Chapitre 4

Traduction dirigée par la syntaxe

Table des matières

Introduction à la problématique de la traduction dirigée par la syntaxe, avec essentiellement des exemples concernant la génération de code. Le point de vue est donc un point de vue « compilation ».

Ce polycopié doit l'essentiel de sa structure et de son contenu au cours de R. Cori « compilation », donné en 1988 à Bordeaux I.

`—Id: crs.lgeF.gramAtt.tex,v 1.4 2003/03/12 10:58:49 amsili Exp —`

12 mars 2003

Introduction

Rappel : avec une grammaire (contrairement à des systèmes comme automates ou MT), on sait non seulement reconnaître des mots d'un langage, mais on sait aussi les « décomposer ».

Mais à quoi peut donc être utile une telle décomposition ? — à donner du sens au « mot » ainsi décomposé. On suppose, en d'autres termes, que la syntaxe peut nous guider pour l'interprétation des mots d'un langage (par exemple le langage de la logique des prédicats, ou le langage Pascal).

Mais du point de vue qui nous occupe ici, donner du sens revient à **traduire** les mots décomposés dans un autre langage. D'où le titre du chapitre « traduction dirigée par la syntaxe ».

4.1 Grammaires dans $X^* \times Y^*$

4.1.1 Définition

Soient X et Y deux alphabets (pas nécessairement disjoints). Alors on peut parler de $X^* \times Y^*$: $\{(f,g)/f \in X^*, g \in Y^*\}$.

On peut définir une grammaire sur $X^* \times Y^*$: les règles ont la forme habituelle, mais les terminaux sont des couples. Forme générale d'une règle : $A \longrightarrow u_1A_1u_2A_2 \dots u_pA_pu_{p+1}$, avec A, $A_i \in V$, et $u_i \in X^* \times Y^*$, c'est-à-dire les u_i sont de la forme $u_i = (f_i, g_i)$, avec $f_i \in X^*$ et $g_i \in Y^*$.

Exemple $S \longrightarrow (a,\varepsilon)S(\varepsilon,a)$
$S \longrightarrow (b,\varepsilon)S(\varepsilon,b)$
$S \longrightarrow (a,a) \mid (b,b) \mid (\varepsilon,\varepsilon)$

Le mot $(abaa, aaba)$ est engendré :

$$S \to (a,\varepsilon)S(\varepsilon,a) \to (a,\varepsilon)(b,\varepsilon)S(\varepsilon,b)(\varepsilon,a) \to (a,\varepsilon)(b,\varepsilon)(a,\varepsilon)S(\varepsilon,a)(\varepsilon,b)(\varepsilon,a)$$
$$\to (a,\varepsilon)(b,\varepsilon)(a,\varepsilon)(a,\varepsilon)S(\varepsilon,a)(\varepsilon,a)(\varepsilon,b)(\varepsilon,a) \to (a,\varepsilon)(b,\varepsilon)(a,\varepsilon)(a,\varepsilon)(\varepsilon,\varepsilon)(\varepsilon,a)(\varepsilon,a)(\varepsilon,b)(\varepsilon,a)$$

Il faut définir une nouvelle version de la concaténation : $(u,v)(u',v') = (uu',vv')$. Cette nouvelle opération a le bon goût d'être associative.

Alors on a bien $(a,\varepsilon)(b,\varepsilon)(a,\varepsilon)(a,\varepsilon)(\varepsilon,\varepsilon)(\varepsilon,a)(\varepsilon,a)(\varepsilon,b)(\varepsilon,a) = (abaa, aaba)$

Cette grammaire engendre $(f, \tilde{f})$.

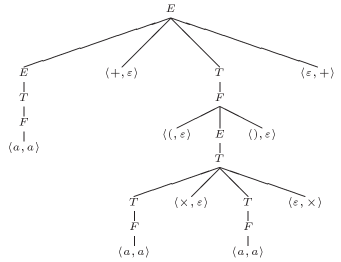
Déf. 1 (Système de traduction)

On appelle souvent de telles grammaires sur $X^* \times Y^*$ des *systèmes de traduction*.
On dit alors que g est la *traduction* de f si $S \xrightarrow{*} (f,g)$.

