Résumé

Une Grammaire Attribuée Générique, notée GAG, est une GA normale qui décrit un problème classique (comme l'identification à l'aide d'une table des symboles) sur une syntaxe abstraite ne comportant que les notions syntaxiques nécessaires pour la description de ce problème (comme la déclaration d'un identificateur, l'utilisation d'un identificateur et la notion de bloc).

Une GAG constitue alors un module suffisamment abstrait dans la description du problème qu'il spécifie pour être utilisable dans plusieurs applications.

On peut ainsi concevoir de construire une application complète pour un langage donné à partir de plusieurs modules de GAG et de la syntaxe décrivant ce langage.

Il faut donc pouvoir projeter la sémantique d'une GAG sur une syntaxe dite *d'entrée*, ce qui se traduit par une instanciation de la GAG sur la syntaxe. Afin de pouvoir effectuer cette instanciation, il faut spécifier les correspondances entre entre ces deux syntaxes. Ce principe d'instanciation est commun à tous les travaux réalisés dans l'optique de la réutilisation de modules ([FMY92], [DuC90],[KaW92]), ce sont les moyens de mise en pratique qui diffèrent.

Dans notre solution, chaque non-terminal de la GAG va être associé à un ensemble de non-terminaux de la syntaxe d'entrée, qui représentent la même notion pour ce problème.

Il est alors possible, moyennant certaines conditions, de construire une nouvelle grammaire attribuée réalisant la même sémantique que celle de la GAG, mais sur la syntaxe d'entrée.

La stratégie que nous avons développée pour instancier la GAG sur une syntaxe se fonde sur la notion de méta-composition de grammaires couplées par attributs, définie par Ganzinger et Giegerich [GaG84].

Dans un premier temps, il s'agit de construire automatiquement une première grammaire attribuée G_1, qui, pour tout arbre de la syntaxe d'entrée, construit un arbre de la syntaxe de la GAG. Cette construction est faite à partir de ces deux syntaxes ainsi que des correspondances établies au préalable entre elles.

La seconde étape consiste à composer, au sens de Ganzinger et Giegerich, cette première grammaire G_1 avec la GAG, appelée ici G_2.

La composition $G_1 \circ G_2$ permet d'obtenir une grammaire attribuée G_3 basée sur la syntaxe de G_1 et qui a la même sémantique que G_2. On a donc bien obtenu la projection de la sémantique de la GAG (G_2) sur la syntaxe d'entrée (syntaxe de base de G_1).

L'instanciation d'une GAG sur une syntaxe d'entrée peut alors se résumer à construire cette première grammaire attribuée G_1.

Cette notion de Grammaire Attribuée Générique permet d'étendre le concept de modularité et facilite l'utilisation des grammaires attribuées en tant qu'outil de programmation. On a alors l'équivalent du module générique classique d'un langage conçu pour le développement de grosses applications tel qu'Ada.

Table des matières

Table des figures

Chapitre I

Introduction

Lors de la confection de compilateurs et de traducteurs, il n'est pas rare de rencontrer des problèmes difficiles à résoudre, et ce, du fait de la diversification et de l'élaboration croissante des concepts utilisés pour les langages de programmation.

Les outils développés pour répondre à cette attente sont donc basés sur des concepts allant, eux aussi, vers une complexité croissante.

L'un des concepts les plus appréciés est celui des Grammaires Attribuées, habituellement notées GA. Il s'agit d'une méthode permettant de résoudre les problèmes dirigés par la syntaxe.

A l'origine, le concept de grammaire attribuée fut introduit par Knuth[Knu68] pour la spécification de langages de programmation.

Depuis, de nombreux travaux ont été effectués afin de permettre d'utiliser les grammaires attribuées comme outil de programmation.

Cette méthode est décrite de manière plus précise dans le chapitre des rappels, section II.1, ce qui permettra au novice de se familiariser avec le concept.

Très vite il s'est révélé que cette méthode de développement offrait de gros avantages, de par ses qualités intrinsèques. En effet, ce formalisme est déclaratif, ce qui permet de ne spécifier que les calculs à effectuer, et non comment on doit les exécuter. De plus, il est structuré par les blocs que constituent les productions de la grammaire. Ceci induit un principe de localité à l'intérieur de ces blocs, ce qui permet de spécifier des calculs locaux à une production, en ne précisant que l'interface possible avec les autres. De plus, les grammaires attribuées permettent de faire du développement de manière incrémentale, en permettant de tester chaque partie de spécification indépendamment des autres.

Bien entendu, le principal avantage de ce concept est qu'il constitue une méthode de spécification exécutable, puisqu'il permet de ne faire qu'une description d'un calcul donné sous la forme d'une grammaire attribuée, et de produire ensuite automatiquement un programme qui le réalise.

Tout ceci permet de réaliser le développement d'une application réelle dans un minimum de temps, et il est dommage que les industriels. Bien entendu, cela n'est pas sans raison, et il est vrai que jusqu'à maintenant, les évaluateurs produits automatiquement grâce aux grammaires attribuées n'étaient pas assez efficaces pour concurrencer avantageusement les "productions" manuelles.

Mais les travaux réalisés dans ce domaine tendent à diminuer très sensiblement cet inconvénient. Ainsi, comme l'a décrit M. JOURDAN([Jou92]), l'analyse statique rend possible l'optimisation d'un évaluateur par la connaissance d'un grand nombre d'informations rassemblées avant sa production.

C'est donc sur cette méthode particulièrement intéressante, capable de fournir des spécifications exécutables, qu'est basé le système FNC–2. En effet, à partir d'une spécification écrite en OLGA, son langage d'entrée, il fournit automatiquement l'application décrite : un traducteur d'un langage vers un autre ou même, un compilateur OLGA (OLGA est "bootstrappé"). La version de FNC–2 sur laquelle j'ai commencé à travailler est décrite dans le chapitre des rappels section II.3 de ce mémoire.

Un autre inconvénient des GA est qu'elles correspondent à une vision très rigide des choses, puisque la spécification de la sémantique dépend en fait de la structure syntaxique.

Or, l'homme est enclin à développer un mode de pensée par concept, ce qui, si on l'applique à la conception d'une spécification, induit une plus grande clarté, pour l'écrivain comme pour le lecteur. Il est donc plutôt désagréable, lors du développement d'une grosse application, de se retrouver devant une unique GA d'une taille considérable. Ceci rend la compréhension presque impossible et interdit pratiquement le travail en parallèle de plusieurs programmeurs.

Le but consiste donc à essayer de rendre possible l'utilisation des GA en tant qu'outil de Génie Logiciel. Il faut réduire la taille des grammaires attribuées et acquérir une plus grande modularité en séparant au maximum les concepts mis en cause lors de l'utilisation d'une GA. Ceci constitue les deux principaux buts des travaux existants décrits dans le chapitre III concernant la mise en œuvre des GA. Ils s'efforcent tous d'aboutir à une plus grande facilité d'utilisation, principal manque de cette méthode, lorsqu'il s'agit de grosses applications.

Notre approche concernant ce sujet est elle aussi abordée dans ce chapitre, section III.4.

Il est donc possible d'aller plus loin dans le confort d'utilisation que l'on peut fournir : ainsi, nous avons nous aussi travaillé sur des modifications permettant d'utiliser la méthode des grammaires attribuées plus largement et plus convivialement.

Une des premières modifications que l'on a apporté à FNC–2 dans cet esprit, a été l'implantation d'une couche supérieure d'OLGA. Ce "sur-langagë permet de produire des règles sémantiques de manière automatique, en regroupant les attributs par classes. Ceci a pour effet de supprimer la redondance liée au comportement similaire de certains attributs dans une application donnée. Cela permet, bien entendu, de diminuer la taille de la grammaire, mais aussi d'offrir au programmeur une autre méthode de travail sur les GA. En effet, celui-ci doit pouvoir décrire le comportement par défaut d'un type d'attribut, pour ne préciser ensuite, dans la grammaire, que les points de réel intérêt. Le nouveau langage incluant cette partie se nomme OLGA++, et est décrit dans le chapitre IV.

Mais cette structure n'est pas complètement satisfaisante car elle ne permet pas de séparation entre les divers concepts véhiculés sur une même grammaire. C'est pourquoi nous avons développer le concept de grammaire attribuée générique.

Une Grammaire Attribuée Générique (GAG) doit être capable de représenter un problème particulier à partir d'une syntaxe "minimale". Elle peut ensuite servir de base à l'application sur une autre GA, dite d'entrée, de la sémantique qu'elle décrit. Ceci est fait moyennant une spécification des correspondances entre les deux syntaxes : la "minimale" et celle de la grammaire d'entrée. On génère ainsi une grammaire attribuée contenant les informations originelles auxquelles s'ajoutent celles de la GAG.

Le chapitre V décrit le principe de l'instanciation d'une GAG sur une grammaire d'entrée. Il se résume à vérifier que les correspondances établies entre les deux syntaxes permettent de définir une fonction et qu'il est possible de la produire automatiquement. Ainsi, on peut utiliser la méthode de la méta-compilation de grammaires couplées, décrite dans le chapitre II, section V.1, qui permet d'obtenir une GA ayant pour base syntaxique la grammaire d'entrée et pour sémantique celle de la GAG.

Vient ensuite le chapitre VI qui permet de généraliser ce principe pour l'instanciation de plusieurs GAG sur une même grammaire d'entrée. Il décrit également l'interfaçage possible entre les GAG et la grammaire d'entrée pour lequel on a définit la notion de Grammaire Attribuée Ouverte (GAO).

Le chapitre VII décrit une extension du principe d'instanciation pour augmenter les possibilités de réutilisation d'une GAG, toujours dans l'optique de la généricité.

Chapitre II

Rappels

II.1 La théorie des Grammaires Attribuées

A l'origine, le concept de grammaire attribuée fut introduit par Knuth [Knu68] pour la spécification de langages de programmation. Un avantage de cette méthode est qu'elle fournit des spécifications exécutables.

Depuis leur invention, les grammaires attribuées ont fait l'objet de beaucoup d'études et de développements dans un vaste domaine d'application : sémantique statique des langages de programmation [Knu68, Lor77], optimisation de code [NAM74], édition de textes [Rep82], présentation graphique de structure arborescente [CFL88], etc ...

Leur principe consiste à associer :

- à chaque symbole non-terminal d'une grammaire, un ensemble d'attributs,
- à chaque production, un ensemble d'équations définissant le calcul des valeurs d'attributs pour toute instance d'une telle production dans un arbre de dérivation.

Le calcul des valeurs des instances d'attributs à travers l'arbre n'est pas spécifié explicitement par l'écrivain de la grammaire, il est implicitement défini par les équations et par la structure de l'arbre.

II.1.1 Définitions

II.1.1.1 Grammaires Attribuées

Une grammaire attribuée, ou GA, est décrite par plusieurs éléments, sa base étant une grammaire indépendante du contexte.

Définition II.1.1 : Grammaire indépendante du contexte (Rappel)

Soit G une grammaire indépendante du contexte *:*

$G = (N, T, P, Z)$ *où*

- *N est un ensemble fini de symboles non-terminaux;*
- *T est un ensemble fini de symboles terminaux tel que $N \cap T = \emptyset$*
  $V = N \cup T$ est appelé le vocabulaire.
- *P est l'ensemble des productions, $P \subset N \times V^*$;*
  Chaque production $(A, \alpha) \in P$, $A \in N$, $\alpha \in V^*$ est notée $A \to \alpha$, avec A la partie gauche et α la partie droite. ($A \to \mathcal{E}$: production vide)
- *Z est l'axiome ou symbole de départ, $Z \in N$.*

♣

Dans toute la suite, une référence à une grammaire indépendante du contexte $G = (N, T, P, Z)$ supposera implicitement la propriété suivante : G est **propre**, c'est-à-dire que tout symbole non terminal est *accessible* depuis l'axiome, et est *productif*, c'est-à-dire dérive dans une phase terminale :

$$\forall A \in N, \exists\ x \in T^*,\ \alpha \in V^*,\ y \in T^*\ :\ \ S \stackrel{*}{\Longrightarrow} \alpha A x \stackrel{+}{\Longrightarrow} \alpha y x$$

A partir de la définition d'une grammaire indépendante du contexte, on peut décrire celle d'une grammaire attribuée.

Une **Grammaire Attribuée** α est un triplet $\alpha = (G, A, F)$ dont les éléments sont définis ci-dessous :

1) une grammaire indépendante du contexte $G = (N, T, P, Z)$ telle que définie ci-dessus.

Les terminaux ne jouent aucun rôle dans la théorie des grammaires attribuées, ils seront donc ignorés et les productions de P se noteront :

$$p\ :\ X_0 \longrightarrow X_1 X_2 \ldots X_{n_p}$$

où $\forall i, 0 \leq i \leq n_p,\ X_i \in N$

2) un ensemble d'*attributs* A tel que

$A = HER \cup SYN$ avec $HER \cap SYN = \emptyset$

où HER est l'ensemble des attributs *hérités* et SYN l'ensemble des attributs *synthétisés*.

Les attributs *synthétisés* propagent l'information depuis les feuilles vers la racine de l'arbre syntaxique construit à partir de la grammaire. Inversement, les attributs *hérités* transmettent l'information vers les feuilles. Chaque attribut est d'un seul type bien défini, c'est-à-dire qu'il ne peut se comporter en *hérité* dans une production puis en *synthétisé* dans une autre.

A chaque non-terminal $X \in N$ est attaché un sous-ensemble de ces attributs $A(X) = HER(X) \cup SYN(X)$, avec $HER(X) \subseteq HER$ et $SYN(X) \subseteq SYN$.

On note $X.a$ une *occurrence d'attribut* de l'attribut a attaché au non-terminal X dans une production.

Dans chaque production $p : X_0 \longrightarrow X_1 X_2 \dots X_{n_p}$, on définit l'ensemble des occurrences d'attribut de p:

$$W(p) = \{X_i.a \mid 0 \leq i \leq n_p \ et \ a \in A(X_i)\}$$

On peut également préciser ici les notions d'attributs d'*entrée* et d'attributs de *sortie*.

L'ensemble $OW(p)$ des occurrences d'attributs définies dans une production p constitue les attributs de sortie.

$$OW(p) = \{X_i.a \mid i = 0 \ et \ a \in SYN(X_0), \ ou \ 1 \leq i \leq n_p \ et \ a \in HER(X_i)\}$$

L'ensemble des attributs d'entrée $IW(p)$ est le complémentaire de l'ensemble des attributs de sortie.

$$IW(p) = \{X_i.a \mid i = 0 \ et \ a \in HER(X_0), \ ou \ 1 \leq i \leq n_p \ et \ a \in SYN(X_i)\}$$

3) un ensemble F de *règles sémantiques*, spécifiant la manière dont les valeurs des différentes occurrences d'attributs sont calculées, tel que :

$F = \{ \ F(p) \mid p \in P \ \}$

où $F(p) = \{f_{p,i,a} \mid X_i.a \in OW(p)\}$ contient exactement une règle sémantique définissant chaque occurrence d'attribut de sortie de p.

Une règle sémantique est une expression fonctionnelle qui exprime la manière dont la valeur d'une occurrence d'attribut est calculée à partir des autres occurrences de $W(p)$:

$$f_{p,i,a} \ : \ X_i.a = f_{p,i,a}(X_{k_1}.b_1, X_{k_2}.b_2, \dots, X_{k_m}.b_m)$$

où $f_{p,i,a}$ est la fonction de calcul,

$m \geq 0$,

et $\forall j \ tq \ 1 \leq j \leq m$, on a $X_{k_j}.b_j \in UW(p)$.

On note $UW(p) \subseteq W(p)$, l'ensemble des occurrences d'attribut utilisées dans la définition.

La théorie des grammaires attribuées demande que chaque occurrence d'attribut soit spécifiée par une et une seule règle sémantique.

Exemple :

Nous allons reprendre l'exemple de l'article de Knuth : le calcul de la valeur décimale d'un nombre binaire.

Soit (G, A, F) la grammaire attribuée effectuant le calcul de la valeur décimale d'un nombre binaire.

