Traduction dirigée par la syntaxe

Cyril Rabat
cyril.rabat@univ-reims.fr

Licence 3 Informatique - Info0602 - Langages et compilation

2021-2022

Cours n°6

Attributs et règles sémantiques
Table des symboles et arbres abstraits

Version 8 mars 2022

Table des matières

Introduction

- Nouvelle phase : l'analyse sémantique
  $\hookrightarrow$ Vérification de la cohérence sémantique
- Plusieurs fonctions :
  - Le contrôle de type : opérations, appels de fonctions/procédures
  - Détermination des instructions, expressions, identificateurs
- La grammaire n'est pas suffisante pour cette phase
  $\hookrightarrow$ Certains éléments contextuels se trouvent à différents endroits du code source
- La grammaire est augmentée :
  - Ajout d'attributs aux symboles de la grammaire
  - Ajout de règles sémantiques
  - Ajout de conditions
- On parle de «Traduction dirigée par la syntaxe» car elle s'appuie sur la grammaire définissant la syntaxe
- On parle de «Traduction dirigée par le modèle» si elle s'appuie sur la représentation intermédiaire (l'arbre syntaxique)

Un attribut

Définition : attribut

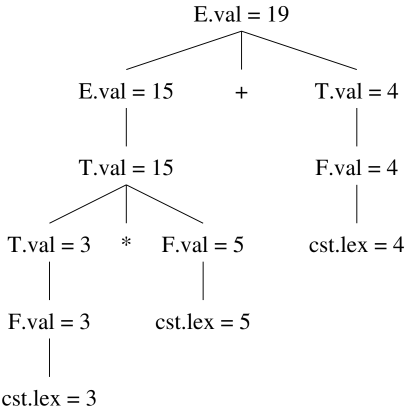
Un attribut est une valeur qui est associée à un nœud de l'arbre syntaxique.

- Un attribut représente une information quelconque : une chaîne de caractères, une adresse mémoire, . . .
- Il est possible d'associer autant d'attributs que l'on souhaite à chaque $A \in N$
- Les attributs sont séparés en deux sous-ensembles :
    - Les attributs *synthétisés*
    - Les attributs *hérités*

Un arbre syntaxique décoré

Définition : arbre syntaxique décoré

Un arbre syntaxique décoré est un arbre syntaxique où l'on fait apparaître les attributs associés aux nœuds.



Attribut hérités vs synthétisés

- L'**attribut synthétisé** est attaché au symbole en partie gauche
- Il est calculé à partir des valeurs des attributs des symboles de partie droite
  $\hookrightarrow X \rightarrow X_1 \dots X_n \; \{attr(X) \leftarrow f(attr(X_1), \dots, attr(X_n))\}$
- L'**attribut hérité** est attaché aux symboles en partie droite
- Il est calculé à partir des attributs du symbole non terminal en partie gauche et des attributs des autres symboles de la partie droite
  $\hookrightarrow X \rightarrow X_1 \dots X_n \; \{attr(X_j) \leftarrow$
  $f(attr(X), attr(X_1), ..., attr(X_{j-1}), attr(X_{j+1}), ..., attr(X_n))\}$
- Si on se représente l'arbre syntaxique :
  - Les attributs synthétisés attachés à un nœud sont calculés en fonction des attributs des fils
  - Les attributs hérités attachés à un nœud sont calculés en fonction du père et des nœuds frères

Une règle sémantique

Définition : règle sémantique

Une règle sémantique est définie par un algorithme qui permet de calculer les attributs d'un nœud en fonction des attributs des fils ou du père.

- Les règles sémantiques sont représentées entre accolades
- Elles peuvent apparaître dans la partie droite des règles
- Elles peuvent se situer entre chaque symbole
- Il peut y en avoir plusieurs

Exemple : une calculatrice (1/2)

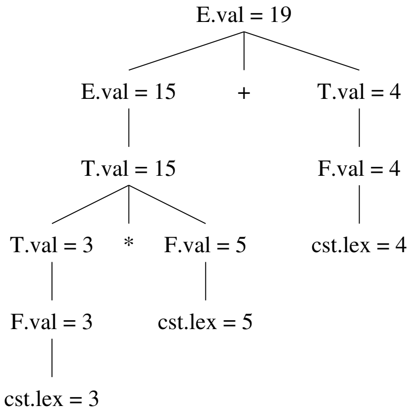
- Utilisation de la grammaire des expressions non ambiguë :
$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow \text{cst} \mid \text{id} \mid (E)$$
- Association d'un attribut synthétisé *val* à chaque non terminal
$\hookrightarrow$ E, T et F
- Règle sémantique : calcul du *val* du non terminal à partir des non terminaux de la partie droite

