Analyse sémantique, grammaires attribuées 1

Mirabelle Nebut

Bureau 223 - extension M3
mirabelle.nebut at lifl.fr

2009-2010

Analyse sémantique - pourquoi ?

À l'issue de la compilation, le programme cible doit avoir la même sémantique que le programme source.

Correction syntaxique insuffisante.

Ex TP2 AVA : `3+true>2` syntaxiquement correct.

Ex TP2 AVA : `int x, x ;` syntaxiquement correct.

$\Rightarrow$ Nécessité d'une analyse sémantique.

Analyse sémantique - quoi ?

Dans le compilateur d'un langage de programmation :

- calcul d'un arbre abstrait
- décoré par des informations qui préparent la génération de code (ex : adresses mémoire)
- vérifications sémantiques : de type, des déclarations, etc.

Applications de traitement de données textuelles : calculs divers.

Ex : évaluation d'une expression, calcul d'un nombre de variables, de la hauteur d'un arbre, etc.

Analyse sémantique - comment ?

L'an. sémantique calcule des données (ex : types, valeurs)

⇒ comment exprimer ces calculs ?

Le plus naturel est de le faire en fonction de sa grammaire : on parle de traduction dirigée par la syntaxe.

⇒ On enrichit les grammaires algébriques par 2 concepts :

- données
- calculs

et on obtient une grammaire attribuée.

Grammaire attribuée, informellement

C'est une grammaire algébrique, plus :

- des attributs (les données) associés aux terminaux et non-terminaux ;
- des règles (ou actions sémantiques) associées aux productions (les calculs des données).

Exemple :

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

Exemple - expression arithmétique

Calcul de la valeur d'une expression arithmétique :

$$E \rightarrow E + T \mid T$$
$$T \rightarrow (E) \mid \texttt{i}$$

Comment aborder l'attribution de la grammaire ?

- Quelles données veut-on calculer $\Rightarrow$ quels attributs ?
- Comment les calculer $\Rightarrow$ quelles actions ?

Exemple - quelles données ?

Ex : pour le mot `i + i + i`, pour `i` représentant 3

Données = valeur :

- de l'expression elle-même : E ;
- de ses sous-expressions : T, `i`.

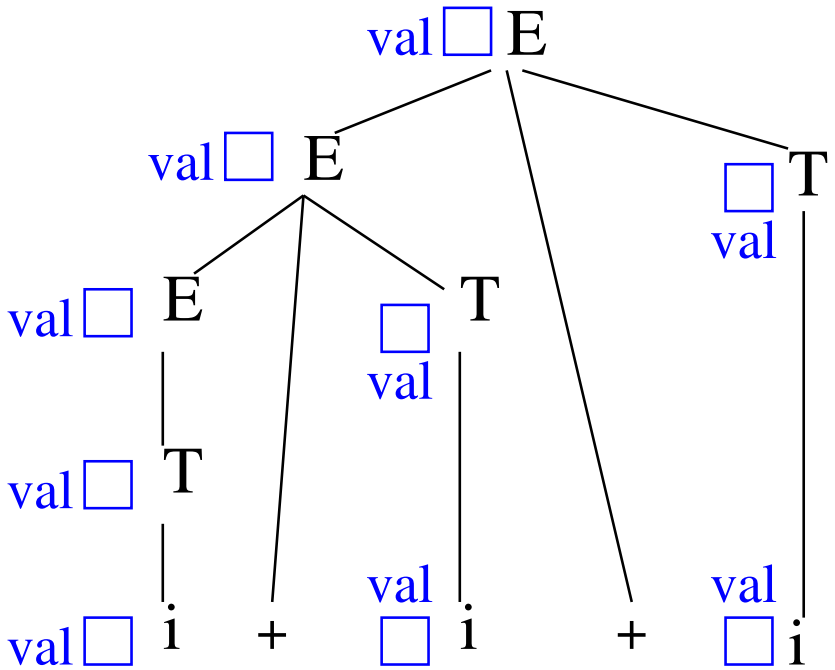
Attributs :

- *val* de type entier associé au non-terminal E, noté $E.val$;
- *val* de type entier associé au non-terminal T, noté $T.val$;
- *val* de type entier associé au terminal `i`, noté `i`.*val*.

Exemple - arbre décoré

Nœuds de l'arbre syntaxique décorés avec la valeur des attributs.

Autant de « cases » que l'analyse sémantique doit remplir.



Exemple - quels calculs ?

Additions, par ex pour $E \rightarrow E + T$:

$$E.val = E.val + T.val$$

On associe une action à la production en distinguant les différentes occurrences de E :

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

«Passage de valeurs» :

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

Exemple - au final

Qui fixe la valeur de l'attribut *val* du terminal `i` ?

$\Rightarrow$ elle est initialisée par l'analyseur lexical.

Au final on obtient la grammaire attribuée :

$$
\begin{array}{ll}
E \rightarrow E + T & \{ E_0.val = E_1.val + T.val \} \\
E \rightarrow T & \{ E.val = T.val \} \\
T \rightarrow (E) & \{ T.val = E.val \} \\
T \rightarrow \mathtt{i} & \{ T.val = \mathtt{i}.val \}
\end{array}
$$

Et si la grammaire était ambiguë ?

On obtiendrait la même valeur pour les 2 arbres admis par

`i + i + i`...

... mais considérons la grammaire :

$$E \longrightarrow E + E \mid E * E \mid \texttt{i}$$

Le mot `i + i * i` a deux significations, E.`val` a 2 valeurs possibles.

$\Rightarrow$ attribuer une grammaire ambiguë n'a aucun sens pratique.

Exemple - remarques

Cette grammaire spécifie comment calculer des valeurs associées à ses symboles.

Mais elle ne dit pas quand effectuer ces calculs. . .

. . . ni dans quel ordre effectuer les actions.

Grammaire attribuée = formalisme de spécification, pas d'exécution.

Au sens strict, actions = équations, pas affectations.

Pour rajouter une sémantique opérationnelle, on introduit 2 types d'attributs : les synthétisés et les hérités.

Attributs synthétisés - exemple

$$
\begin{array}{ll}
E \rightarrow E + T & \{\ E_0.val = E_1.val + T.val\ \} \\
E \rightarrow T & \{\ E.val = T.val\ \} \\
T \rightarrow (\ E\ ) & \{\ T.val = E.val\ \} \\
T \rightarrow \mathtt{i} & \{\ T.val = \mathtt{i}.val\ \}
\end{array}
$$

Ces actions respectent le schéma :

```
val_att_gauche_prod = f (val_att_droite_prod)
```

Pour les symboles non-terminaux :

- en partie gauche de production : occurrences de définition des attributs
- en partie droite de production : occurrences d'utilisation des attributs

Attributs synthétisés - exemple

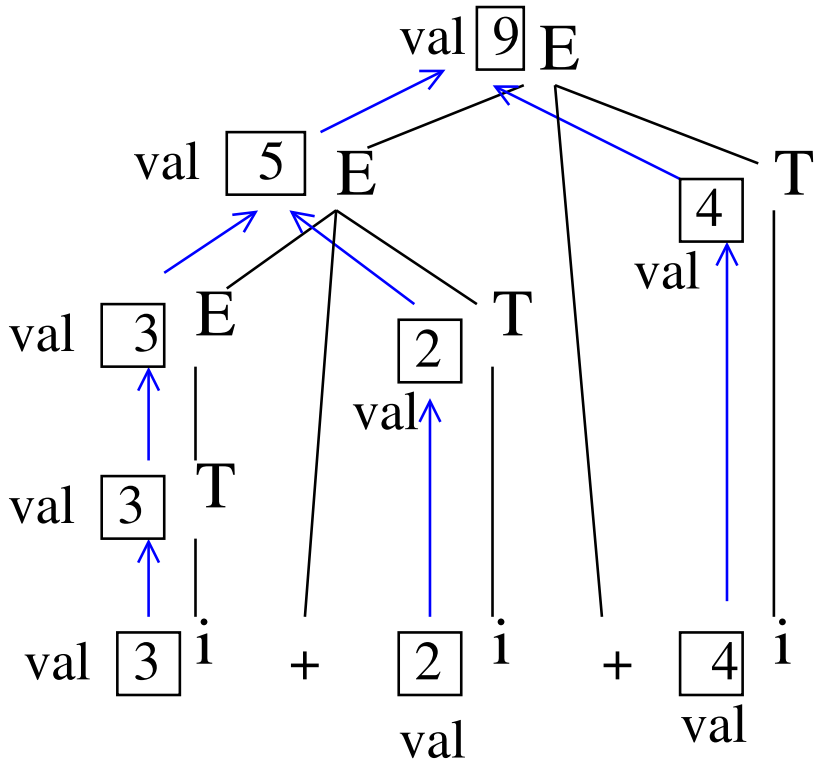
Le schéma : `val_att_gauche_prod = f (val_att_droite_prod)` correspond à un attribut synthétisé :

- défini quand associé à un non-terminal en partie gauche de production ;
- utilisé quand associé à un non-terminal ou terminal en partie droite de production.

Exception : attribut du terminal `i`.*val* :

- valeur fixée par analyseur lexical ;
- n'apparaît qu'en partie droite (utilisation seule) ;
- dit synthétisé par convention.

Attributs synthétisés - exemple



Les attributs synthétisés « remontent » dans l'arbre décoré.

Attributs synthétisés - cas général

L'attribut $X.a$ est synthétisé si X apparaît en partie gauche de production et si la valeur de $X.a$ est calculée en fonction de la valeur d'attributs associés à des symboles apparaissant en partie droite de production.

$X \rightarrow X_1 X_2 \ldots X_n \{ X.a = f(X_1.x_1, \ldots, X_n.x_n) \} \qquad X_i \in V_T \cup V_N$

Toute production doit calculer la valeur des attributs synthétisés de son membre gauche (via une action).

Attributs synthétisés - remarque

Calcul du nombre n pour un mot $a^n b^n$: attribut nb de type entier associé à S, synthétisé.

$$S \rightarrow aSb \quad \{ S_0.nb = S_1.nb + 1 \}$$
$$S \rightarrow \epsilon \quad \{ S.nb = 0 \}$$

0 (calcul cas vide) est l'élément neutre de l'addition (calcul cas général).

Ce schéma se produit souvent.

Exemple : contrôle de type pour *Init* étendu

En vue du contrôle de type, on souhaite associer aux identificateurs de `Init` leur type (`Type.ENTIER` ou `Type.LISTE`).

Ex : `int x,y ; list z ;`

decl $\rightarrow$ DECLINT *listeId* `PV` | DECLLIST *listIdent* `PV`
listIdent $\rightarrow$ IDENT | IDENT `SEP` *listeId*

NB : L'attribut *type* de IDENT peut/doit-il être fixé par l'analyseur lexical/syntaxique ?

NB : IDENT d'une part et DECLINT, DECLLIST d'autre part ne sont pas dans les mêmes sous-arbres : contexte différent.

$\Rightarrow$ A-t-on toujours envie de faire remonter l'information ?

Attributs hérités : exemple

Attributs :

- *type* de type `Type` associé au terminal `IDENT` ;
- *type* de type `Type` associé au non-terminal *listeId*.

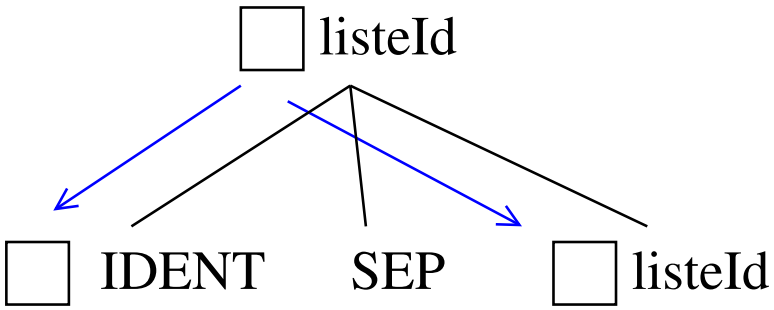
decl → DECLINT *listeId* `PV`	{ *listeId.type* = `Type.ENTIER`}
decl → DECLLIST *listeId* `PV`	{ *listeId.type* = `Type.LISTE`}
listeId → IDENT	{ IDENT.*type* = *listeId.type* }
listeId → IDENT `SEP` *listeId*	{ IDENT.*type* = $listeId_0.type$ $listeId_1.type = listeId_0.type$ }

occurrences de définition occurrences d'utilisation

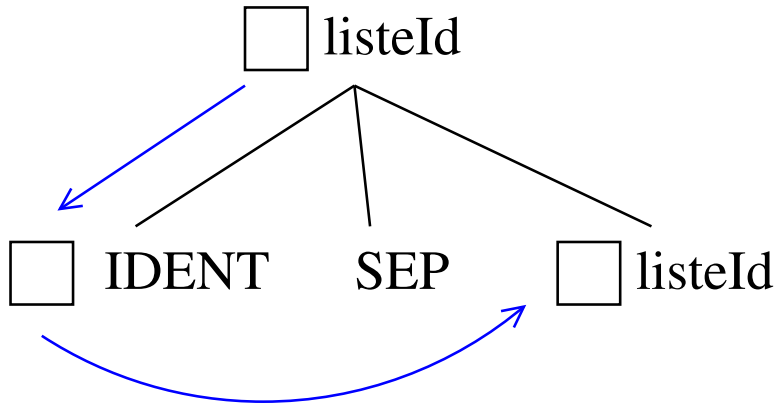
Les attributs *type* sont hérités pour IDENT et *listeId*.

Attributs hérités : exemple

Deux écritures possibles :



$\text{IDENT}.type = listeId_0.type$
$listeId_1.type = listeId_0.type$



$\text{IDENT}.type = listeId_0.type$
$listeId_1.type = \text{IDENT}.type$

Les attributs hérités « descendent » du père, ou viennent des frères de gauche ou des frères de droite.