- $G = (N, T, P, Z)$ une grammaire indépendante du contexte où
  - $N = \{Z, L, B\}$ l'ensemble des non-terminaux de G
  - $T =$\{"0", "1","."\} l'ensemble des terminaux de G
  - $P = \{p_1, p_2, p_3, p_4, p_5, p_6\}$ l'ensemble des productions de G
  - Z l'axiome
- $A = \{v, p, l\}$ l'ensemble des attributs,
  où $A(Z) = \{v\}$, $A(L) = \{v, p, l\}$, $A(B) = \{v, p\}$ avec
  $X.v$ la valeur décimale de la chaîne dérivée de X, de type réel,
  $X.p$, la position du bit le plus à droite de la chaîne dérivée de X, de type entier,
  $X.l$ la longueur de la chaîne dérivée de X, de type entier.
- F l'ensemble des règles sémantiques

Les règles sémantiques associées à chaque production sont décrites ci-dessous :

$$
\begin{array}{ll}
p_1: & Z \rightarrow L \\
 & Z.v = L.v \\
 & L.p = 0 \\
p_2: & Z \rightarrow L_1\text{“.”}L_2 \\
 & Z.v = L_1.v + L_2.v \\
 & L_1.p = 0 \\
 & L_2.p = -L_2.l \\
p_3: & L_0 \rightarrow L_1\ B \\
 & L_0.v = L_1.v + B.v \\
 & B.p = L_0.p \\
 & L_1.p = L_0.p + 1 \\
 & L_0.l = L_1.l + 1
\end{array}
\qquad
\begin{array}{ll}
p_4: & L \rightarrow B \\
 & L.v = B.v \\
 & B.p = L.p \\
 & L.l = 1 \\
p_5: & B \rightarrow \text{“1”} \\
 & B.v = 2^{B.p} \\
p_6: & B \rightarrow \text{“0”} \\
 & B.v = 0
\end{array}
$$

Ainsi, la phrase "1101.01" a pour arbre de dérivation celui de la figure II.1.

♢

II.1.1.2 Graphe de dépendances local

Les règles sémantiques introduisent une notion de relation de dépendance locale entre les attributs: on dit qu'une occurrence d'attribut dépend d'une autre si celle-ci

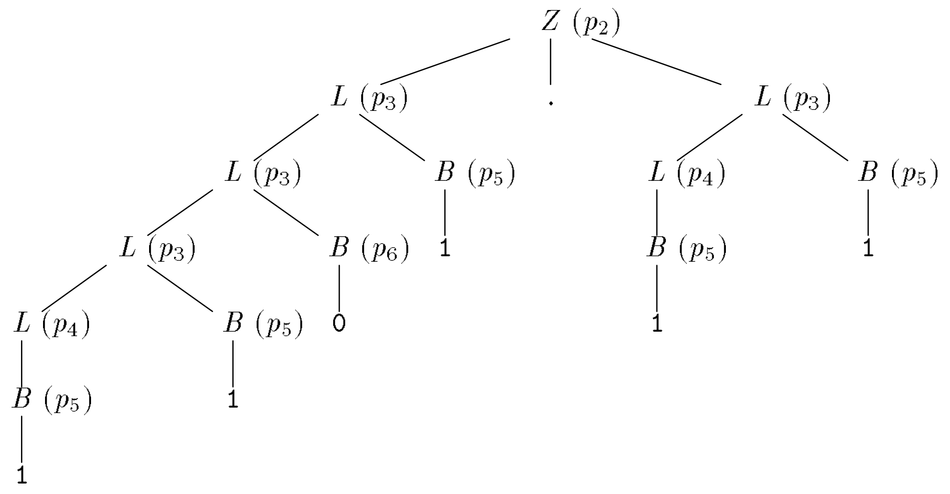


Figure II.1 : Arbre de dérivation du texte “1101.01”.

apparaît dans sa définition.

Ainsi, la règle sémantique $f_{p,i,a}$: $X_i.a = f_{p,i,a}(X_{k_1}.b_1, X_{k_2}.b_2, \ldots, X_{k_m}.b_m)$ induit le fait que $X_i.a$ dépend de $X_{k_1}.b_1, X_{k_2}.b_2, \ldots, X_{k_m}.b_m$ dans la production p, ce qu'on note $X_{k_j}.b_j \underset{p}{\rightarrow} X_i.a$

On peut ainsi construire un *graphe de dépendance local* $D(p)$ pour chaque production p, qui schématise les relations entre les attributs.

Ce graphe de dépendance est celui de la relation $\underset{p}{\rightarrow}$ dont les sommets sont les éléments de W(p).

Un attribut de sortie peut être défini par des attributs d'entrée ou par des attributs de sortie ($UW(p) \subseteq W(p)$).

Une grammaire est dite en **Forme Normale** (de Bochman) [Boc76] si les attributs de sortie sont définis uniquement en fonction des attributs d'entrée dans toutes les productions, c'est-à-dire lorsque $UW(p) \subseteq IW(p), \forall p \in P$.

Une GA en forme normale est considérée comme plus facile à lire, et de plus, toute GA peut être transformée en une GA équivalente en forme normale, à condition qu'aucun des graphes de dépendance des productions ne comporte de cycle. Cette restriction ne diminue pas le pouvoir d'expression et rend plus facile la caractérisation des grammaires.

Dans la suite, nous considérerons que les grammaires sont en forme normale.

Exemple :

La figure II.2 représente l'ensemble des graphes de dépendance de toute les productions pour la grammaire attribuée décrite précédemment.

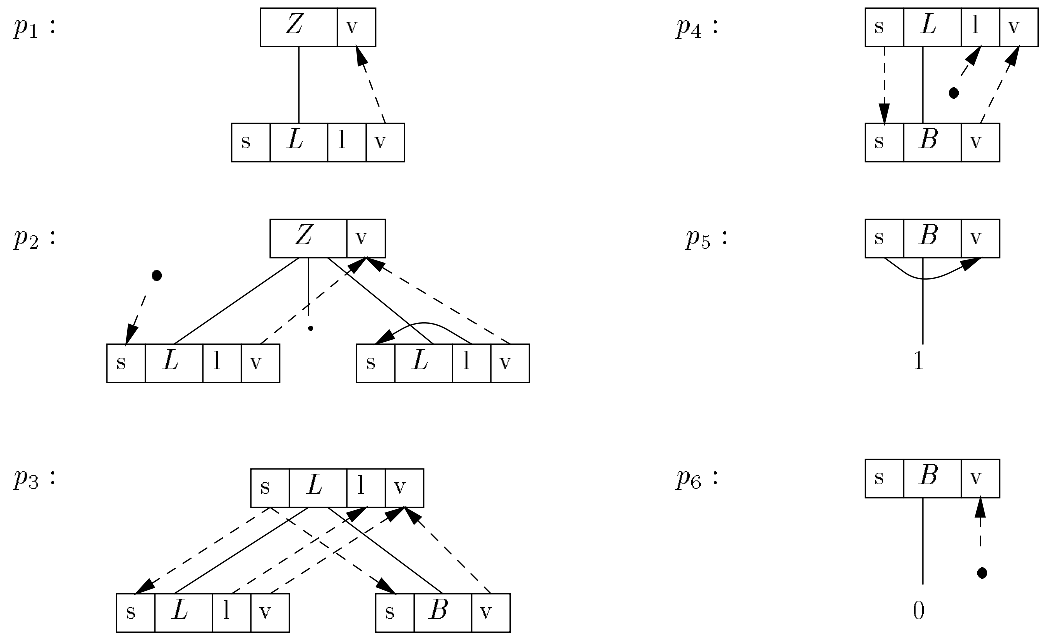


Figure II.2 : Graphes de dépendance des productions.

◊

II.1.1.3 Graphe de dépendances complet

Un *arbre attribué* est un arbre syntaxique t augmenté à chaque nœud des valeurs des attributs associés. Par convention, les attributs hérités sont représentés à gauche des non-terminaux et les synthétisés à droite. Chacune de ces boîtes est appelée *instance d'attribut.*

Tout nœud u de l'arbre t est étiqueté par la production p qui s'y applique. On note $label(u)$ la fonction qui associe à un nœud u son étiquette p. Si la production $p : X_0 \longrightarrow X_1 X_2 \ldots X_{n_p}$ s'applique au nœud u, tous les attributs de $A(X_0)$ seront attachés à u et chacune de ces variables est une instance d'attribut. L'ensemble des instances d'attributs d'un nœud u est alors défini par :

$$IA(u) = \{a(u) \mid a \in A(LHS(label(u))\}$$

où LHS est la fonction qui associe à une production son non-terminal en partie gauche.

L'ensemble de toutes les instances d'attributs d'un arbre t, noté $W(t)$, peut alors être considéré comme l'ensemble des variables auxquelles le processus d'évaluation devra affecter une valeur :

$$W(t) = \{a(u) \mid u \in t \text{ et } a \in A(LHS(label(u)))\}$$

Pour un arbre de dérivation t donné, on définit un graphe de dépendances complet noté $D(t)$, qui représente les dépendances entre les attributs instanciés aux nœuds de t. Ce graphe est obtenu en combinant les graphes des dépendances locales $D(p)$ correspondant aux productions utilisées dans t, selon la structure syntaxique de t. Les sommets du graphe $D(t)$ sont alors les instances d'attributs de l'arbre, c'est-à-dire les éléments de $W(t)$.

On note $D(u)$ le graphe construit à partir du graphe de dépendance local $D(p)$ où $p = label(u)$ avec $p : X_0 \longrightarrow X_1 \ldots X_{n_p}$. Les sommets de $D(u)$ sont les instances d'attributs $\{\ u.i.a \mid 0 \leq i \leq n_p \ et \ a \in A(X_i)\ \}$. Les arcs de $D(u)$ sont dérivés de ceux de $D(p)$: pour tout arc $X_j.b \underset{p}{\rightarrow} X_i.a$, il y a un arc $u.j.b \underset{u}{\rightarrow} u.i.a$ dans $D(u)$.

Exemple :

Les graphes de dépendance locaux décrits dans l'exemple précédent permettent de construire, lors de l'analyse du texte "1101.01", un graphe de dépendance complet illustré par la figure II.3, associé à l'arbre de la figure II.1.

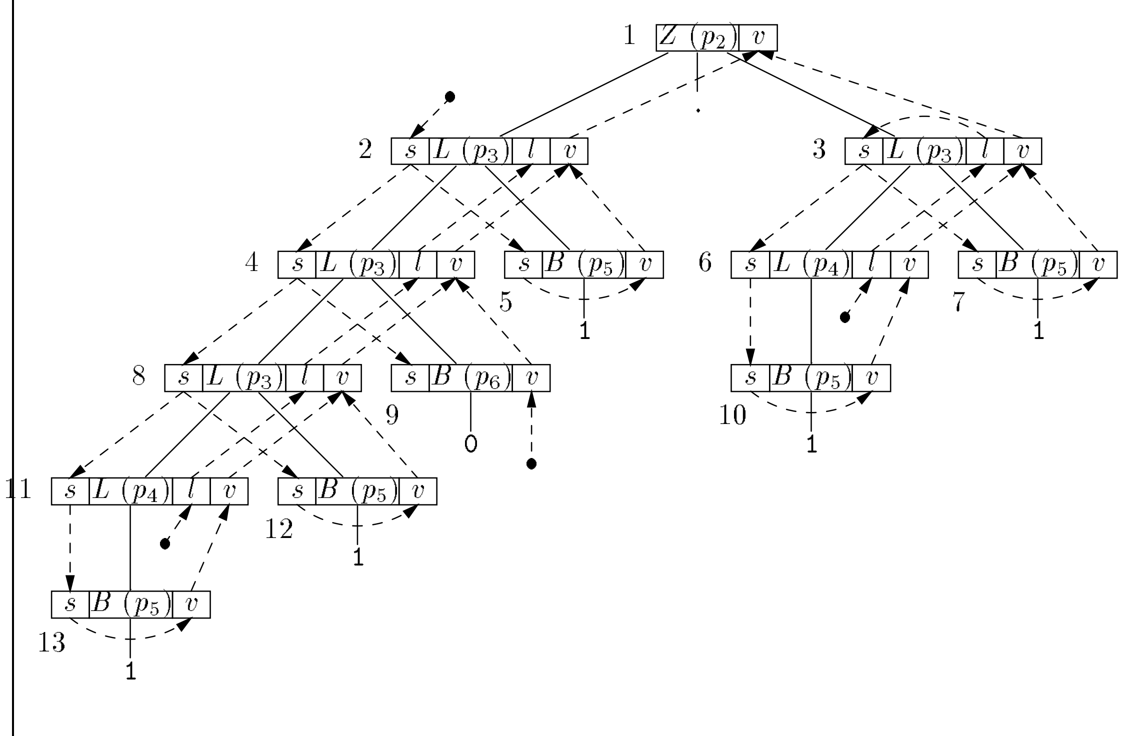


Figure II.3 : Analyse de "1101.01" : graphe de dépendance complet.

◊

Le graphe $D(t)$ peut contenir des cycles, on parlera alors de graphe circulaire. Au contraire, si pour tout arbre de dérivation t, ce graphe est sans circuit, la grammaire attribuée est dite *non circulaire* ou *bien formée*. Cette propriété de non-circularité, intrinsèque à la grammaire attribuée, nécessite un test qui s'est révélé être l'un des premiers points noirs de la théorie des grammaires attribuées : Knuth, dans la correction de l'article original, a proposé un algorithme pour tester la non-circularité d'une GA, algorithme dont l'exponentialité intrinsèque a malheureusement été démontrée [JOR75]. Les travaux de recherche ont alors porté sur des optimisations possibles de ce test et on sait aujourd'hui tester des formes plus fortes de non-circularité de façon polynômiale. D'autres travaux d'optimisation montrent que le caractère exponentiel peut ne subsister que pour des cas pathologiques de grammaires attribuées [DJL84].

Lorsque le graphe $D(t)$ est sans circuit, il induit sur les instances d'attributs de t un ordre partiel qui peut être utilisé pour calculer les valeurs des instances d'attributs.

II.1.2 Évaluation des attributs

II.1.2.1 Généralités

Étant donné un arbre syntaxique dont chaque nœud est étiqueté par la production qui s'y applique, on construit un système d'équations dont les inconnues sont toutes les instances d'attributs de l'arbre qui sont définies par les règles sémantiques.

L'évaluation des instances d'attributs consiste à résoudre ce système d'équations noté $K(t)$ dans son intégralité. Cette évaluation peut néanmoins se restreindre au calcul de la valeur sémantique de l'arbre, définie comme étant l'ensemble des attributs synthétisés de la racine.

Le système d'équations est dit **non circulaire** si et seulement si le graphe de dépendance complet ne comporte pas de cycle.

De même, on dira que la grammaire attribuée est **non circulaire** si les systèmes d'équations résultants de tous les arbres qu'elle peut engendrer sont non circulaires.

L'évaluation des fonctions ou des expressions dénotées par les $f_{p,a,k}$ ne doit pas produire d'effet de bord. Ceci est l'une des contraintes les plus importantes associée à la programmation par grammaire attribuée : les règles sémantiques doivent être écrites dans un langage **applicatif**. En effet, l'ordre d'évaluation de ces termes n'etant pas spécifié, il y a des fortes chances pour que ces effets de bord ne produisent pas le résultat attendu. Et surtout, dans le cadre de l'évaluation incrémentale, ces termes seront évalués plusieurs fois de suite; les effets de bord sont alors particulièrement dangereux.

Il existe plusieurs classes d'évaluateurs correspondant aux évaluations ascendante, descendante, etc...

L'implantation des évaluateurs d'instances d'attributs est un des problèmes majeurs de la théorie des GA. Tout algorithme d'évaluation doit analyser les relations de dépendances entres les attributs. Ces relations peuvent être très complexes, d'où le caractère exponentiel du test de non-circularité et la difficulté à trouver un algorithme d'évaluation général et efficace.

II.1.2.2 Différentes méthodes d'évaluation

La propriété fondamentale concernant les grammaires attribuées est :

> Il existe un ordre partiel d'évaluation pour tout arbre syntaxique t si et seulement si la grammaire attribuée est non circulaire.

La première famille de techniques d'évaluation est fondée sur la détermination dynamique de cet ordre.

Nous pouvons distinguer deux étapes dans l'évaluation :

1. la construction d'un ordre d'évaluation $T(t)$;
2. le calcul effectif des attributs de manière à affecter des valeurs aux variables de l'ensemble $W(t)$.

Il existe deux manières de suivre l'ordre $T(t)$: soit en partant des éléments minimaux de la relation, soit en partant des éléments maximaux. On obtient ainsi deux catégories d'évaluateurs :

- Les évaluateurs *ascendants* ou bottom-up;
- Les évaluateurs *descendants* ou top-down.

L'avantage principal d'un évaluateur *top-down* est la possibilité de choisir parmi les éléments maximaux un sous-ensemble constitué généralement des attributs synthétisés de l'axiome. Ainsi on peut distinguer, parmi toutes les instances d'attributs, celles qui sont effectivement utiles pour le calcul de la valeur sémantique de l'arbre.

Le fait que ces méthodes reconstruisent le graphe de dépendances complet à chaque exécution, puisque l'ordre d'évaluation dépend aussi de l'arbre syntaxique, rend les systèmes à évaluation dynamique le plus souvent très inefficaces en temps et en place [Fan72, Lor74, Lor77, CoH79, KeR79, Jou84].

Pour tenter de pallier cette inefficacité, on a vu apparaître une deuxième famille de méthodes d'évaluation : celles qui utilisent un **ordre d'évaluation statique**.

Un tel ordre est valide quel que soit l'arbre syntaxique.

