Traduction orientée-syntaxe

Sections 5.1 à 5.4 et 5.5.1 à 5.5.3

* Contenu *

- Buts et fonctionnement
- Outils
  - Définitions orientées-syntaxe
    - Définition S-Attribuée
    - Attributs hérités
    - Ordre d'évaluation (graphe des dépendances et tri topologique)
    - Arbres de syntaxe abstraite
    - Représentation de types
    - Définition L-Attribuée
  - Système de traduction
- Traduction descendante
  - Élimination de la récursion à gauche
  - Analyseur récursif
- Outils automatisés
  - Bison

Introduction

Chapitre 5

À propos de la technique de traduction orientée-syntaxe:

- c'est une technique qui permet d'effectuer la traduction de langages ou des calculs quelconques en étant guidé par une grammaire hors-contexte;
- but: **analyse sémantique**;
- on attache des *attributs* à chaque symbole de la grammaire;
- on donne des *règles sémantiques* qui spécifient de quelle façon le calcul des attributs et diverses actions doivent être effectués.

Conceptuellement, les étapes reliées à la traduction orientée-syntaxe sont:

- l'analyse [syntaxique] de la séquence de jetons;
- la construction de l'arbre de dérivation;
- la traversée de l'arbre de dérivation pour effectuer l'exécution des règles sémantiques.

Outils

Il existe deux sortes d'outils pour permettre de spécifier les règles sémantiques:

- les *définitions orientées-syntaxe*;
- les *systèmes de traduction*.

Les définitions orientées-syntaxe se situent à un plus haut niveau d'abstraction puisqu'elles ne font que déclarer de quelle façon on peut calculer la valeur des attributs sans spécifier l'ordre dans lequel ils doivent être calculés ni à quel moment.

Traduction orientée syntaxe:

Définitions orientées-syntaxe

Définitions orientées-syntaxe

Une définition orientée-syntaxe:

- est une généralisation d'une grammaire hors-contexte;
- où un ensemble d'attributs est attaché à chaque symbole de la grammaire;
- où un attribut est soit *synthétisé*, soit *hérité*;
- où un attribut peut représenter une valeur de n'importe quel type;
- comporte des règles sémantiques qui définissent la façon de calculer la valeur des attributs;
- où ces règles fixent des dépendances entre les attributs.

L'ensemble des attributs et des dépendances peuvent être représentés à l'aide d'un *graphe de dépendance* où les attributs sont les noeuds et où les dépendances sont les arcs.

On dit qu'on *décore* ou qu'on *annote* l'arbre de dérivation lorsqu'on calcule la valeur de ses attributs.

Définitions orientées-syntaxe

Dans une définition orientée-syntaxe, chaque production $A \rightarrow X_1\, X_2\, \ldots\, X_n$ de la grammaire est accompagnée de règles sémantiques de la forme $b := f(c_1, \ldots, c_k)$ où f est une fonction et où b, c_1, ..., c_k sont des attributs.

Un attribut est dénoté par $Y.nom$ où Y est le propriétaire de l'attribut du nom indiqué. Le propriétaire est soit A ou l'un des X_i.

- b est appelé un attribut *synthétisé* s'il appartient à A; i.e. b est $A.nom$;
- b est appelé un attribut *hérité* s'il appartient à l'un des X_i; i.e. b est $X_i.nom$.

Dans les deux cas, on dit que b *dépend* des attributs c_i.

Le choix du vocabulaire utilisant "hérité" et "synthétisé" vient du fait que, dans un arbre de dérivation, on dessine A comme parent des enfants X_1, ..., X_n.

On dit qu'une définition orientée-syntaxe est une *grammaire attribuée* si chacune des fonctions f ne cause aucun effet de bord.

Définitions orientées-syntaxe

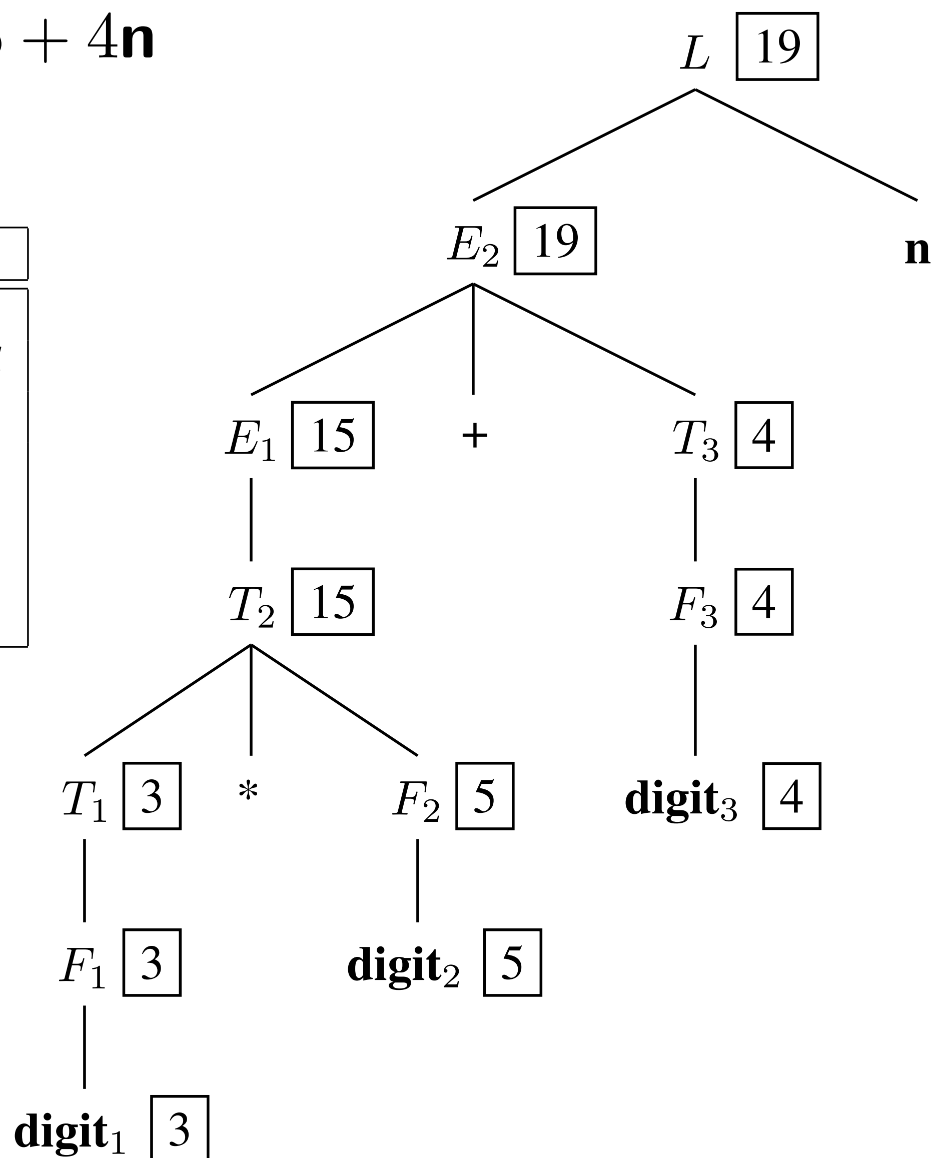
Exemple 5.1. Définition orientée-syntaxe pour des expressions arithmétiques simples

Productions	**Règles Sémantiques**
$L \rightarrow E$ **n**	$L.val = E.val$
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow$ **digit**	$F.val =$ **digit**$.lexval$

Définitions orientées-syntaxe

Exemple 5.2. Entrée: $3 * 5 + 4\mathbf{n}$

Productions	Règles Sémantiques
$L \rightarrow E$ **n**	$L.val = E.val$
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow$ **digit**	$F.val =$ **digit**$.lexval$



Définitions orientées-syntaxe S-attribuées

Une définition orientée-syntaxe est dite *S-attribuée* si tous les attributs qu'elle comporte sont synthétisés.

On peut toujours décorer l'arbre de dérivation d'une définition S-attribuée de manière ascendante.

Bien que l'on n'étudie pas l'analyse syntaxique ascendante (LR), on mentionne qu'on peut facilement adapter un analyseur syntaxique LR pour qu'il implante une définition S-attribuée.

Utilisation des attributs hérités

- Les attributs hérités sont utiles pour exprimer la dépendance d'un élément de programme sur le contexte dans lequel il se trouve. Ex:
  - déclaration préalable des variables;
  - type des variables;
  - la position (gauche ou droite) par rapport à un opérateur d'affectation;
  - etc.
- En principe, on peut récrire une définition orientée-syntaxe de façon à ce qu'elle ne comporte que des attributs synthétisés, mais on risque de perdre le naturel des règles et des attributs.

