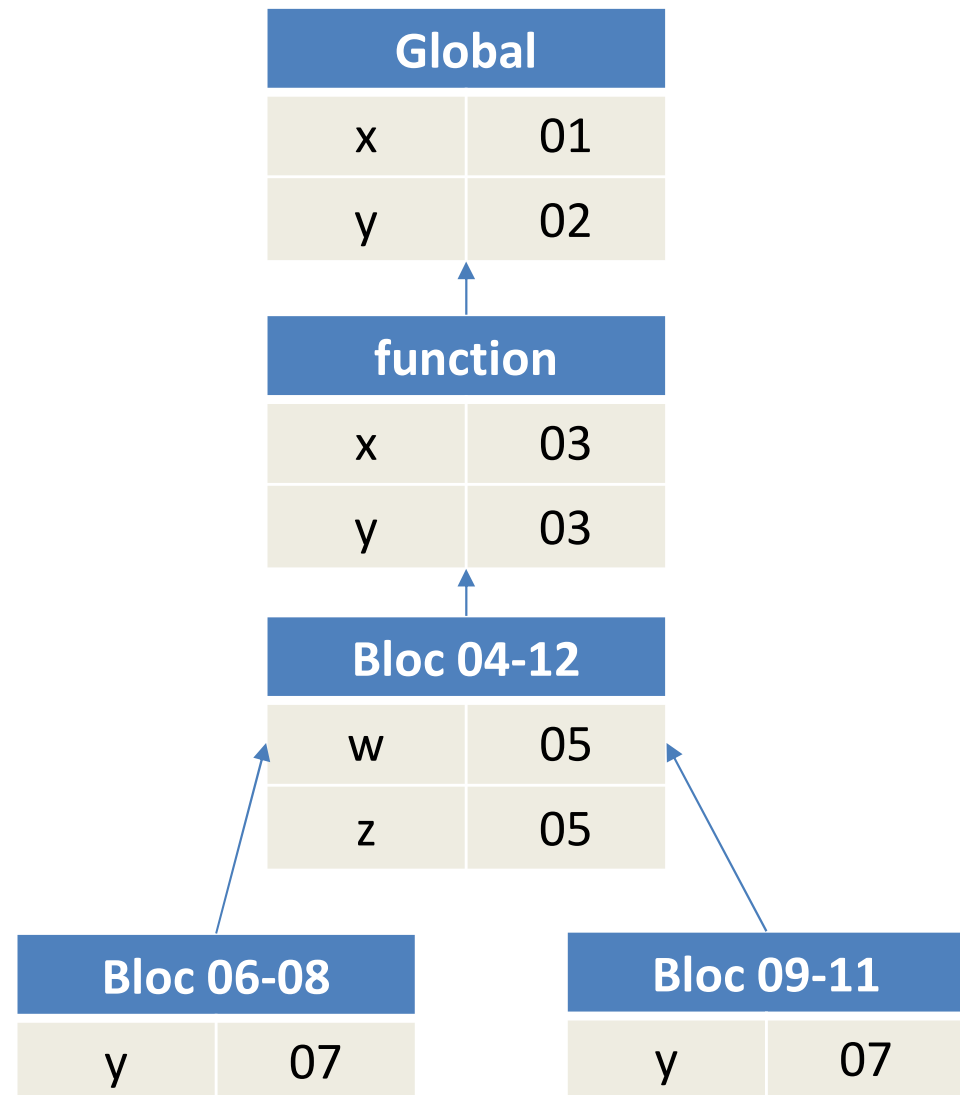
Une autre vision

```
01: int x;  
02: int y;  
03: int function(int x, int y)  
04: {  
05:   int w,z;  
06:   {  
07:     int y;  
08:   }  
09:   {  
10:     int w;  
11:   }  
12: }
```



Une autre vision

```
01: int x;  
02: int y;  
03: int function(int x, int y)  
04: {  
05:   int w,z;  
06:   {  
07:     int y;  
08:   }  
09:   {  
10:     int w;  
11:   }  
12: }
```



Implémentation 2 : spaghetti stacks

- La table des symboles est un arbre de tables de correspondances (associées à des contextes)
- Chaque contexte possède un pointeur sur le contexte englobant
- En chaque point du programme, la table des symboles apparaît comme une pile
 - Mêmes propriétés que la première implémentation
- *Mais : peut être stockée et calculée une et une seule fois!*