Attributs hérités - cas général

L'attribut $Y.a$ est hérité si Y apparaît en partie droite et si la valeur de $Y.a$ est calculée en fonction de la valeur d'attributs associés à des symboles apparaissant en partie gauche et/ou des autres symboles apparaissant en partie droite.

$$X \rightarrow X_1 \ldots Y \ldots X_n \{ Y.a = f(X.x, X_1.x_1, \ldots, X_n.x_n) \}$$

pour $Y, X_i \in V_T \cup V_N$.

Toute production doit calculer la valeur des attributs hérités des symboles en partie droite (via une action).

Attributs hérités - cas de l'axiome

L'axiome ne peut avoir d'attributs hérités.

En effet l'axiome « à la racine » n'a ni père ni frères.

Au besoin, on rajoute un axiome bidon.

Ex : attribut nb de type entier associé à S, hérité

$S \rightarrow aaSb \quad \{ S_1.nb = S_0.nb + 2 \}$

$S \rightarrow \epsilon \quad \{ \text{afficher}(S.nb) \}$

$S' \rightarrow S \quad \{ S.nb = 0 \}$

Attributs synthétisés et hérités

Dans le cas général, on mélange les deux types d'attributs.

$decl \rightarrow$ DECLINT $listeId$ PV	$\{ listeId.type = \texttt{Type.ENTIER}\}$
$decl \rightarrow$ DECLLIST $listeId$ 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	$\{ exprAffect.type = \text{IDENT}.type \}$

Se passer des att. hérités : grammaire S-attribuée

Pour toute grammaire attribuée GA, il existe une grammaire attribuée GA' :

- qui engendre le même langage ;
- calculant les mêmes valeurs d'attributs ;

et qui possède uniquement des attributs synthétisés.

Une grammaire attribuée ne comportant que des attributs synthétisés est dite S-attribuée.

Comment faire pour les types de INIT ?

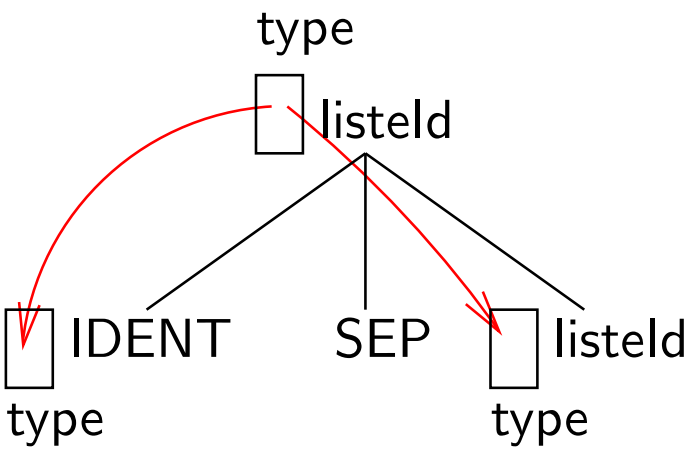
Ordre d'évaluation et graphe de dépendances - 1

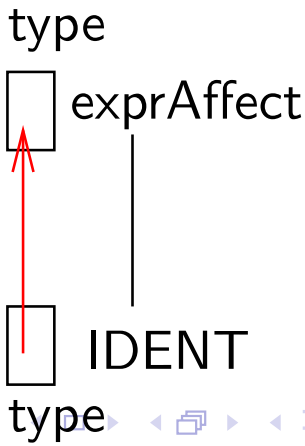
Dans quel ordre évaluer les valeurs des attributs ?

- Dépendances de données ds les actions (" = " lu comme " :="") ;
- $\Rightarrow$ construction d'un graphe de dépendances.

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

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





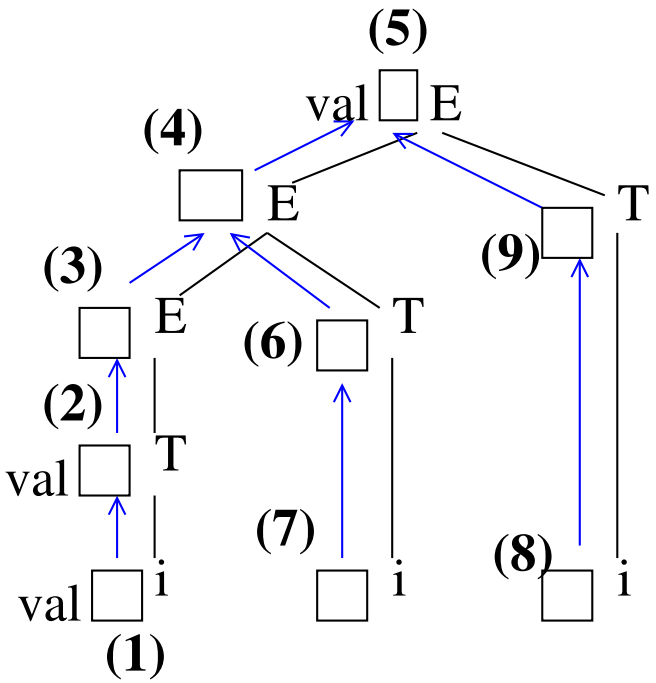
Ordre d'évaluation et graphe de dépendances - 2

Si ce graphe est cyclique : grammaire mal formée.

Sinon (grammaire bien formée) : on peut trouver un ordre d'évaluation, une numérotation des actions.

On ne verra pas comment dans ce cours : seulement des exemples avec numérotation de nœuds (**(i)** dans les schémas).

Ordre d'évaluation et graphe de dépendances - ex 1

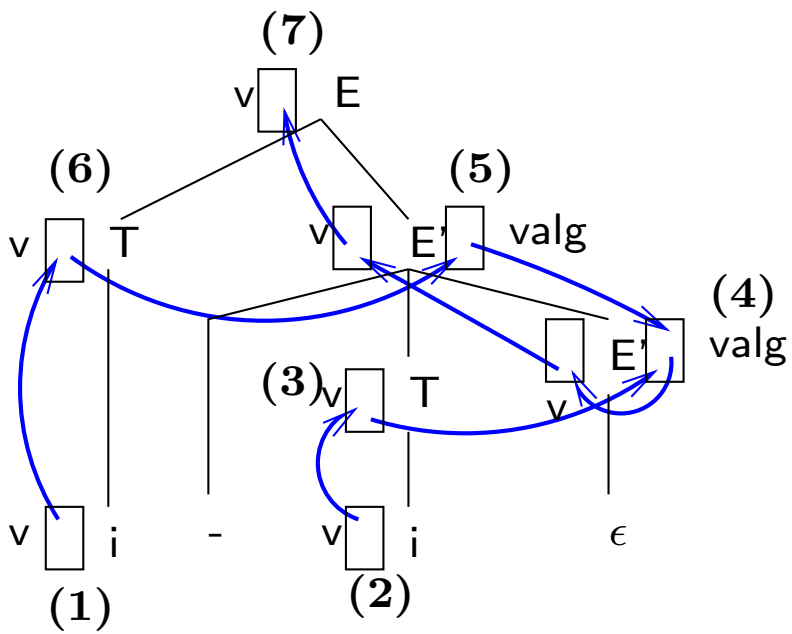


$$E \rightarrow E + T \quad \{ E_0.val = E_1.val + T.val \}$$
$$E \rightarrow T \quad \{ E.val = T.val \}$$
$$T \rightarrow (E) \quad \{ T.val = E.val \}$$
$$T \rightarrow \mathtt{i} \quad \{ T.val = \mathtt{i}.val \}$$

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)

Ordre d'évaluation et graphe de dépendances - ex 2



$E \rightarrow TE'$ { $E'.\text{valg} = \text{f}(T.\text{v})$; $E.\text{v} = \text{f}(E'.\text{v})$ }

$E' \rightarrow -TE'$ { $E'_1.\text{valg} = \text{f}(E'_0.\text{valg},\ T.\text{v})$; $E'_0.\text{v} = \text{f}(E'_1.\text{v})$ }

$E' \rightarrow \epsilon$ { $E'.\text{v} = \text{f}(E'.\text{valg})$ }

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

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

Problème et solutions

Problème :

- détection des dépendances cycliques pour un arbre syntaxique donné ;
- en cas d'absence de dépendances cycliques : calcul d'un ordre.

Deux grandes approches :

- utilisation d'un évaluateur d'attributs ;
- restriction des grammaires attribuées pour garantir l'absence de dépendances cycliques.

Évaluateurs d'attributs

Outil qui prend en entrée une grammaire attribuée :

- règles sémantiques écrites dans le format de l'outil, plus ou moins expressif ;
- sait détecter les dépendances cycliques au moyen de graphes de dépendances ;
- sait trouver d'après ces graphes un ordre d'évaluation pour les règles sémantiques.

Deux types d'évaluateurs :

- évaluateurs statiques ;
- évaluateurs dynamiques.

Évaluateurs dynamiques

- prennent en entrée une grammaire attribuée et un mot à analyser ;
- construisent à l'exécution un graphe de dépendance pour le mot à traiter ;
- s'il ne contient pas de cycle en déduisent un ordre d'exécution des règles pour ce mot ;
- exécution des règles dans la foulée.

Un évaluateur dynamique ne sait pas dire si une grammaire est bien formée.

Évaluateurs statiques

- prennent en entrée une grammaire attribuée ;
- analysent la grammaire en considérant tous les graphes de dépendances associés à n'importe quel arbre syntaxique ;
- en déduisent si la grammaire est bien formée ;
- si c'est le cas déterminent statiquement (avant exécution) un ordre d'évaluation des règles ;
- génèrent un module d'analyse sémantique qui prend en entrée le mot à analyser.

Beaucoup plus puissants que les évaluateurs dynamiques.

Inconvénients

Pour trouver un ordre d'évaluation + évaluer les attributs :

- construire un arbre syntaxique + graphe de dépendances ;
- parcourir (éventuellement plusieurs fois) les nœuds de l'arbre.

Or les an. synt. ne construisent pas un arbre syntaxique :

- ils construisent une dérivation ;
- l'arbre syntaxique est implicitement parcouru.

La sémantique du langage étant décrite en terme de sa syntaxe, on peut aller plus loin :

- guider l'analyse sémantique par l'analyse syntaxique ;
- effectuer les règles sémantiques lors de la construction de la dérivation (pendant l'analyse syntaxique).

Avantages et inconvénients

Pour une analyse sémantique faite en parallèle de l'analyse syntaxique :

- - l'évaluation des attributs doit pouvoir suivre le parcours de l'arbre imposé par l'analyse syntaxique :
  - analyse descendante : parcours préfixe ;
  - analyse ascendante : parcours postfixe.
- - il faut donc restreindre la forme des grammaires attribuées.

- + par construction il n'y aura pas de cycle dans les dépendances ;
- + pas besoin de construire un arbre syntaxique ;
- + par construction un seul passage par nœud.

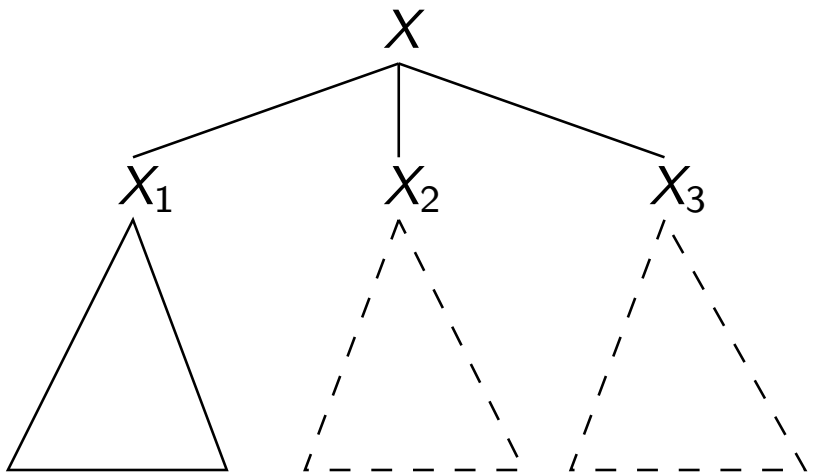
Restrictions posées aux grammaires attribuées

Analyse descendante

Parcours de l'arbre en ordre préfixe, en profondeur d'abord :

père puis fils de gauche à droite, récursivement

Soit la production $X \rightarrow X_1 X_2 X_3$, et la pile $\boxed{\begin{array}{c} X_2 \\ X_3 \end{array}}$



Expansion pour X_2

Restrictions posées aux grammaires attribuées

Analyse descendante

$X_2 \rightarrow \ldots \{X_2.val = \ldots\}$

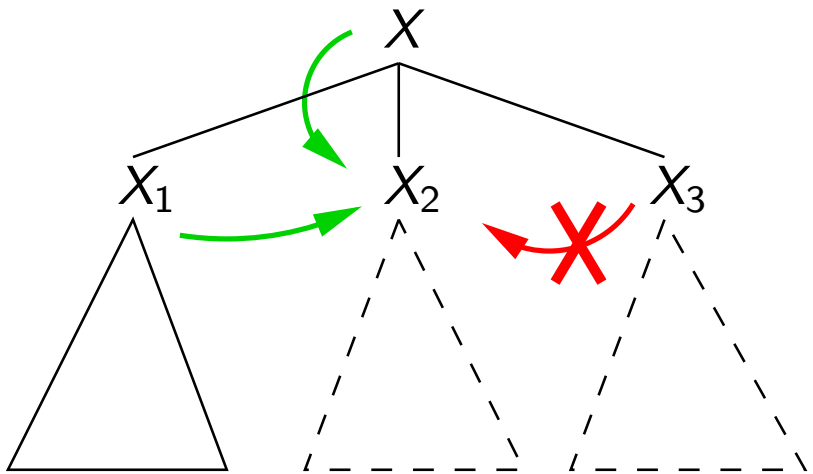
X_2 doit par définition :

- calculer ses attributs synthétisés ;
- avoir à sa disposition ses attributs hérités.