Il convient de distinguer les méthodes dites par *passes*, qui fixent à priori une stratégie de parcours d'arbre, de celles qui déterminent l'ordre d'évaluation en fonction des dépendances locales, au moment de la construction de l'évaluateur.

Ainsi c'est la structure de la grammaire attribuée qui détermine le choix de la stratégie de parcours pour l'évaluation.

Ces méthodes produisent des évaluateurs efficaces mais au détriment de la puissance d'expression de la grammaire. Ceci est particulièrement vrai pour les méthodes par passes [LRS74, JaW75, Boc76, RaU81, EnF82, KoR83].

On peut également se poser le problème de l'évaluation incrémentale lorsqu'on effectue une modification sur un nœud de l'arbre. Ceci va probablement modifier un certain nombre d'instances d'attributs de ce nœud.

La solution naïve, qui consiste à faire une évaluation exhaustive, est à écarter.

Le système (d'évaluation incrémentale) doit être capable d'utiliser au mieux le travail déjà fait pour rendre le même résultat que celui de la version exhaustive.

En effet, on peut facilement imaginer qu'une modification dans une partie de l'arbre n'entraîne pas nécessairement la modification de l'ensemble des valeurs des instances d'attributs associées à cet arbre. Certaines (nombreuses), communes aux deux arbres, peuvent avoir leurs valeurs inchangées et il devient inutile de les réévaluer.

C'est le même problème posé par la réévaluation incrémentale (au sens compilateur incrémental) étudiée dans [Rep82, Par88].

L'idée d'un tel algorithme est que la connaissance de l'ensemble des dépendances entre les instances d'attributs sur un arbre d'entrée t donné permet de propager une éventuelle modification seulement là où c'est nécessaire.

II.1.3 Classification des grammaires attribuées

Les études théoriques et pratiques ont conduit à définir des catégories de grammaires attribuées selon certaines propriétés des relations d'ordre et à déterminer comment, pour chacune d'elles, un évaluateur pouvait être construit.

Nous donnons dans la figure II.4 un résumé de cette classification et des relations d'inclusions entre les principales classes de grammaires attribuées. Sont aussi indiquées les complexité en temps des tests d'appartenance aux classes des grammaires NC (ou Bien Formées), FNC, DNC et L-Ordonnées ([EnF82, CFZ82, Eng84]).

Il n'existe pas de classe de GA meilleure que les autres, qui ait le pouvoir d'expression le plus fort, et qui permette la construction d'un évaluateur "optimal".

On constate néanmoins que la première classe ayant un test d'appartenance polynomial est la classe FNC qui regroupe une trés large sous-classe des GA, mais les évaluateurs produits sont trés gourmands en place mémoire.

En revanche, la classe des GA L-Ordonnées permet de produire des évaluateurs

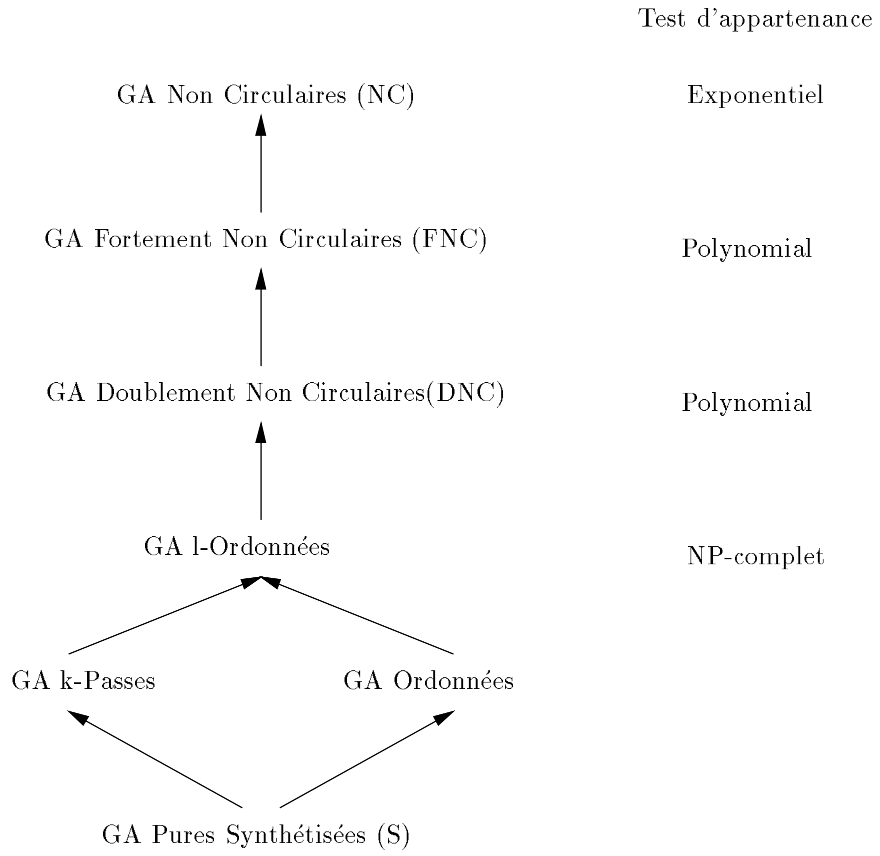


Figure II.4 : Schema simplifié des classes de Grammaires attribuées

statiques et bénéficiant des techniques d'optimisation mémoire, car basés sur des *séquences de visites* respectant un ordre d'évaluation total. Mais le test d'appartenance est NP-Complet.

Il existe d'ailleurs une transformation permettant de passer d'une GA FNC à une GA L-Ordonnée. Cette transformation est toujours possible car elle est un cas particulier d'un cas plus général [Fil83, Der84, Fil86]. Ceci permet d'avoir les avantages de la grande puissance d'expression des GA FNC ainsi que l'efficacite en encombrement mémoire des GA L-Ordonnées. Mais en revanche il existe un risque d'exponentialité de la taille de la GA L-Ordonnée produite. Une optimisation de cette transformation est décrite dans [Par88] qui permet de se situer dans des cas où la taille de la GA reste trés raisonnable, tout en n'empêchant pas les cas pathologiques d'exponentialité.

II.1.4 Les grammaires attribuées en tant qu'outil de spécification

Les grammaires attribuées constituent une technique très prisée de spécification de divers processeurs dirigés par la syntaxe, comme les compilateurs, éditeurs, etc.

La spécification traditionnelle des grammaires attribuées a été orientée par la syntaxe: chaque règle syntaxique est associée à un ensemble de règles sémantiques servant à calculer la sémantique correspondante.

L'intérêt des grammaires attribuées est maintenant clair :

- C'est une méthode déclarative (non-procédurale): on spécifie *ce* qui doit être calculé et non *comment* le calculer (dans quel ordre).
- C'est une méthode structurée : chaque production associée à ses règles sémantiques, agit comme une "boîte noire" indépendante et ne communique avec les autres qu'à travers des interfaces bien définies (les non-terminaux et leurs attributs).
- On ne peut voir dans les règles sémantiques que les attributs locaux à cette production et ainsi les opérations sont divisées en parties plus simples à manipuler et la spécification n'en est que plus lisible.
- Et surtout, il est possible de construire automatiquement un programme qui implantera la traduction spécifiée par la grammaire attribuée.

II.2 Les Grammaires Couplées par Attributs

Les grammaires couplées par attributs, ou GCA, représentent une variation de la notion traditionnelle des GA. Elle a été introduite par H. Ganzinger et R. Giegerich en 1984 ([GaG84]), et permet de voir les grammaires attribuées comme des fonctions allant d'une forme d'arbre abstrait à une autre.

Il ne s'agit plus seulement d'effectuer un calcul par la décoration sémantique d'un arbre abstrait, mais bien de permettre la construction d'un autre arbre abstrait en sortie.

L'intérêt de cette vision est que cela permet de considérer les grammaires attribuées comme des modules traitant chacun une étape de calcul, et s'enchaînant les uns après les autres.

On peut également alors définir une opération de composition, décrite dans la suite.

Il devient alors très utile d'utiliser cette modularité pour la construction d'une application (type compilateur), où les étapes de constructions de termes intermédiaires sont nombreuses.

II.2.1 Description

Tout d'abord, il s'agit de ne plus faire de différence entre un arbre syntaxique (abstrait), traditionnellement non décoré, et un arbre attribué, qui sont deux représentations intermédiaires possibles (une GA usuelle faisant passer de la première à la seconde).

On va alors considérer deux sortes de types :

- le type *sémantique*, auquel appartiennent les attributs, au sens usuel des GA, et les terminaux, considérés comme des informations sémantiques,
- le type *syntaxique* auquel appartiennent les non-terminaux.

Ainsi, les symboles terminaux et les attributs sont du même type et sont alors traités de la même manière dans l'arbre :

- les attributs associés à un symbole non-terminal X deviennent des fils de tous les nœuds étiquetés par X.
- un symbole terminal T n'est pas autre chose qu'un attribut de type T, dont la valeur est fournie par l'analyseur lexical.

Une production $p\ :\ E\ \rightarrow\ E\ or\ T$, quant à elle, peut être considérée comme une fonction qui a des arguments de type sémantique (terminal or) et de type syntaxique (non-terminaux E et T), et donnant un résultat de type syntaxique (non-terminal E).

Ceci nous amène à donner une redéfinition d'une grammaire indépendante du contexte, après avoir rappelé ce qu'est une signature sémantique.

Définition II.2.1 : Signature sémantique

On appelle **signature sémantique** *un couple* $\Sigma = (S, F)$.

- *S est un ensemble de types contenant le type* **Bool**.
- *F est un ensemble de fonctions sur S contenant la fonction :* **IfThenElse** *: $Bool \times S \times S \rightarrow S$.*

♣

Définition II.2.2 : Grammaire indépendante du contexte

On appelle grammaire indépendante du contexte sur une signature sémantique Σ, $G(\Sigma)$ le quadruplet (N,T,Z,P) tel que :

- *N est un ensemble de non-terminaux ou types syntaxiques :* $N \cap T = \emptyset$,

- *T est un ensemble de terminaux ou types sémantiques, $T \subseteq S$,*
- *Z est le type de la racine, $Z \in N \cup T$,*
- *P est l'ensemble des productions ou fonctions syntaxiques :*
  $N \rightarrow [\ (N \cup T) - \{Z\}]^*$.

♣

Une grammaire peut donc être vue comme un enrichissement syntaxique d'une signature sémantique. Les programmes peuvent être considérés comme des arbres syntaxiques dont les feuilles sont des informations sémantiques.

Un mécanisme de passage d'un arbre d'une grammaire $G_1(\Sigma)$ à un arbre d'une grammaire $G_2(\Sigma)$ est le **couplage** de ces deux grammaires. Si ce *couplage* est assuré par une grammaire attribuée, on dit alors que G_1 et G_2 sont deux grammaires couplées par attributs.

Définition II.2.3 : Grammaires couplées par attributs

Soit $G_1(\Sigma) = (N_1, T_1, Z_1, P_1)$ et $G_2(\Sigma) = (N_2, T_2, Z_2, P_2)$ deux grammaires sur une même signature Σ = (S,F).

Soit (G_1 , α) une grammaire attribuée, où G_1 représente la grammaire indépendante du contexte sous-jacente, et α l'association traditionnelle d'attributs et de règles sémantiques.

α est appelée **fonction de couplage**, *et G_1 et G_2 sont deux* **grammaires couplées par attributs**, *si les conditions suivantes sont remplies :*

1. *A Z_1, α associe un unique attribut z, tel que type(z) = Z_2. La valeur de cet attribut est le résultat de la traduction spécifiée par α.*
2. *Un attribut a est, soit de type sémantique, type(a) $\in S$, soit de type syntaxique, type(a) $\in N_2$.*
3. *Les attributs de type sémantique sont définis dans les règles sémantiques de α, soit par la fonction identité, soit par des fonctions $f_1 \in F$.*
4. *Les attributs de type syntaxique sont définis dans les règles sémantiques de α, soit par la fonction identité, soit par la conditionnelle IfThenElse : Bool $\times N_2 \times N_2 \rightarrow N_2$, soit par une fonction $f_2 \in P_2$.*
5. *Le typage est fort, deux attributs ne peuvent être égaux que s'ils sont de même type.*

♣

A partir de cette définition, nous pouvons introduire une notion qui correspond à une autre vision des grammaires couplées par attributs. Il s'agit des grammaires attribuées fonctionnelles.

Définition II.2.4 : Grammaire attribuée fonctionnelle

Soit $G_1(\Sigma) = (N_1, T_1, Z_1, P_1)$ *et* $G_2(\Sigma) = (N_2, T_2, Z_2, P_2)$ *deux grammaires couplées par attributs, et* α *leur fonction de couplage.*

On note $G_1 \xrightarrow{\alpha} G_2$, *la* **grammaire attribuée fonctionnelle** *associée* $(G_1 \ , \alpha)$.

♣

Une grammaire attribuée fonctionnelle est donc une fonction qui à un arbre d'une grammaire G_1 associe un arbre d'une grammaire G_2.

Dans la suite, la fonction de couplage α représentera, par extension, la grammaire attribuée fonctionnelle qu'elle définit, et on pourra la noter $\alpha = (G_1, A_\alpha, F_\alpha)$, selon la notation classique d'une grammaire attribuée.

On peut maintenant introduire la description du principe de la *méta-composition* qui s'applique sur les GA fonctionnelles.

II.2.2 La méta-composition

La propriété des grammaires fonctionnelles qui les différencient des grammaires classiques est que l'on peut les voir comme des morphismes sur lesquels on peut définir une opération de composition. La composition de:
$\alpha{:}G_1(\Sigma) \rightarrow G_2(\Sigma)$ et $\beta{:}G_2(\Sigma) \rightarrow G_3(\Sigma)$ est $\gamma{:}G_1(\Sigma) \rightarrow G_3(\Sigma)$ qui spécifie la même traduction que β appliquée au résultat de α.

Pour la différencier d'une opération de composition classique de deux fonctions, qui pourrait se rapprocher de la composition de deux évaluateurs produits par deux grammaires attribuées, nous appellerons cette opération structurelle la *méta-composition* (plutôt que *composition descriptionnelle* dans [GaG84]) de grammaires attribuées.

La Méta-composition

On notera $\gamma = \alpha \circ \beta$ la grammaire attribuée qui spécifie la même traduction que $\beta = (G_2, A_\beta, F_\beta)$ appliquée après $\alpha = (G_1, A_\alpha, F_\alpha)$

La GA fonctionnelle $\gamma = (G_\gamma, A_\gamma, F_\gamma)$ est définie comme suit.

1. Syntaxes d'entrée et de sortie :
   - La grammaire d'entrée G_γ est la grammaire d'entrée de α : $G_\gamma = G_1$.
   - La grammaire de sortie de γ est la grammaire de sortie de β : G_3.

2. Attributs de $A_\gamma = \bigcup_{X \in N_1} \; A_\gamma(X)$:
   Chaque ensemble $A_\gamma(X)$ est décomposable en deux ensembles disjoints $A_\gamma/_S(X)$ qui représente l'ensemble des attributs sémantiques et $A_\gamma/_{P_2}(X)$ qui représente l'ensemble des attributs syntaxiques déclarés sur $X \in N_1$.

   - Cas des attributs sémantiques de $A_\alpha(X)$:
     $A_\gamma/_S(X) = \{ \; a \mid a \in A_\alpha(X) \; , \; type(a) \in S \; \}$.
   - Cas des attributs syntaxiques de $A_\alpha(X)$:
     $$A_\gamma/_{P_2}(X) = \{ \; x.y \mid x \in A_\alpha(X) \; , \; type(x) = Y \in N_2 \; , \; y \in A_\beta(Y) \; , \; type(x.y) = type(y) \}$$
     Si x est synthétisé alors $x.y$ a la même direction (synthétisé / hérité) que y, sinon $x.y$ a la direction inverse de celle de y.

     Un attribut syntaxique y (resp. sémantique) de A_β donne un attribut syntaxique (resp. sémantique) $x.y$ dans A_γ.

3. Règles sémantiques de $F_\gamma = \bigcup_{p \in P_1} \; F_\gamma(p)$:

   Chaque attribut de sortie d'une production est défini exactement par une règle sèmantique.

   Pour chaque production $p \in P_1$, on définit $F_\gamma(p)$:

   - Attributs sémantiques de sortie $X.a \in OW(p)$ provenant de α :
     $\forall a \in A_\gamma(X)$ tel que $type(a) \in S$ et $X.a \in OW(p)$
     soit $f \in F_\alpha(p)$ définissant $X.a$ alors $f \in F_\gamma(p)$.
   - Attributs de sortie $X.x.y \in OW(p)$ provenant d'attributs $X.x$ de type syntaxique $Y \in N_2$ dans A_α ;
     On note $f_\alpha \in F_\alpha(p)$ la règle sémantique où $X.x$ apparaît.
     On a alors deux cas :

     - f_α est une fonction syntaxique : $f_\alpha = p_2 \in P_2$.
       On trouve la définition $f \in F_\beta(p_2)$ de l'attribut $Y.y$ correspondant à $X.x.y$.
       On remplace dans f toutes les occurrences d'attributs de A_β par les occurrences d'attributs de A_γ qui leurs sont associées .
       On obtient ainsi une règle sémantique ne contenant plus que des occurrences d'attributs de γ et qui définit $X.x.y \in A_\gamma$.
     - f_α est l'égalité de $X.x$ avec un attribut $X'.x'$ du même type syntaxique Y.
       On génère alors dans $F_\gamma(p)$ l'égalité $X.x.y = X'.x'.y$.