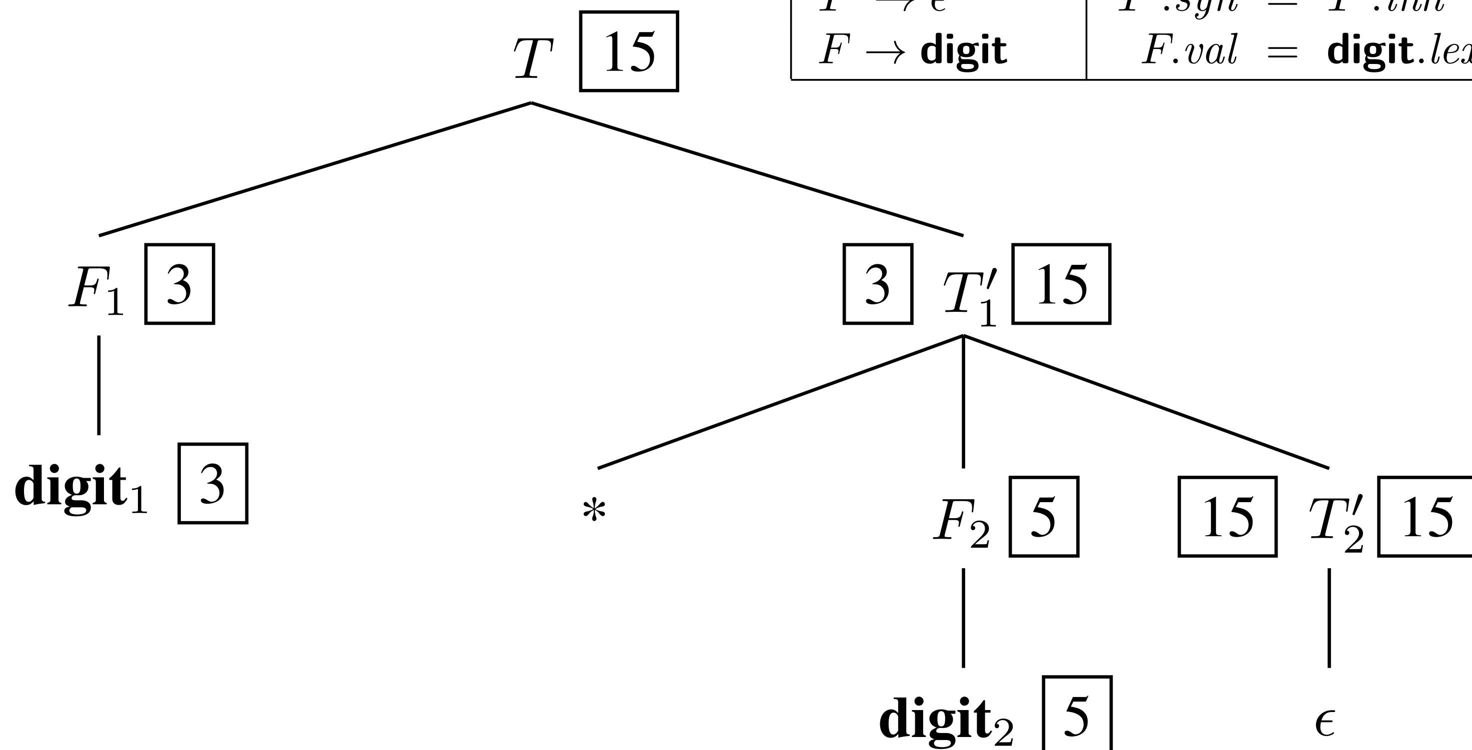
Exemple 5.3

Productions	Règles Sémantiques
$T \to F\ T'$	$T'.inh = F.val$
	$T.val = T'.syn$
$T' \to *\ F\ T'_1$	$T'_1.inh = T'.inh \times F.val$
	$T'.syn = T'_1.syn$
$T' \to \epsilon$	$T'.syn = T'.inh$
$F \to \textbf{digit}$	$F.val = \textbf{digit}.lexval$

Utilisation des attributs hérités (exemple 5.3)

Entrée: `3 * 5`

Productions	Règles Sémantiques
$T \rightarrow F\ T'$	$T'.inh = F.val$ $T.val = T'.syn$
$T' \rightarrow *\ F\ T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
$F \rightarrow$ **digit**	$F.val =$ **digit**$.lexval$



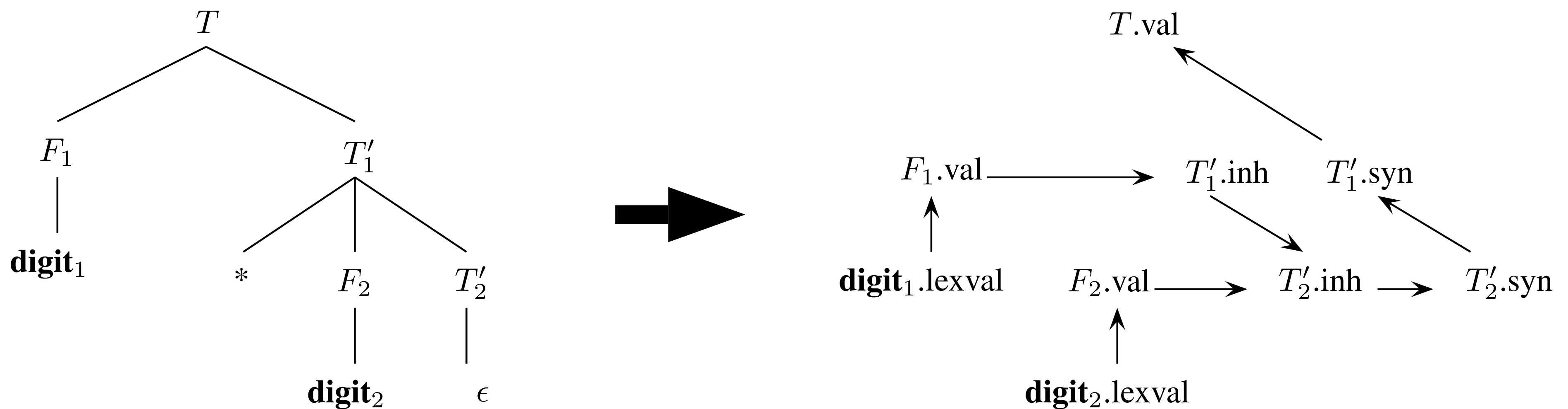
Les attributs hérités sont illustrés à gauche, et les attributs synthétisés, à droite.

Graphes de dépendance

Voici un algorithme permettant de construire le graphe de dépendance associé à un arbre de dérivation donné.

Pour chaque noeud n de l'arbre de dérivation **faire**
  pour chaque attribut a du symbole associé à n **faire**
    ajouter un noeud dans le graphe de dépendance pour a
Pour chaque noeud n de l'arbre de dérivation **faire**
  pour chaque règle sémantique $b := f(c_1, \ldots, c_k)$ associée à la production employée en n **faire**
    pour i allant de 1 à k **faire**
      ajouter un arc partant du noeud de c_i et aboutissant au noeud de b

Exemples 5.4, 5.5



Ordre d'évaluation des attributs

Un *tri topologique* d'un graphe acyclique G:

- est un ordonnancement $m_1, \ldots, m_k$ des noeuds de G;
- l'ordonnancement est tel que pour tout arc dans G, l'arc part d'un noeud situé *avant* le noeud où l'arc aboutit (selon l'ordonnancement);
- en d'autres mots, si $m_i \to m_j$ est un arc dans G, alors m_i apparaît avant m_j dans l'ordonnancement.