Restrictions posées aux grammaires attribuées

Analyse descendante et attribués hérités

Les attributs hérités de X_2 ne peuvent venir de ses frères de droite :



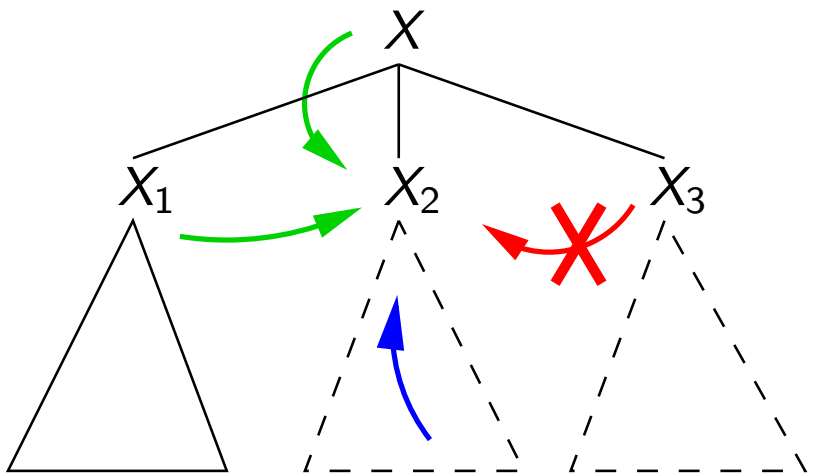
$\Rightarrow$ restriction sur les dépendances liées au calcul des attributs hérités.

Restrictions posées aux grammaires attribuées

Analyse descendante et attributs synthétisés

Les attributs synthétisés de X_2 sont calculés en fonction :

- de ses attributs hérités ;
- des attributs calculés dans le sous-arbre dont il est racine.



$\Rightarrow$ pas de restriction sur les dépendances liées au calcul des attributs synthétisés.

Restriction : grammaire L-attribuée

Pour permettre une évaluation des attributs lors de l'analyse descendante, une grammaire attribuée doit être L-attribuée : pour toute production

$$X \rightarrow X_1 X_2 \dots X_n$$

Les attributs hérités de X_i $(1 \leq i \leq n)$ dépendent uniquement de la valeur :

- d'attributs hérités du père X ;
- d'attributs des frères X_j, avec $j \leq i$

= pas de dépendances de la droite vers la gauche.

Exemple de grammaire L-attribuée

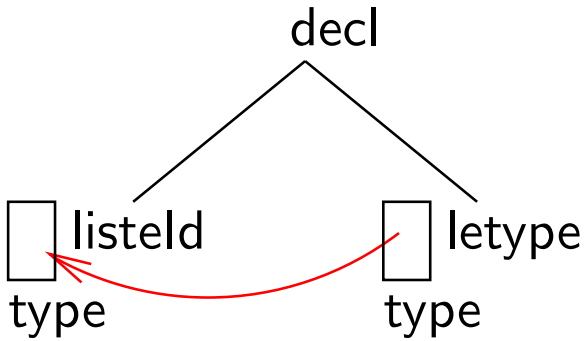
Beaucoup de grammaires sont « naturellement » L-attribuées.

$decl \rightarrow$ DECLINT $listeId$ PV	$\{ listeId.type = \texttt{Type.ENTIER}\}$
$decl \rightarrow$ DECLLIST $listeId$ PV	$\{ listeId.type = \texttt{Type.LISTE}\}$
$listeId \rightarrow$ IDENT	$\{$ IDENT$.type = listeId.type$ $\}$
$listeId \rightarrow$ IDENT SEP $listeId$	$\{$ 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	$\{ exprAffect.type =$ IDENT$.type \}$

Exemple de grammaire non L-attribuée

(pour un autre langage, type Pascal)

$decl \longrightarrow listeId\ letype \quad \{\ listeId.type = letype.type\ \}$
$letype \longrightarrow \text{DECLINT} \quad \{\ letype.type = \text{Type.ENTIER}\ \}$
$letype \longrightarrow \text{DECLLIST} \quad \{\ letype.type = \text{Type.LISTE}\ \}$
$listeId \longrightarrow \ldots$



Grammaire LL(1) attribuée

Grammaire L-attribuée et LL(1) : LL-attribuée.

Exemple des expressions arithmétiques avec addition.

Grammaire non ambiguë, mais non LL(1)

$$E \rightarrow E - T \quad \{ E_0.\mathrm{v} = E_1.\mathrm{v} - T.\mathrm{v} \}$$
$$E \rightarrow T \quad \{ E.\mathrm{v} = T.\mathrm{v} \}$$
$$T \rightarrow \mathtt{i} \quad \{ T.\mathrm{v} = \mathtt{i}.\mathrm{v} \}$$

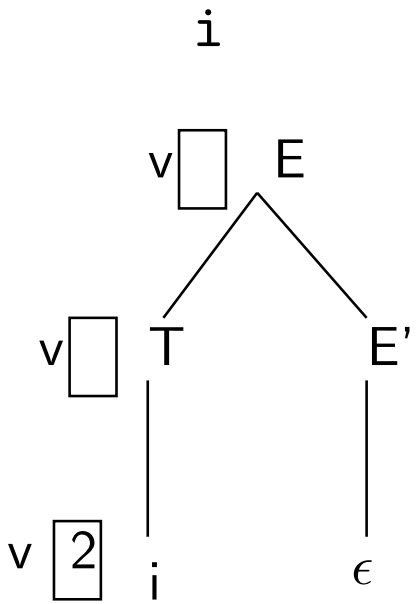
Grammaire LL(1) équivalente :

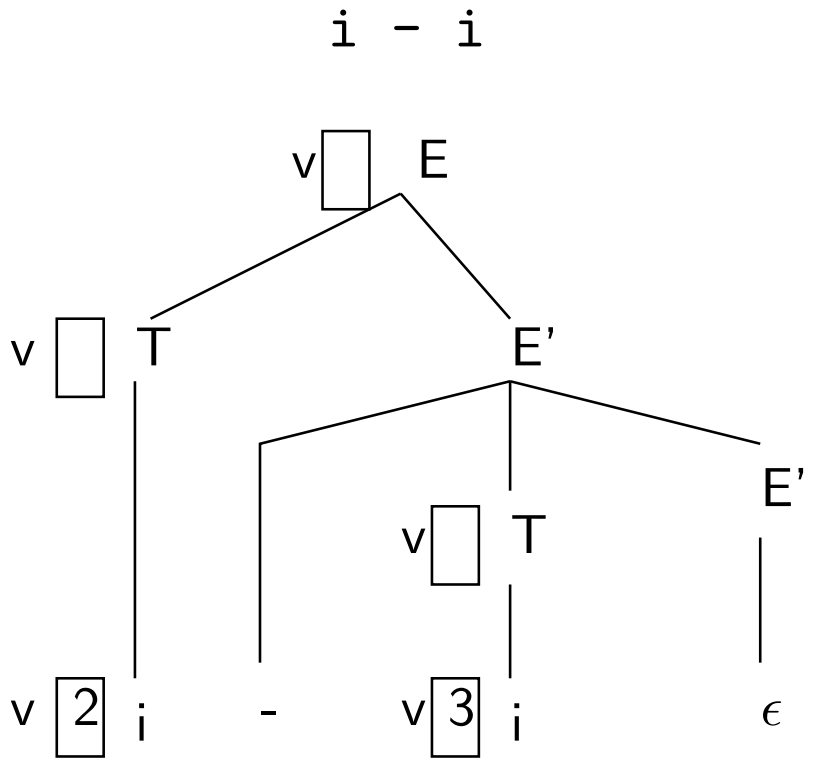
$$E \rightarrow TE'$$
$$E' \rightarrow -TE' \mid \epsilon$$
$$T \rightarrow \mathtt{i}$$

Comment aborder l'attribution :

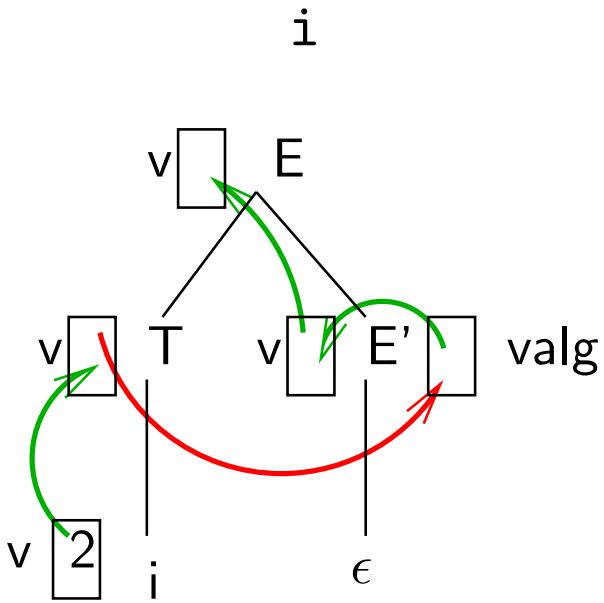
- Évident : pour i, E et T : attribut v synthétisé, type entier.
- Mais quels attributs pour E' ?

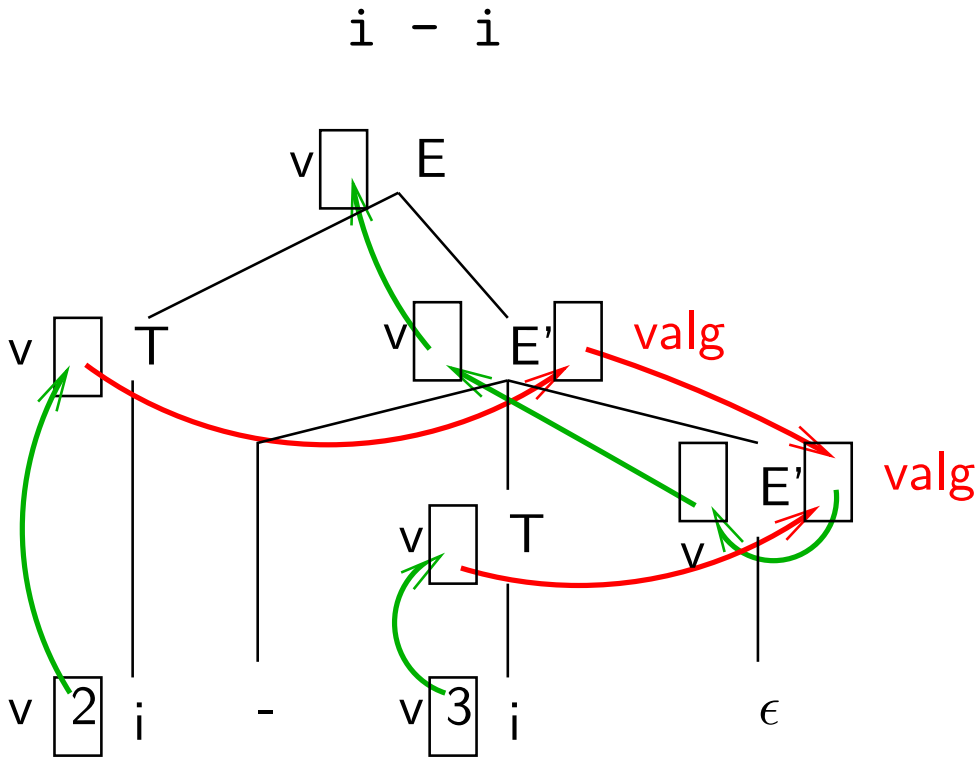
Grammaire LL(1)-attribuée - exemple d'arbres





Grammaire LL(1)-attribuée - ex avec dépendances





v attribut synthétisé de E, T, i et E'

valg attribut hérité de E'

Grammaire LL(1)-attribuée - exemple avec actions

$$
\begin{array}{ll}
E \rightarrow TE' & \{\ E'.\text{valg} = T.\text{v} \\
 & \quad E.\text{v} = E'.\text{v} \quad \} \\
E' \rightarrow -TE' & \{\ E'_1.\text{valg} = E'_0.\text{valg} - T.\text{v} \\
 & \quad E'_0.\text{v} = E'_1.\text{v} \quad \} \\
E' \rightarrow \epsilon & \{\ E'.\text{v} = E'.\text{valg}\ \} \\
T \rightarrow \texttt{i} & \{\ T.\text{v} = \texttt{i}.\text{v}\ \}
\end{array}
$$

La valeur de $E'.\text{valg}$ dépend de valeurs :

- de l'attribut hérité *valg* du père E' ;
- et éventuellement de l'attribut *v* du frère gauche T

$\Rightarrow$ la grammaire est LL-attribuée.

Implantation dans un analyseur récursif

Schéma de traduction

Peut-être plus facile à expliquer en terme de schéma de traduction (on ne les utilisera pas en TD/TP).

On indique quand les actions ont lieu en partie droite :

- pour une action qui calcule une valeur héritée pour X._ : placée avant X ;
- pour une action qui calcule ou utilise une valeur synthétisée de X._ : placée après X.

$E \rightarrow T \ \{ \ E'.\text{valg} = T.\text{v} \ \} \ E' \ \{ \ E.\text{v} = E'.\text{v} \ \}$
$E' \rightarrow - T \ \{ \ E'_1.\text{valg} = E'_0.\text{valg - } T.\text{v} \ \} \ E' \ \{ \ E'_0.\text{v} = E'_1.\text{v} \ \}$
$E' \rightarrow \epsilon \ \{ \ E'.\text{v} = E'.\text{valg} \ \}$
$T \rightarrow \texttt{i} \ \{ \ T.\text{v} = \texttt{i}.\text{v} \ \}$

Implantation dans un analyseur récursif

Schéma de traduction

Pour la production $X \longrightarrow AbC$:

$X \longrightarrow$ { calcul Her(A) } A { calcul Synth(A) }
b { calcul Synth(b) }
{ calcul Her(C) } C { calcul Synth(C) }
{ calcul Synth(X) }

Implantation dans un analyseur récursif

Rappel : 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 $Her(X) = \{h_1^X, \ldots, h_n^X\}$ sont des entrées de X ;
- les attributs synthétisés $Synth(X) = \{s_1^X, \ldots, s_m^X\}$ sont des sorties de X.