La figure II.5 donne un exemple simple de méta-composition qui permet de mieux comprendre le mécanisme.

Les propriétés de la méta-composition concernant la conservation des appartenances aux sous-classes des GA sont décrites en détail dans [Gie86]. Dans cet article

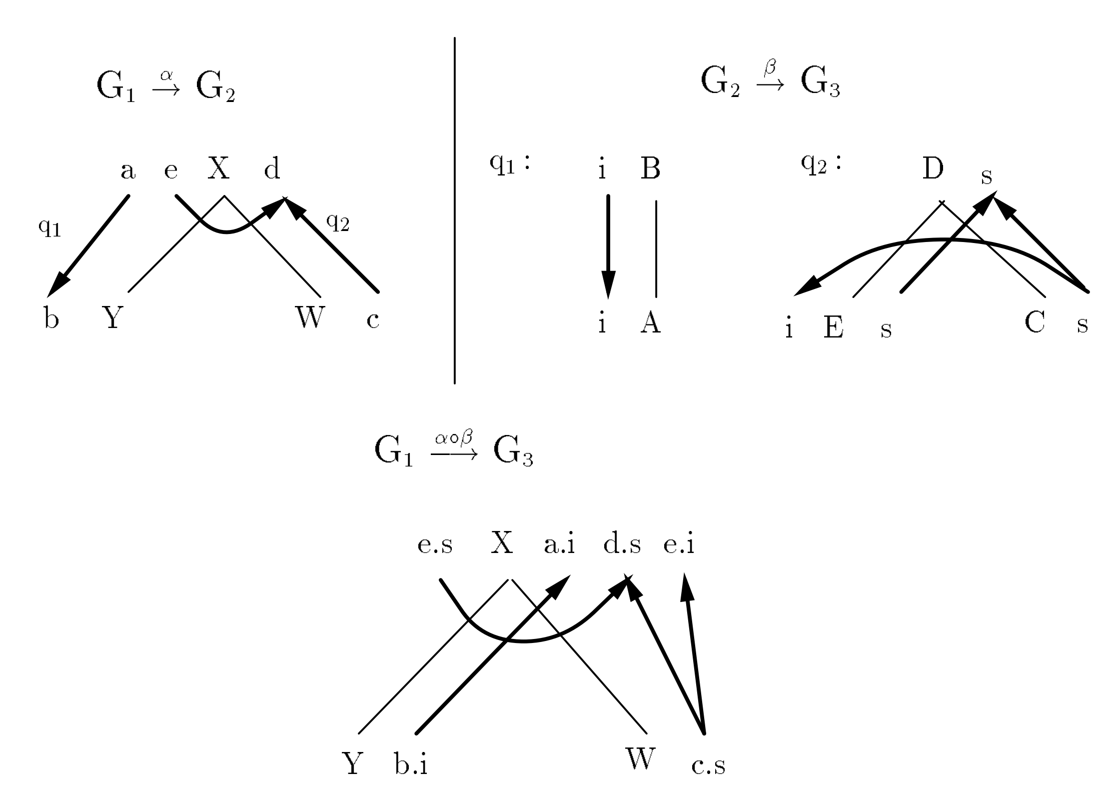


Figure II.5 : Exemple de méta-composition.

R. GIEGERICH discute entre autre de la réponse à la question :

> Le résultat de la méta-composition de deux grammaires attribuées fonctionnelles d'une classe C donnée est-elle aussi de la classe C (clôture pour la méta-composition) ?
>
> Par abus de notation : $C \circ C \subset C$?

Nous retiendrons un résultat concernant la sous-classe des GA qui nous intéresse le plus, à savoir les GA FNC.

On a ici :

$$\mathrm{FNC/S} \circ \mathrm{FNC/S} \subset \mathrm{FNC/S}$$

où FNC/S signifie que la GA est globalement FNC mais ne contient que des attributs syntaxiques synthétisés.

Un résultat tout aussi intéressant est directement déduit de la définition de la méta-composition. Ainsi, pour une classe C appartenant aux classes des GA Synthétisées Pures, GA Ordonnées, L-Ordonnées, DNC, FNC, NC[1] on a :

$$S \circ C \subset C$$

avec S la classe des GA Synthétisées Pures, c'est-à-dire ne contenant que des attributs synthétisés, donc en particulier des attributs syntaxiques uniquement synthétisés, avec une restriction : une occurrence d'attribut syntaxique ne doit pas être utilisée plusieurs fois sur une production (occurrence utilisée dans les deux branches d'un *IfThenElse* par exemple).

En effet, dans ce cas les attributs et les règles sémantiques du résultat sont directement calqués sur ceux et celles de la grammaire de la classe C sans modification de direction et les graphes de dépendances conservent leurs propriétés, ce qui garantit son appartenance à cette même classe C (les règles de calcul des attributs sémantiques de la grammaire synthétisée sont mises telles quelles dans la GA résultat et n'influencent en aucun cas la détermination de sa classe).

Ceci est vrai en particulier pour la classe FNC, qui nous intéresse car elle offre une grande puissance d'expression et que l'on sait transformer une grammaire de cette classe en une GA L-Ordonnée pour laquelle on sait produire un évaluateur efficace et économe en place mémoire.

II.3 Le système FNC–2

II.3.1 Caractéristiques de FNC–2

FNC–2 est un système moderne de traitement de grammaires attribuées en cours de développement dans le projet "ChLoE" de l'INRIA. Il se veut puissant, efficace et souple à utiliser.

FNC–2 est un système ayant une grande *puissance* d'expression puisqu'il est capable de traiter des grammaires attribuées Fortement Non Circulaires, qui représentent une large sous-classe des GA.

Il est également d'une grande *efficacité* : les évaluateurs générés sont déterministes, grâce à la transformation des GA FNC en GA l-ordonnées présentée, entre autre, dans la thèse de Didier PARIGOT [Par88].

Cela permet d'utiliser de puissantes techniques d'optimisation aussi bien en temps d'exécution, qu'en place mémoire, pour les évaluateurs produits.

Sa *facilité* d'utilisation est due à l'emploi d'un langage d'entrée, OLGA, spécialement conçu pour la spécification des grammaires attribuées, contrairement à

[1] ainsi que pour une sous-classe des GA k-Passes, les GA k-sweep pour lesquelles l'ordre d'évaluation des fils d'un nœud, qui est éventuellement modifié par la composition, ne joue pas sur la détermination de la classe

d'autres systèmes qui utilisent des langages classiques simplement augmentés de constructions pour accéder aux attributs.

De plus, une spécification de GA est totalement indépendante de son implantation, ce qui confère à FNC–2 sa *souplesse*. Il est ainsi facile de l'intégrer dans divers environnements.

On peut utiliser plusieurs langages cibles : il suffit de spécifier un traducteur vers un langage donné pour avoir l'évaluateur correspondant.

Pour des raisons de rapidité et de fiabilité, FNC–2 a d'abord été implanté en C, ce qui permet d'utiliser les tables générées par SYNTAX[2] [BDL87] et un certain nombre des fonctionnalités du système UNIX.

Un traducteur vers LE_LISP [Dur89], en plus du traducteur vers C, a été réalisé, ce qui s'imposait pour l'intégration de FNC–2 à CENTAUR[3] [BCD87].

La stratégie d'évaluation utilisée par FNC/ERN [Jou84], l'ancêtre de FNC–2 était de calculer la valeur des attributs à la demande. Mais la détermination statique de "séquences de visites" à partir de la grammaire permet d'optimiser les calculs [Par88].

Il existe également une possibilité de produire une version incrémentale des évaluateurs [Par88].

II.3.2 Le langage d'entrée : OLGA *(Ouf, un Langage pour les Grammaires Attribuées)*

Pourquoi un nouveau langage?

En fait, plusieurs systèmes de traitement des grammaires attribuées ont été conçus avant FNC–2 et les divers langages d'entrée utilisés pour spécifier les règles sémantiques ont été par exemple ALADIN[4] pour le système GAG, développé par U. Kastens [KHZ82].

Mais la plupart des autres systèmes utilisent comme langage d'entrée des langages existants (comme Algol-68 pour HLP [RSS83], MACLISP pour FNC/ERN, ou Pascal pour MUG2 [Gie79]) auxquels sont adjointes des notations permettant la déclaration et l'utilisation d'attributs.

OLGA se veut donc un langage dédié aux GA. Ses objectifs sont multiples :

- facilité d'apprentissage
- sureté et fiabilité de la programmation
- conception de programmes lisibles et modulaires

[2]SYNTAX est le logiciel développé par Pierre BOULLIER qui génère des *parser* et des *scanner* pour un langage à partir d'une spécification syntaxique.

[3]Générateur d'environnements de programmation écrit en LE_LISP.

[4]"A Language for Attributed DefINitions"

- être le plus proche possible de la théorie des grammaires attribuées
- être indépendant de la méthode d'évaluation proposée par le système
- être aisément traduisible dans la plupart des langages d'implantation des évaluateurs (C, LE_LISP, ADA, ...)
- intégrer des constructions facilitant l'optimisation de l'encombrement mémoire et du temps de calcul
- la simplicité.

Le compilateur OLGA effectue deux types d'opérations sur une GA. La première est la vérification de types et la seconde la traduction.

Lors du typage, des informations sont construites qui vont permettre de traduire la GA dans un langage donné (actuellement, C et LE_LISP).

La figure II.6 permet de mieux comprendre le fonctionnement du compilateur OLGA dans FNC–2, et ainsi de situer les éléments dont on va parler par la suite : le générateur d'évaluateur et les traducteurs, entre autres.

Le générateur d'évaluateur intégré dans le compilateur OLGA détermine, par une analyse globale des dépendances induites par les règles sémantiques, un ordre d'évaluation statique.

Certaines informations destinées au générateur d'évaluateur sont des *schémas de dépendances*. Ceux-ci sont des listes d'occurrences d'attributs servant de paramètres aux *fonctions* répertoriées pour le traducteur.

II.3.2.1 Les principaux concepts

- OLGA est un langage **applicatif**, c'est-à-dire sans affectation de variable mais avec seulement des expressions et des fonctions. Cela signifie donc qu'il ne peut y avoir d'effets de bord.

  Or la théorie des grammaires attribuées suppose qu'un attribut associé à un nœud possède une et une seule valeur. Ceci sous-entend que cette valeur ne peut par la suite être modifiée : OLGA satisfait donc bien cette exigence.

  OLGA permet néanmoins des connections avec des fonctions externes pouvant faire des effets de bord, mais ceci est laissé sous l'entière responsabilité du programmeur!

  OLGA n'est pas un langage *fonctionnel*: les fonctions ne peuvent être passées en paramètres.

- OLGA est un langage **fortement typé**.

  Une vérification de types est effectuée sur chaque expression OLGA afin de détecter le maximum d'inconsistances et d'erreurs lors de la compilation. Les

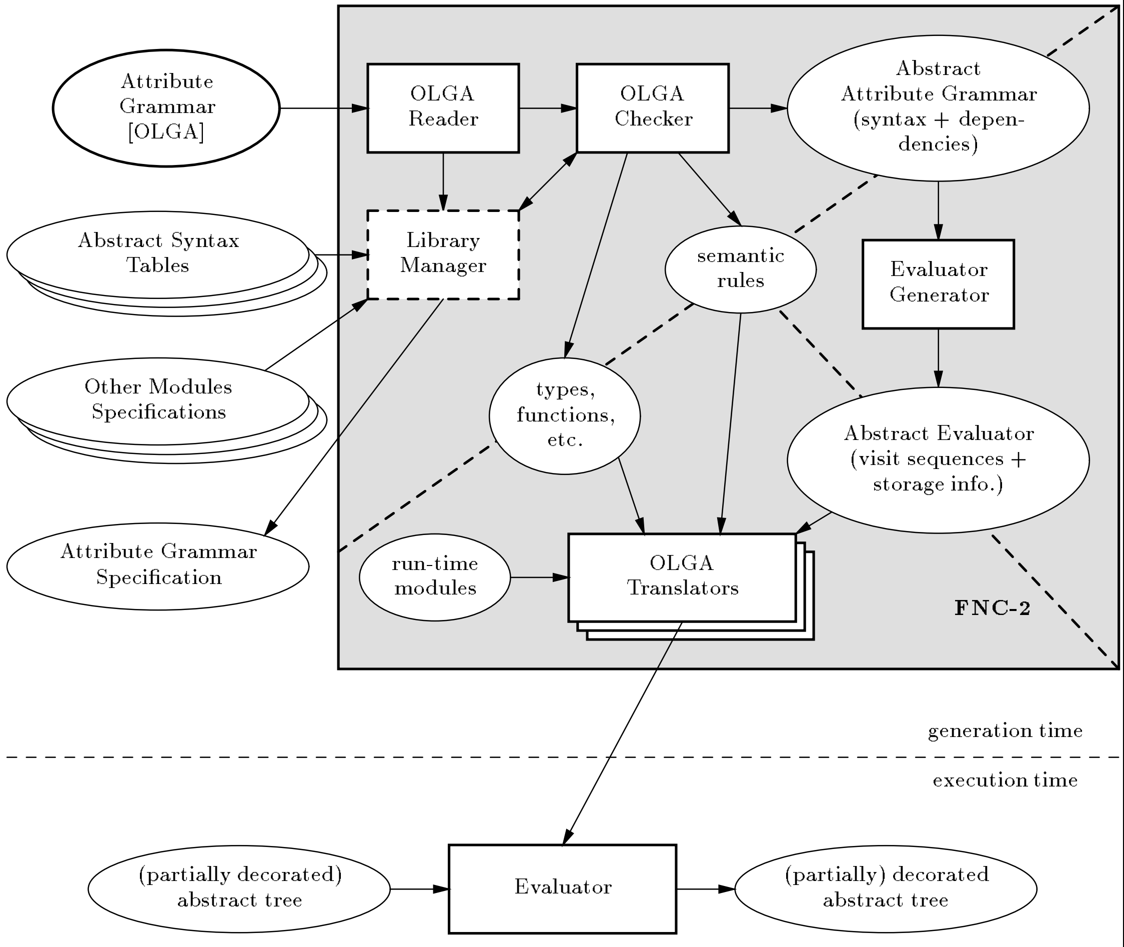


Figure II.6 : Le système FNC–2.

déclarations de types peuvent être optionnelles. Il est d'ailleurs prévu d'implanter un mécanisme d'inférence de types pour déterminer le type de chaque objet à la compilation. L'utilisation de fonctions polymorphiques est pour l'instant limitée à certaines fonctions prédéfinies.

Les types prédéfinis sont : les entiers (*int*), les réels (*real*), les booléens (*bool*), les chaînes de caractères (*string*), les caractères (*char*), les symboles lexicaux (*token*: objet représentant une chaîne pour SYNTAX), le type vide (*void*), la position dans le texte source (*source-index*).

Avec les types structurés, on peut représenter :

- un enregistrement, ensemble de champs de types prédéfinis ou structurés (*record*);

- une union de types prédéfinis ou structurés (*union*);

- une liste homogène d'éléments de type prédéfini ou structuré (*liste*);

- un ensemble d'éléments de type scalaire (*set*).

Les trois premiers types peuvent être récursifs.

On peut également représenter un ensemble d'éléments énumérés sur lesquels on peut faire des opérations de comparaison (type énuméré).

- La **modularité** est un concept de base d'OLGA.

  En effet, les définitions de constantes, types et fonctions peuvent être écrites dans des modules séparés. Même une grammaire attribuée peut être structurée en *phases*, et, bien sûr, en *productions*.

  Chacun de ces niveaux structurels peut comporter des définitions d'objets locaux. Il existe d'ailleurs des modules de déclaration dans lesquels sont déclarés les objets “visibles” par les autres blocs et des modules de définition où l'on définit les objets cachés.

  Chaque bloc peut comporter des clauses d'importation permettant d'inclure un module entier ou seulement une partie. Ceci implique qu'il existe des dépendances entre modules.

- La base syntaxique des GA écrites en OLGA n'est pas une syntaxe concrète, comme dans la théorie classique [Knu68], mais plutôt une **syntaxe abstraite** comme dans [VoM82].

  Elle permet la description de la structure des notions d'un langage au lieu de son apparence réelle.