Exemple : une calculatrice (2/2)

Productions	Règles sémantiques
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow cst$	$F.val = cst.lex$

- *Note* : $cst.lex$ est la valeur retournée par l'analyseur lexical

L'arbre syntaxique décoré



Exemple avec «$3 * 5 + 4$»

Autre exemple : les attributs hérités (1/2)

- Soit la grammaire de déclaration de variables :

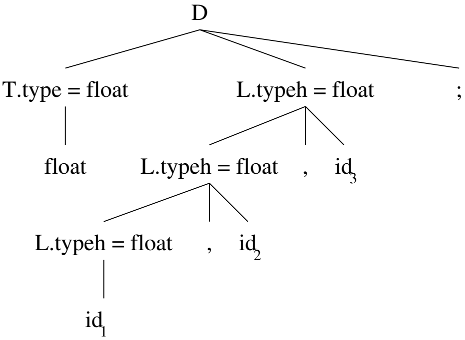
$$D \rightarrow TL;$$
$$T \rightarrow int \mid float$$
$$L \rightarrow L, id \mid id$$

- D est une déclaration, T est le type, L est une liste de déclarations
- L'attribut *typeh* est le type hérité
- L'attribut *entrée* est l'entrée des identificateurs dans la table des symboles
  $\hookrightarrow$ La procédure *ajouterType* permet de spécifier son type

Autre exemple : les attributs hérités (2/2)

Productions	**Règles sémantiques**
$D \rightarrow T\ L\ ;$	$L.typeh = T.type$
$T \rightarrow int$	$T.type = int$
$T \rightarrow float$	$T.type = float$
$L \rightarrow L_1, id$	$L_1.typeh = L.typeh$ $ajouterType(id.entrée,\ L.typeh)$
$L \rightarrow id$	$ajouterType(id.entrée,\ L.typeh)$

L'arbre syntaxique décoré



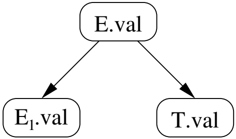
Exemple avec «float $id_1, id_2, id_3;$»

Graphes de dépendances (1/2)

- Les règles sémantiques impliquent des dépendances entre les attributs
  $\hookrightarrow$ Il existe un ordre pour les évaluations
- Les dépendances sont représentées sous la forme d'un graphe :
  - Les nœuds sont les attributs
  - Les arcs représentent les dépendances entre les attributs

Exemple

$E \rightarrow E_1 + T \; \{E.val = E_1.val + T.val\}$:



(les calculs de $E_1.val$ et $T.val$ doivent être réalisés avant $E.val$)

Graphes de dépendances (2/2)

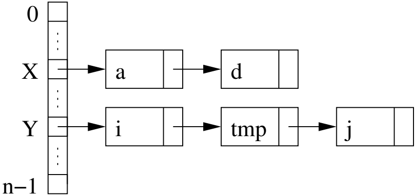
- Possible d'avoir des boucles dans le graphe
  $\hookrightarrow$ Ce qui pose des problèmes lors du calcul des valeurs
- La table des symboles permet de résoudre la plupart des problèmes rencontrés
- Pour résumer :
  - Les attributs synthétisés : très utilisés
  - Les attributs hérités : utiles pour les déclarations de variables

La table des symboles

- Contient la trace de la portée et des informations concernant la liaison des noms
- A chaque fois qu'un nom est trouvé, une recherche dans la table est effectuée
- Quand est-elle modifiée ?
  - Une nouvelle entrée est créée si un nouveau nom est trouvé
  - Une entrée peut être modifiée au cours de l'analyse
    $\hookrightarrow$ Valeur, type, *etc.*
- Chaque entrée correspond à un nom mais le format de l'entrée n'est pas uniforme
  $\hookrightarrow$ Il dépend de l'utilisation du nom

Implantation sous la forme d'une table de hachage

- Utilisation d'une table de hachage
  - Un tableau de n listes chaînées
  - Une fonction de hachage
    $\hookrightarrow h(w) \in [0; n-1]$ où w est le nom de l'entrée



- Fonctions :
  - `init(TS)` : initialise une table des symboles
  - `ajouter(nom, type, TS)` : ajoute une entrée
  - `recuperer(nom, TS)` : récupère une entrée