Implantation dans un analyseur récursif

h_i^X attribut hérité de X, entrée de `X()`
s_j^X attribut synthétisé de X, calculé par `X()`

Dans un langage à la Pascal :

```
X(in h1X, ..., in hnX, out s1X, ..., out smX)
```

En Java :

```
SynthX X(h1X, ..., hnX)
```

avec `SynthX` type objet regroupant les attributs de $Synth(X)$.

Implantation dans un analyseur récursif : exemple

$E' \longrightarrow -TE' \mid \epsilon$

`public int T() throws ...`

```
public int Eprime(int valg) throws ... {
  if (courant == Type.Moins) {
    // E' -> - T E'
    this.consommer(Type.Moins);
    int vt = T();
    int vep = EPrime(vt - valg);
    return vep;
  } else if (courant == Type.EOF)
    // E' ->
   return valg;
}
```

Implantation dans un analyseur récursif : exemple

$T \longrightarrow$ id

```
public int T() throws ... {
  if (courant == Type.Id) {
    // T -> i
    int vi = this.courant.getValue(); // manque cast
    this.consommer(Type.Id);
    return vi;
  } else throw ...
}
```

Attention à bien récupérer les attributs d'un terminal avant de bouger la tête de lecture.

Implantation dans un analyseur récursif : cas général

production $X \rightarrow Ab$

```
public SynthX X(h1X, ..., hnX) {
  ...
  eval des Her(A); // en fonction de Her(X)
  SynthA synth_A = A(h1A, ..., hnA);

  if (courant == Type.b) {
    Synthb synth_b = (Synthb) courant.getValue();
    this.consommer(b);
  } else throw ... ;

  eval des Synth(X); // en fonction de Her(X),
                     // Synth(A), Synth(b)
  return Synth(X);
}
```

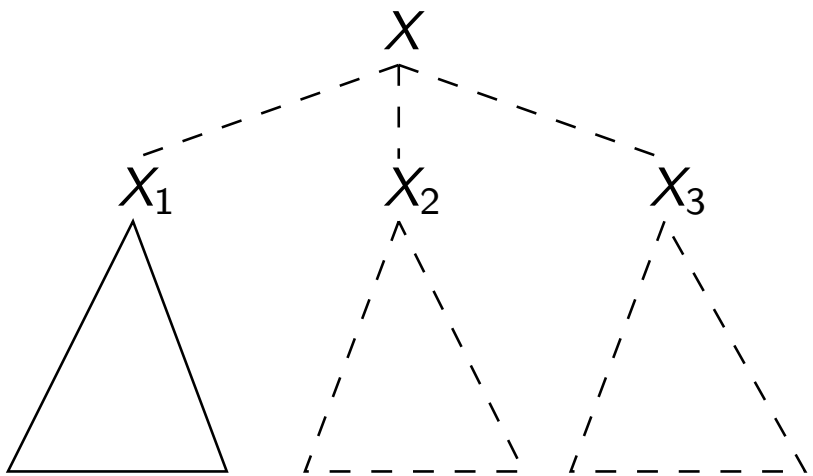
Restrictions posées aux grammaires attribuées

Analyse ascendante

Analyse ascendante : parcours de l'arbre en ordre postfixe :

fils de gauche à droite, puis père

Pour l'item $[X \rightarrow X_1 \bullet X_2 X_3]$:



Expansion pour X_2

Restrictions posées aux grammaires attribuées

Analyse ascendante

$X_2 \rightarrow \ldots \{X_2.val = \ldots\}$

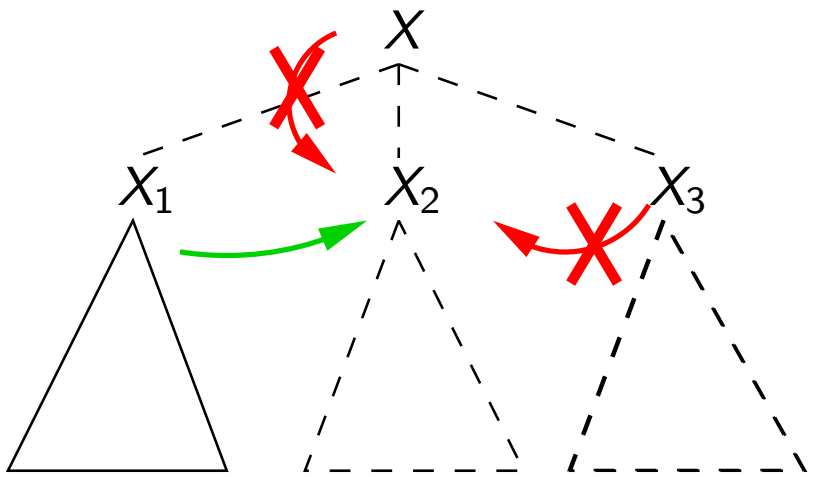
X_2 doit par par définition :

- calculer ses attributs synthétisés ;
- avoir à sa disposition ses attributs hérités ;

Restrictions posées aux grammaires attribuées

Analyse ascendante et attributs hérités

Les attributs hérités de X_2 ne peuvent venir que de ses frères de gauche (père et frères de droite pas construits) :



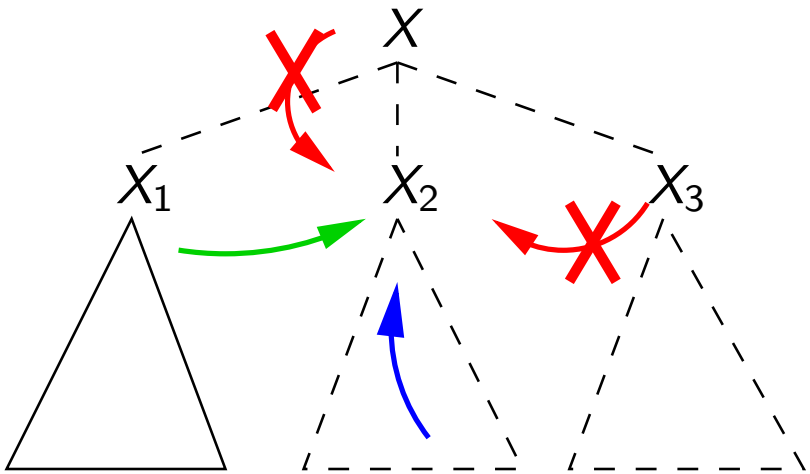
$\Rightarrow$ restrictions posées sur les dépendances liées au calcul des attributs hérités.

Restrictions posées aux grammaires attribuées

Analyse ascendante et attributs synthétisés

Les attributs synthétisés de X_2 sont calculés en fonction :

- de ses attributs hérités ;
- des attributs calculés dans le sous-arbre dont il est racine.



$\Rightarrow$ Pas de restriction pour les attributs synthétisés.

Restrictions posées aux grammaires attribuées

Analyse ascendante, en TP

En TP on utilisera des grammaires S-attribuées.

N'utilisent pas d'attributs hérités, donc conviennent pour une analyse ascendante.

En pratique certaines grammaires L-attribuées sont utilisables.

Quand une grammaire n'est pas S-attribuée

Cette grammaire n'est pas S-attribuée.

Attribut *type* hérité pour `listeId`.

$decl \rightarrow$ DECLINT $listeId$ PV	$\{ listeId.type = \texttt{Type.ENTIER}\}$
$decl \rightarrow$ DECLLIST $listeId$ PV	$\{ listeId.type = \texttt{Type.LIST}\}$
$listeId \rightarrow$ IDENT	$\{$ setType(IDENT.nom,$listeId.type$) $\}$
$listeId \rightarrow$ IDENT SEP $listeId$	$\{$ setType(IDENT.nom,$listeId_0.type$) $listeId_1.type = listeId_0.type$ $\}$

Supprimer les attributs hérités : solution 1

On collecte l'information et on la fait remonter jusqu'à savoir la traiter.

listeId → `IDENT` { *listeId*.*l* = `new MaListe(IDENT.`*nom*`)` }

listeId → `IDENT SEP` *listeId*
{ *listeId*$_0$.l = *listeId*$_1$.*l*`.ajoutTete(IDENT.`*nom*`)` }

decl → `DECLINT` *listeId* `PV`
{ itérer sur `it` = *listeId*.*l*`.iterator()`
`setType(it.next(), Type.ENTIER)`}

decl → `DECLLIST` *listeId* `PV`
{ itérer sur `it` = *listeId*.*l*`.iterator()`
`setType(it.next(), Type.LISTE)`}

Supprimer les attributs hérités : solution 2

On modifie la grammaire en gardant une grammaire équivalente (et on repasse les tests de l'analyseur syntaxique).

decl → DECLLIST *listeIdList* PV
decl → DECLINT *listeIdInt* PV
listeIdInt → IDENT { setType(IDENT.nom, Type.ENTIER) }
listeIdInt → IDENT SEP *listeIdInt*
{ setType(IDENT.*nom*, Type.ENTIER) }
listeIdList → IDENT { setType(IDENT.*nom*, Type.LISTE) }
listeIdList → IDENT SEP *listeIdList*
{ setType(IDENT.*nom*, Type.LISTE) }

Cup et les grammaires attribuées

Attributs

Un seul attribut par symbole (utiliser des types objets pour regrouper plusieurs attributs au besoin).

Pour déclarer un attribut, on donne uniquement son type (objet) :

```
terminal Integer ENTIER;
non terminal Integer expr;
```

L'attribut du non-terminal membre gauche est par convention `RESULT`.

Cup et les grammaires attribuées

Actions

Les actions sémantiques sont écrites en Java.

Elles sont placées :

- entre `{: :}`
- en fin de production avant le ;
- mais avant un %prec.

Elles sont exécutées lors de la réduction de la production.

Cup et les grammaires attribuées

Actions et accès aux attributs

Accès aux attributs par :, nommage local à la production.

```
expr ::= ENTIER:val {: RESULT = val; :}
       | MOINS expr:val
         {: RESULT = new Integer(- val.intValue()); :}
         %prec MOINSU
       | expr:val1 MOINS expr:val2
         {: RESULT = ... val1 ... val2; :}
```

Cup et les grammaires attribuées

Attributs de l'axiome

L'analyseur syntaxique décoré fournit les attributs de l'axiome.

En supposant que l'axiome de la grammaire pour `Ava` possède un attribut de type `Programme` :

```
import java_cup.runtime.Symbol;
...
ParserAva parser = new ParserAva(scanner);
...
Symbol s = parser.parse();
Programme p = (Programme) s.value;
...
```

Quel que soit le type d'analyse...

En effectuant l'analyse sémantique pendant l'analyse syntaxique :

- analyseur syntaxique difficile à lire ;

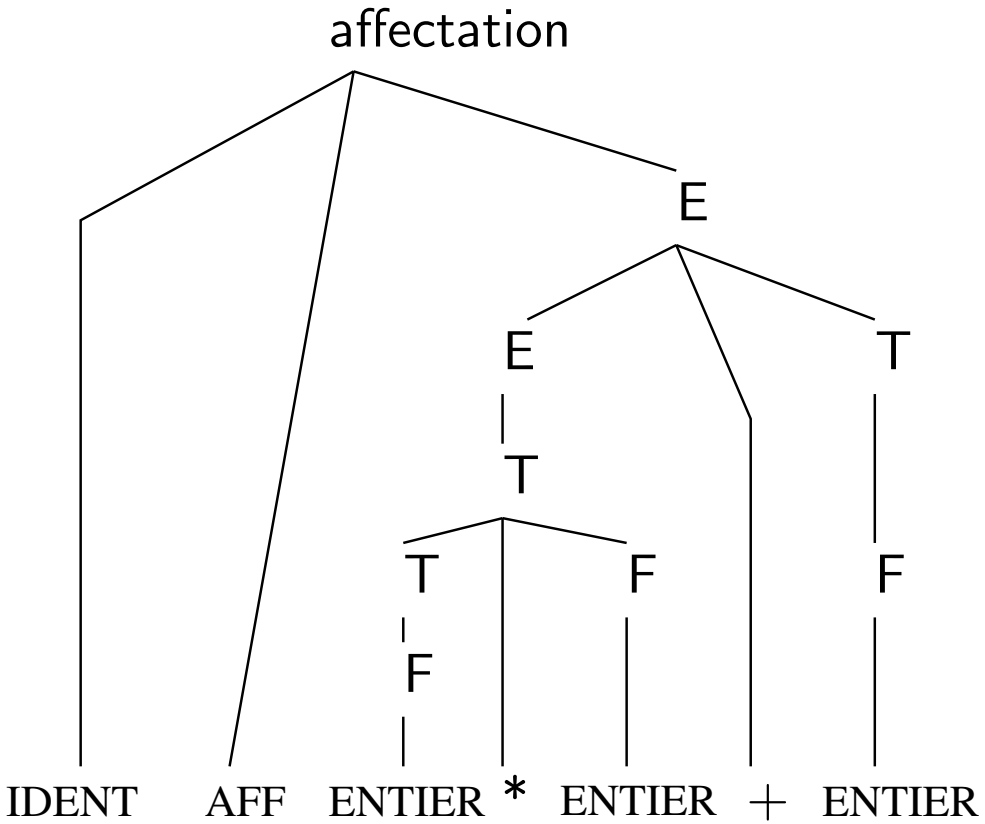
```
gauche ::= droite {: code module 1
                      code module 2
                      // code module supprimé
                      ...
                   :}
```

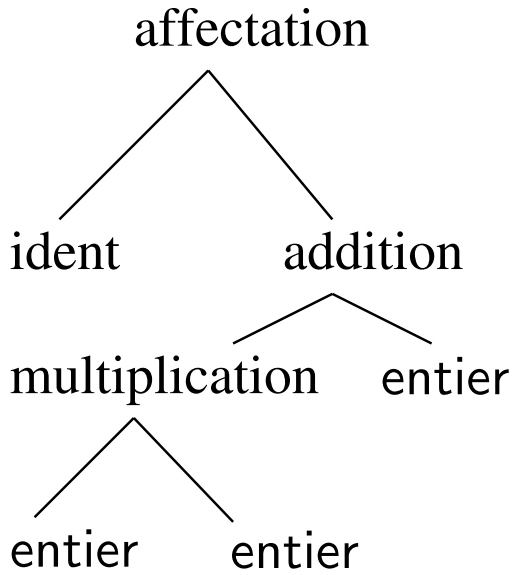
- difficilement maintenable ;
- difficile à étendre.