Exemple (classique)

$X = \{+, \times, a, (,)\}$; $Y = \{+, \times, a\}$

$E \to E\langle +, \varepsilon\rangle T\langle \varepsilon, +\rangle$
$E \to T$
$T \to T\langle \times, \varepsilon\rangle F\langle \varepsilon, \times\rangle$
$T \to F$
$F \to \langle a, a\rangle$
$F \to \langle (, \varepsilon\rangle E\langle), \varepsilon\rangle$



Traduit les expressions infixes en postfixes ;
ex. $\langle a + (a \times a), a\,a\,a\ \times +\rangle$

4.1.2 Utilisation des systèmes de traduction

Pratiquement, il n'y a pas d'intérêt à utiliser de tels systèmes de traduction pour *générer* des mots sur $X^* \times Y^*$. Ce qu'on voudrait, plutôt, c'est utiliser ces grammaires pour résoudre des problèmes de traduction.

Déf. 2 (Problème de traduction)

Le problème de traduction consiste à répondre à la question suivante :

Soit $f \in \mathcal{L}(\mathcal{G}_1)$, trouver $g \in \mathcal{L}(\mathcal{G}_2)$ tel que $(f, g) \in \mathcal{L}(\mathcal{G})$

Pour répondre pratiquement à cette question, on va commencer par découpler les deux « jeux » de règles.

Soit $\mathcal{G}$ un système de traduction sur $X^* \times Y^*$. On peut tirer de $\mathcal{G}$ deux grammaires, l'une sur X^*, et l'autre sur Y^* : pour chaque règle $A \longrightarrow u_1 A_1 u_2 \ldots$ de $\mathcal{G}$, avec $u_i = (f_i, g_i)$, on peut définir la règle $A \longrightarrow f_1 A_1 f_2 \ldots$ pour $\mathcal{G}_1$, et la règle $A \longrightarrow g_1 A_1 g_2 \ldots$ pour $\mathcal{G}_2$.

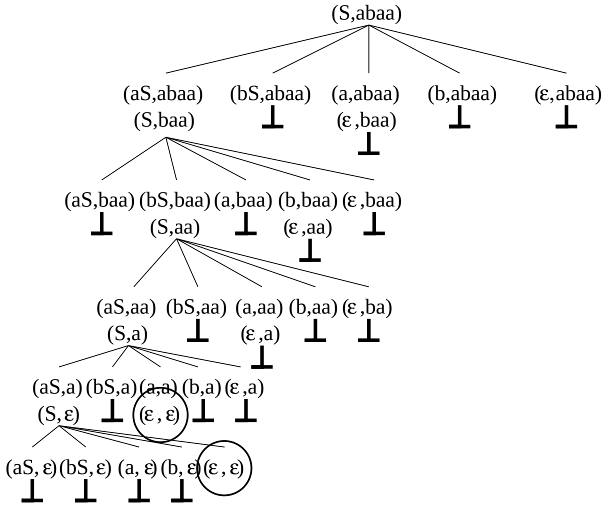
Alors à chaque règle de $\mathcal{G}_1$ correspond une règle de $\mathcal{G}_2$, et réciproquement.

Exemple en reprenant la grammaire ETF précédente, on obtient comme grammaire sur Y^* $(\mathcal{G}_2)$:
$E \to ET+ \mid T$
$T \to TF\times \mid F$
$F \to a \mid E$

On dispose alors d'une méthode de traduction effective : on réalise une analyse syntaxique du mot f, qui nous indique le n° de chaque règle de $\mathcal{G}_1$ utilisée. Alors il suffit d'utiliser « en parallèle » les règles de $\mathcal{G}_2$ pour engendrer la traduction de f.