  Une grammaire attribuée est considérée par le système FNC–2 comme une fonction allant d'une Syntaxe Abstraite Attribuée dans une autre, c'est pourquoi il est important d'en connaître les principales caractéristiques. La SAA *d'entrée* sert de base à la GA décrite, et les règles sémantiques construisent un arbre décoré de la SAA de sortie. Il s'agit là d'une grammaire attribuée fonctionnelle.

  La partie suivante est destinée à décrire la Syntaxe Abstraite Attribuée utilisée par OLGA.

II.3.2.2 Syntaxe Abstraite Attribuée *(SAA)*

Celle-ci présente deux principaux avantages :

- Une syntaxe abstraite est libre de toutes les contraintes syntaxiques gênantes que l'on peut rencontrer avec les méthodes d'analyse utilisées par les générateurs d'analyseurs. Elle exprime ainsi plus clairement l'imbrication des constructions sémantiques.

- Les arbres abstraits sont généralement plus petits que les concrets, car ils ne contiennent pas de mots-clés et autres informations contextuelles.

Le formalisme utilisé pour décrire les arbres abstraits est très proche de celui de Metal [KLM83], utilisé dans l'environnement de programmation Mentor développé à l'INRIA (ainsi que dans CENTAUR). Il consiste à définir des **phyla** et des **opérateurs**, que l'on peut respectivement assimiler à des *non-terminaux* et à des *productions* d'une grammaire indépendante du contexte usuelle.

Les opérateurs et les phyla

Un *phylum* est une notion syntaxique du langage que l'on assimile à un ensemble d'opérateurs.

Chaque occurrence dans un arbre abstrait est associée à un phylum qui indique quels opérateurs sont autorisés à cette place.

Les *opérateurs* représentent en fait les nœuds des arbres abstraits, et les feuilles sont constituées par les opérateurs d'arité nulle.

Ils sont divisés en deux classes :

- Les opérateurs d'*arité fixe* sont des opérateurs *hétérogènes*, c'est-à-dire qu'ils peuvent avoir différents phyla comme opérandes, et ce en nombre fixe.

  Parmi eux, on peut distinguer une certaine catégorie d'opérateurs d'arité fixe : les *opérateurs nullaires prédéfinis*, d'arité 0, et destinés au constructeur d'arbre abstrait. Ils sont automatiquement ajoutés à la liste des opérateurs de chaque phylum de la syntaxe abstraite et peuvent être alors considérés comme des “opérateurs-erreurs”.

- les opérateurs d'*arité variable* sont des opérateurs *homogènes*, c'est-à-dire que leurs opérandes appartiennent au même phylum, mais sont en nombre variable.

  Ceux-ci sont également appelés *opérateurs de liste* et donnent lieu à des constructions spéciales, présentées ultérieurement, permettant d'accéder aux attributs des productions de liste ou de les définir.

Le langage asx

Définir une *SAA* revient en fait à décrire des opérateurs, des phyla, mais aussi des **attributs**.

Le langage de définition d'une *SAA* utilisé dans FNC–2 se nomme *asx*.

Il permet de définir :

- un opérateur : par son arité et les phyla associés à ses fils,

- un phylum : par les opérateurs qu'il représente,

- un attribut : par les phyla auxquels il est rattaché, son nom (*$nom*) et son type (type Olga prédéfini ou déclaré dans un module de déclaration importé par un module de description *asx*).

En effet, un module *asx* peut importer un module Olga de déclaration, ce qui permet, entre autres, d'utiliser les types déclarés. Les *dépendances* entre les modules *asx* et les modules Olga sont différentes selon la nature de ces derniers.

De plus, une GA étant considérée par Fnc–2 comme une fonction d'une *SAA* dans une autre, un module Olga de spécification de GA prend en entrée un arbre abstrait attribué, donc un module *asx*, et peut en fournir un ou plusieurs en sortie.

Le nom d'un attribut ne doit pas faire partie des noms réservés. Ces noms correspondent à des attributs prédéclarés contenant des informations syntaxiques et utilisés uniquement dans les règles sémantiques.

II.3.2.3 Les modules Olga de spécification de GA

Un module Olga comporte donc une base syntaxique constituée par la SAA d'entrée, des attributs, et des règles sémantiques.

Nous avons décrit la composition d'une SAA, voyons maintenant

Les attributs

Les attributs manipulés dans un module Olga de spécification de GA sont de plusieurs sortes :

- les attributs de la *SAA d'entrée*, qui constituent des données pour la grammaire attribuée et se comportent donc comme des constantes dans chaque production qui en contient une occurrence,

- les attributs de la *SAA de sortie*, représentant le résultat du calcul sémantique effectué dans la production qui en contient une occurrence,

- les attributs *locaux* à la grammaire attribuée, appelés attributs de travail, contenant les résultats des calculs effectués sur chaque nœud de l'arbre.

  Ceux-ci sont définis dans le module de spécification Olga de la même façon que les autres, mais comportent une information supplémentaire : la manière dont on les utilise dans les règles sémantiques des productions. Il faut donc préciser s'il s'agit d'attributs *hérités*, *synthétisés*, *mixtes*, ou *globaux*.

Un attribut *mixte*, ou *composé*, est défini par un couple d'attributs, l'un *synthétisé*, et l'autre *hérité*. Tous deux se comportent comme leur genre l'indique, mis à part qu'ils sont associés au même phylum et ont le même type.

L'information véhiculée par un attribut *mixte* peut donc partir du haut de l'arbre pour y être remontée ensuite avec les modifications effectuées lors du parcours (ceci est très utile pour construire une table de symboles cohérente en parcourant la partie déclaration d'un bloc dans un langage).

Un attribut *global* se comporte comme un attribut *hérité* qu'on peut l'utiliser, sans le modifier, dans tout le sous-arbre à partir duquel il est défini sans avoir à le redéfinir à chaque production (cas d'une table de symboles dans une partie instructions d'un bloc dans un langage).

Les noms réservés aux attributs prédéclarés et contenant des informations relatives aux productions sont :

$arity, de type *int*, donnant l'arité d'un opérateur,

$position, de type *int*, donnant la position d'un phylum dans une production,

$scx, de type *source-index*, attaché à chaque phylum et donnant sa position dans la grammaire.

II.3.2.4 La spécification des règles sémantiques

Les règles sémantiques sont attachées à chaque production d'une syntaxe abstraite et spécifient les valeurs qui vont être assignées aux occurrences d'attributs de sortie de cette production. Ces valeurs sont définies par une expression pouvant comporter des occurrences d'attributs d'entrée. La grammaire attribuée est ainsi en Forme Normale [Boc76].

Les productions

Les *productions* OLGA sont exactement le pendant des productions des grammaires concrètes, c'est-à-dire la spécification d'un opérateur et de ses descendants. Cette spécification n'est autre que la répétition de celle définie dans la *SAA* d'entrée.

À une production, on associe une liste de règles sémantiques définissant les occurrences d'attributs.

Le groupe ainsi formé a cette forme :

> **where** *production* **use**
>   *règles sémantiques*
> **end where;**

Une production est considérée comme un bloc dans lequel des déclarations locales peuvent apparaître.

Les déclarations de *valeurs* que l'on définit avec des occurrences d'attributs d'entrée sont comme des règles sémantiques. Ces valeurs jouent le rôle d'*attributs locaux*, c'est-à-dire d'attributs attachés à une production, plutôt qu'à un phylum. La spécification associée à une production peut alors avoir cette forme :

> **where** *production*
> **declare**
>   **value** *valeur : type := expression* ,
>   *déclarations ...*
> **use**
>   *règles sémantiques*
> **end where;**

Les règles sémantiques

Les règles sémantiques sont des fonctions de calcul des occurrences d'attributs de sortie d'une production en fonction des occurrences d'entrée.

Les occurrences d'attributs servent à décrire un attribut attaché à un symbole de la production (opérateur ou phyla de la partie droite). Spécifier le nom du symbole n'est pas toujours suffisant pour savoir de quelle occurrence de l'attribut il s'agit (lorsque le symbole apparaît plus d'une fois dans la production, ou s'il s'agit d'une production de liste), c'est pourquoi il est parfois nécessaire d'ajouter un *qualificateur*.

Les règles sémantiques pour les productions d'arité fixe peuvent comporter des occurrences d'attributs que l'on distingue alors par des nombres entiers qui joue le rôle de qualificateurs :

La première occurrence d'un phylum *PH*, c'est-à-dire la plus à gauche, est désignée par *PH.1*, la suivante par *PH.2*, et ainsi de suite.

La spécification des occurrences d'attributs dans une production de liste est ici plus compliquée car le nombre des descendants d'une production de liste n'est pas connu statiquement : on ne peut donc distinguer un symbole particulier ou une occurrence d'attribut donnée.

Les qualificateurs utilisés ici sont *left, right, first* et *last.*

Pour une production $op \rightarrow PH*$, *PH.left* désignera le frère gauche du phylum *PH*, *PH.right* celui de droite, et *PH.first* et *PH.last* les premier et dernier fils de *op*.

Il existe en particulier deux constructions spécifiques aux règles sémantiques des productions de liste :

- La première construction :

> **case position is**
>   **first** : *expression* ;
>   **last** : *expression* ;
>   **other** *expression* ;   clause obligatoire
> **end case**;

concerne la définition des attributs hérités d'un membre droit de règle dont la position peut correspondre à l'une des trois sortes proposées, et où ne sont visibles (c'est-à-dire, dans laquelle on ne peut utiliser) que les frères droit et gauche et le membre gauche de la production.

La clause *other* est obligatoire et doit être accompagnée d'au moins une des deux autres.

- La seconde construction :

case arity is
  0 : *expression1* ;
  1 : *expression2* ;
  other *expression3* ; clause obligatoire
end case;

concerne la définition des attributs synthétisés d'un membre gauche de règle, où ne sont visibles que les premier et dernier fils de celui-ci.

Là aussi la clause *other* est obligatoire et doit être accompagnée d'au moins une des autres.

De plus, il existe un opérateur spécial permettant d'utiliser les attributs des fils : l'opérateur **map**.

Les principales expressions autorisées dans les règles sémantiques

- L'opérateur **map** peut être utilisé dans des productions de liste pour calculer une valeur d'attribut synthétisé ou d'attribut local en accédant aux attributs synthétisés de tous les fils par un parcours gauche-droite ou droite-gauche de la liste de ces fils.

  map *direction fonction*
    value *init*
    other *attributs*
  end map;

  La *direction* peut prendre les valeurs *left* ou *right*, selon que le parcours du corps de la production se fait dans le sens gauche-droite ou droite-gauche.

  La *fonction* est la fonction à appliquer sur les occurrences d'attributs décrites dans la clause *other*.

  init est la valeur initiale de l'expression : elle sert lors du premier appel de la fonction.

- La construction **case ... is** détermine une expression de sélection permettant de choisir une valeur parmi d'autres selon plusieurs types de tests (égalité d'une valeur scalaire avec une constante, filtrage structurel d'une valeur selon des schémas faits à partir de constructeurs, test de l'appartenance d'une valeur à une variante d'un type union).

Notons que les structures décrites précédemment pour les règles sémantiques des productions de liste sont basées sur cette construction (elles sélectionnent sur l'arité d'une production, en prenant comme valeur l'attribut prédéfini *\$arity*, ou sur la position d'un phylum, en prenant cette fois l'attribut *\$position*).

Exemple :

> **case** *valeur* **is**
> *val1* : *expression1* ;
> *val2* : *expression2* ;
> **other** *expression3* ;   clause obligatoire
> **end case**;

- La construction **let ... in** permet de définir certaines valeurs pouvant dépendre des attributs d'entrée de la production, avant de les utiliser dans les règles sémantiques.

  Exemple :

> **where** *production* **use**
> *\$attribut1* :=
>   **let** *valeur* : *type* := *expression*
>   **in**
>   *expression* utilisant *valeur* ;
> *\$attribut2* := *expression* ;
> **end where;**

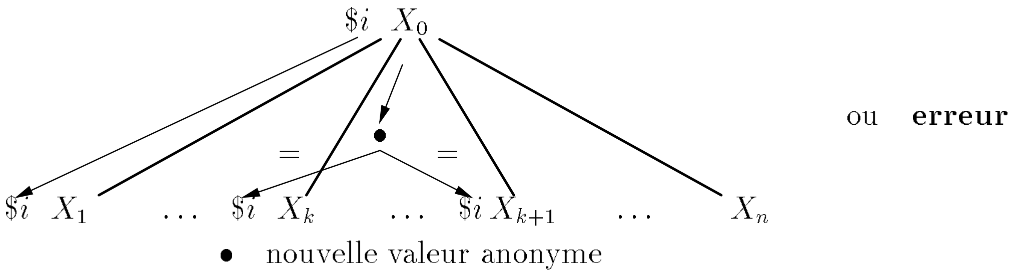
II.3.2.5 Les règles sémantiques de copie

Les règles sémantiques de copie sont des règles chargées de transférer la valeur d'une occurrence d'attribut $\$a(X)$ dans une autre occurrence d'attribut $\$b(Y)$.

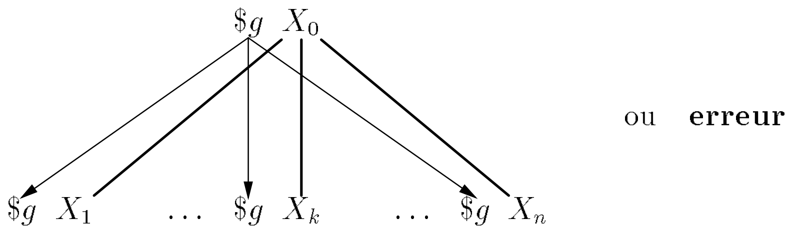
Ces règles n'ont d'autre sémantique que de véhiculer des informations sans les modifier selon des directions particulières qui dépendent du type de l'attribut.

Elles constituent souvent une grande partie des règles sémantiques d'une GA. Dans certains cas, il est possible de les induire d'après les attributs eux-mêmes. Ainsi, si dans une production $X \rightarrow \ldots Y \ldots$ on a l'occurrence $\$a(X)$ à définir et que l'on a en entrée$\$a(Y)$, il y a de grandes chances pour que la règle sémantique soit $\$a(X) = \$a(Y)$. C'est dans cette idée que l'on a été amené à décrire des comportement par défaut pour les attributs de même nature dans une production. Ces comportements ne sont décrits qu'à partir de règles de copie.

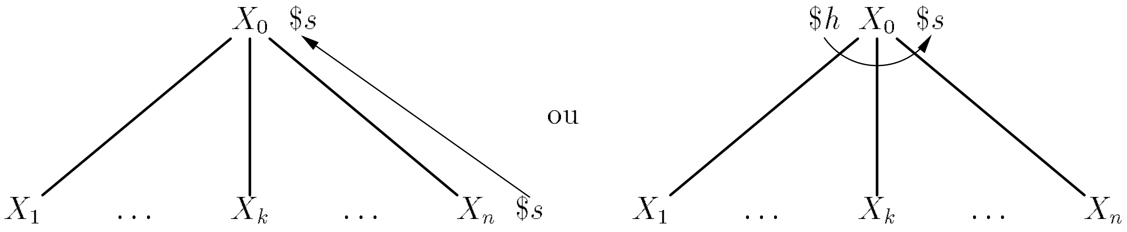
A défaut d'une autre règle, un attribut *hérité* attaché à un membre droit d'une production prend la même valeur que ceux de même nom de ses frères (valeur égale à celle de l'attribut du père).



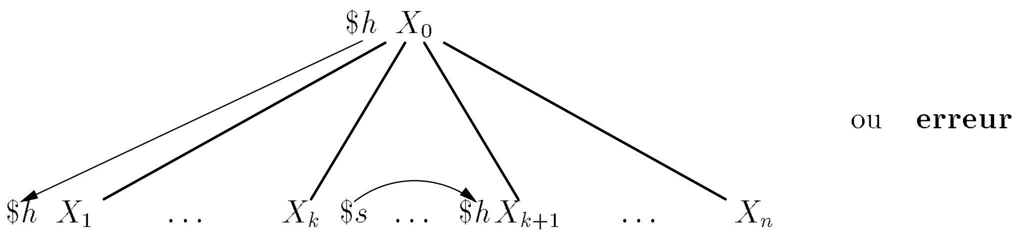
Un attribut *global* prendra, lui, la valeur du même attribut de l'opérateur, son père.



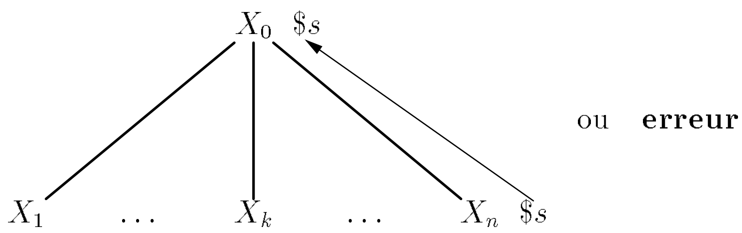
Ainsi, un attribut *composé-synthétisé* de l'opérateur prendra la valeur du *composé-synthétisé* du dernier fils, si possible, et sinon celle du *composé-hérité* de l'opérateur.