$\Rightarrow$ construire, lors de l'analyse syntaxique, un arbre abstrait.
et parcourir l'arbre lors de chaque module sémantique.

Arbre abstrait - exemple

Structure de données qui représente le programme source en s'abstrayant des détails syntaxiques.





Arbre abstrait - remarques

Plus de parenthésage :

- interprétation d'un mot (priorités, associativité des opérateurs) fixée par l'arbre ;
- les parenthèses, c'est de la syntaxe !

Des nœuds :

- d'arité fixée ;
- d'arité variable.

Arité fixée : exemple des affectation et expressions

Une affectation (ici racine de l'arbre abstrait) a 2 fils :

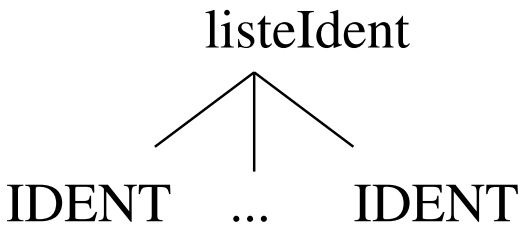
- l'identificateur à gauche du :=;
- l'expression à droite du :=.

Une addition, sous-arbre de l'arbre abstrait, a 2 fils :

- l'expression à gauche du +;
- l'expression à droite du +.

Un identificateur ou un entier est une feuille de l'arbre.

Arité variable : exemple des listes



Arité variable : exemple des listes

Une liste d'identificateurs est un nœud d'arité n-aire :

- au moins un fils ;
- autant de fils que d'identificateurs.

Une liste de déclaration est un nœud d'arité n-aire :

- possiblement 0 fils ;
- autant de fils que de déclarations.

Choix d'un arbre

Pas une solution unique :

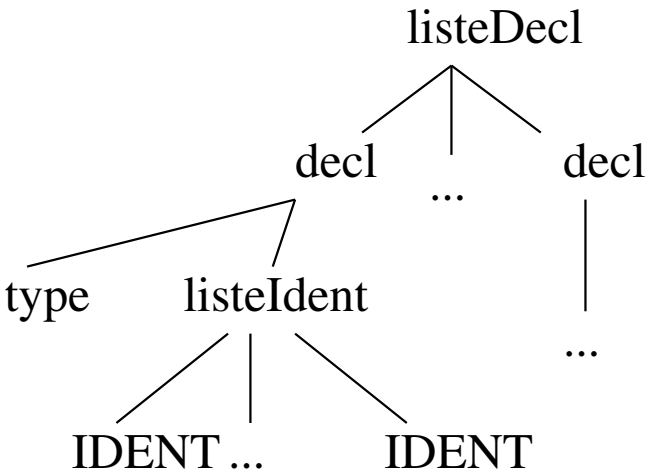
- l'arbre doit contenir les informations essentielles,
- mais est + ou - proche de la grammaire.

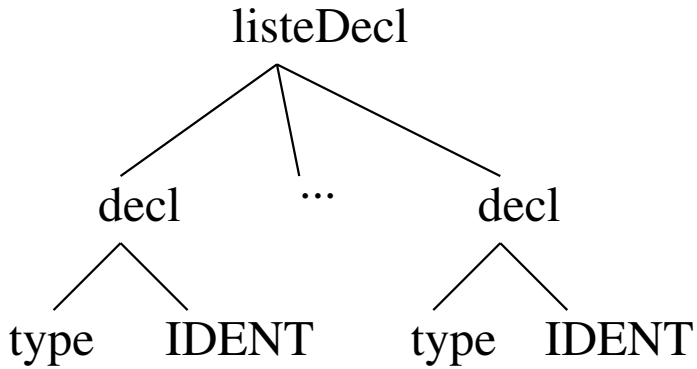
Choix d'un arbre : ex des déclarations de Init

listeDecl $\longrightarrow \epsilon$ | decl listeDecl

decl $\longrightarrow$ `DECLINT` listeIdent | `DECLLIST` listeIdent

listeIdent $\longrightarrow$ `IDENT` | `IDENT` listeIdent





Mise en œuvre en objet (Java)

Un peu de génie logiciel !

Construction d'un arbre abstrait pour cet exemple :

affectation $\longrightarrow$ `IDENT AFF` expr
expr $\longrightarrow$ expr + terme | terme
terme $\longrightarrow$ terme * facteur | facteur
facteur $\longrightarrow$ `IDENT` | `ENTIER` | (expr)

Quels types d'attribut pour quels terminaux et non-terminaux ?

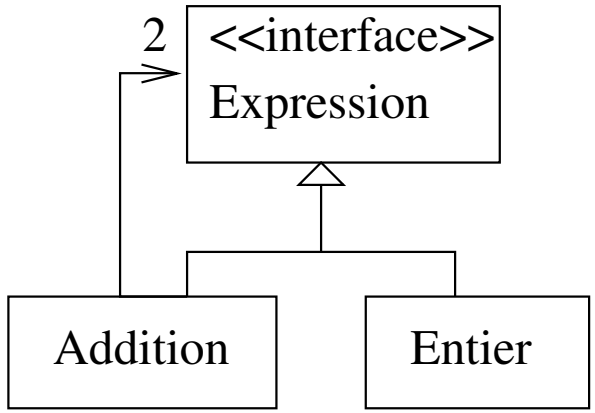
Mise en œuvre en Java : les types

Les nœuds sont typés : un type de nœud = un type Java.

Ex : `Affectation`, `Expression`, `Entier`, etc

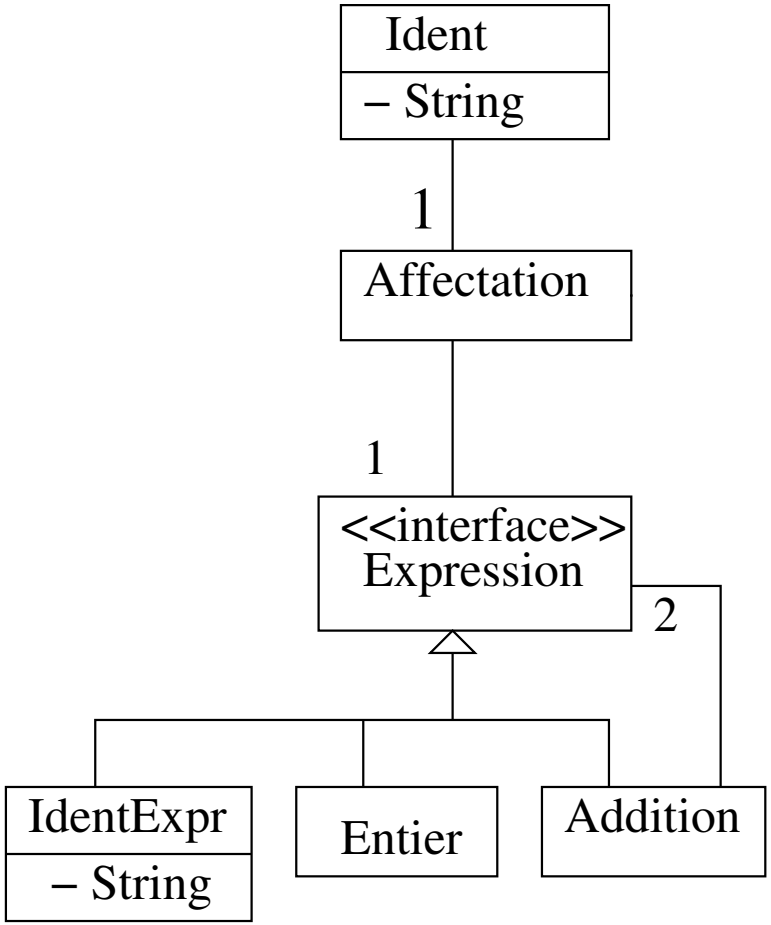
Quel type ?

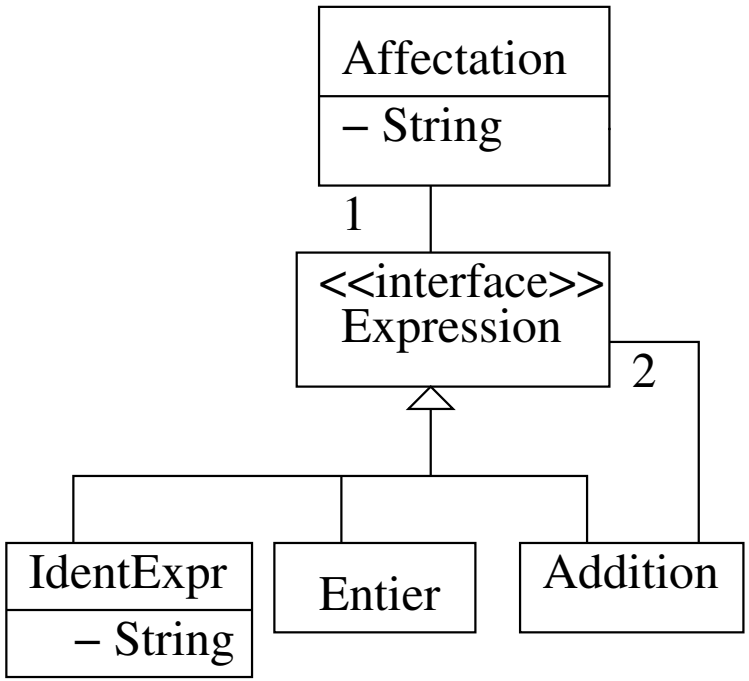
- type abstrait avec plusieurs spécialisations possibles : interface ou classe abstraite ;
  Ex : `Expression`, `Instruction`
- type concret : classe, éventuellement implantant un super-type.
  Ex : `Addition`, `Affectation`



Mise en œuvre en Java : exemple de l'affectation

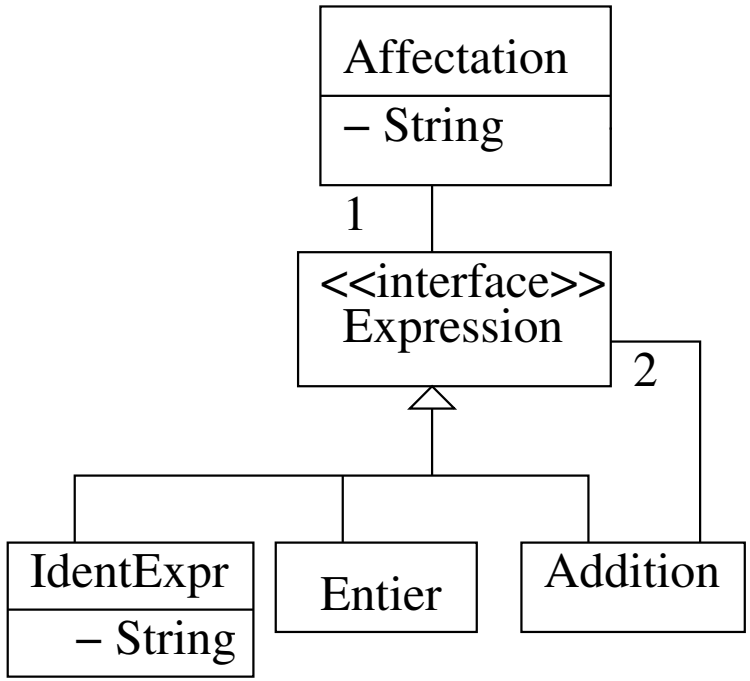
Les 2 solutions conviennent.





Mise en œuvre en Java : exemple des expressions

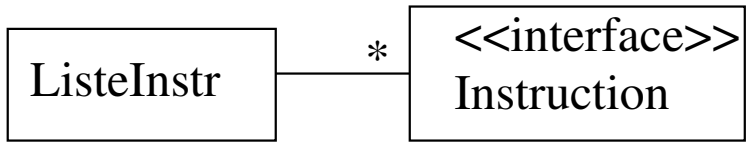
on doit garder le type `IdentExpr` : `String` n'est pas un sous-type de `Expression`.



Mise en œuvre en Java : arité

Suivant l'arité des nœuds :

- arité fixée : un fils est représenté par un attribut ;
- arité n-aire : on utilise un conteneur.



Mise en œuvre en Java - attributs synthétisés pour l'ex

`IDENT` : *s* de type `String`
`ENTIER` : *i* de type `Integer`
affectation : *a* de type `Affectation`
expr, *terme*, *facteur* : *e* de type `Expression` (interface)

affectation $\longrightarrow$ `IDENT AFF` expr
{ *affectation*.*a* = `new Affectation(IDENT.`*s*`,`*expr*.*e*`) ;` }
expr $\longrightarrow$ *expr* + *terme*
{ $expr_0$.*e* = `new Addition(`$expr_1$.*e*`,` *terme*.*e*`) ;` }
expr $\longrightarrow$ *terme* { *expr*.*e* = *terme*.*e* ; }
facteur $\longrightarrow$ `IDENT` { *facteur*.*e* = `new IdentExpr(IDENT.`*s*`) ;` }
facteur $\longrightarrow$ (*expr*) { *facteur*.*e* = *expr*.*e* ; }
...