Exemple (à partir du premier exemple) Soit le mot *abaa*. La grammaire $\mathcal{G}_1$ peut s'écrire $S \to aS \mid bS \mid a \mid b \mid \varepsilon$. Une analyse descendante pourrait prendre la forme suivante représentée à la figure 4.1. Les deux chemins possibles, si on numérote les règles de 1 à 5, correspondent aux règles (1,2,1,3) et (1,2,1,1,5). Il ne reste plus qu'à appliquer les

Fig. 4.1 – Analyse descendante complète de *abaa*



règles correspondante de la grammaire $\mathcal{G}_2$, ici $S \to Sa \mid Sb \mid a \mid b \mid \varepsilon$. Cela donne : $S \to Sa \to Sba \to Saba \to aaba$.

Remarque Avec une analyse descendante, on peut faire la traduction en même temps que l'analyse (même s'il y a du backtrack). Si on fait une analyse ascendante, il faut attendre que l'analyse soit terminée pour faire la traduction.

4.2 Les STDS pratiques

STDS : Systèmes de Traduction Dirigée par la Syntaxe.

Les ST utilisés en compilation sont un peu différents de ce qu'on vient de voir : ils reprennent l'idée de coupler aux règles qui reconnaissent (et décomposent) les mots d'un langage des ***actions*** à accomplir par le compilateur.

Deux exemples avec ETF

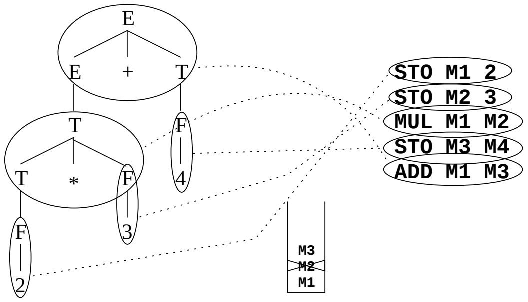
$E \to E + T$	*empiler(dépiler(P) + dépiler(P))*
$E \to T$	
$T \to T \times F$	*empiler(dépiler(P) × dépiler(P))*
$T \to F$	
$F \to (E)$	
$F \to a$	*empiler(P,valeur(a))*

Les actions associées aux règles permettent d'évaluer (calculer) l'expression en même temps qu'on la reconnaît (en analyse descendante).

Ce qu'on vient de réaliser, ce n'est pas un *compilateur*, mais un *interpréteur*. Mais sur la même idée, on peut construire un vrai compilateur :

$E \to E + T$	*Engendrer l'opération :* – additionner le contenu des mémoires dont les adresses sont en sommet de pile[1] ; – mettre le résultat dans la première des deux ; – supprimer un élément de la pile[1]
$E \to T$	
$T \to T \times F$	*Engendrer l'opération :* – multiplier le contenu des mémoires dont les adresses sont en sommet de pile[1] ; – mettre le résultat dans la première des deux ; – supprimer un élément de la pile[1]
$T \to F$	
$F \to (E)$	
$F \to a$	*Engendrer l'opération :* – mettre a dans une case libre[1] ; – mettre l'adresse de cette case dans la pile[1]

Fig. 4.2 – Exemple de compilateur ETF

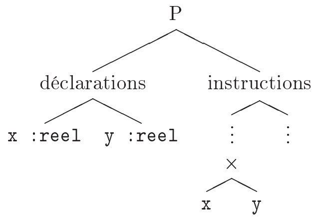


Résumé Avec une grammaire et des « actions », on peut faire :

Analyse syntaxique Action : imprimer n° de règle
Traduction dans $X^* \times Y^*$ Action : appliquer la règle dans $\mathcal{G}_2$.
Interprétation Action : calculer sur des valeurs
Compilation Action : générer du code

[1]On suppose que le compilateur gère des cases mémoire libres, ainsi qu'une pile, qui ne sera pas nécessaire *at run-time*.