De même, un attribut *composé-hérité* d'un membre droit prendra la valeur, du *composé-hérité* de l'opérateur s'il s'agit du premier fils, et celle du *composé-synthétisé* de son frère de gauche sinon.



Un attribut *synthétisé* de l'opérateur prend, la valeur du *synthétisé* du même nom du dernier fils, s'il existe.



Lorsque des occurrences d'attributs de sortie d'une production n'ont pas de règles sémantiques pour les définir, ce sont ces règles que le système essaie de construire à leur place. Elles n'ont donc pas besoin d'apparaître explicitement dans la spécification, ce qui constitue un gain certain de taille en nombres de règles sémantiques à écrire.

II.3.3 Conclusion

FNC–2 est un système permettant de générer des compilateurs et des évaluateurs efficaces pour la grande majorité des applications dirigées par la syntaxe.

Le principal inconvénient est commun avec tous les principaux systèmes de GA : il devient assez lourd à maniouler dés que l'on traite de grosses applications nécessitant l'écriture d'un grand nombre de règles sémantiques. Néanmoins, il comporte un certain nombre de notions qui lui permettent de se situer parmi les systèmes les plus prometteurs.

En effet, il inclut déjà modularité (modules de déclaration, de définitions, de spécification de GA et phases de GA), efficacité, incrémentalité et puissance d'expression (grammaire FNC). Tel qu'il vient d'être décrit, il lui manque l'inclusion de concepts plus puissants que la génération de règles de copie par défaut pour permettre d'écrire le moins de choses possibles.

Chapitre III

Mise en œuvre des Grammaires attribuées

III.1 Problèmes rencontrés

Globalement, si l'on veut pouvoir utiliser les grammaires attribuées comme un outil de spécification agréable et facile d'emploi, se posent alors des problèmes basés essentiellement sur l'utilisation d'une unique grammaire attribuée qui décrit toute une application.

En effet, on obtient alors une grammaire attribuée pouvant atteindre une taille impressionnante, et quasiment illisible et non maintenable en raison du trop grand nombre d'attributs regroupés dans une production et ne concernant pas les mêmes concepts sémantiques.

De plus, il est bien sûr hors de question de parler de partage des tâches lors d'un développement!

Il se dégage donc deux principaux problèmes : la taille, en nombre de règles sémantiques écrites, et la non modularité, en terme de séparation de concepts (et non pas seulement en terme de nombre de fichiers).

Les sections suivantes décrivent les différentes solutions existantes pour résoudre ces problèmes.

III.2 Solutions au problème de la taille

Dans chaque production, chaque occurrence d'attribut de sortie doit être définie une (et une seule) fois, ce qui peut, à partir d'un certain nombre d'attributs, devenir très "encombrant". C'est pourquoi il est nécessaire de réduire le plus possible la quantité d'informations à fournir explicitement.

Les deux parties suivantes concernent les méthodes mises en œuvre pour permettre l'élimination dans l'écriture d'une GA, de certains types de règles sémantiques. Il s'agit de certaines *règles de copie* et des *règles par défaut*.

III.2.1 Règles de copie

Dans la plupart des GA, une forte proportion de règles ne sont en fait que des règles de copie (identités ou règles de transfert), c'est à dire de la forme:

$X_1.a_1 = X_2.a_2$

De plus, une très large majorité d'entre elles concerne des attributs de même nom ($a_1 = a_2$). A partir de cette constatation, il a été possible de définir un service de génération automatique de règles sémantiques manquantes. En effet, ces types de règles peuvent alors ne pas être écrites, mais générées à partir d'un certain nombre de règles de copie "préconstruites" associées à chaque type d'attribut (hérité, synthétisé ou autre).

Ce type de service a été introduit par Bernard LORHO dans le système DELTA [Lor77], puis mis en place dans FNC/ERN [Jou84], ancêtre de FNC–2, ainsi que dans FNC–2 lui-même.

Cette méthode permet d'éviter d'écrire ces règles de copie, souvent nécessaires mais sans réel intérêt sémantique car elles ne servent qu'à transporter des informations le long de l'arbre sans les modifier. Cela s'avère très efficace puisque la taille des applications, nombre de règles, est réduite d'environ 50 % en moyenne.

III.2.2 Règles par défaut

Mais l'expérience de programmation avec les grammaires attribuées montre qu'il reste encore un trop grand nombre de règles autres que les règles de tcopie. Or, nombreuses sont celles qui sont construites suivant un modèle similaire pour un même attribut dans différentes productions. Ces règles répétitives prennent une place "inutile" dans la grammaire, sans compter l'ennui, pour l'utilisateur, d'avoir à les recopier (même avec un éditeur de texte performant!).

Cela nécessite bien sûr de se munir d'une structure supplémentaire permettant de spécifier ces règles sémantiques et, à partir de celles-ci, de générer les règles manquantes.

Un tel outil a déjà fait l'objet de travaux tels que ceux de M. D. Tiemann [Tie87], où il décrit une approche "orientée sémantique" permettant de réduire le nombre de règles sémantiques de façon significative. Il présente en fait une notation qui permet de spécifier des modèles de comportement généraux des attributs, appelés *équations de flux*. Un générateur crée ensuite les règles sémantiques associées aux attributs selon leur comportement et le contexte syntaxique dans lequel ils apparaissent.

Il utilise des concepts intéressants, comme celui de *classe*.

Une classe regroupe ici tous les attributs dont on veut décrire le comportement.

Prenons un exemple:

Soit l'attribut a défini sur un certain nombre de non-terminaux: A, B ,C .

La classe correspondante sera définie ainsi:

defclass $<a>$ $A.a$, $B.a$, $C.a$

et l'équation de flux associée, si l'attribut est synthétisé:

flows $<a>$ (*liste de choix de règles*)

ou, si l'attribut est hérité:

flowi $<a>$ (*liste de choix de règles*)

Les règles de la liste sont classées dans un ordre particulier: elles vont de la plus spécifique à la moins spécifique. Ainsi, l'opérateur de choix, en les examinant dans l'ordre, va choisir celle qui correspond le mieux au contexte.

Notons qu'ici, le choix s'opère à travers un *filtrage par la sémantique*, puisqu'il s'agit en fait de vérifier si chaque règle est applicable dans le contexte de la production, c'est à dire vérifier l'existence des occurrences d'attributs mises en cause et leur bonne utilisation dans la règle.

III.3 Solutions au problème de la modularité

III.3.1 Le langage MARVIN

Le langage applicatif MARVIN a été spécifié pour permettre l'écriture de compilateurs modulaires. Il constitue un enrichissement de Modula-2, par l'apport des concepts de signatures, termes et couplage par attributs comme décrit section II.2.

Schématiquement, une signature est un module exportant des types et des fonctions construisant des valeurs de ces types. Les types de données de base sont des termes de différentes natures, utilisés pour représenter des valeurs syntaxiques et sémantiques. Les grammaires couplées par attributs sont utilisées pour définir des fonctions unaires sur des objets de ces types.

Le monde sémantique de Modula-2 n'est donc pas affecté par cette extension.

MARVIN est fortement typé et propose deux types de modularité : le concept de module hérité de Modula-2 et la modularité obtenue par le mécanisme de composition de ses fonctions.

En effet, un des concepts permettant d'augmenter la modularité des spécifications, outre le concept de module, est celui de la composition fonctionnelle d'opérateurs. En effet, il est clair que les termes qui apparaissent comme résultat d'une passe de compilateur peuvent être la donnée source d'une autre passe, sans donner lieu à un quelconque changement de représentation. Ceci permet de structurer les compilateurs en passes, d'où une séparation des concepts et une plus grande sécurité dans le développemnt.

Les langages intermédiaires, décrits par des termes, peuvent ainsi être spécifiés suivant les besoins et les possibilités relatifs à un problème donné.

On peut alors appliquer la méthode de la méta-composition des grammaires couplées par attributs, décrite dans la section II.2.2, sur un certain sous-ensemble d'opérateurs définis à l'aide des grammaires couplées.

On peut ainsi écrire une spécification de manière modulaire, consistant en un certain nombre de phases de transformations simples, sans obtenir un compilateur qui ait besoin de construire toutes ces formes intermédiaires. Cela n'exclut cependant pas la possibilité de tester chacune de ces phases indépendamment.

III.3.2 Les Grammaires Attribuées Incomplètes

III.3.2.1 Description

Cette vision est issue de la thèse de Claude Kleinhans [Kle91].

Le principe en est simple puisqu'il s'agit de construire des Grammaires Attribuées représentant un certain nombre de concepts que l'on veut retrouver dans une seule application. Ces GA sont dites Incomplètes car leur syntaxe de base ne prend en compte qu'un certain nombre d'objets types (règles syntaxiques, non-terminaux, attributs, types, règles sémantiques) pour le concept qu'elle représente.

Elles sont basées sur des syntaxes abstraites : ainsi les terminaux sont inexistants, et apparait la notion de *phylum* qui représente un ensemble d'*opérateurs*, assimilés aux productions.

Il existe des attributs de *sortie* et des attributs *d'entrée*, permettant éventuellement un lien avec d'autres GAIs.

Les occurrences d'attributs en entrée d'une GAI ne sont pas définie : il faut donc les mettre en correspondance avec des occurrences d'attributs en sortie d'autres GAIs.

Ces GAIs sont réunies en une seule à l'aide d'une spécification de connexion et d'une opération de fusion.

La spécification de connexion consiste à renommer chaque objet d'une GAI (règles syntaxiques, non-terminaux, attributs, types, règles sémantiques), ce qui réalise en quelque sorte une projection.

L'opération de fusion permet de projeter un ensemble de GAIs sur un même "plan", selon la spécification de connexion. Elles peuvent donc se chevaucher : certains objets appartenant à des GAIs différentes sont renommés de la même manière, ils sont alors partagés (cf figure III.1).

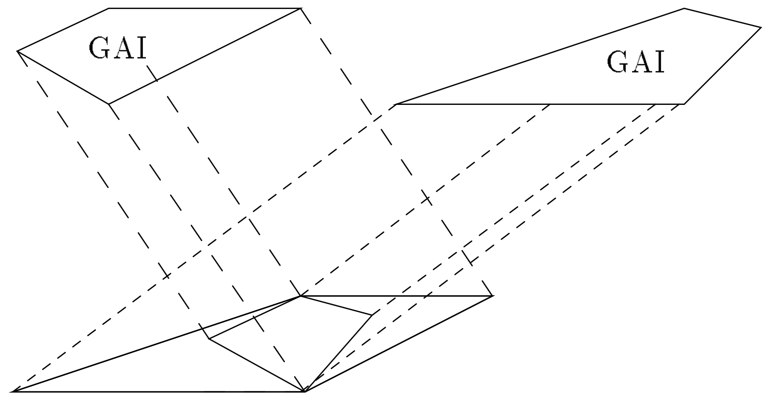


Figure III.1 : Projection de deux GAIs sur un plan : GA résultat.

III.3.2.2 Opération de fusion

L'opération de fusion permet de composer des GAIs en renommant chacun de leurs identificateurs, de manière à projeter ces GAIs l'une sur l'autre, en partageant certaines parties communes, ce qui réalise l'amalgame des syntaxes et des calculs sémantiques des différentes GAIs.

Il apparaît alors deux concepts :

- Aucun objet d'une GAI (règle syntaxique, phylum, attribut, type, règle sémantique) ne disparait et aucun objet nouveau n'apparait.

  Tous les objets se projettent en se renommant.

- La sémantique des fonctions de compilation (règles sémantiques) et des types des attributs ne change pas après projection.

Afin de réaliser la fusion, il est nécessaire de définir un concept de connexion. En effet, il faut que les GAIs se connectent entre elles sans erreurs ou conflits.

Plusieurs conditions garantissent la bonne connection d'une famille de GAIs, et notamment deux portant sur les définitions des attributs ainsi mis en commun :

1. Tous les attributs synthétisés qui résultent de l'opération de fusion et apparaissent en partie gauche d'une production doivent être définis par une règle sémantique.

2. Tous les attributs hérités qui résultent de l'opération de fusion et apparaissent en partie droite d'une production doivent être définis par une règle sémantique.

La figure III.2 montre un exemple de fusion où la condition 1 n'est pas vérifiée : les deux GAIs ne peuvent donc pas être bien connectées, la fusion n'est pas valide.

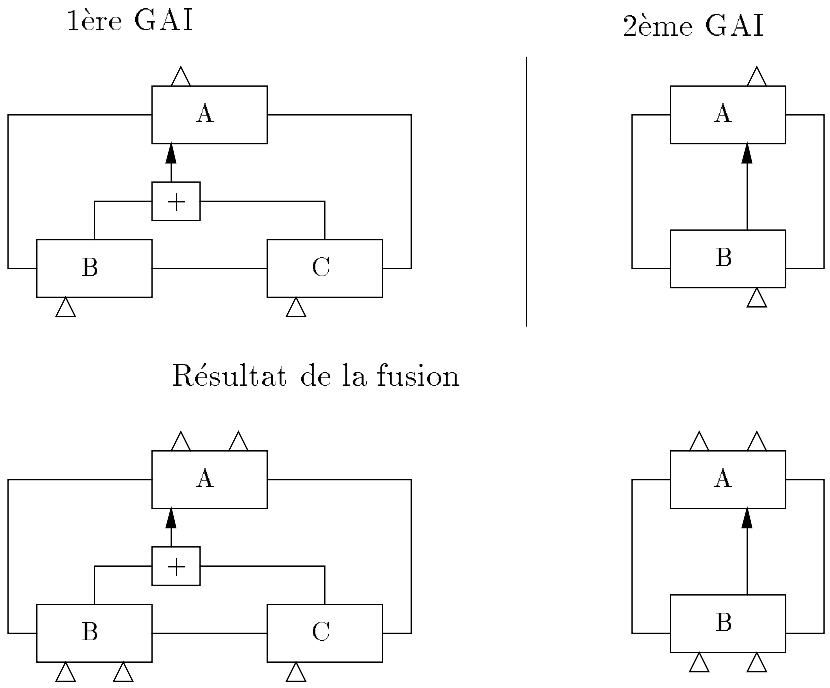


Figure III.2 : Fusion de deux GAIs.

Le type des attributs est déterminé par le sens des têtes de flèches $\triangle$ qui les représentent.

La figure III.3 nous montre une fusion valide de deux GAIs. Les occurrences d'attributs en entrée de la grammaire sont symbolisés par $\square$, et les occurrences d'attributs en sortie qui peuvent s'y rattacher, par $\diamondsuit$.

III.3.2.3 Remarques

Cette méthode des GAIs est principalement un outil de structuration, permettant de séparer les différents concepts apparaissant dans la construction d'une application type compilateur.

Contrairement à ses ambitions, ce système n'offre pas une souplesse suffisante, car il est encore trop lié à une structure syntaxique particulière. La réutilisabilité des modules est limitée aux applications ayant une ressemblance structurelle très forte.