La table des symboles et les règles sémantiques

- Soit la grammaire suivante :

$$\begin{array}{ll} P & \rightarrow D\ I \\ D & \rightarrow \epsilon \mid LD \\ LD & \rightarrow T\ id\ ;\ LD \mid T\ id\ ; \\ T & \rightarrow int \mid float \end{array}$$

  $\hookrightarrow$ Un programme est une suite de déclarations (D) suivi par une liste d'instructions (I)
- La table est construite dans la partie déclarations
- Les types peuvent être vérifiés dans la partie instructions

Construction de la table des symboles

- Première solution : ajout d'un non terminal ITS (initialisation de la table des symboles) :

$$\begin{aligned} P &\rightarrow ITS\ D\ I \\ ITS &\rightarrow \epsilon \ \{ \ \texttt{init(TS)} \ \} \end{aligned}$$

$\hookrightarrow$ TS est une variable globale (et non un attribut)

- Deuxième solution : ajout d'une règle ϵ

$$\begin{aligned} P &\rightarrow D\ I \\ D &\rightarrow \epsilon \ \{ \ \texttt{init(TS)} \ \} \\ &\rightarrow LD \\ LD &\rightarrow T\ id\ ;\ LD \\ &\rightarrow T\ id\ ;\ \{ \ \texttt{init(TS)} \ \} \end{aligned}$$

$\hookrightarrow$ La table est créée après la dernière déclaration

Utilisation de la table des symboles

- Ajout des symboles :

$$
\begin{aligned}
LD &\rightarrow T\ id\ ;\ LD\ \{\ \texttt{ajouter(id.lex, T.type, TS)}\ \} \\
&\rightarrow T\ id\ ;\ \{\ \texttt{ajouter(id.lex, T.type, TS)}\ \} \\
T &\rightarrow int\ \{\ \texttt{T.type} = int\ \} \\
&\rightarrow float\ \{\ \texttt{T.type} = float\ \}
\end{aligned}
$$

  $\hookrightarrow$ Vérifier que les symboles n'existent pas déjà !
- Utilisation pour vérifier les types :
  $\hookrightarrow$ Comparer le type de l'expression et le type de l'identificateur

Portée des variables

- Dans les langages classiques, la déclaration visible d'un identificateur est celle qui se trouve dans le bloc englobant le plus interne
- Exemple :

```
int f() {
  int f, i;
  if(i = 0; i < 10; i++) {
    int f;
    f = ...
    {
      int f = 2;
      f = ...
    }
  }
}
```

Portée des variables et table des symboles

- Comment savoir de quel identificateur il est question ?
- Une solution : utilisation d'une pile de tables de symboles
- On cherche d'abord dans la table au sommet, puis dans la suivante
- On peut définir ainsi un niveau de portée
  $\hookrightarrow$ Niveau 0 table située en bas de la pile
  $\hookrightarrow$ Niveau 1 table suivante
  $\hookrightarrow$ etc.

Inconvénient de la pile des tables de symboles

- La recherche d'un identificateur peut être longue
- Solution : utilisation d'une table externe indexée par les noms
- Chaque entrée comporte le niveau ainsi que les propriétés
- Problème : que faire à la sortie d'un bloc ?
  $\hookrightarrow$ Parcours nécessaire de toute la table pour supprimer les entrées
- Autre solution : une table de niveaux
- Listes chaînées indexées par le niveau, chaque élément pointe vers les symboles
  $\hookrightarrow$ Lorsque l'on descend d'un niveau, on supprime tous les symboles de la dernière liste

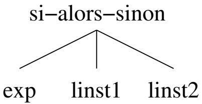
Un peu plus loin

- Surcharge (C++, *Java*, ...)
  $\hookrightarrow$ Un seul identificateur déclaré plusieurs fois
- Espaces de nommage
  $\hookrightarrow$ La portée est réduite à un espace

Les arbres abstraits

- Permettent de séparer l'étape de traduction (génération de code) de l'analyse syntaxique
- Construction de l'arbre pendant la phase d'analyse
- Évaluation pour générer le code
  $\hookrightarrow$ L'ordre de construction et d'évaluation peut ainsi être différent
- Correspond à une forme condensée de l'arbre syntaxique
- Basé sur le langage analysé

Exemple



Arbre correspondant à la production

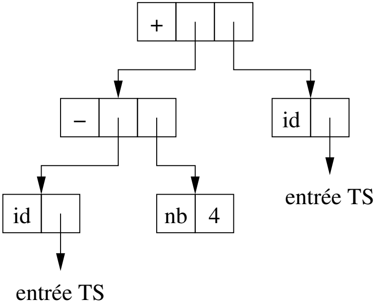
inst $\rightarrow$ *si exp alors linst_1 sinon linst_2*