Limite Il n'est pas toujours possible de faire de la compilation (du moins aussi simplement). Par exemple, si on a dans l'arbre syntaxique d'un programme Pascal d'une part les déclarations, et d'autre part les opérations, il est nécessaire au moment où on génère le code correspondant aux opérations de connaître le type des variables[2] :



Autrement dit, cette approche « locale » (il suffit de savoir quelle règle s'applique pour savoir quelle action effectuer) rencontre des limites : certaines informations doivent être manipulées de façon globale. Deux types de réponses possibles (les deux sont mises en œuvre dans les compilateurs) :
– les structures de données statiques modifiées et accessibles à tout moment (table des symboles, pile...)
– les systèmes d'attributs, informations associées aux nœuds de l'arbre syntaxique prévus pour permettre une « propagation » des informations dans l'arbre syntaxique.

4.3 Définition des systèmes d'attributs

4.3.1 Définition

Idée intuitive : à chaque nœud de l'arbre syntaxique (c'est-à-dire à chaque occurrence d'un non terminal dans une dérivation), on associe une ou plusieurs « cases mémoires » dont les valeurs vont être calculées en même temps que les dérivation interviennent.

Déf. 3 (Système d'attributs)

Un *système d'attributs* (ou « *grammaire attribuée* ») est défini par la donnée de $\langle X, V, S, P, A, R\rangle$, où
– X, V, S, P sont définis de la façon habituelle,
– A est un ensemble d'*attributs*. On peut voir l'attribut comme un variable au sens informatique du terme : il a un nom, et prend à tout instant une valeur parmi un ensemble de valeurs possibles.
– R est un ensemble de *règles de calcul*, associées à chaque règle de P. Ces règles de calcul peuvent prendre une des deux formes suivantes :
Si $A = \{\alpha_1, \alpha_2, \dots, \alpha_k\}$, pour une règle de P de la forme $A \longrightarrow f_1 A_1 f_2 A_2 \dots f_l A_{l+1}$,
(a) $A.\alpha_i := \Phi(A_p.\alpha_{q\ (p \in [1,l],\, q \in [1,k])}, A.\alpha_{j\ (j \in [1,k] \,\&\, j \neq i)})$
(b) $A_j.\alpha_i := \Psi(A.\alpha_{j\ (j \in [1,k])}, A_p.\alpha_{q\ (p \in [1,l],\, q \in [1,k] \,\&\, p \neq j)})$
Dans le cas (a), on dit que α_i est un attribut *synthétisé* ; dans le cas (b) on dit que α_i est un attribut *hérité*.

[2]Ce problème particulier est lié à ce qu'on appelle le polymorphisme des opérateurs.

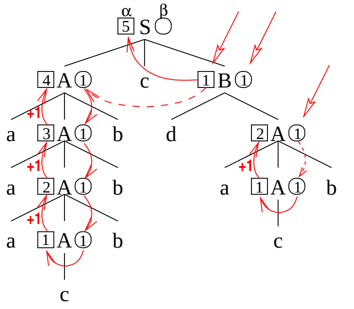
4.3.2 Exemple

Soit la grammaire suivante : (1) $S \to AcB$
(2) $A \to aAb$
(3) $A \to c$
(4) $B \to dA$

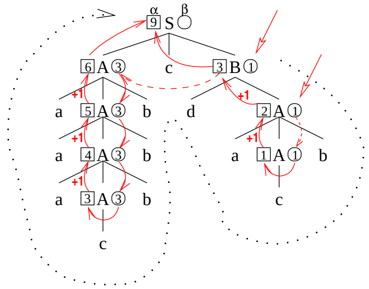
On veut lui associer un système à deux attributs : α (synthétisé) et β (hérité). Les règles sont définies de la façon suivante :

règle (1) : $S.\alpha := A.\alpha + B.\alpha$
$A.\beta := B.\alpha$
$B.\alpha := 1$
règle (2)[3] *:* $A^0.\alpha := A^1.\alpha + 1$
$A^1.\beta := A^0.\beta$
règle (3) : $A.\alpha := A.\beta$
règle (4) : $B.\alpha := 1$
$A.\beta := 1$