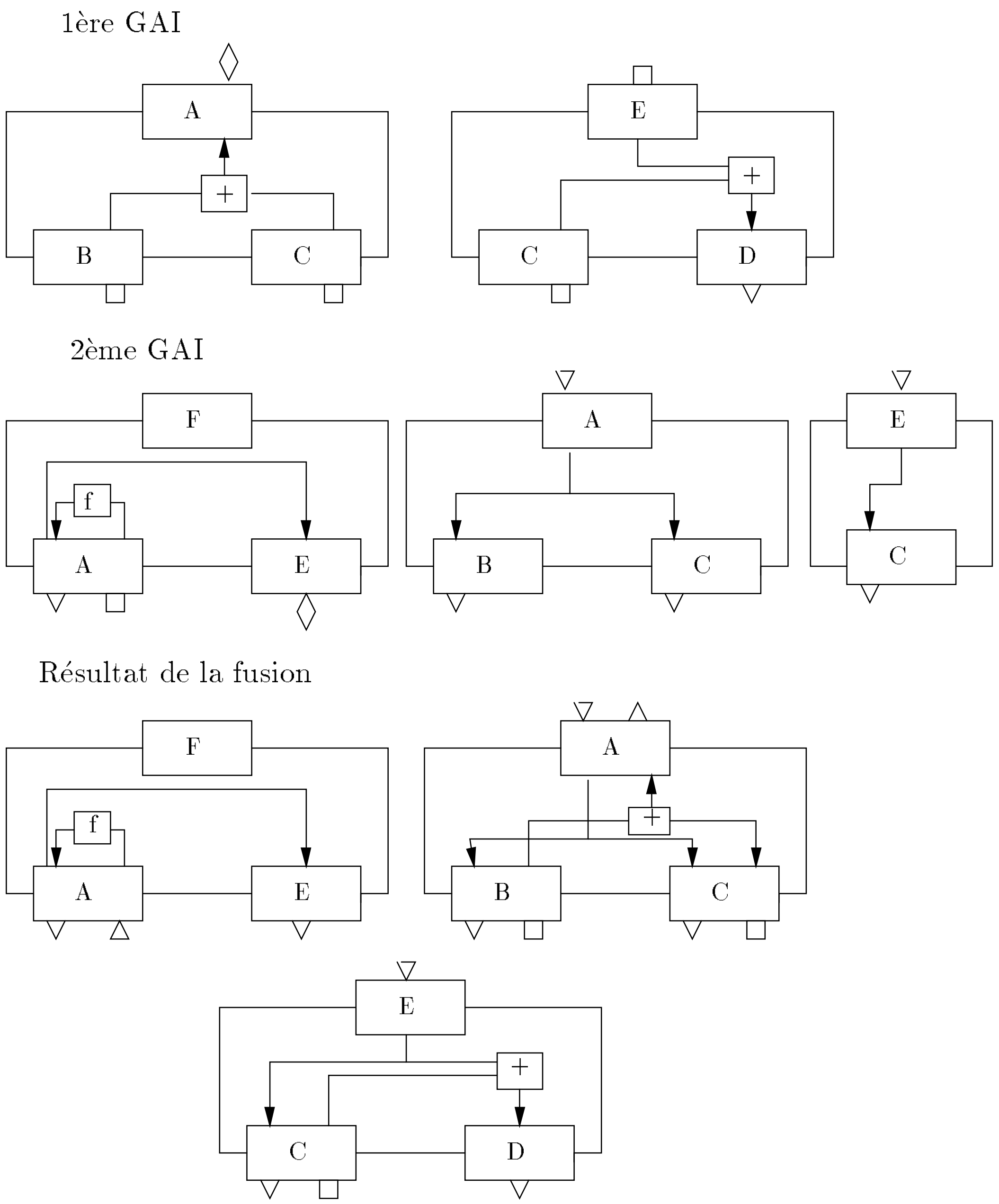


Figure III.3 : Fusion valide de deux GAIs.

III.3.3 Les Grammaires Attribuées Modulaires

Le concept de MAG (Modular Attribute Grammar) est la réponse de G.D.P. Dueck et G.V. Cormack [DuC90] à l'introduction de la modularité dans les grammaires attribuées.

Il s'agit en fait d'un générateur de GA à partir de règles regroupées en grammaires attribuées modulaires (MAG).

Chaque règle décrit une classe de calculs d'attributs, rendant ainsi les MAG nettement moins complexes que les GA équivalentes. Elles peuvent alors être structurées selon les critère d'une décomposition modulaire.

Une MAG est donc en fait un ensemble de filtres et de schémas associés. Les filtres sont “essayés” sur la grammaire indépendante du contexte servant de base à la spécification concernée.

Pour chaque production filtrée on génère les règles sémantiques correspondant aux schémas associés, selon des contraintes de définissabilité et de besoin. Il s'agit en fait de générer les calculs qui peuvent être effectués et de sélectionner ceux qui sont utiles dans le contexte.

L'interface entre les MAG se fait par l'intermédiaire d'attributs d'entrée et de sortie, ce qui peut induire une dépendance, voir une interdépendance de l'usage des MAG pour reconstituer une grammaire attribuée.

Le schéma de la figure III.4 représente le système MAGGIE, qui inclue la traduction de plusieurs MAG en une seule GA, ainsi que la génération de l'évaluateur correspondant.

III.3.3.1 Description

Une règle de MAG est une production particulière servant de filtre, à laquelle est accroché un ensemble de schémas de règles sémantiques.

Une production d'une grammaire indépendante du contexte ne contient que des symboles du vocabulaire de cette grammaire. Une production d'une MAG, qui sert de filtre, contient, elle :

- des variables qui peuvent représenter n'importe lequel des symboles du vocabulaire,
- des symboles quotés, qui représentent ces symboles particuliers,
- des points de suspension, représentant zéro ou plusieurs symboles du vocabulaire.

Les schémas associés à une production d'une MAG font référence aux attributs associés aux variables et aux symboles quotés de la production.

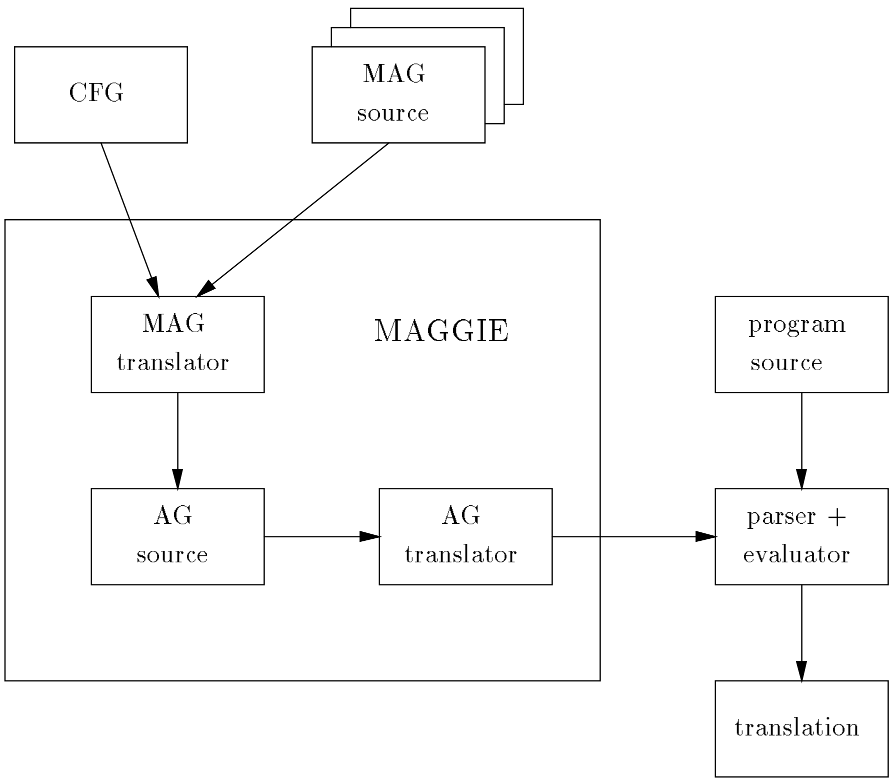


Figure III.4 : Le système MAGGIE.

Exemple :
Vocabulaire $V = \{A, B, C, D, E\}$ de la grammaire indépendante du contexte de base.
La règle d'une MAG :

$P \longrightarrow P\,'B \ldots Q$

$P.w = P_1.x + B.y + Q.z$

filtre :

$A \longrightarrow A\ B\ C\ D\ E$

et produit :

$A.w = A_1.x + B.y + E.z$

seulement si :

- $A_1.x$, $B.y$ et $E.z$ sont produits ailleurs
- $A.w$ est utile, c'est-à-dire utilisé ailleurs.

Les tests sur la forme normale des règles ne sont pas faits au moment de la traduction des MAG en grammaire attribuée, mais juste après, c'est-à-dire sur les règles produites.

Un exemple d'interdépendance de modules MAG concerne la définition et la propagation d'un environnement le long d'un arbre de dérivation, de haut en bas et de la gauche vers la droite.

Le premier module concerne la propagation de l'environnement vers le bas de l'arbre de dérivation. Il produit l'attribut de sortie (de type hérité) env, et utilise en entrée l'attribut def :

module env

$'goal \longrightarrow A \ \ldots$

$\quad A.env = 0;$

$A \longrightarrow B \ \ldots$

$\quad B.env = A.env;$

$A \longrightarrow \ldots \ B \ C \ \ldots$

$\quad C.env = B.def;$

Le second produit l'attribut de sortie (synthétisé) def, à partir de l'attribut d'entrée env. Il représente l'environnement que l'on modifie dans la passe ascendante du parcours de l'arbre :

module def

$A \longrightarrow \ldots \ B \ \ldots$

$\quad B.def = B.env;$

$A \longrightarrow \ldots \ B$

$\quad A.def = B.def;$

$A \longrightarrow$

$\quad A.def = A.env;$

Ce module def doit lors d'une application réelle, être enrichi de règles premettant de reconnaître les définitions d'identificateurs et de générer les définitions d'attributs qui modifient l'environnement.

Une technique communément utilisée dans la construction de compilateur est de transformer l'arbre de dérivation en un arbre de syntaxe abstraite.

On peut ici utiliser une MAG pour spécifier la transformation d'un arbre de syntaxe concrète en un arbre de syntaxe abstraite. L'évaluation des attributs pouvant ensuite être spécifiée dans une seconde MAG basée sur la grammaire abstraite.

Mais l'abstraction peut être obtenue par un autre moyen, qui est de regrouper dans une seule règle de MAG les productions issues d'un même type de nœud et dont les attributs se calculent de la même manière. Le filtre ne contient plus alors que des variables qui permettent de remplacer n'importe quel symbole du vocabulaire, ce qui permet d'obtenir cette abstraction. Les symboles quotés qui apparaissent dans les filtres déterminent l'attachement des attributs définis, à la syntaxe concrète.

On peut alors aller jusqu'à construire des modules suffisamment abstraits pour pouvoir être réutiliser dans d'autres applications, à partir d'un même ensemble d'attributs spécifiques, et en s'assurant que les symboles quotés qui peuvent subsister existent et correspondent aux mêmes notions dans ces applications.

III.3.3.2 Remarques

L'expérience montre que la programmation par utilisation de filtres n'est pas si facile, et nécessite toujours une vérification de la GA produite.

Le but de cette méthode est bien sûr d'arriver à ne programmer qu'au niveau des MAG et nécessite donc une expertise des domaines concernés afin de déterminer une méthodologie pour la création des modules MAG. La combinaison de plusieurs modules, seule, pourrait ainsi aboutir à la création d'une application complète.

L'existence de modules, comme ceux utilsés pour la propagation d'un environnement décrits précédemment, pouvant être utilisés dans beaucoup d'applications amène à la possibilité d'élaborer des modules génériques effectuant ces calculs.

Ces modules seraient paramétrés par des attributs et des symboles de production, instanciés lors d'une application, ce qui conduirait à la génération d'un module MAG, instance de ce module.

Mais cette approche n'a pas encore été développée. En revanche, l'accent a été mis sur la création de modules réutilisables, c'est-à-dire les plus abstraits possibles.

III.3.4 Le système LIGA

Le système LIGA [KaW92] est une partie du système de construction de compilateur Eli [GHL92]. Dans ce système est implanté un certain nombre de structures permettant d'étendre le concept de calcul de base des GA (tout en pouvant s'y ramener) et allant ainsi vers l'introduction de la modularité et la réutilisabilité.

Une particularité de ce système est qu'il introduit une vue du calcul réalisé par une GA pouvant comporter des effets de bord. Ainsi, les attributs sont considérés comme des post- et des pré-conditions aux calculs plutôt que comme des moyens de propager des valeurs. Les dépendences deviennent alors des moyens de synchroniser les calculs :

- chaque attribut représente une postcondition atteinte après l'exécution des calculs définissant cet attribut.

- les attributs utilisés dans un calcul représentent des préconditions pour effectuer ce calcul.

Ce concept plus général permet d'obtenir des implantations efficaces de la plupart des applications.

Les structures permettant d'obtenir une solution pour introduire la modularité et la réutilisabilité sont décrites dans les sections suivantes.

III.3.4.1 Accès aux attributs distants

Le calcul sous-jacent aux GA décrit précédemment impose le principe de localité dans les spécifications : toutes les préconditions et la postcondition d'un calcul donné doivent apparaître dans le contexte d'un unique nœud et de ses fils. Le localité des spécifications est censée favoriser le lisibilité d'une GA, mais une précondition d'un calcul dans le contexte local d'un nœud particulier peut parfois être atteinte à un nœud plus lointain dans l'arbre.

Ces dépendances lointaines peuvent être de trois sortes :

1. La précondition est établie à la racine du sous-arbre contenant le contexte local au calcul.
2. La précondition est l'union des postconditions d'une partie des calculs effectués sur des nœuds descendants dans le sous-arbre issu du contexte local.
3. La dépendance utilise un invariant pour un calcul itératif qui visite les nœuds selon une stratégie gauche-droite.

A ces trois cas de figure correspondent trois structures particulières.

Pour décrire le cas 1, voici un exemple de spécification du calcul de la profondeur d'encapsulement d'un bloc, utilisée pour former les adresses des variables déclarées dans le bloc.

```
RULE Program: Root ::= Block COMPUTE
        Block.Depth = 0;
END;
RULE Inner: Statement ::= Block COMPUTE
        Block.Depth = ADD(INCLUDING Block.Depth, 1);
END;
```

Le mot clef INCLUDING indique que la précondition est la première instance de l'attribut spécifié lorsqu'on remonte l'arbre à partir du nœud courant.

Le cas 2 est illustré par le problème de la détermination du coût de chaque instruction dans un programme. On suppose que chaque opérateur a un coût (longueur du code généré) et que le coût d'une instruction est la somme des coûts de ses opérateurs qu'elle contient.

```
RULE Calculation: Statement ::= Expression COMPUTE
        Statement.Cost = CONSTITUENTS Operator.Cost
                         WITH (int, ADD, IDENTICAL, ZERO);
END;
```

Le mot clef CONSTITUENTS indique que la précondition est l'union des instances de l'attribut *Operator.Cost* apparaissant dans le sous-arbre issu du nœud courant.

La clause WITH spécifie le type de la valeur de l'union (*int*) et nomme trois fonctions utilisées dans ce calcul : la première (*ADD*) combine deux valeurs d'union et produit une valeur d'union, la deuxième (*IDENTICAL*) crée une valeur d'union à partir d'une valeur d'un *Operator.Cost*, et la troisième (*ZERO*) crée une valeur d'union à partir de rien.

Le cas 3 est illustré par le calcul des valeurs des constantes énumérées en Pascal. Une énumération de constantes Pascal est définie par une liste d'identificateur et la valeur représentée par chaque identificateur est simplement le nombre d'identificateurs qui le précèdent.

Un invariant est décrit par un attribut spécial appelé “attribut chaîné”, ou simplement une chaîne. Sa valeur initiale est établie dans une clause CHAINSTART à la racine d'un sous-arbre. Les calculs qui mettent à jour cet invariant sont associés à chaque nœud de ce sous-arbre.

```
CHAIN Count: int;
RULE  Enumeration: Type ::= '(' EnumConstList ')' COMPUTE
      CHAINSTART EnumConstList.Count = 0;
END;
RULE  ConstListElt: EnumConst ::= Identifier COMPUTE
      EnumConst.Count = ADD(EnumConst.Count, 1);
END;
```

Ces invariants peuvent s'exprimer en terme de deux attributs par nœud : un hérité qui représente la précondition des calculs sur chaque nœud, et un synthétisé qui représente la postcondition établie par chaque calcul. Cahque nœud interne du sous-arbre a ainsi une paire d'attributs pour chaque invariant.

III.3.4.2 Calcul sur les symboles et héritage

Lorsqu'un calcul est liés à des attributs d'un seul symbole ou à des attributs distants et qu'il est effectué pour presque toutes les instances de ce symbole dans un arbre, on peut alors l'associer à ce symbole plutôt qu'à chaque production, ou contexte local, où il apparaît.

Ainsi, dans l'exemple de calcul de la profondeur d'imbrication d'un bloc, si l'on prend un langage comme ALGOL 60, on peut implanter le corps d'une procédure comme un bloc. On alors :

```
RULE Inner: Statement ::= Block COMPUTE
        Block.Depth = ADD(INCLUDING Block.Depth, 1);
END;
RULE Proc: Body ::= Block COMPUTE
        Block.Depth = ADD(INCLUDING Block.Depth, 1);
END;
```

Dans ce cas, le fait d'associer le calcul au symbole *Block* évite la redondance de la règle. Cela s'écrit ainsi :

```
SYMBOL Block COMPUTE
            INH.Depth = ADD(INCLUDING Block.Depth, 1);
END;
```

Le mot clef INH indique que *Depth* est un attribut hérité et que son calcul doit être associé avec chaque contexte contenant le symbole *Block* dans son membre droit. Inversement le calcul d'un attribut synthétisé *Block.a* noté SYNT.a sera associé à chaque contexte dont le membre gauche est *Block*.

Dans cet exemple, il ne faut pas oublier de donner explicitement une règle pour l'initialisation de *Depth* qui prend le pas sur celle introduite implicitement par l'application sur le symbole du membre droit *Block* :

```
RULE Program: Root ::= Block COMPUTE
        Block.Depth = 0;
END;
```

Le principe de l'héritage permet de décrire une fois un calcul pour un symbole qui n'appartient pas au langage que l'on veut décrire. Ce calcul est dit abstrait car il ne dépend pas du langage. Le symbole introduit symbolise en fait une ou plusieurs notions du langage qui se comportent de la manière pour le calcul qui lui est associé. Ces notions héritent alors du comportement décrit par le symbole introduit.