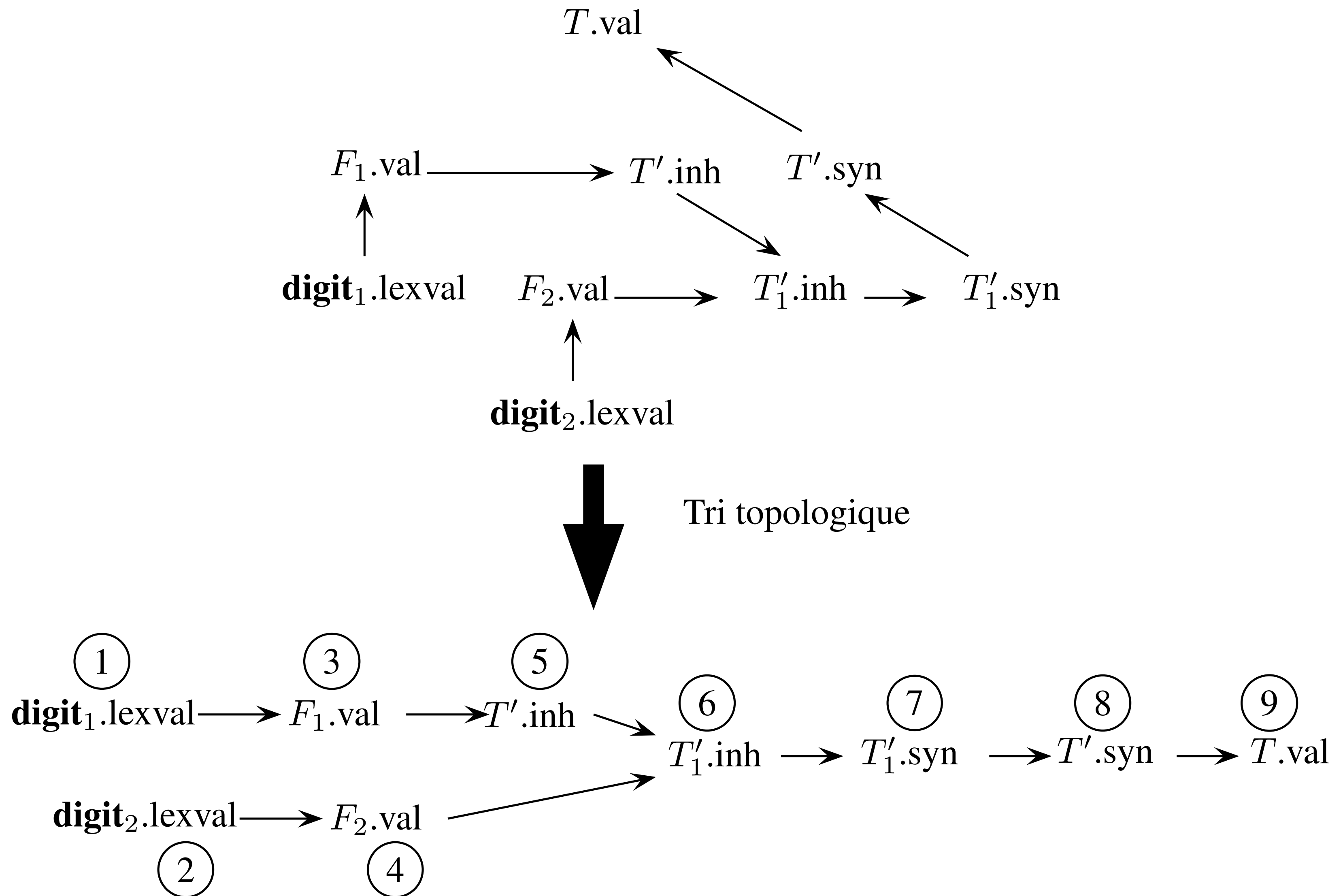
Étant donné un arbre de dérivation et ses attributs, le fait d'effectuer un tri topologique sur le graphe de dépendance correspondant permet de fixer un ordre *valide* d'évaluation des attributs.

L'ordre est valide au sens où pour chaque règle $b := f(c_1, \ldots c_k)$ qui calcule la valeur de b, le calcul de b est fait seulement après que tous les attributs c_i dont il dépend sont déjà évalués.

Si le graphe de dépendance associé à un arbre de dérivation *n*'est *pas* acyclique, c'est-à-dire qu'il comporte au moins un cycle, alors il est impossible d'évaluer tous les attributs.

Ordre d'évaluation des attributs

Exemple 5.6



Méthodes d'évaluation des règles sémantiques

On peut employer diverses méthodes pour faire l'évaluation des attributs, parmi lesquelles on retrouve:

- *La méthode de l'arbre de dérivation*. On construit l'arbre de dérivation, on crée le graphe de dépendance, on fait un tri topologique et on évalue les attributs dans l'ordre obtenu.
- *La méthode à base de règles*. Au moment d'implanter le compilateur, les règles sémantiques sont examinées afin de déterminer un ordre d'évaluation à suivre.
- *La méthode aveugle*. Un ordre d'évaluation arbitraire est choisi sans tenir compte des règles sémantiques. L'ordre d'évaluation est souvent lié à la technique d'analyse syntaxique employée. C'est à l'implanteur de créer ses règles sémantiques de façon à ce qu'elles soient compatibles avec l'ordre d'évaluation choisi.

Construction d'arbres de syntaxe

Section 5.3.1

On montre comment on peut construire des arbres de syntaxe à l'aide de définitions orientées-syntaxe.

On obtient plus de flexibilité en séparant la traduction (le reste du compilateur) de l'analyse syntaxique.

Les méthodes de traduction intégrées à l'analyse syntaxique sont sujettes à certaines contraintes:

1. Une grammaire adéquate pour l'analyse syntaxique ne reflète pas nécessairement la structure hiérarchique naturelle du langage analysé.
2. La méthode d'analyse syntaxique impose un certain ordre de visite des noeuds de l'arbre de syntaxe.

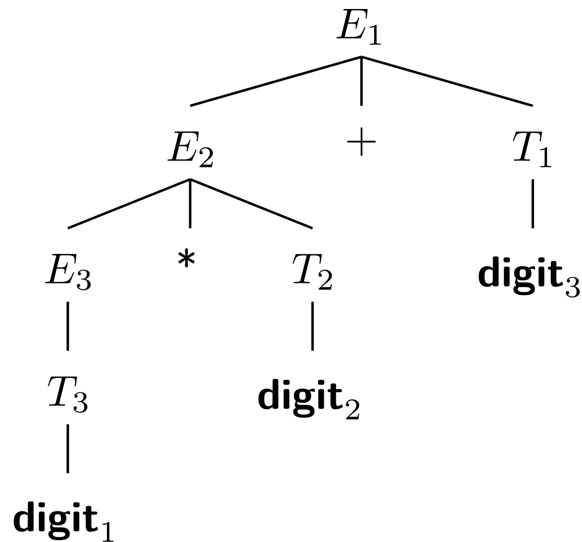
Un *arbre de syntaxe abstraite* est une version compacte de l'arbre de dérivation où:

- seules les informations pertinentes pour la suite de la compilation sont présentes;
- les mots-clés ou opérateurs sont les noeuds internes (ex. +) et où les entités conceptuellement subordonnées sont leurs enfants (ex. deux expressions).

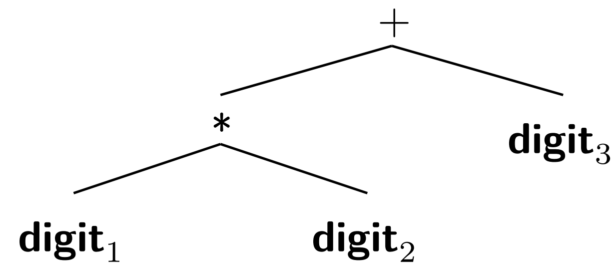
Construction d'arbres de syntaxe

Exemple: l'expression `3 * 5 + 4`

Arbre de syntaxe concrète

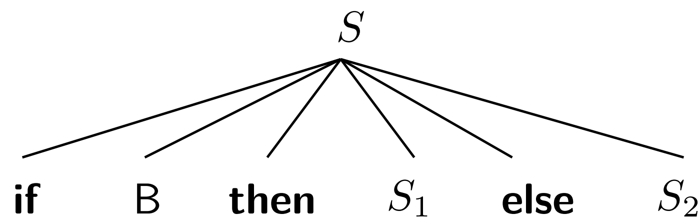


Arbre de syntaxe abstraite

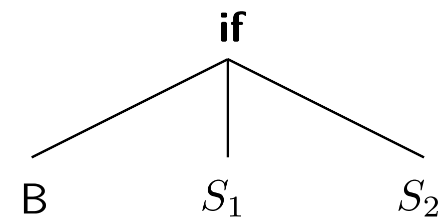


Exemple: la production $S \rightarrow \textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2$

Arbre de syntaxe concrète



Arbre de syntaxe abstraite



Construction d'arbres de syntaxe

Si on souhaite construire l'arbre de syntaxe abstraite pour des expressions:

- on définit des attributs $node$ pour chacun des non-terminaux décrivant la syntaxe des expressions;
- on écrit des règles sémantiques qui construisent l'arbre de syntaxe abstraite au fur et à mesure que les informations sont disponibles et qui stockent les noeuds ainsi construits dans les attributs $node$;
- les noeuds de l'arbre de syntaxe sont construits à l'aide des expressions suivantes:
  - $new\ Node(op, c_1, c_2, \ldots, c_k)$;
  - $new\ Leaf(op, val)$.