Pour un mot donné reconnu par la grammaire, on évalue (calcule) les attributs sur l'arbre de dérivation correspondant. Avec le mot *aaacbbbcdacb*, par exemple, cela donne :



Exercice Calculer la valeur obtenue pour le même mot, avec le même système d'attributs, sauf la règle de calcul associée à la règle (4) : $B.\alpha := 1 + A.\alpha$ (au lieu de $B.\alpha := 1$). On observe alors un sens différent pour le calcul.



4.3.3 Problèmes

En ajoutant ainsi un système d'attributs à une grammaire, on augmente considérablement sa « puissance », mais de nouvelles questions se posent :

1. Peut-on toujours calculer les attributs dans un système d'attributs (dans tous les cas, c'est-à-dire pour tout mot reconnu) ?

[3] Il faut distinguer les deux occurrences de A dans la règle (2). On « ré-écrit » à cet effet la règle de la façon suivante : (2) $A^0 \to aA^1b$.

2. Quelles sont les stratégies possibles (algorithmes) de calcul des attributs ?

De façon moins théorique, on peut aussi se demander comment exploiter concrètement un système d'attributs pour faire de la compilation, ou plus généralement de la « traduction dirigée par la syntaxe » (par exemple, pour revenir au problème déjà évoqué de la déclaration des variables, on voit que l'on peut envisager d'avoir les types des variables qui remontent dans la partie "déclaration" (synthèse), et qui descendent dans la partie "évaluation" (héritage)).

4.3.4 Exemple 2

Expressions arithmétiques postfixées, 1 attribut synthétisé. Calcul du nombre de registres nécessaires à une évaluation. (1) $S^0 \to S^1S^2b$
$S^0 := \left[(S^1.\alpha == S^2.\alpha)\ ?\ S^1.\alpha + 1\ :\ max(S^1.\alpha, S^2.\alpha)\right]$
(2) $S \to a$
$S.\alpha := 0$

4.3.5 Exemple 2′

On repart de l'exemple précédent, on ajoute une règle (3), et un attribut hérité (β).

(1) $S^0 \to S^1S^2b$
$S^0 := \left[(S^1.\alpha == S^2.\alpha)\ ?\ S^1.\alpha + 1\ :\ max(S^1.\alpha, S^2.\alpha)\right]$
$S^1.\beta := S^0.\beta + 1$
$S^2.\beta := S^0.\beta \times 2$
(2) $S \to a$
$S.\alpha := S.\beta$
(3) $S' \to S$
$S.\beta := 1$

4.3.6 Exemple 2′′

Variante de l'exemple précédent (seul changement : 3^e règle associée à (1)), où le calcul est plus compliqué : on ne peut pas d'abord calculer tous les β, puis tous les α.

(1) $S^0 \to S^1S^2b$
$S^0 := \left[(S^1.\alpha == S^2.\alpha)\ ?\ S^1.\alpha + 1\ :\ max(S^1.\alpha, S^2.\alpha)\right]$
$S^1.\beta := S^0.\beta + 1$
$S^2.\beta := S^1.\alpha \times 2$
(2) $S \to a$
$S.\alpha := S.\beta$
(3) $S' \to S$
$S.\beta := 1$

On voit bien sur ce exemple la difficulté de prédire qu'un système d'attributs donné sera calculable.

4.3.7 Exemple 3

Deux attributs synthétisés (α et γ), et un attribut hérité (β).