Ce qui donne en Cup

```
terminal String IDENT;
terminal Integer ENTIER;
nonterminal Affectation affectation;
nonterminal Expression expr, terme, facteur;
...
affectation ::= IDENT:s AFF expr:e
        {: RESULT = new Affectation(s,e); :}
;
expr ::= expr:e1 + terme:e2
        {: RESULT = new Addition(e1,e2); :}
        | terme:e {: RESULT = e; :}
;
...
```

Mise en œuvre en Java - concrètement

Pour le mot `x := 3 + x` on aura l'arbre :

```
new Affectation("x",
        new Addition(new Entier(3), new IdentExpr("x")));
```

ou

```
new Affectation(new Ident("x"),
        new Addition(new Entier(3), new IdentExpr("x")));
```

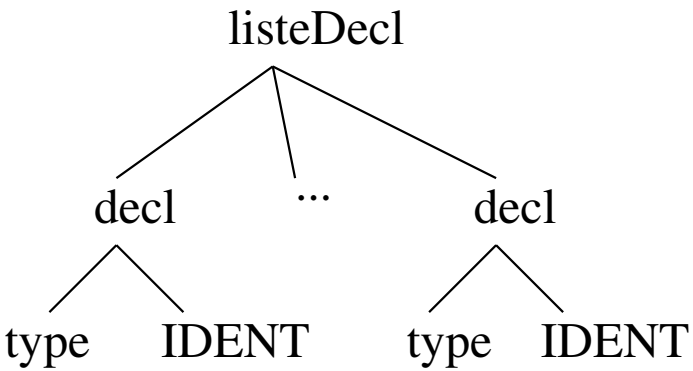
Noter le type différent des deux "x".

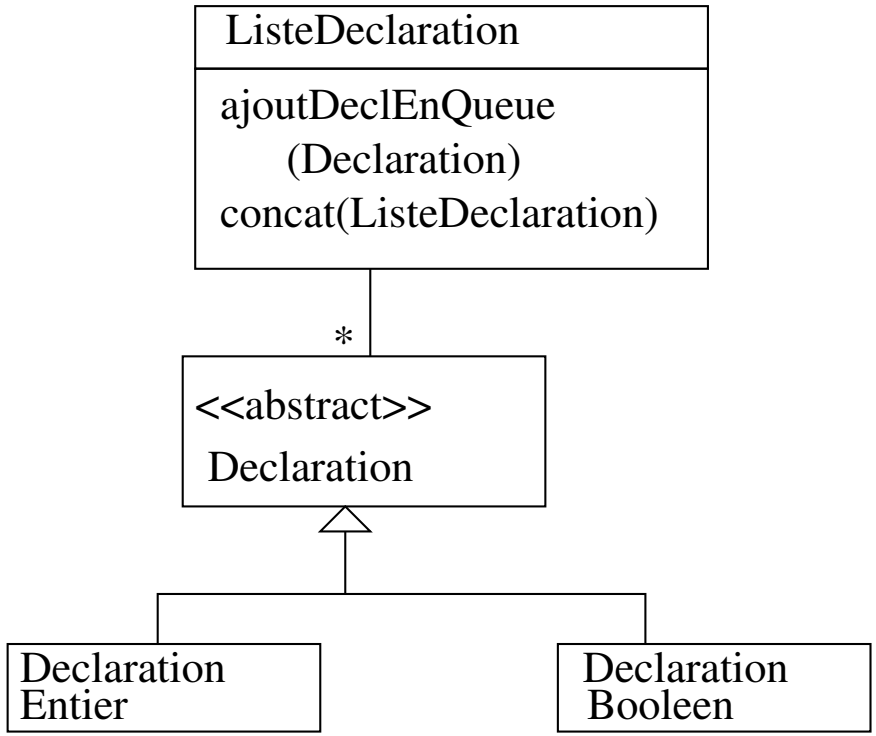
Lors du traitement d'une affectation on ne traite pas de la même manière :

- l'identificateur à gauche ;
- un identificateur dans l'expression de droite.

Les déclarations de Ava

Paquetage `ava.arbreAbstrait`





Les déclarations de Ava - Ex 1 - 1

Cas où la grammaire rend l'attribution un peu compliquée.

```
declarations ::=
      | DECLINT listeIdent FININSTR declarations
      | DECLBOOL listeIdent FININSTR declarations;
listeIdent ::= IDENT
       | IDENT SEP listeIdent;
```

Comment associer une `ListeDeclaration` (une liste plate de déclarations) à `declarations` ?

Les déclarations de Ava - Ex 1 - 2

```
listeIdent ::= IDENT | IDENT SEP listeIdent;
```

- Attribut de type `String` pour `IDENT`.
- On rassemble les identificateurs d'une déclaration : attribut de type `LinkedList` pour `listeIdent`

```
listeIdent ::=
    IDENT:s {: RESULT = new LinkedList();
               RESULT.add(s); :}
  | IDENT:s SEP listeIdent:l; {: l.addFirst(s);
                                 RESULT = l;
                              :};
```

Les déclarations de Ava - Ex 1 - 3

`declarations ::= INT listeIdent FININSTR declarations`

- on récupère la liste d'identificateurs de `listeIdent` ;
- on itère dessus pour créer des `DeclarationEntier`...
- ... qui alimentent une `ListeDeclaration` en construction.

```
declarations ::=
   INT listeIdent:idents FININSTR declarations:decls
      {: Iterator it = idents.iterator();
         ListeDeclaration ldecl = new ListeDeclaration();
          while (it.hasNext())
             ldecl.ajoutDeclEnQueue(
                new DeclarationEntier((String)it.next()));
          ...
       :};
```

Les déclarations de Ava - Ex 1 - 4

```
declarations ::=
     | INT listeIdent FININSTR declarations;
```

Et on concatène la `ListeDecl` créée à celle qui suit :

```
declarations ::=
   /* mot vide */ {: RESULT = null; :}
   | INT listeIdent:idents FININSTR declarations:decls
       {: ...
          ListeDeclaration ldecl = new ListeDeclaration();
          ... /* calcul précédent de ldecl */
          if (decls != null) ldecl.concat(decls);
          RESULT = ldecl;
        :};
```

Les déclarations de Ava - Ex 2 - 1

Autre cas.

```
declarations ::=
        | declaration declarations
;
declaration ::= DECLINT listeIdent FININSTR
              | DECLBOOL listeIdent FININSTR
;
listeIdent ::= IDENT
            | IDENT SEP listeIdent
;
```

On traite `listeIdent` comme précédemment.

Les déclarations de Ava - Ex 2 - 2

Comme précédemment, on itère sur la liste de nom pour remplir une `ListeDeclaration` :

```
declaration ::= INT listeIdent:idents FININSTR
    {: ListeDeclaration ldecl = new ListeDeclaration();
       Iterator iterateur = idents.iterator();
       while(iterateur.hasNext())
            ldecl.ajoutDeclEnQueue(
               new DeclarationEntier((String)it.next()));
       RESULT = ldecl;
     :}
;
```

Les déclarations de Ava - Ex 2 - 3

Et on concatène la `ListeDeclaration` créée à celle qui suit (avec petite variante) :

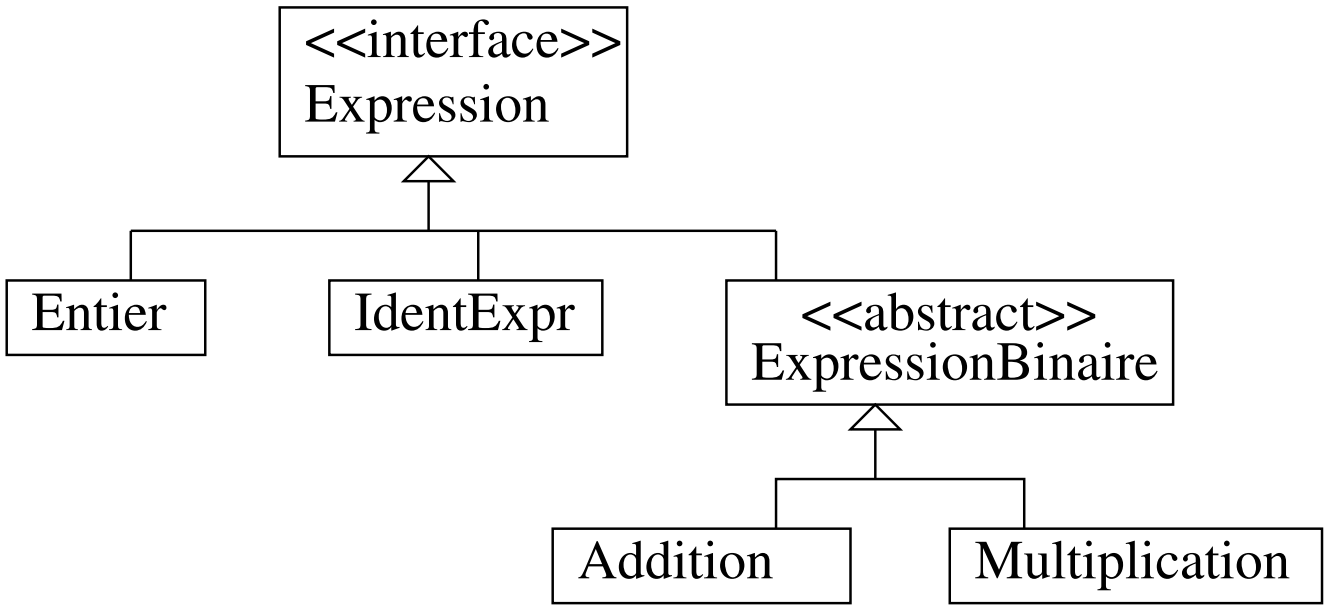
```
declarations ::=
        /* mot vide */ {: RESULT = new ListeDeclaration(); :}
        | declaration:decl1 declarations:listedecl2
           {: decl1.concat(listedecl2);
              RESULT = decl1; :}
;
```

Conclusion :

- autant d'attributions que de grammaires différentes ;
- autant d'actions que de manières différentes de les coder.

Le problème (de génie logiciel)

Un arbre abstrait représentant une expression arithmétique.



Le problème (de génie logiciel)

On veut appliquer au moins deux analyses successives, parmi :

- contrôle de type ;
- décompilation ;
- passage infixe vers postfixe ;
- génération de code ;
- etc.

Comment fait-on pour appliquer séquentiellement ces analyses à l'arbre ?

Remarque : parcours de l'arbre

Le parcours de l'arbre lui-même est standard.

Par exemple on prévoit l'arbre :

- pour que son parcours infixe (père gauche droite)
- reconstitue l'ordre de lecture initial du programme source.

Remarque : parcours de l'arbre, exemple

Pour une `Addition` possédant deux fils de type `Expression` :

```
void visit(Addition a) {
    ...
    visit(a.getGauche());
    ...
    visit(a.getDroite());
    ...
}
```

Remarque : parcours de l'arbre , exemple

Pour un `Programme` possédant un nom, une liste de déclarations et une liste d'instructions :

```
void visit(Programme p) {
    ...
    visit(p.getNom());
    ...
    visit(p.getListeDecl());
    ...
    visit(e.getListeInstr());
    ... }
```

Le problème (de génie logiciel), enfin.

Le parcours de l'arbre n'est pas un problème.

Le problème est qu'on veut mettre. . .

. . . plusieurs analyses dans la visite. . .

. . . en étant modulaire.

Comment assurer une architecture modulaire

Tentative 1

Autant de classes que d'analyses :

- `TypeChecker`
- `Decompilateur`
- etc.

Comment assurer une architecture modulaire

Tentative 1, exemple

```
public class Decompilateur {

  private Expression expr;
  private String texte;

  public Decompilateur(Expression e) {
     this.expr = e;
     this.texte = "";
  }
}
```

Comment assurer une architecture modulaire

Tentative 1, arg !

Mise au rencart du polymorphisme.

```
public void decompile() throws DecompilateurException {
    ...
    if (this.expr instance of Entier)
      this.texte = this.texte + ... ;
    else if (this.expr instance of Ident)
      this.texte = this.texte + ... ;
    ...
}
```

Entendez-vous JC Routier cracher le feu ? « Solution » à proscrire !

Comment assurer une architecture modulaire

Tentative 2

Ajout des fonctionnalités dans la structure de données.

```
public class Entier implements Expression {
    ...
    public String decompile() {
        return new String(this.valeur);
    }

    public void typeCheck() throws ... {
    ...
    }
    ...
}
```

Comment assurer une architecture modulaire

Tentative 2

Autant de classes que d'analyses.

```
public class Decompilateur {
   ...
   public void decompile() throws... {
      this.texte = this.expr.decompile();
   }
}

public class TypeCkecker {
   ...
   public void typeCheck() ... {
   ... this.expr.typeCheck()...
   }
}
```

Comment assurer une architecture modulaire

Tentative 2, peut mieux faire

Entendez-vous JC Routier maugréer ?

Viole le principe «open-close» :

- pour ajouter une fonctionnalité. . .
- = ouvrir le code aux extensions. . .
- il faut retoucher chaque classe de l'arbre abstrait. . .
- = ne pas fermer le code aux modifications

Arbre abstrait Ava $\sim$ 40 types Java.