Nœuds et opérations

- Un nœud de l'arbre contient un champ pour chaque opérande de l'opérateur
  ↪ Appelé *étiquette*
- Trois fonctions :
  - `creerNoeud(op, gauche, droit)`
    ↪ `op` est un opérateur, `gauche` et `droit` sont eux-mêmes des arbres
  - `creerFeuille(id, entree)` :
    ↪ `entree` est l'entrée de `id` dans la table des symboles
  - `creerFeuille(nb, val)` :
    ↪ `val` est la valeur de `nb` calculée par l'analyseur lexical

Exemple

Arbre abstrait pour l'expression $a - 4 + c$ (TS = Table des Symboles)



- $a_1 \leftarrow creerFeuille(id, recuperer(a, TS))$
- $a_2 \leftarrow creerFeuille(nb, 4)$
- $a_3 \leftarrow creerNoeud('-', a_1, a_2)$
- $a_4 \leftarrow creerFeuille(id, recuperer(c, TS))$
- $a_5 \leftarrow creerNoeud('+', a_3, a_4)$

Règles sémantiques de construction

La grammaire avec les règles sémantiques, en considérant l'attribut `arbre` :

$$
\begin{array}{ll}
E \rightarrow E_1 + T & \{ E.arbre = \text{creerNoeud('+', } E_1.arbre, \ T.arbre) \ \} \\
E \rightarrow T & \{ E.arbre = T.arbre \ \} \\
T \rightarrow T_1 * F & \{ T.arbre = \text{creerNoeud('*', } T_1.arbre, \ F.arbre) \ \} \\
T \rightarrow F & \{ T.arbre = F.arbre \ \} \\
F \rightarrow id & \{ F.arbre = \text{creerFeuille(id, id.entree)} \ \} \\
F \rightarrow cst & \{ F.arbre = \text{creerFeuille(nb, cst.val)} \ \}
\end{array}
$$

Les grammaires LL(1)

Toutes les règles sont de la forme $A \rightarrow \alpha_1 | \ldots | \alpha_n$

Procédure A()
  Si fe $\in$ Premier(α_1) **Alors**
    $\alpha_1()$
  Sinon Si fe $\in$ Premier(α_2) **Alors**
    $\ldots$
  Fin Si

Avec les règles sémantiques

Avec :

- $A.attr$ les attributs de A
- $\alpha_1 = X_1 \ldots X_n$ et $X_i.attr$ les attributs de X_i si $X \in N$
- $A \to \{act_1\}X_1\{act_2\}X_2 \ldots \{act_n\}X_n\{act_{n+1}\}$

Procédure A()

Si fe $\in$ Premier(α_1) **Alors**

act_1

$X_1(X_1.attr)$

act_2

$\ldots$

act_n

$X_n(X_n.attr)$

act_{n+1}

Sinon Si fe $\in$ Premier(α_2) **Alors**

$\ldots$

Fin Si

Problème de la récursivité gauche (1/4)

Exemple :

$$
\begin{array}{lll}
E \rightarrow E_1 + T & \{ E.v = E_1.v + T.v \} \\
E \rightarrow E_1 \text{ - } T & \{ E.v = E_1.v - T.v \} \\
E \rightarrow T & \{ E.v = T.v \} \\
T \rightarrow cst & \{ T.v = cst.lex \}
\end{array}
$$

Après transformation :

$$
\begin{array}{rl}
E & \rightarrow T\ X \\
X & \rightarrow + \ T\ X \\
 & \rightarrow \text{-}\ T\ X \\
 & \rightarrow \epsilon \\
T & \rightarrow cst
\end{array}
$$

Que deviennent les règles sémantiques ?