Construction d'arbres de syntaxe

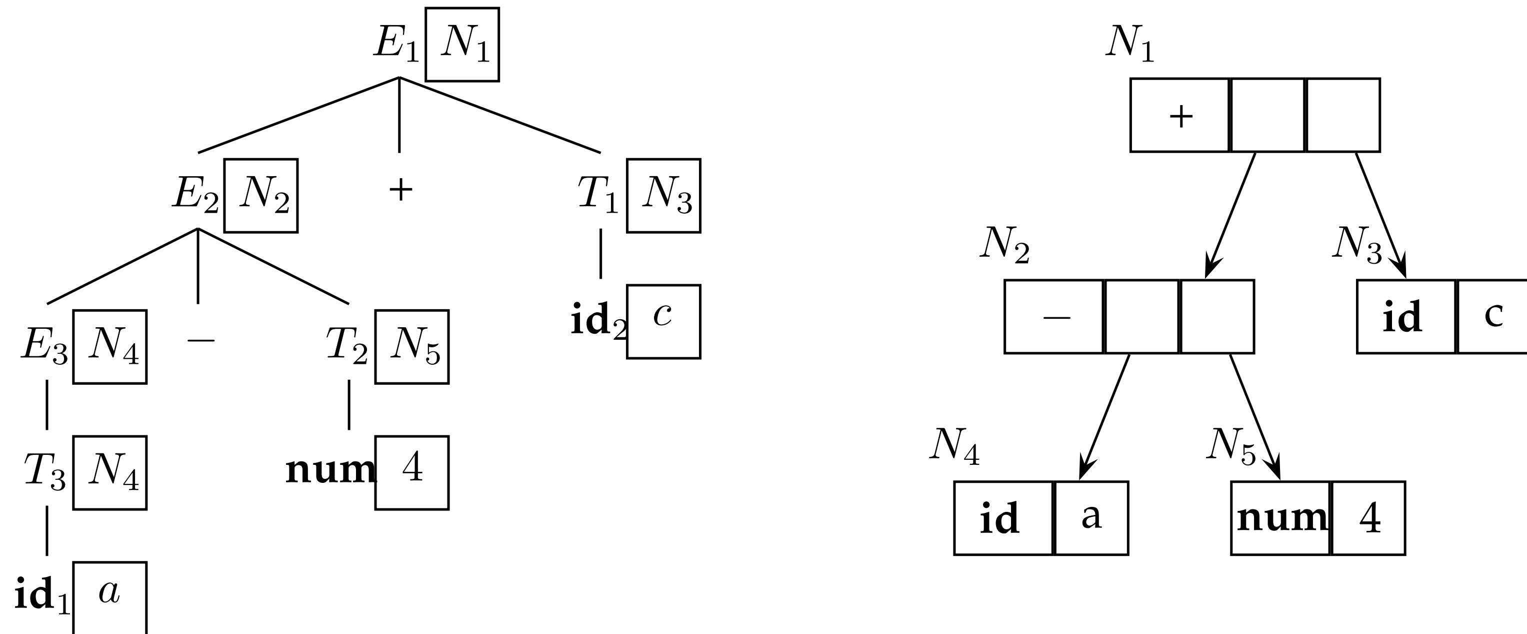
Exemple 5.11, figures 5.10 - 5.11

Voici une définition orientée-syntaxe S-attribuée qui construit un arbre de syntaxe abstraite pour des expressions arithmétiques simples.

Production	Règles Sémantiques
$E \to E_1 + T$	E.*node* := **new** Node('$+$', E_1.*node*, T.*node*)
$E \to E_1 - T$	E.*node* := **new** Node('$-$', E_1.*node*, T.*node*)
$E \to T$	E.*node* := T.*node*
$T \to (E)$	T.*node* := E.*node*
$T \to$ **id**	T.*node* := **new** Leaf(**id**, **id**.*entry*)
$T \to$ **num**	T.*node* := **new** Leaf(**num**, **num**.*val*)

Construction d'arbres de syntaxe (Exemple 5.11)

Entrée: $a - 4 + c$



Étapes de la construction:

$$
\begin{aligned}
N_4 &:= \textbf{new}\ Leaf(\textbf{id}, \textit{entry-a});\\
N_5 &:= \textbf{new}\ Leaf(\textbf{num}, 4);\\
N_2 &:= \textbf{new}\ Node(`-', N_4, N_5);\\
N_3 &:= \textbf{new}\ Leaf(\textbf{id}, \textit{entry-c});\\
N_1 &:= \textbf{new}\ Node(`+', N_2, N_3)
\end{aligned}
$$

Construction d'arbres de syntaxe

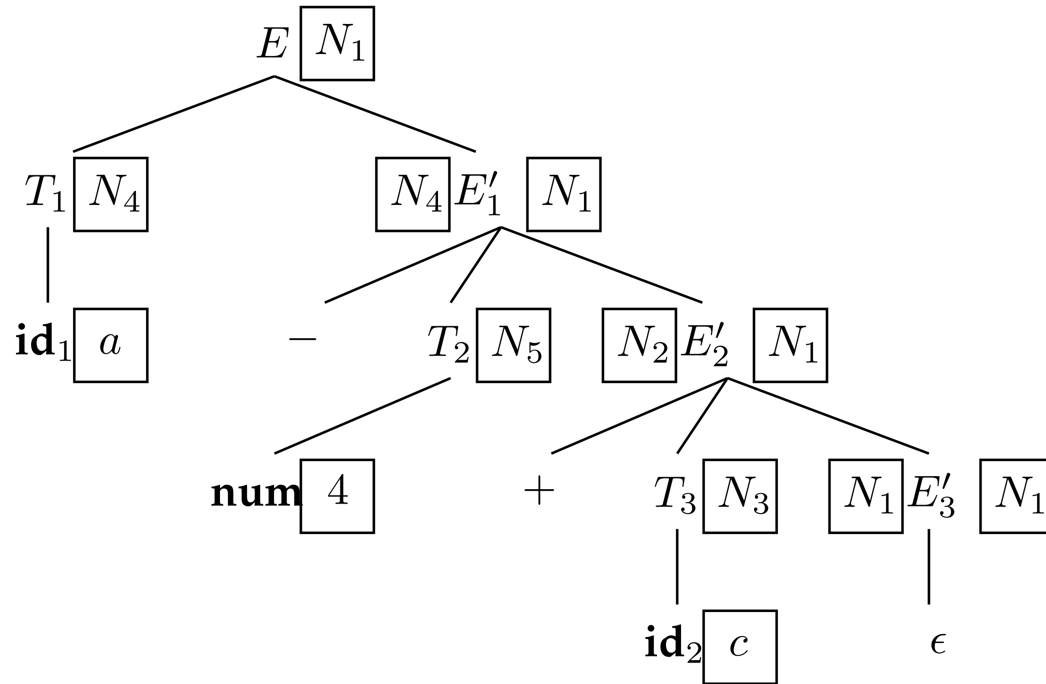
Exemple 5.12, figure 5.13

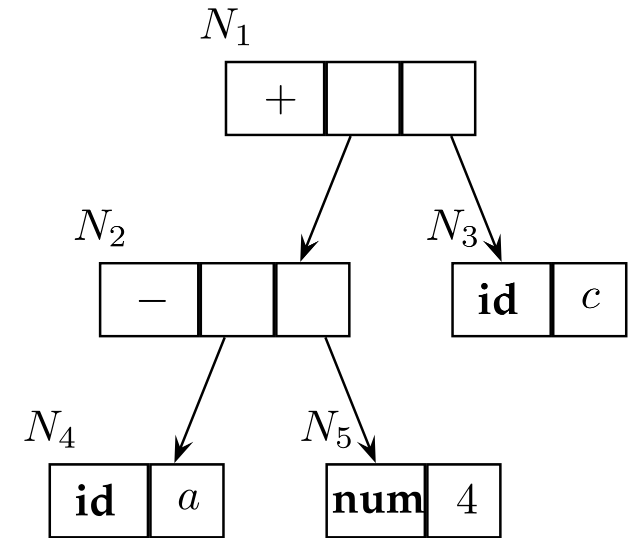
Voici une définition orientée-syntaxe *L-attribuée* (notion présentée à la page 28) qui donne le même résultat:

Production	Règles Sémantiques
$E \rightarrow TE'$	E.*node* := E'.*syn* E'.*inh* := T.*node*
$E' \rightarrow +TE'_1$	E'_1.*inh* := **new** Node('+', E'.*inh*, T.*node*) E'.*syn* := E'_1.*syn*
$E' \rightarrow -TE'_1$	E'_1.*inh* := **new** Node('−', E'.*inh*, T.*node*) E'.*syn* := E'_1.*syn*
$E' \rightarrow \epsilon$	E'.*syn* := E'.*inh*
$T \rightarrow (E)$	T.*node* := E.*node*
$T \rightarrow$ **id**	T.*node* := **new** Leaf(**id**, **id**.*entry*)
$T \rightarrow$ **num**	T.*node* := **new** Leaf(**num**, **num**.*val*)

Construction d'arbres de syntaxe (Exemple 5.12)

Entrée: $a - 4 + c$





Étapes de la construction:

$$\begin{aligned}
N_4 &:= \textbf{new } Leaf(\textbf{id}, \textit{entry-a});\\
N_5 &:= \textbf{new } Leaf(\textbf{num}, 4);\\
N_2 &:= \textbf{new } Node(`-', N_4, N_5);\\
N_3 &:= \textbf{new } Leaf(\textbf{id}, \textit{entry-c});\\
N_1 &:= \textbf{new } Node(`+', N_2, N_3)
\end{aligned}$$

Construction d'arbres de syntaxe

Omission complète des arbres de dérivation possible dans certains cas.

Définition orientée-syntaxe *L-attribuée*
(notion présentée plus loin, p.28)

$+$ Intégration de la traduction orientée-syntaxe à l'analyse syntaxique
(notion présentée plus loin, p.37)

$\Rightarrow$ Construction directe de l'arbre de syntaxe abstraite
sans passer par l'arbre de dérivation

Représentation des types

Section 5.3.2

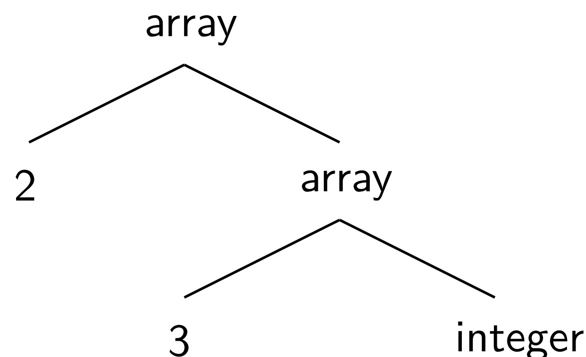
La représentation interne des types ressemble à celle des arbres de syntaxe.