Comment assurer une architecture modulaire

Tentative 3 : le pattern Visiteur

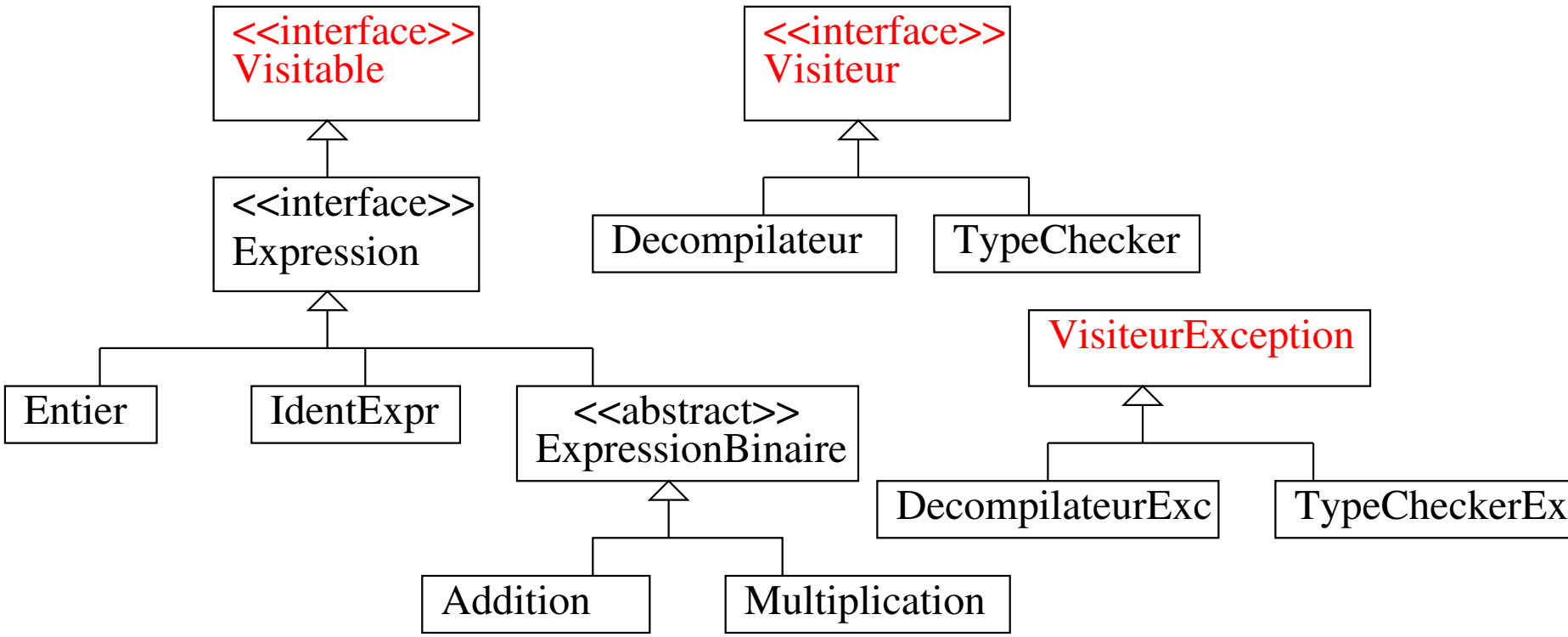
But : assurer un couplage faible entre structure de données et traitements.

Les traitements ne sont pas internes à la structure de données.

La structure de données offre la possibilité de se faire visiter (sans pour autant rendre publics ses attributs !).

Elle devient « visitable » par un « visiteur ».

Pattern Visiteur : interfaces



L'interface Visitable

Un type visitable (qui implante Visitable) autorise des traitements (qui implantent Visiteur) à le visiter.

Le corps de la méthode indique comment se fait la visite.

```
public interface Visitable {
    /** Permet la visite d'un nœud de l'arbre par un
     * visiteur passé en paramètre.
     * @param v le visiteur pour ce nœud.
     */
    public void visite(Visiteur v)
                              throws VisiteurException;
 }
```

Comment implanter `Visitable`

Une `Expression` ne sait pas comment se faire visiter : c'est une interface.

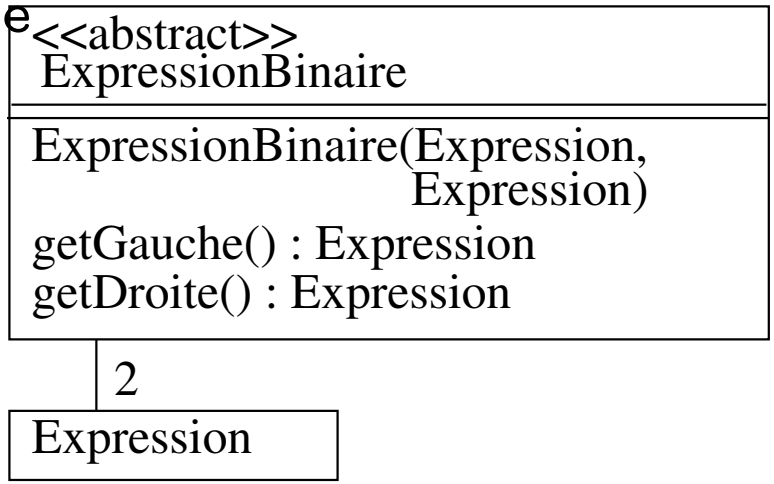
Délégué aux sous-types (polymorphisme).

```
public interface Expression extends Visitable {
}
```

Comment implanter `Visitable`

Une `ExpressionBinaire` ne sait pas non plus comment se faire visiter : c'est une classe abstraite. Aussi délégué aux sous-types.

```
public abstract ExpressionBinaire
    implements Expression {
  ...
  public abstract void
      visite(Visiteur v)
      throws VisiteurException;
}
```



Comment implanter `Visitable`

Une `Addition` est une feuille de l'arbre d'héritage : elle sait a priori comment se faire visiter.

Mais... comment au fait ? Puisque la visite dépend du traitement ?

⇒ inversion du contrôle :

- on renvoie la balle (l'`Addition`) au `Visiteur` ;
- ce visiteur sait quel traitement il applique aux additions
  ⇒ méthode `Visiteur.visiteAddition(Addition)`.

La méthode visite d'Addition

```
public void visite(Visiteur v) throws VisiteurException {
  if (v == null)
    throw new IllegalArgumentException("...");
  v.visiteAddition(this);
}
```

Pattern Visiteur : interface `Visiteur`

L'interface contient des méthodes `visite*` prenant en paramètre les différents types de nœuds à traiter (qui sont des `Visitable`) :

```
public interface Visiteur {
  public void visiteAddition(Addition a)
                    throws VisiteurException;
  public void visiteMultiplication(Multiplication m)
                    throws VisiteurException;
  public void visiteEntier(Entier e) throws VisiteurException;
  public void visiteIdent(IdentExpr e) throws VisiteurException;
}
```

Pattern Visiteur : les visiteurs

Chaque module d'analyse sémantique implante l'interface `Visiteur` :

```
public class Decompilateur implements Visiteur {
...
}
public class TypeChecker implements Visiteur {
...
}
```

On a ainsi découplé structure de données et traitements.

Inconvénient : modification structure de données $\Rightarrow$ modification de tous les visiteurs en conséquence.

Exemple : le visiteur Decompilateur

Décompile une Expression.

```
public class Decompilateur implements Visiteur {

  private Expression expr;
  ...

  public Decompilateur(Expression e) {
     this.expr = e;
     ...
  }
  ...
  public String decompile() {
    ???????
  }
  ...
}
```

Lancement de la décompilation : inversion du contrôle

La décompilation se fait statiquement sur une `Expression` dont on ne connait pas le type dynamique effectif.

⇒ appel de la méthode `Expression.visite(Visiteur)` qui, dynamiquement, délègue à son sous-type effectif `X`, donc renvoie la balle à la méthode `Decompilateur.visiteX(...)`.

```
public String decompile()
  throws DecompilateurException, VisiteurException {
  this.expr.visite(this);
  return this.texte;
}
```

Noter l'exception `VisiteurException` dans la signature, due à la signature de `Expression.visite(Visiteur)`.

Exemple de cascade /délégation d'appel

Effet « ping-pong » de l'inversion du contrôle pas évident à saisir.

```
Expression e = new Addition(...,...) ;
Decompilateur d = new Decompilateur(e) ;
d.decompile() ;
===> e.visite(d) ; (délégué à l'Addition)
===> d.visiteAddition(e) ; (inversion du contrôle)
```

Effets de bord

Le décompilateur produit une représentation textuelle, type `String`.

Dans cette version du visiteur, les méthodes `visite*` ne retournent pas de valeur.

$\Rightarrow$ chaque `visite*` alimente par effet de bord un attribut du visiteur.

Effets de bord

```
public class Decompilateur implements Visiteur {
  private Expression expr;
  private String texte;
  public Decompilateur(Expression e) {
     this.expr = e;
     this.texte = "";
  }
  ...
}
```

Décompiler un entier

```
public void visiteEntier(Entier e)
        throws DecompilateurException {
  if (e == null)
    throw new DecompilateurException("...");
  this.texte = this.texte + e.getValeur();
}
```

Décompiler un identificateur

```
public void visiteIdent(IdentExpr id)
          throws DecompilateurException {
  if (id == null)
    throw new DecompilateurException("...");
  this.texte = this.texte + id.getNom();
}
```

Décompiler une addition

Avec parenthésage.

Nécessite de visiter ses opérandes de type `Expression`.

```
public void visiteAddition(Addition a) throws ... {
  if (a == null) throw new DecompilateurException("...");
  this.texte = this.texte + "(";
  Expression gauche = expr.getGauche(); // opérande gauche
  gauche.visite(this);
  this.texte = this.texte + " + ";
  Expression droite = expr.getExprDroite(); // opérande droit
  droite.visite(this);
  this.texte = this.texte + ")";
}
```

Exemple de cascade /délégation d'appel

```
Expression e1 = new Entier(3) ;
Expression e2 = new IdentExpr("x") ;
Expression e = new Addition(e1,e2) ;
Decompilateur d = new Decompilateur(e) ;
d.decompile() ;
===> e.visite(d) ; (délégué à l'Addition)
===> d.visiteAddition(e) ; (inversion du contrôle)
===> e1.visite(d) ; (délégué à Entier)
===> d.visiteEntier(e1) ; (positionne this.texte)
===> e2.visite(d) ; (délégué à IdentExpr)
===> d.visiteIdent(e2) ; (positionne this.texte)
```

Pattern Visiteur : remarques

Vive le polymorphisme !

Pas évident à comprendre ;

- inversion du contrôle ;
- fonctionnement par effet de bord.

Mais on s'habitue vite ! Vous coderez 2 `Visiteur` en TP.

Le contrôle de type

But :

- inférer un type pour chaque variable ;
- contrôler que chaque utilisation de variable est conforme à son type.

Les classiques :

- opérandes d'une addition : de type numérique ;
- problèmes de conversion entre types numériques ;
- type booléen dans les structures de contrôle ;
- détection des déclarations multiples ou omises ;
- etc.

Le contrôle de type pour AVA

Crucial car la grammaire algébrique est beaucoup trop permissive pour les expressions.

Programme syntaxiquement correct, sémantiquement incorrect :

```
program "aie";
int x;
x := true + 3 + x<=5;
```

Table des symboles

Comment mémoriser les associations (identificateur,type) ?

$\Rightarrow$ structure de données = table des symboles

Table associative implantée par une table de hachage :

- clé = l'identificateur
- avant il fallait la programmer en C, ça occupait un TP. . . ;
- avec les langages type Java on peut utiliser un type existant (comme `Map`, en TP).

Quand compléter la table des symboles ?

Deux approches possibles : compilateurs étroits et larges.

Compilateur étroit :

- historiquement les premiers conçus pour cause de mémoire coûteuse ;
- lecture d'un fragment de code source puis traitement immédiat (y compris génération de fragment de code) ;
- donc remplissage de la table au plus tôt ;
- l'analyseur lexical y entre les identificateurs sans préciser leur type ;
- permet de traiter les doubles déclarations dès l'analyse lexicale.

Peu structuré, pas l'approche suivie en TP.

Quand compléter la table des symboles ?

Compilateur large :

- apparaissent quand on parvient à stocker un programme entier en mémoire ;
- on lit tout le programme source d'un coup ;
- puis on applique une succession d'analyses (modules) ;
- le remplissage de la table se fait lors de l'analyse sémantique.

Approche du TP.

Table des symboles

Qu'associe-t-on à un identificateur ?

- son type ;
- sa portée ;
- etc.

En TP :

(clé = `String`, valeur = `AttributsIdentificateur`)

Table des symboles

Pour le TP.