(1) $S \to STb$

$S^0.\alpha := max(S^1.\alpha, T.\alpha)$
$S^0.\gamma := S^1.\gamma + T.\gamma$
$S^1.\beta := S^0.\alpha$
$T.\beta := S^0.\alpha + S^0.\beta$

(2) $T^0 \to T^1c$

$T^0.\alpha := T^1.\alpha + 1$
$T^0.\gamma := T^1.\gamma + 1$
$T^1.\beta := T^0.\beta$

(3) $T \to c$

$T.\alpha := 1$
$T.\gamma := T.\beta$

(4) $S \to a$

$S.\alpha := 2$
$S.\gamma := S.\beta + 1$

4.3.8 Exemple 4

Exemple de système impossible

(1) $S \to S^1S^2b$

$S^0.\alpha := S^1.\alpha + S^2.\alpha$
$S^1.\beta := S^0.\beta$
$S^2.\beta := S^0.\beta$

(2) $S \to a$

$S.\alpha := S.\beta$

(3) $S' \to S$

$S.\beta := 2 \times S.\alpha$

4.4 Test de circularité, algorithme d'évaluation d'attributs

Remarque On peut avoir des systèmes d'attributs calculable pour certains arbres de dérivation et pas pour d'autres ; dans l'exemple précédent le calcul est impossible sur tout arbre de dérivation.

Comment établir formellement l'impossibilité du calcul ?

4.4.1 Test de circularité

On associe à chaque arbre syntaxique dans une grammaire attribuée donnée, un graphe : chaque nœud du graphe est une valeur d'attribut (autrement dit, chaque nœud (branchant) de l'arbre correspond à k sommets du graphe, s'il y a k attributs). Le graphe est orienté, on trace un arc du sommet s vers le sommet t ssi le calcul de t a besoin de la valeur de

s (les flèches vont dans le sens de la propagation des valeurs[4]). On remarque que pour les attributs synthétisés, les arcs vont des fils vers le père (ou dans le sommet lui-même), alors que pour les attributs hérités, les arcs vont du père vers les descendants (ou dans le sommet lui-même).

Théorème Le calcul d'attributs est possible pour le mot $f \in \mathcal{L}(\mathcal{G})$ ssi le graphe associé à son arbre de dérivation est sans circuit.

Remarque Dans la pratique, ce théorème n'est pas intéressant : on a besoin de savoir si un système d'attribut est calculable pour **tous** les arbres de dérivation.
Il existe des versions plus « fortes » de ce théorème, qui permettent de vérifier qu'un système est calculable en étudiant un nombre fini d'arbres.

4.4.2 Algorithme

Rappel Propriété des graphes sans circuit : il existe au moins un sommet sans prédécesseur. Par ailleurs, on sait aussi que si on ôte un sommet à un graphe sans circuit, il reste sans circuit. Il est donc possible de munir l'ensemble des sommets d'un ordre partiel (« tri topologique »).

C'est de l'algo classique de tri topologique que l'on s'inspire pour proposer un algorithme de parcours du graphe associé à un arbre de dérivation, qui permet de calculer les attributs :

> $\mathcal{G}$ = *graphe associé initial ;*
> *Tant que* $\mathcal{G}$ *non vide {*
> *Calculer la valeur des attributs sur le(s) sommet(s) sans prédécesseur ;*
> $\mathcal{G} := \mathcal{G} \setminus \{$*sommet(s) sans prédécesseur*$\}$
> *}*

Exercice Faire le graphe associé à l'exemple $2''$.

[4]On peut aussi concevoir le graphe « inverse », dans lequel les flèches représentent la « dépendance » : arc de a vers b si a a besoin de b.

4.5 Génération de code avec des grammaires attribuées

L'idée est simple : l'un des attributs va contenir le code généré petit-à-petit (synthétisé), et on va ajouter d'autres attributs pour les informations nécessaires *at compile time*.