Les attributs hérités peuvent aider à construire la représentation de certains types.

Exemple 5.13: Nous pouvons représenter le type des tableaux à n dimensions en définissant un type “array” qui possède deux paramètres:

- Le nombre d'éléments dans le tableau
- Le type des éléments dans le tableau

On pourrait alors représenter `int[2][3]` (qui correspond à “tableau de 2 tableaux de 3 entiers”) par:



Représentation des types (exemple 5.13)

La définition orientée-syntaxe suivante permet de construire la représentation interne de types **int**, **float** ou encore *tableau à n dimensions* d'un de ces deux types.

Production	Règles Sémantiques
$T \to BC$	$T.t := C.t$
	$C.b := B.t$
$B \to$ **int**	$B.t :=$ *integer*
$B \to$ **float**	$B.t :=$ *float*
$C \to [$ **num** $] \; C_1$	$C.t :=$ *array*(**num**.*val*, $C_1.t$)
	$C_1.b := C.b$
$C \to \epsilon$	$C.t := C.b$

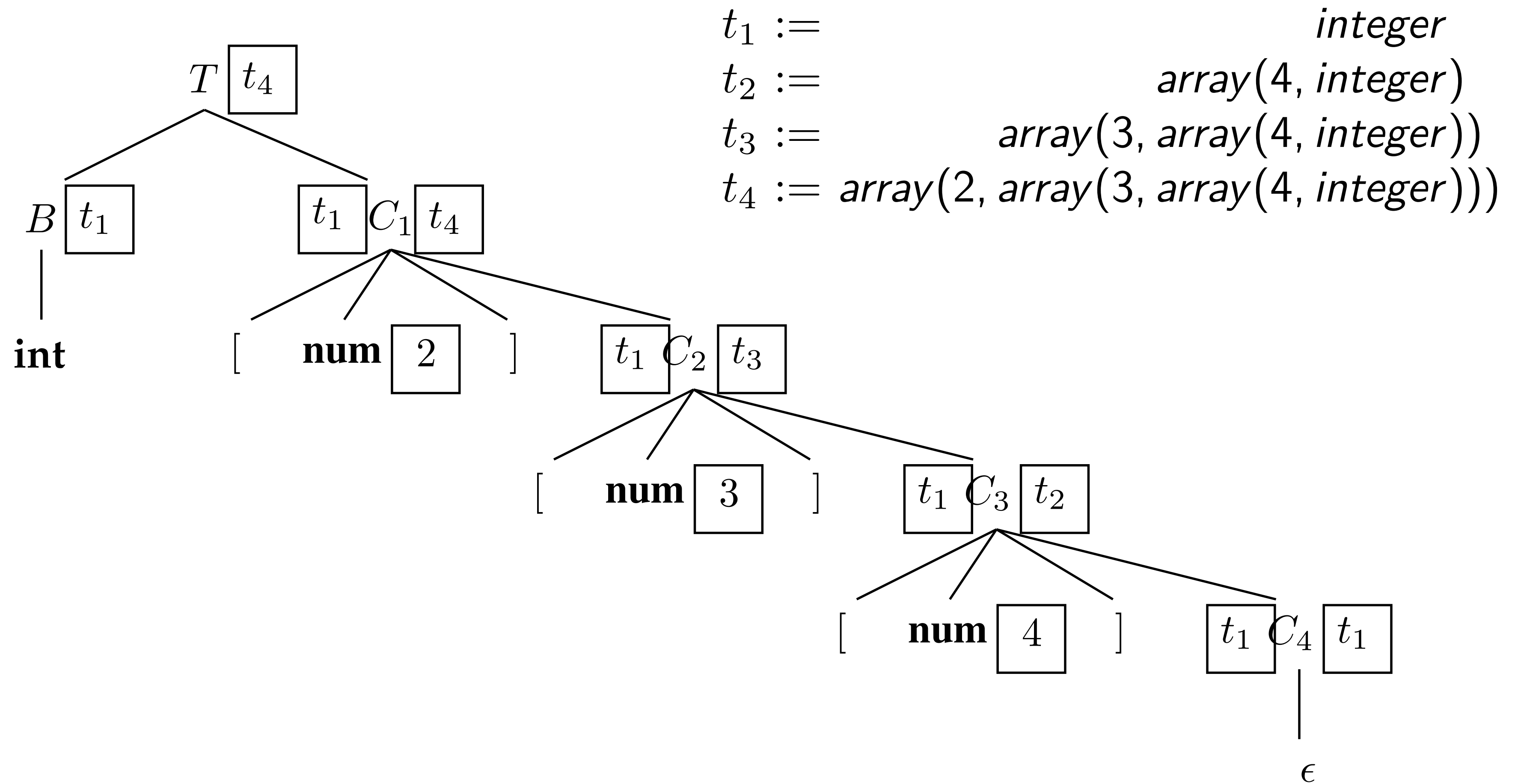
Le non-terminal B sert à générer un type de base et le non-terminal C sert à ajouter des dimensions afin de former des tableaux.

L'attribut b sert à faire hériter le type de base à partir du point de déclaration (B), tout le long de la séquence de dimensions. L'attribut t sert à synthétiser le type construit. Ainsi,
l'entrée **int** est convertie en le type interne *integer*,
l'entrée **float**$[4]$ est convertie en le type interne *array*$(4,$ *float*$)$ et
l'entrée **int**$[2][3][4]$ est convertie en le type interne *array*$(2,$ *array*$(3,$ *array*$(4,$ *integer*$)))$.

Note: la première dimension est dite *majeure* et la dernière dimension est dire *mineure*.

Représentation des types (exemple 5.13)

Entrée: **int**$[2][3][4]$



Définitions L-attribuées

Section 5.2.4

Un peu plus général que l'ordre de parcours strictement ascendant de l'arbre de dérivation, il y a l'ordre de parcours en profondeur d'abord.

Cet ordre de parcours est décrit par l'algorithme suivant:

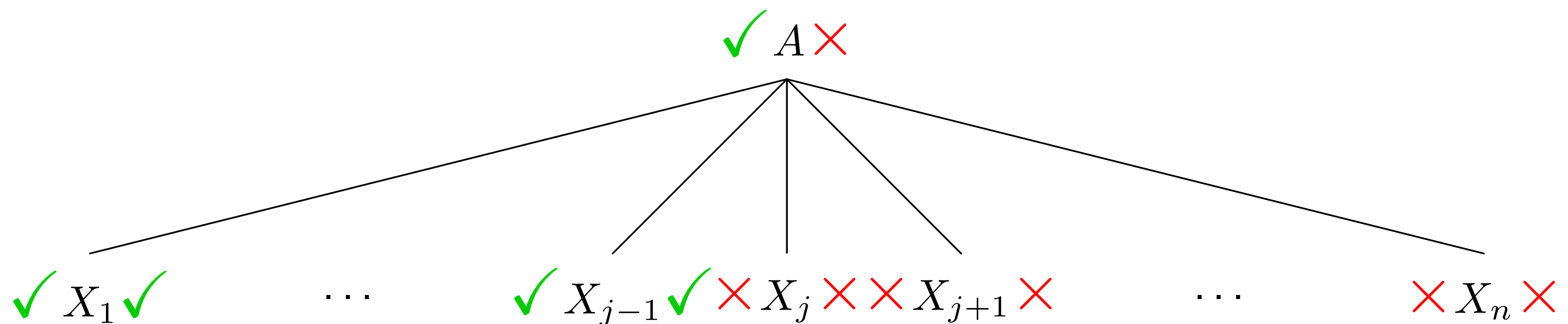
> **procédure** *dfvisit*(n: node);
> **début**
>   **pour** chaque fils m de n, de gauche à droite **faire**
>   **début**
>   évaluer les attributs hérités de m;
>   *dfvisit*(m)
>   **fin**;
>   évaluer les attributs synthétisés de n
> **fin**

Définitions L-attribuées

Une définition orientée-syntaxe est *L-attribuée* si chaque attribut hérité de X_j, $1 \leq j \leq n$, dans le membre droit de $A \to X_1 \ldots X_n$, ne dépend que:

- des attributs des symboles X_1, ..., X_{j-1} et
- des attributs hérités de A.

Informellement, une définition est L-attribuée si on lit uniquement des attributs à gauche et/ou en haut du symbole courant.



Propriété: Toute définition S-attribuée est aussi L-attribuée.

Note: Ces restrictions sont naturelles si on imagine que les attributs sont calculés au cours d'une traversée en profondeur d'abord de l'arbre de dérivation.