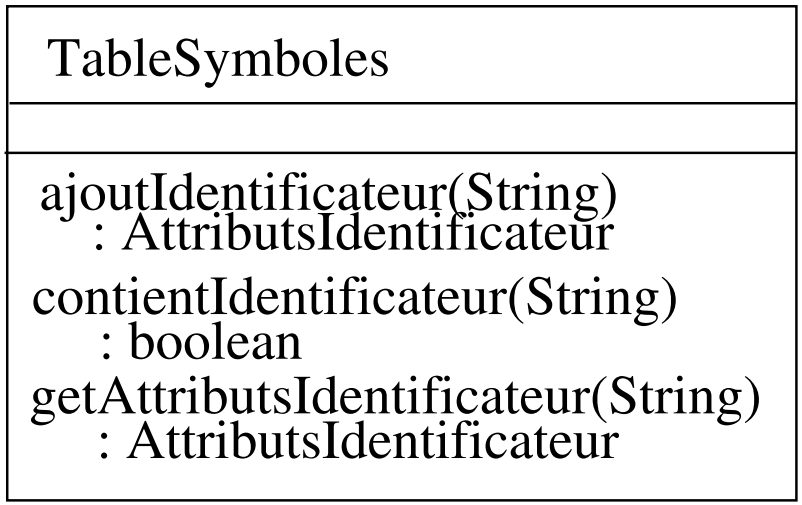
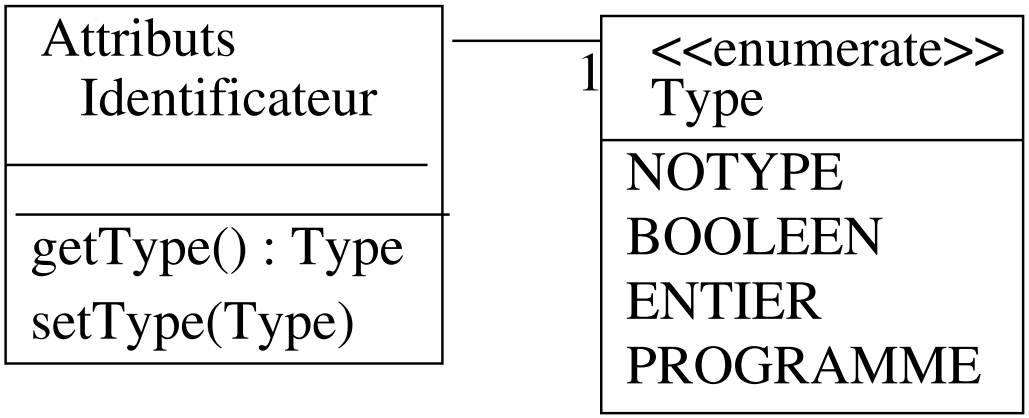


Table des symboles

Pour le TP.



En TP : implantation par un Visiteur

Le contrôleur est paramétré par la table des symboles et l'arbre abstrait (noter les dépendances explicites).

```
public class TypeChecker implements Visiteur {
  private TableSymboles ts;
  private Programme program;
  ...

  public TypeChecker(TableSymboles ts, Programme p) {
    this.ts = ts; this.program = p;
  }
  ...
```

Inférence de type pour AVA

Pour certains langages comme CAML, le type des variables n'est pas déclaré, c'est le compilateur qui l'infère..

En comparaison l'inférence de type pour AVA est un jeu d'enfant car les types sont déclarés et statiques :

- quand le contrôleur rencontre une déclaration entière/booléenne pour `x` ;
- il vérifie que `x` n'est pas dans la table des symboles ;
- il ajoute `x` dans la table des symboles ;
- il fixe le type de `x` à `Type.ENTIER`/`Type.BOOLEEN`.

Le nom du programme est déclaré de type `Type.PROGRAM`.

Inférence de type pour AVA : exemple

```
public void visiteDeclInt(DeclarationEntier decl)
        throws TypeCheckingException {
  String name = decl.getIdent();
  if (this.ts.contientIdentificateur(name))
  throw new TypeCheckingException("la variable " + name
     + " a deja ete declaree");
  AttributsIdentificateur att =
        this.ts.ajoutIdentificateur(name);
  att.setType(Type.ENTIER);
}
```

Contrôle de type AVA : exemple facile, la lecture

La variable lue :

- doit déjà avoir été déclarée ;
- doit être de type entier.

Contrôle de type AVA : exemple moins facile, l'affectation

Une affectation c'est :

- une variable (affectée) ;
- une expression qui doit être du même type.

Mais quel est le type d'une expression ?

Doit être inféré à partir du type des expressions de base : identificateurs et constantes.

Type des expressions AVA

Utilisation d'un visiteur à effet de bord : attribut qui mémorise le type de la dernière expression visitée.

```
public class TypeChecker implements Visiteur {
  private TableSymboles ts;
  private Programme program;
  private Type typeDerniereExpressionVisitee;

  public TypeChecker(TableSymboles ts, Programme p) {
    this.ts = ts; this.program = p;
    this.typeDerniereExpressionVisitee = Type.NOTYPE;
  }
  ...
```

Inférence pour les `Expression` de base

Ex : `Entier`.

Pas de contrôle ici : on infère juste qu'un `Entier` est de type `Type.ENTIER`.

```
public void visiteEntier(Entier expr) throws ... {
   ...
   this.typeDerniereExpressionVisitee = Type.ENTIER;
}
```

Inférence pour une Addition

Les deux opérandes doivent être de type entier, le résultat est de type entier.

```
public void visiteAddition(Addition expr) throws ... {
  ...
  Expression gauche = expr.getExprGauche();
  gauche.visite(this);
  if (this.typeDerniereExpressionVisitee != Type.ENTIER)
    throw new TypeCheckingException("...");
  Expression droite = expr.getExprDroite();
  ... // même traitement qu'à gauche
  this.typeDerniereExpressionVisitee = Type.ENTIER;
}
```

Génération de code

Problématiques dans un compilateur réel :

- génération de code pour un processeur donné ;
- optimisation de code.

Pas dans ce cours.

En TP

Génération de bytecode Java.

- utilisation de l'API BCEL (Byte Code Engineering Library - un projet Jakarta Apache)
  `http://jakarta.apache.org/bcel/`
- pour vous : utilisation d'une API maison plus simple.

Quelques connaissances à avoir sur la structure d'un `.class`.

Structure d'un `.class`

Suite de bytecodes (suite d'octets) découpée en sections.

Première section : séquence d'octets particulière permettant à la JVM de vérifier la validité du `.class`.

Sa génération vous sera fournie.

Structure d'un `.class` : table des constantes

Seconde section : table des constantes utilisées dans le `.class`

- chaînes de caractères, constantes entières ;
- noms et profils des méthodes, etc.

Table gérée par index entier : une constante est référencée par son index dans cette table.

Ex :

...	...
4	"unechaine"
5	23
...	...

Structure d'un `.class` : instructions

Parmi les sections suivantes : code du corps des méthodes.

TP : `main` contenant les instructions du programme AVA.

Code = suite d'instructions rangées séquentiellement par adresse croissante.

Une adresse est un index de type entier.

Ex :

...	...
6	lcd 4
7	ldc 12
8	iadd
9	goto 3
...	...

Structure d'un .class : table des variables

Dans le code des méthodes : des variables locales.

Elles sont référencées par leur index dans une table des variables.

Ex :

...	...
4	"x"
5	"y"

code de méthode

...	...
10	store 5
...	...

y := ...

Les instructions de bytecode

Le code s'exécute grâce à une pile d'exécution.

Les instructions de byte-code manipulent cette pile :

- chargement d'une valeur dans la pile
- stockage d'une valeur dans une variable
- opérations arithmétiques, booléennes, sur pile
- manipulation de pile (pop, etc)

Les instructions de bytecode

Autre instructions :

- instructions de saut ;
- instructions créées pour les besoins du TP
  ex : `SPRINT` = appel de `java.io.PrintStream.print()`

Chargement dans la pile, stockage

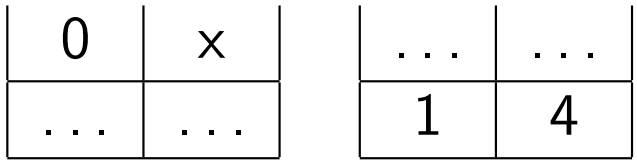
`iconst_0` : empile la constante 0 (existe avec 1).

`ldc <index>` : empile la valeur de la constante de numéro `index`

`iload <index>` : empile la valeur de la variable de numéro `index`

`istore <index>` : stocke le sommet de pile dans la variable d'index `index`, et le dépile.

Exemple : `x := 4`




```
ldc 1 ; istore 0 ;
```

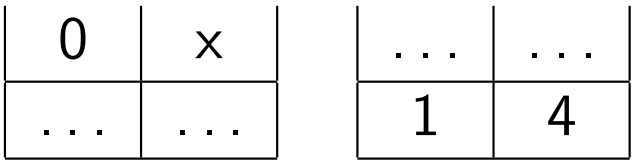
Opérations arithmétiques et booléennes

`iadd` : additionne le sommet de pile et le sous-sommet de pile, remplace le sommet et le sous-sommet par le résultat de l'addition

`iand` : idem avec le $\wedge$ bit à bit

NB : pas de type booléen, vrai $= 1$, faux $= 0$

Ex : `x := x + 4`




```
iload 0 ; ldc 1 ; iadd ; istore 0 ;
```

Instructions à saut

Permettent de sauter à une instruction d'index donné

Permettent l'encodage des structures de contrôle.

`goto <index>` : positionne le compteur de programme (PC) à l'instruction de numéro `index`.

`if_eq <index>` : positionne le PC à l'instruction d'index `index` si le sommet de pile vaut 0 ; dépile le sommet de pile.

$\Rightarrow$ permet de coder un `if b` si `b` est une variable ou une constante booléenne.

Exemple : `if b`

Exemple : pour `if b then writeln ; end if ; read x ;`

- les instructions de bytecode sont numérotées par leur index ;
- `b` a l'index 1.

```
18 - iload 1        on empile la valeur de b
20 - ifeq --> 30    si b vaut faux (0) on saute au read (30)
23 - writeln        sinon on fait le writeln
30 - iread          et on passe en séquence (pas de else)
33 - istore 0
```

Noter :

- visiter `b` : empiler la valeur de `b` ;
- on saute si la condition est fausse : `ifeq`

Exemple : `if b`, positionnement index de saut

Exemple : pour `if b then writeln ; end if ; read x ;`

En fait on procède en deux temps.

```
18 - iload 1      on empile la valeur de b
20 - ifeq -->?    si b vaut faux (0) on saute où ?
```

- au moment où on génère le `ifeq`...
- ... le code vers lequel sauter n'a pas encore été généré !
- $\Rightarrow$ on mémorise l'index du test (20) ;
- on génère le bloc `then`...

Exemple : `if b`, positionnement index de saut

Exemple : pour `if b then writeln ; end if ; read x ;`

```
18 - iload 1
20 - ifeq -->?
23 - writeln
30 -????
```

la prochaine instruction sera en 30

- on utilise un `setTarget(int from, int to)` pour mettre à jour l'index de saut du `ifeq`
- ici `setTarget(20,30)`

Exemple : `if b`, positionnement index de saut

Exemple : pour `if b then writeln ; end if ; read x ;`

```
18 - iload 1
20 - ifeq --> 30
23 - writeln
30 - ????          la prochaine instruction sera en 30
```

- et on génère la suite du `if` en 30.

Instructions à sauts, suite

`if_cmplt <index>` : (lt = lower than) positionne le PC à l'instruction de numéro `index` si le sous-sommet de pile est strictement inférieur au sommet de pile, incrémente le PC sinon. Dépile le sommet et le sous-sommet de pile.

`if_compge <index>` : idem avec greater or equal, etc.

$\Rightarrow$ permettent d'encoder les comparaisons.

Exemple : 3<4

Exemple : pour `write(%b,3<4)`
si le 3 est à l'index 75 et le 4 à l'index 76

`12 - ldc 75`	on empile 3
`14 - ldc 76`	on empile 4
`16 - if_icmpge --> 23`	si $3 \geq 4$, sauter à `iconst_0` (empiler faux)
`19 - iconst_1`	si pas sauté, cas $3 < 4$: empiler vrai (1)
`20 - goto --> 24`	on saute à la fin
`23 - iconst_0`	si on a sauté, cas $3 \geq 4$: empiler faux (0)
`24 - iprint`	on imprime le sommet de pile

Remarque : visiter une comparaison

Visiter une comparaison c'est :

- visiter ses opérandes ;
- utiliser le bytecode qui réalise la négation de la comparaison ; 3<4 $\Rightarrow$ `if_icmpge`
- empiler 0 si la comparaison est fausse
- empiler 1 si la comparaison est vraie

Exemple : 3<4, suite

Exemple : pour `write(%b,3<4)`
Comme précédemment, on fixe les index de saut après coup.

```
12 - ldc 75                  on empile 3
14 - ldc 76                  on empile 4
16 - if_icmpge -->???        sauter où ???
```

- on mémorise l'index 16 ;
- et on génère la suite. . .

Exemple : 3<4, suite

Exemple : pour `write(%b,3<4)`

```
12 - ldc 75              on empile 3
14 - ldc 76              on empile 4
16 - if_icmpge -->??     sauter où ??
19 - iconst_1            empiler 1
20 - goto -->??          sauter où ??
23 -????                 le iconst_0 sera ici
```

- on fait le `setTarget(16,23)`
- on génère la suite...

Exemple : 3<4, suite

Exemple : pour `write(%b,3<4)`

```
12 - ldc 75              on empile 3
14 - ldc 76              on empile 4
16 - if_icmpge --> 23
19 - iconst_1            empiler 1
20 - goto -->??          sauter où ??
23 - iconst_0
24 - ...                 la suite sera ici
```

- on fait le `setTarget(20,24)`
- et on génère la suite

Exemple : 3<4, suite

Exemple : pour `write(%b,3<4)`

```
12 - ldc 75                 on empile 3
14 - ldc 76                 on empile 4
16 - if_icmpge --> 23
19 - iconst_1               empiler 1
20 - goto --> 24
23 - iconst_0
24 - iprint                 on affiche le sommet de pile
```

Exemple plus complet : `if x < 4 then...`

Exemple : `if x < 4 then writeln ; end if ; read x ;`
si `x` est en 0 et 4 en 75

`15 - iload 0`	on empile la valeur de `x`
`17 - ldc 75`	on empile 4
`19 - if_cmpge --> 26`	si `x>=4` sauter pour empiler 0
`22 - iconst_1`	si `x<4` : empiler 1
`23 - goto --> 27`	sauter fin comparaison
	`setTarget(19,26)`
`26 - iconst_0`	empiler 0, fin code comparaison
	`setTarget(23,27)`
`27 - ifeq --> 39`	`if` cond fausse, sauter après le bloc `then`
`30 - println`	bloc `then`
	`setTarget(27,39)`
`39 - iread`	et on passe en séquence (pas de else)

Sauts pour les structures de contrôle

Pour le `if then else` (à chercher : `while`)