4.5.1 Exemple 1 : ETF

On introduit deux attributs : `nb` pour le nombre de registres nécessaires, et `code` pour le code lui-même. Ils sont tous deux synthétisés.

$$
\begin{array}{lcll}
F \to a_i & : & F.\mathtt{code} & := \mathtt{LOAD}\ i\ \mathtt{R1} \\
 & & F.\mathtt{nb} & := 1 \\
F \to (E) & : & F.\mathtt{code} & := E.\mathtt{code} \\
 & & F.\mathtt{nb} & := E.\mathtt{nb} \\
T \to F & : & T.\mathtt{code} & := F.\mathtt{code} \\
 & & T.\mathtt{nb} & := F.\mathtt{nb} \\
E \to T & : & E.\mathtt{code} & := T.\mathtt{code} \\
 & & E.\mathtt{nb} & := T.\mathtt{nb} \\
E \to E + T & : & E^0.\mathtt{code} & := \left[\begin{array}{l} (E^1.\mathrm{nb} \geq T.\mathtt{nb})\ \ ? \\ E^1.\mathtt{code}\,;\ \mathcal{T}(T.\mathtt{code})\,;\ \mathtt{ADD\ R1\ R2}\ \ : \\ \mathrm{T}.\mathtt{code}\,;\ \mathcal{T}(E^1.\mathtt{code})\,;\ \mathtt{ADD\ R1\ R2} \end{array}\right] \\
 & & E^0.\mathtt{nb} & := \left[(E^1.\mathrm{nb} == T.\mathtt{nb})\ ?\ E^1.\mathtt{nb}+1 :\ max(E^1.\mathrm{nb},T.\mathtt{nb})\right] \\
T \to T \times F & : & T^0.\mathtt{code} & := \left[\begin{array}{l} (T^1.\mathrm{nb} \geq F.\mathtt{nb})\ \ ? \\ T^1.\mathtt{code}\,;\ \mathcal{T}(F.\mathtt{code})\,;\ \mathtt{MUL\ R1\ R2}\ \ : \\ \mathrm{F}.\mathtt{code}\,;\ \mathcal{T}(T^1.\mathtt{code})\,;\ \mathtt{MUL\ R1\ R2} \end{array}\right] \\
 & & T^0.\mathtt{nb} & := \left[(T^1.\mathrm{nb} == F.\mathtt{nb})\ ?\ T^1.\mathtt{nb}+1 :\ max(T^1.\mathrm{nb},F.\mathtt{nb})\right]
\end{array}
$$

Code « translaté » : Décalage des numéros de registre.
Par exemple, $\mathcal{T}(\mathtt{LDA\ A\ R1}) = \mathtt{LDA\ A\ R2}$.

Exercice Reprendre l'exemple donné plus haut : $2 \times 3 + 4$.

4.5.2 Exemple 2 : `if`, `while` et les booléens

à voir en cours.

4.6 Récapitulatif

Utilisation de la méthode des attributs pour la génération des compilateurs. Pour écrire un compilateur, il faut :

- une grammaire du langage sous une forme qui permet d'écrire un analyseur syntaxique (*parser*) ; cette grammaire décrit la *syntaxe* du langage **source**.
- une représentation de la *sémantique* du langage **source**. Il n'existe pas de formalisme aussi systématiquement approprié pour la sémantique que l'est une grammaire (hors-contexte) pour la syntaxe. Les options qui s'offrent sont :
  - actions associées à chaque construction de langage source (*sémantique procédurale*)
  - système d'attributs associé à la grammaire source
  - en sémantique formelle des langues naturelles, on envisagera aussi des règles d'interprétation dans un modèle tarskien
  - ...
- enfin, il faut connaître la syntaxe et la sémantique du langage **objet**. En général, c'est beaucoup plus simple.

À titre d'exemple, la grammaire d'attributs (GA) pour le langage Pascal représente entre 50 et 100 pages, celle du langage ADA plus de 500 pages.

Muni de tous ces éléments, on peut alors soit construire le compilateur « à la main », soit utiliser un générateur de compilateurs. Par exemple `yacc`, qui, étant donnée une grammaire et une série d'actions associées, génére un analyseur syntaxique en C. Autre exemple, les systèmes `SYNTAX` et `FNC2` (développés à l'INRIA), qui prennent directement en entrée des grammaires attribuées, et fournisse un "évaluateur de systèmes d'attributs".