Définitions L-attribuées

Exemple 5.8: Définition L-attribuée:

Production	**Règles Sémantiques**
$T \to FT'$	$T'.inh := F.val$
$T' \to *FT'_1$	$T'_1.inh := T'.inh \times F.val$

Exemple 5.9: Définition qui **n**'est **pas** L-attribuée:

Production	**Règles Sémantiques**
$A \to BC$	$A.s := B.b$
	$B.i := f(C.c, A.s)$

Traduction orientée-syntaxe:

Systèmes de traduction

Systèmes de traduction

Un *système de traduction*:

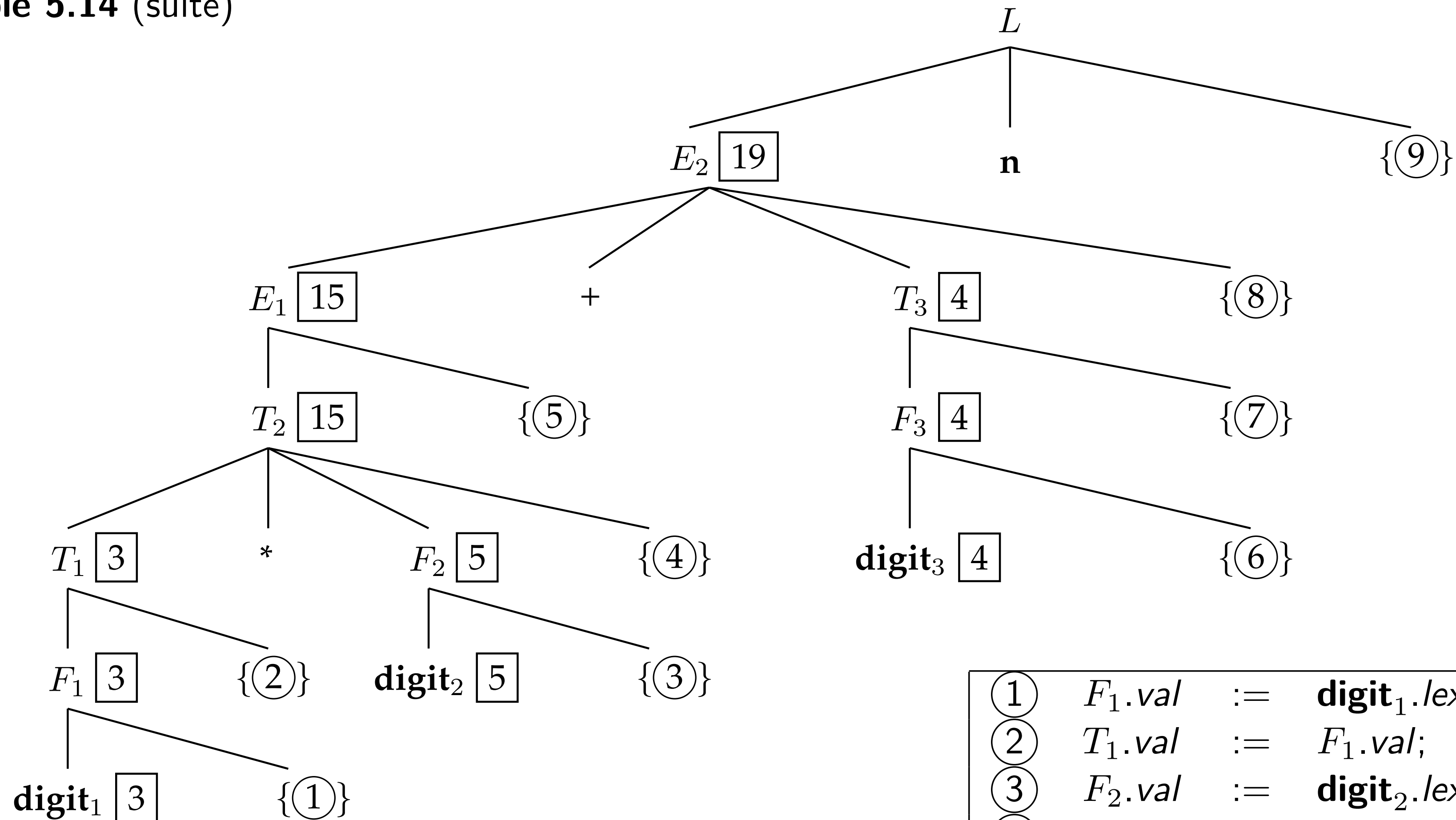
- est une grammaire hors-contexte;
- associe des attributs aux symboles de la grammaire;
- permet l'insertion d'*actions sémantiques* à l'intérieur des membres droits des productions; ces actions sémantiques sont inscrites entre accolades ($\{\}$);
- permet l'utilisation d'attributs hérités et synthétisés (moyennant certaines précautions).

Systèmes de traduction

Exemple 5.14: système de traduction pour calculatrice rudimentaire

$L \rightarrow E$ **n**	{ print(E.*val*); }
$E \rightarrow E_1 + T$	{ 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*; }
$F \rightarrow$ **digit**	{ F.*val*:= **digit**.*lexval*; }

Exemple 5.14 (suite)



①	$F_1.\mathit{val}$	$:=$	$\mathbf{digit}_1.\mathit{lexval}$;
②	$T_1.\mathit{val}$	$:=$	$F_1.\mathit{val}$;
③	$F_2.\mathit{val}$	$:=$	$\mathbf{digit}_2.\mathit{lexval}$;
④	$T_2.\mathit{val}$	$:=$	$T_1.\mathit{val} \times F_2.\mathit{val}$;
⑤	$E_1.\mathit{val}$	$:=$	$T_2.\mathit{val}$;
⑥	$F_3.\mathit{val}$	$:=$	$\mathbf{digit}_3.\mathit{lexval}$;
⑦	$T_3.\mathit{val}$	$:=$	$F_3.\mathit{val}$;
⑧	$E_2.\mathit{val}$	$:=$	$E_1.\mathit{val} + T_3.\mathit{val}$;
⑨	print($E_2.\mathit{val}$);		

Systèmes de traduction

Lorsqu'un système de traduction ne comporte que des attributs synthétisés, il suffit d'ajouter les actions sémantiques à la fin des productions. Ainsi, on s'assure que les attributs des noeuds enfants sont déjà calculés lorsqu'on tente de calculer les attributs des noeuds parents.

Toutefois, si le système de traduction comporte aussi des attributs hérités, on doit faire attention aux choses suivantes:

1. Un attribut hérité d'un symbole dans le membre droit d'une production doit avoir été calculé par une action précédant ce symbole.

2. Une action ne peut tenter d'utiliser un attribut d'un symbole suivant cette action.

3. Un attribut synthétisé du symbole dans le membre gauche d'une production ne peut être calculé que lorsque tous les attributs desquels il dépend ont été calculés.

Systèmes de traduction

Exemple de système de traduction qui ne respecte pas l'ordre de traversée en profondeur d'abord dans son calcul des attributs. (Il est naïvement traduit de la définition orientée-syntaxe vue à la page 26.)

$$
\begin{array}{lcll}
T & \rightarrow & B\ C & \{T.t := C.t;\ C.b := B.t\} \\
B & \rightarrow & \textbf{int} & \{B.t := \textit{integer}\} \\
B & \rightarrow & \textbf{float} & \{B.t := \textit{float}\} \\
C & \rightarrow & [\ \textbf{num}\ ]\ C_1 & \{C.t := \textit{array}(\textbf{num}.\textit{val},\ C_1.t);\ C_1.b := C.b\} \\
C & \rightarrow & \epsilon & \{C.t := C.b\}
\end{array}
$$

En plaçant correctement les actions, on peut s'arranger pour faire en sorte que le calcul des attributs respecte l'ordre de traversée en profondeur d'abord.

$$
\begin{array}{lcl}
T & \rightarrow & B\ \{C.b := B.t\}\ C\ \{T.t := C.t\} \\
B & \rightarrow & \textbf{int}\ \{B.t := \textit{integer}\} \\
B & \rightarrow & \textbf{float}\ \{B.t := \textit{float}\} \\
C & \rightarrow & [\ \textbf{num}\ ]\ \{C_1.b := C.b\}\ C_1\ \{C.t := \textit{array}(\textbf{num}.\textit{val},\ C_1.t)\} \\
C & \rightarrow & \epsilon\ \{C.t := C.b\}
\end{array}
$$

Traduction orientée syntaxe:

Traduction descendante (élimination de la récursion à gauche)

Traduction descendante

Section 5.4.4

Nous étudions l'intégration de la traduction orientée-syntaxe avec un analyseur syntaxique descendant.

Afin d'être plus précis sur la forme des calculs, des systèmes de traduction sont utilisés.

La technique d'élimination de la récursion à gauche dans les grammaires hors-contexte est étendue pour éliminer la récursion à gauche dans les systèmes de traduction.

En particulier, lorsque le système de traduction original contient des attributs synthétisés, les actions qui servent à les évaluer doivent être adaptées pour correspondre à la forme du nouveau système.