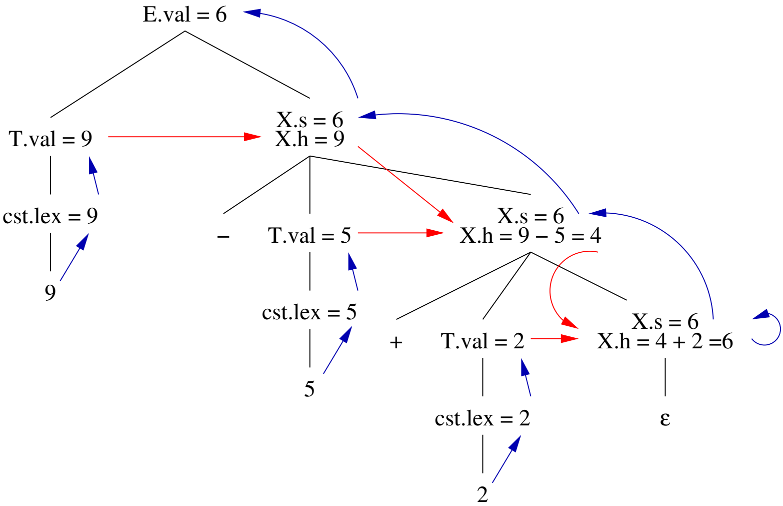
Problème de la récursivité gauche (2/4)

On définit 2 attributs pour X :

- h un attribut hérité
- s un attribut synthétisé

$$
\begin{aligned}
E &\rightarrow T \; \{ \; X.h = T.val \; \} \; X \; \{ \; E.val = X.s \; \} \\
X &\rightarrow + \; T \; \{ \; X_1.h = X.h + T.val \; \} \; X_1 \; \{ \; X.s = X_1.s \; \} \\
&\rightarrow \text{-} \; T \; \{ \; X_1.h = X.h \text{ - } T.val \; \} \; X_1 \; \{ \; X.s = X_1.s \; \} \\
&\rightarrow \epsilon \; \{ \; X.s = X.h \; \} \\
T &\rightarrow cst \; \{ \; T.val = cst.lex \; \}
\end{aligned}
$$

Problème de la récursivité gauche (3/4)



Arbre décoré correspondant à l'opération «9-5+2»

→ : attributs synthétisés ; → : attributs hérités

Problème de la récursivité gauche (4/4)

De façon générale :

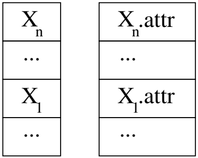
$$
\begin{aligned}
A &\rightarrow A_1 \; B \; \{ \; A.attr = g(A_1.attr, B.attr) \; \} \\
A &\rightarrow C \; \{ \; A.attr = f(C.attr) \; \}
\end{aligned}
$$

Devient :

$$
\begin{aligned}
A &\rightarrow C \; \{ \; X.h = f(C.attr) \; \} \; X \; \{ \; A.attr = X.s \; \} \\
X &\rightarrow B \; \{ \; X_1.h = g(X.h, B.attr) \; \} \; X_1 \; \{ \; X.s = X_1.s \; \} \\
&\rightarrow \epsilon \; \{ \; X.s = X.h \; \}
\end{aligned}
$$

Les grammaires LR(1)

- Les attributs peuvent être calculés lors des réductions
  $\hookrightarrow$ Nécessite de récupérer les attributs des $X_i \in N$ pour $A \rightarrow X_1 \ldots X_n$
- Avec les valeurs des attributs, on calcule les attributs de A
- Stockés dans une nouvelle pile appelée **pile des attributs**
  $\hookrightarrow$ En pratique, il est plus simple d'accéder directement aux attributs



Procédure liée à une règle

Soit la procédure associée à la règle de production n°k $A \rightarrow X_1 \dots X_n$

Procédure réduction$_k$()

  dépiler(n, PS) */* n symboles dépilés */*
  empiler(transition[sommet(PS), A]) */* Symbole suivant empilé */*
  Pour i = n à 1 par -1 **Faire** */* Calcul des attributs */*
    $X_i.attr$ = sommet(PA)
    dépiler(PA)
  Fin Pour
  $A.attr = f(X_1.attr, \dots, X_n.attr)$ */* Calcul des attributs de A */*
  empiler($A.attr$, PA)

Avec PS = Pile Syntaxique et PA = Pile des Attributs

Conclusion

LL(1) :

- Possible de définir des règles sémantiques où on le souhaite dans une règle de production
- Le calcul des attributs est plus complexe si la grammaire est récursive gauche

LR(1) :

- Règles sémantiques exécutées uniquement durant une réduction
- Solution pour “contourner” cette restriction :

$$A \rightarrow \alpha_1 \ \{ \ act_1 \ \} \ \alpha_2 \ \{ \ act_2 \ \}$$

devient :

$$\begin{array}{l} A \rightarrow B \ \alpha_2 \ \{ \ act_2 \ \} \\ B \rightarrow \alpha_1 \ \{ \ act_1 \ \} \end{array}$$