Élimination de la récursion à gauche

Exemple 5.17

$$
\begin{array}{lll}
E & \rightarrow & E_1 + T \quad \{ \text{print('+');} \} \\
E & \rightarrow & T
\end{array}
$$

se transforme en:

$$
\begin{array}{lll}
E & \rightarrow & T\ R \\
R & \rightarrow & +\ T\ \{ \text{print('+');} \}\ R \\
R & \rightarrow & \epsilon
\end{array}
$$

Élimination de la récursion à gauche

Considérons plus abstraitement le problème de l'élimination de la récursion à gauche dans un système de traduction. Soit le système suivant:

$$
\begin{array}{lcll}
A & \rightarrow & A_1 \; Y & \{A.a := g(A_1.a, Y.y)\} \\
A & \rightarrow & X & \{A.a := f(X.x)\}
\end{array}
$$

À partir de la grammaire hors-contexte sous-jacente, la technique d'élimination de la récursion à gauche produit la grammaire suivante:

$$
\begin{array}{lcl}
A & \rightarrow & X \; R \\
R & \rightarrow & Y \; R \mid \epsilon
\end{array}
$$

Comment choisir des actions qui permettent de retrouver des attributs ayant les mêmes valeurs? Dans le système original, les attributs $A_i.a$ sont fonctions des $A_j.a$ précédents. Dans la nouvelle grammaire, la récursion se fait à droite.

Dans le cas de notre système abstrait, on doit choisir les actions suivantes:

$$
\begin{array}{lcll}
A & \rightarrow & X & \{R.i := f(X.x)\} \\
 & & R & \{A.a := R.s\} \\
R & \rightarrow & Y & \{R_1.i := g(R.i, Y.y)\} \\
 & & R_1 & \{R.s := R_1.s\} \\
R & \rightarrow & \epsilon & \{R.s := R.i\}
\end{array}
$$

Élimination de la récursion à gauche

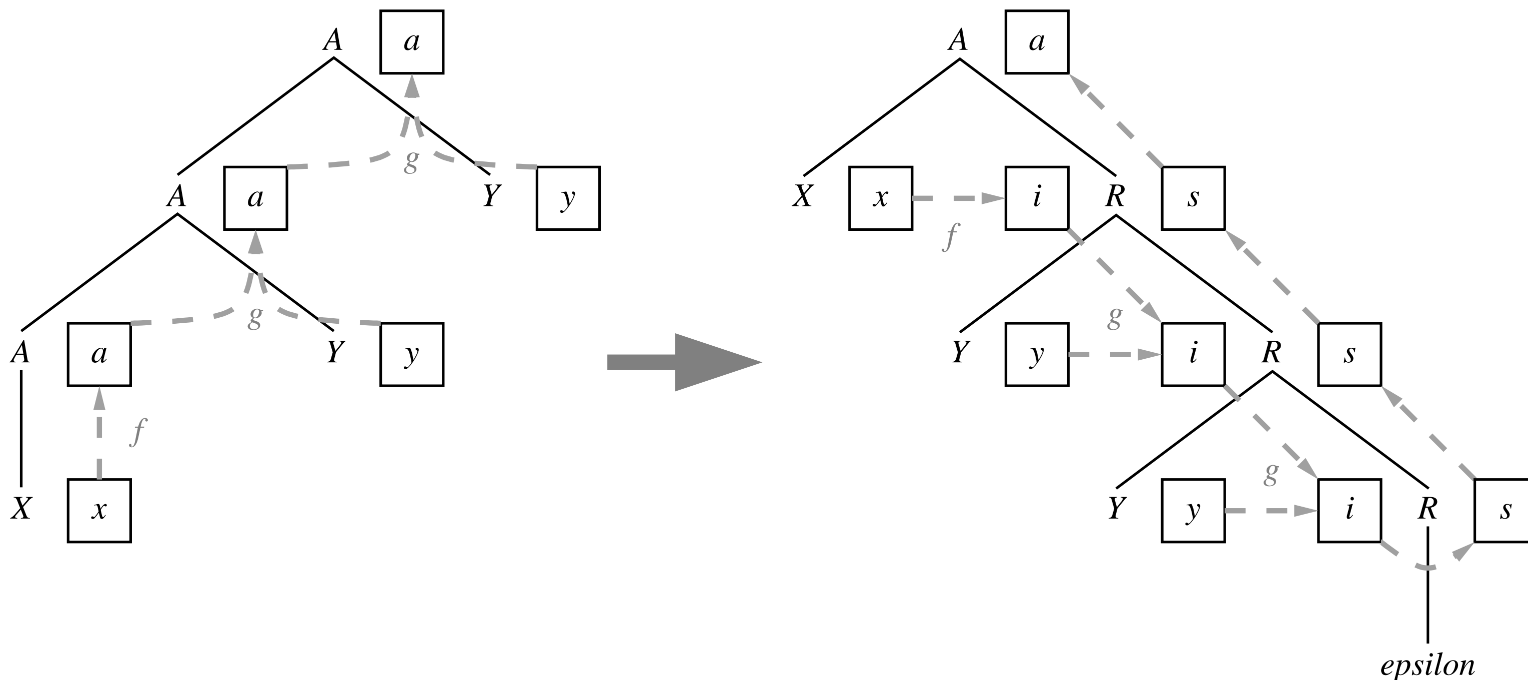
Système initial

$$
\begin{array}{llll}
A & \rightarrow & A_1\ Y & \{A.a := g(A_1.a, Y.y)\} \\
A & \rightarrow & X & \{A.a := f(X.x)\}
\end{array}
$$

Système transformé

$$
\begin{array}{llll}
A & \rightarrow & X & \{R.i := f(X.x)\} \\
 & & R & \{A.a := R.s\} \\
R & \rightarrow & Y & \{R_1.i := g(R.i, Y.y)\} \\
 & & R_1 & \{R.s := R_1.s\} \\
R & \rightarrow & \epsilon & \{R.s := R.i\}
\end{array}
$$

Illustration avec “XYY”:



Élimination de la récursion à gauche

Exemple: application au système suivant

$$
\begin{array}{lcll}
E & \rightarrow & E_1 + T & \{ E.\textit{node} := \textbf{new} \text{ Node}(`+', E_1.\textit{node}, T.\textit{node}) \} \\
E & \rightarrow & E_1 - T & \{ E.\textit{node} := \textbf{new} \text{ Node}(`-', E_1.\textit{node}, T.\textit{node}) \} \\
E & \rightarrow & T & \{ E.\textit{node} := T.\textit{node} \} \\
T & \rightarrow & (E) & \{ T.\textit{node} := E.\textit{node} \} \\
T & \rightarrow & \textbf{id} & \{ T.\textit{node} := \textbf{new} \text{ Leaf}(\textbf{id}, \textbf{id}.\textit{entry}) \}
\end{array}
$$

On obtient:

$$
\begin{array}{lcll}
E & \rightarrow & T & \{ R.i := \text{T}.\textit{node} \} \\
 & & R & \{ E.\textit{node} := R.s \} \\
R & \rightarrow & + \, T & \{ R_1.i := \textbf{new} \text{ Node}(`+', R.i, T.\textit{node}) \} \\
 & & R_1 & \{ R.s := R_1.s \} \\
R & \rightarrow & - \, T & \{ R_1.i := \textbf{new} \text{ Node}(`-', R.i, T.\textit{node}) \} \\
 & & R_1 & \{ R.s := R_1.s \} \\
R & \rightarrow & \epsilon & \{ R.s := R.i \} \\
T & \rightarrow & (E) & \{ T.\textit{node} := E.\textit{node} \} \\
T & \rightarrow & \textbf{id} & \{ T.\textit{node} := \textbf{new} \text{ Leaf}(\textbf{id}, \textbf{id}.\textit{entry}) \}
\end{array}
$$

Traduction orientée syntaxe:

Analyseur récursif

Traduction descendante: analyseur récursif

Section 5.5.1

Entrée. Un système de traduction basée sur une grammaire hors-contexte adéquate pour effectuer de l'analyse prédictive.

Sortie. Le code d'un traducteur orienté-syntaxe récursif.

Méthode.

1. Pour chaque non-terminal A, construire une fonction qui a un paramètre par attribut hérité et qui retourne le (présumé) unique attribut synthétisé. (Quand il y a plusieurs attributs synthétisés, on peut utiliser une structure.)

2. Tel que vu précédemment, le code de la fonction commence par déterminer quel membre droit doit être utilisé.

3. Le code généré pour un membre droit donné comporte une suite d'instructions pour chacun des symboles du membre droit, en les considérant de gauche à droite:

   (a) Pour un terminal X qui possède l'attribut synthétisé x, le bout de code consiste à sauvegarder la valeur de $X.x$ dans une variable prévue à cet effet et, ensuite, à consommer le terminal.

   (b) Pour un non-terminal B, le bout de code est l'affectation $c := B(b_1, \ldots, b_k)$ où les variables b_i contiennent la valeur des attributs hérités de B et c, celle de l'attribut synthétisé de B.

   (c) Pour une action, le bout de code est celui de l'action tel quel, en renommant les attributs utilisés dans le système de traduction pour mentionner les variables utilisées par l'analyseur.

Traduction descendante: analyseur récursif

Exemple: en partant de ce système de traduction:

$$
\begin{array}{lcll}
E & \rightarrow & T & \{\ E'.\mathit{inh}\ := T.\mathit{node}\ \} \\
 & & E' & \{\ E.\mathit{node} := E'.\mathit{syn}\ \} \\
E' & \rightarrow & +\ T & \{\ E'_1.\mathit{inh}\ := \textbf{new}\ \text{Node}(`+', E'.\mathit{inh}, T.\mathit{node})\ \} \\
 & & E'_1 & \{\ E'.\mathit{syn}\ := E'_1.\mathit{syn}\ \} \\
E' & \rightarrow & -\ T & \{\ E'_1.\mathit{inh}\ := \textbf{new}\ \text{Node}(`-', E'.\mathit{inh}, T.\mathit{node})\ \} \\
 & & E'_1 & \{\ E'.\mathit{syn}\ := E'_1.\mathit{syn}\ \} \\
E' & \rightarrow & \epsilon & \{\ E'.\mathit{syn}\ := E'.\mathit{inh}\ \} \\
T & \rightarrow & (E) & \{\ T.\mathit{node} := E.\mathit{node}\ \} \\
T & \rightarrow & \textbf{id} & \{\ T.\mathit{node} := \textbf{new}\ \text{Leaf}(\textbf{id}, \textbf{id}.\mathit{entry})\ \} \\
T & \rightarrow & \textbf{num} & \{\ T.\mathit{node} := \textbf{new}\ \text{Leaf}(\textbf{num}, \textbf{num}.\mathit{val})\ \}
\end{array}
$$

On écrit un analyseur récursif ainsi:

- Une fonction E qui ne reçoit aucun paramètre et qui retourne une valeur de type `Node`;
- Une fonction Eprime qui reçoit un paramètre `inh` de type `Node` et qui retourne une valeur de type `Node`;
  - Cette fonction peut distinguer entre les diverses productions en observant si le prochain jeton est un '$+$', un '$-$' ou si on a atteint la fin de l'expression;
- Une fonction T qui ne reçoit aucun paramètre et qui retourne une valeur de type `Node`;
  - Cette fonction peut distinguer entre les diverses productions en observant si le prochain jeton est une parenthèse ouvrante, un **id** ou encore un **num**.

Traduction descendante: analyseur récursif (exemple)

Pseudo-code (version longue):

```
/** Analyseur pour E -> T E' */
Node E() {
    Node T_node, Eprime_inh, Eprime_syn, E_node;

    T_node = T();
    Eprime_inh = T_node;
    Eprime_syn = Eprime(Eprime_inh);
    E_node = Eprime_syn;
    return E_node;
}
```

Traduction descendante: analyseur récursif (exemple)

Pseudo-code (version longue, suite. . .):

```
/** Analyseur pour E' -> + T E'_1 | - T E'_1 | epsilon */
Node Eprime(Node Eprime_inh) {
    Node T_node, Eprime1_inh, Eprime1_syn, Eprime_syn;

    if (peek_token().type == JETON_PLUS) {
        read_token(JETON_PLUS);
        T_node  = T();
        Eprime1_inh = new Node(NODE_PLUS, Eprime_inh, T_node);
        Eprime1_syn = Eprime( Eprime1_inh );
        Eprime_syn = Eprime1_syn;
    }
    else if (peek_token().type == JETON_MOINS) {
        read_token(JETON_MOINS);
        T_node  = T();
        Eprime1_inh = new Node(NODE_MOINS, Eprime_inh, T_node);
        Eprime1_syn = Eprime( Eprime1_inh );
        Eprime_syn = Eprime1_syn;
    }
    else /* epsilon */ {
        Eprime_syn = Eprime_inh;
    }
    return Eprime_syn;
}
```

Traduction descendante: analyseur récursif (exemple)

Pseudo-code (version longue, suite. . .):

```
/** Analyseur pour T -> ( E ) | id | num */
Node T() {
    Node E_node, T_node;

    if (peek_token().type == JETON_PARENTHESE_OUVRANTE) {
        read_token(JETON_PARENTHESE_OUVRANTE);
        E_node = E();
        read_token(JETON_PARENTHESE_FERMANTE);
        T_node = E_node;
    }
    else if (peek_token().type == JETON_ID) {
        T_node = new Leaf( LEAF_ID, peek_token().attribute );
        read_token(JETON_ID);
    }
    else /* JETON_NUM */ {
        T_node = new Leaf( LEAF_NUM, peek_token().attribute );
        read_token(JETON_NUM);
    }
    return T_node;
}
```

Traduction descendante: analyseur récursif (exemple)

Pseudo-code (version courte):

```
/** Analyseur pour E -> T E' */
Node E() {
    return Eprime(T());
}

/** Analyseur pour E' -> + T E'_1 | - T E'_1 | epsilon */
Node Eprime(Node Eprime_inh) {
    if (peek_token().type == JETON_PLUS) {
        read_token(JETON_PLUS);
        return Eprime( new Node(NODE_PLUS, Eprime_inh, T()) )
    }
    else if (peek_token().type == JETON_MOINS) {
        read_token(JETON_MOINS);
        return Eprime( new Node(NODE_MOINS, Eprime_inh, T()) )
    }
    else return Eprime_inh; /* epsilon */
}
```

Traduction descendante: analyseur récursif (exemple)

Pseudo-code (version courte, suite...):

```
/** Analyseur pour T -> ( E ) | id | num */
Node T() {
    Node T_node;

    if (peek_token().type == JETON_PARENTHESE_OUVRANTE) {
        read_token(JETON_PARENTHESE_OUVRANTE);
        T_node = E();
        read_token(JETON_PARENTHESE_FERMANTE);
    }
    else if (peek_token().type == JETON_ID) {
        T_node = new Leaf( LEAF_ID, peek_token().attribute );
        read_token(JETON_ID);
    }
    else /* JETON_NUM */ {
        T_node = new Leaf( LEAF_NUM, peek_token().attribute );
        read_token(JETON_NUM);
    }
    return T_node;
}
```

Traduction descendante: analyseur récursif

Remarques

Tel que mentionné à la page 24,
grâce à cette implantation du système de traduction,
nous sommes en mesure de construire un arbre de syntaxe abstraite
sans construire d'arbre de syntaxe concrète.

Les appels récursifs des fonctions d'analyse entre elles
suivent la forme qu'aurait eu l'arbre de syntaxe concrète,
s'il avait existé.

Traduction orientée-syntaxe

Conclusion

Le cheminement suivant peut être suivi lorsqu'on développe un analyseur syntaxique qui construit un arbre de syntaxe abstraite destiné à être transmis au reste du compilateur.

Spécifications du langage: strictement textuelles

$\mapsto$

Définition orientée-syntaxe: formalisation des calculs à faire

$\mapsto$

Système de traduction: fixation de la façon de faire les calculs

$\mapsto$

Implantation de l'analyseur: utilisable sur un ordinateur

Traduction orientée syntaxe:

Outils automatisés: Bison

Bison

Il existe des outils permettant de générer automatiquement un analyseur syntaxique. Bison est un logiciel permettant de générer un analyseur syntaxique à partir d'un système de traduction orientée-syntaxe (`http://www.gnu.org/software/bison/`). Bison supporte les grammaires LR(1).

La syntaxe est similaire aux systèmes de traductions vus en cours.

$$stmt \rightarrow \text{`+'} \ \textbf{num}$$

s'écrit ainsi en Bison:

```
stmt :  '+' NUM;
```

(Par convention, en Bison, les non-terminaux sont écrits en minuscules, et les terminaux (jetons), en majuscules).

L'**attribut synthétisé** d'un symbole est noté `$$` (pour synthétiser plusieurs attributs, on utilise une `struct`). On peut lire l'attribut synthétisé d'un symbole en utilisant $1, $2, ... où le nombre est la position du symbole dans la production.

Bison s'utilise comme ceci (sur le terminal):

```
bison --output=TestBison.c ConfigurationBison.y
```

Cette commande lit le système de traduction défini dans "ConfigurationBison.y" et génère un analyseur syntaxique dans "TestBison.c"

Note: Bison peut aussi générer des analyseurs syntaxiques en C++ et en Java.

Bison

Voici un fichier de configuration Bison qui définit un système de traduction qui calcule les expressions en notation postfixe:

```
%{  #define YYSTYPE float  /* Type des attributs synthétisés */
    #include <stdio.h>
    int yylex ();  /* Déclarations de fonctions utilitaires */
    void yyerror (char const* s);
%}

/* Déclaration des types de jetons qui ne sont pas représentés par un caractère */
%token NUM

%%  /* Section 2: Productions et règles sémantiques */
input: exp ';'          { printf ("%f\n", $1); } ;  // Le symbole de départ s'appelle 'input'
exp: NUM                { $$ = $1;             }
   | exp exp '+'        { $$ = $1 + $2;        }
   | exp exp '-'        { $$ = $1 - $2;        }
   | exp exp '*'        { $$ = $1 * $2;        }
   | exp exp '/'        { $$ = $1 / $2;        } ;

%% /* ==== Section 3: Code supplémentaire ==== */
/** Fct. que Bison appelle pour recevoir les jetons */
int yylex() {
    if (lexer.toutEstLu()) return 0;  // On fait appel à l'analyseur lexical
    if (lexer.jetonAUneValeur()) yylval = lexer.valeurJeton();
    return lexer.typeJeton();
}

/** Fonction pour la gestion des erreurs */
void yyerror (char const *s) { fprintf (stderr, "Erreur de type '%s' à %s\n", s, lexer.endroit()); }

/* Exemple d'invocation de Bison */
int main (int argc, char *argv[]) { return yyparse(); }
```