Dans l'exemple précédent, on peut introduire le symbole *Contour* :

```
SYMBOL Contour COMPUTE
        INH.Depth = ADD(INCLUDING Contour.Depth, 1);
END;
```

La description des calculs associés aux symboles *Block* et *Procedure* se fera alors par héritage de celui de *Contour*:

```
SYMBOL Block INHERITS Contour END;
SYMBOL Procedure INHERITS Contour END;
```

Ces calculs hérités peuvent bien entendu être masqués par des description de calculs sur les symboles ou dans les contextes.

On peut ainsi décrire des symboles de manière abstraite et les utiliser ou non dans son langage.

III.3.4.3 Attribution cumulée et modularité

L'attribution cumulée est simplement une technique qui permet de décomposer les spécifications d'une grammaire attribuée importante.

En effet, pour chacune des productions, on peut écrire un nombre arbitraire de règles sémantiques. Un contexte dans un arbre est défini par une unique production et est donc associé à l'union des calculs décrits pourcette production.

Il est possible de décomposer une spécification en composants qui couvrent chacun un aspect du problème global. Chacune de ces tâches peut être décrite par un nombre restreint de calculs sur un petit nombre de contextes, si l'on utilise la technique de l'accès aux attributs distants décrites précédemment. On peut donc également faire cette décomposition pour les calculs sur les symboles.

Une décomposition en modules d'attribution peut s'effectuer en stockant chaque composant dans un fichier distinct. A partir d'un ensemble de ces modules, on peut alors obtenir une spécification globale au sens habituel du calcul sous-jacent à la méthode des GA.

Certains de ces fichiers peuvent constituer une librairie de modules réutilisables. Ils ne sont alors constitués que de calculs sur les symboles et peuvent être instanciés sur un autre module par le mécanisme d'héritage.

III.3.4.4 Remarques

Ces idées constituent des outils permettant d'adapter cette solution d'apport de modularité et réutilisabilité dans n'importe quel système basé sur les GA. Ainsi, FNC–2 contient déjà toutes les structures qui permettraient d'aboutir à ce type de solution.

Le problème est que cela constitue plus une méthodologie dérivée d'une implantation du concept des GA qu'un formalisme mis au point à partir de la notion de modularité.

D'autre part, la sémantique décrite pour chaque symbole ne peut s'appliquer que sur une production donnée (qui contient ce symbole), ce qui implique que la structure du langage pour lequel on veut pouvoir hériter des calculs, soit assez proche de celle dont s'est inspiré l'écrivain du module.

De plus, l'application du mécanisme d'héritage n'est pas totalement claire. Par exemple, comment se comporte l'instanciation de symboles abstraits référencés dans les règles sémantiques associées à un symbole abstrait, lorsque ceux-ci sont hérités par plusieurs symboles dans une grammaire: comment choisir le bon ? ...

III.3.5 Les Grammaires Attribuées Composables

Les grammaires attribuées composables ou CAGs (Composable Attribute Grammars), sont un formalisme décrit par R. FARROW, T.J. MARLOWE et D.M. YELLIN dans [FMY92]. Il étend celui des grammaires attribuées pour permettre la composition modulaire de spécifications de traduction et de traducteurs.

L'optique est bien sûr d'obtenir les avantages de la modularité que sont entre autre la lisibilité, la réutilisabilité et l'incrémentalité du développement.

Une CAG est construite à partir de plusieurs petits composants, qui sont des grammaires attribuées chargées de résoudre des sous-problèmes particuliers.

On peut alors combiner plusieurs composants décrivant des problèmes divers et les utiliser sur un langage source. Ceci est fait dans une GA maîtresse nommée la *Glue*, par de simples règles sémantiques associées aux productions de la syntaxe du langage source.

La sémantique et la méthode d'évaluation d'une CAG sont décrites dans les sections III.3.5.3 et III.3.5.4 qui suivent.

III.3.5.1 Les GA composants

Un composant est basé sur une structure syntaxique simplifiée qui reflète les particularités liées à un sous-problème, plutôt que la syntaxe du langage source.

On peut associer des attributs d'entrée et de sortie aux symboles terminaux d'un composant.

Ceci permet de renvoyer le résultat de la résolution d'un sous-problème partout où l'on trouve une référence à ce sous-problème. Il s'agit en quelque sorte de paramètres importés ou exportés par la grammaire.

Mais voyons plutôt un exemple de GA-composant. La figure III.5 tirée de [FMY92] nous montre un composant chargé de traiter la construction et l'utilisation d'une table de symboles pour Pascal.

```
- - grammar symbols

envREF    : t   NAME: in: tp_NTX,      OBJ: out: entry,      MSGS: out: tp_msgs;
envDCL    : t   NAME: in: tp_NTX,      OBJ: in: entry;
envgoal   : n;
envitems  : n   ENVin: h: symbol_table, ENVout: s: symbol_table;

- - productions
env_goal   : envGoal  ::= envitems.
    envitems.ENVin = emptyENV;

env_empty: envitems ::= .
    envitems.ENVout = envitems.ENVin;

env_list    : envitems ::= envitems1 envitems2.
    envitems1.ENVin = envitems.ENVin;
    envitems2.ENVin = envitems1.ENVout;
    envitems.ENVout = envitems2.ENVout;

- - this production introduces a new scope
env_nest   : envitems ::= envitems1.
    envitems1.ENVin = envitems.ENVin;
    envitems.ENVout = envitems.ENVin;

- - this introduces a variable reference
env_ref    : envitems ::= envREF.
    envREF.OBJ = envLookup(envitems.ENVin, envREF.NAME),
    envREF.MSGS = if envREF.OBJ =nullObj then "invalid reference" else "" fi,
    envitems.ENVout = envitems.ENVin;

- - this introduces a variable declaration
env_dcl    : envitems ::= envDCL.
    envitems.ENVout = envUpdate(envitems.ENVin, envDCL.NAME, envDCL.OBJ);
```

Figure III.5 : Composant pour une table de symboles.

On peut faire alors trois remarques. La première est que la syntaxe utlisée pour représenter ce problème est très simple, puisqu'elle ne comporte que trois entités qui sont les déclarations, l'imbrication de blocs et la référence à un identificateur. La deuxième remarque porte sur le fait que la sémantique de ce composant peut correspondre à d'autres langages que Pascal. Et enfin, il est clair que ce type de syntaxe peut être réutilisé pour résoudre d'autres problèmes basés sur les mêmes entités.

On peut bien entendu envisager d'exprimer bon nombre de ce type de sous-problèmes à l'aide de ces composants réutilisables, et construire ainsi une librairie permettant de spécifier un compilateur pour un langage donné.

Ces applications de composants se font à l'aide d'une grammaire attribuée particulière nommées la Glue.

III.3.5.2 La Glue

Les syntaxes des composants sont en général plus ou moins éloignés de la syntaxe du langage source, en tous cas plus simples, il faut donc introduire un moyen de relier les structures de ce langage avec les structures correspondantes des différents composants utilisés.

De plus, il faut spécifier explicitement les liens entre les composants que sont les informations de sortie des uns qui sont utilisées comme informations d'entrée des autres.

Pour faire ce travail, la solution choisie a été celle de grouper ces actions dans une grammaire attribuée maîtresse, appelée Glue car elle colle les différents composants pour obtenir une spécification de sémantique unique relative au langage décrit par la structure syntaxique sous-jacente.

La figure III.6 nous donne un extrait de la Glue dans laquelle on veut utiliser le composant décrit précédemment figure III.5 pour une production du langage Pascal.

```
asgn_stmt ::= variable ASGN expression  - - Pascal grammar production

- - rules for constructing the phrase structure for id-resolution
      asgn_stmt.envitems = env_list(env_ref(REF),expression.envitems),
      REF:envREF = {NAME ⇒ variable.name},

      asgn_stmt.MSGS = concat(REF.MSGS, expression.MSGS);
```

Figure III.6 : Extrait de la Glue.

Il s'agit ici de la production décrivant l'instruction d'affectation en Pascal. Du composant de construction de table de symbole, on veut utiliser la sémantique chargée de gérer la référence à une variable et la mise à jour de la table en fonction des variables de l'expression.

Les règles sémantiques de la production de la Glue spécifient les structures syntaxiques du composant correspondantes, ainsi que les valeurs des attributs d'entrée des symboles terminaux de ces structures.

Pour obtenir les structures syntaxiques de composant, on définit des attributs syntaxiques du type des non-terminaux du composant, dont la valeur est déterminée à l'aide de constructeurs de productions.

Ainsi, dans l'exemple de la figure III.6, le non-terminal *asgn_stmt* a un attribut *envitems* correspondant au non-terminal du même nom du composant.

Le constructeur *env_list* construit la production de ce nom dans le composant. La partie gauche de cette production est constituée du non-terminal *envitems* (type du résultat de cette "fonction") et la partie droite, de deux non-terminaux du type *envitems* (type des 'paramètres).

Le premier de ces deux non-terminaux est produit, dans la glue, par l'application du constructeur *env_ref* du composant, prenant en unique paramètre un élément du type du non-terminal en partie droite, c'est-à-dire *envREF*. Ce paramètre, *REF*, est un symbole terminal introduit par une déclaration locale. Il est déclaré de type *envREF* et le seul attribut d'entrée qui lui est associé, *name*, est défini par la valeur de l'attribut *variable.name* de la Glue.

Ici, un attribut d'entrée de composant est défini dans la Glue à partir d'un attribut existant, mais il aurait pu tout aussi bien dépendre d'un attribut de sortie d'un autre composant.

Ceci montre bien que les interfaces entre les différents composants pouvant être utilisés pour définir une CAG se définissent dans les règles sémantiques de la grammaire Glue.

Les composants ne pouvant interagir que via cet intermédiaire, ils n'ont pas à connaître le contenu de ceux avec qui ils sont en relation.

Il est donc possible de modifier chaque composant indépendamment des autres, et ce sans forcément toucher à la Glue, dans le cas où les interfaces restent les mêmes.

III.3.5.3 La sémantique d'une CAG

La sémantique d'une CAG, constituée de ces composants et de la Glue, est donnée par le définition d'une transformation en une GA monolithique (schéma III.7). Si cette GA est circulaire, alors la CAG n'a pas une sémantique bien définie. Sinon la sémantique de la CAG est celle de la GA monolithique.

Cette transformation s'effectue en utilisant la méthode de la méta-composition décrite précédemment dans la section II.2.2.

Chaque attribut syntaxique de la Glue correspondant à un non-terminal X d'un composant est remplacé par de nouveaux attributs induits correspondants aux attributs de X dans le composant.

Le mécanisme de méta-composition est étendu dans la mesure où les sémantiques des composants sont projetées simultanément dans la Glue.

De plus, les règles permettant le calcul des attributs d'entrée et de sortie sont ajoutées.

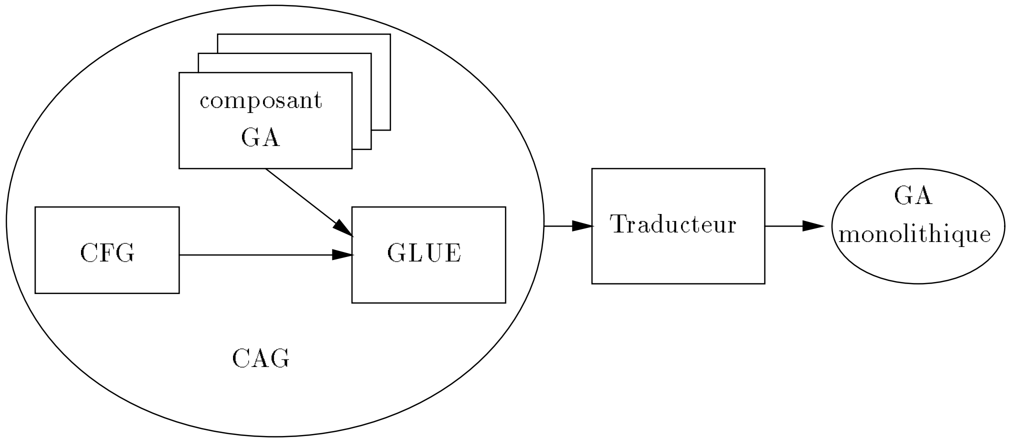


Figure III.7 : Schéma de transformation d'une CAG.

La figure III.8 nous montre le résultat de la méta-composition pour la production décrivant l'instruction d'affectation en Pascal.

```
asgn_stmt ::= variable ASGN expression  - - Pascal grammar production

      envitem1ENVin : symbol_table = asgn_stmt.ENVin,
      envitem1ENVout : symbol_table = envitem1ENVin,
      expression.ENVin = envitem1ENVout,
      asgn_stmt.ENVout = expression.ENVout,
      REFname : tp_NTX = variable.name,
      REFobj : entry = envLoolup(envitem1ENVin, REFname),
      REFmsgs : MSGS = if REFobj = nullObj then "invalid reference" else "" fi,
      asgn_stmt.MSGS = concat(REFmsgs, expression.MSGS);
```

Figure III.8 : Extrait du résultat de la méta-composition.

A la place de l'attribut syntaxique *envitems* du non-terminal *asgn_stmt*, la transformation lui induit deux nouveaux attributs *ENVin* et *ENVout*, qui sont les attributs de *envitems* dans le composant. La même chose pour le non-terminal *expression*.

Quand les règles sémantiques associées aux constructeurs de productions *env_list* et *env_ref* ont été introduites dans la grammaire transformée, il a fallu définir des variables locales pour jouer le rôle qu'avaient les attributs d'entrée et de sortie du composant dans la grammaire Glue initiale. Ainsi les variables locales *REFname*, *REFobj* et *REFmsgs* remplacent les attributs *REF.NAME*, *REF.OBJ* et *REF.MSGS*.

La composition de constructeurs de productions peut aussi introduire des variables locales remplaçant des attributs associés à des non-terminaux implicites. C'est le cas des variables *envitem1ENVin* et *envitem1ENVout* à la place des attributs *envitems1.ENVin* et *envitems1.ENVout*.

III.3.5.4 Méthode d'évaluation

Une des stratégie d'évaluation envisagée pour les CAGs est donc la transformation en une GA monolithique par la méthode de la méta-composition, sur laquelle on applique une évaluation classique.

Mais cette solution présente des inconvénients qui ont amenés les inventeurs des CAGS à envisager une autre solution.

En effet, appliquer la méta-composition et ensuite construire un évaluateur de la grammaire produite ne permet pas de faire d'analyses séparées ni d'appliquer le principe bien connu de compilation séparée.

De plus, on aimerait être capable de raisonner uniquement sur les composants et la Glue individuellement pour en déduire si la CAG est bien formée ou non.

Or si les composants et la Glue sont non circulaires, il n'est pas prouvé que la CAG le soit aussi.

Il existe néanmoins une classe de CAGs pour lesquelles l'analyse de ses composants et de leur Glue permet de savoir qu'elles ne sont pas circulaires. Il s'agit des CAG séparables.

Définition III.3.1 : CAG séparable

Une CAG G est séparable si

- *chaque grammaire attribuée composant est non circulaire.*
- *la Glue est non circulaire même si l'on considère que chaque paramètre de sortie dépend de chaque paramètre d'entrée dans n'importe quel arbre de composant construit dans la Glue.*

♣

La grammaire monolithique induite de ce type de grammaire est alors forcément non circulaire.

En effet, l'analyse de la Glue transformée par l'ajout de dépendances, prend forcément en compte n'importe quelle dépendance entrant en jeu réellement. Si, malgré cette contrainte, la Glue est toujours non circulaire, la CAG le sera donc aussi.

Si une CAG est séparable, on peut alors tester chaque composant et la Glue séparément, ainsi que construire des évaluateurs séparés.

La stratégie d'évaluation des CAGs séparables est donc une stratégie statique d'évaluation modulaire, basée sur la stratégie d'évaluation ordonnée des grammaires attribuées classiques.

III.3.5.5 Remarques

Le fait d'utiliser une Glue à écrire soi-même pour instancier un module laisse à la fois baucoup de liberté, mais aussi beaucoup de responsabilité à l'utilisateur. Il doit bien connaître le contenu des composants et savoir où et comment utiliser la sémantique qu'ils contiennent. Cela ne permet pas de fonctionner avec des modules type "boîte noire" dont on connaît l'interface, mais pas la manière dont les calculs sont effectués à l'intérieur.

De plus, la méthode d'evaluation implantée dans le système concerne une sous-classe de grammaires attribuées composables. La modularité ainsi atteinte est importante mais n'est implantée dans leur système que pour une petite partie des CAGs. Reste à savoir si le pouvoir d'expression est suffisant pour décrire la plupart des problèmes que l'on aimerait pouvoir résoudre ainsi.

Le fait d'utiliser la méta-composition pour décrire la sémantique d'une CAG, puis de ne pas l'utiliser pour l'évaluation car il ne permet pas d'aboutir à la compilation séparée est quelque peu paradoxal.

En effet, on peut se demander si le fait de descendre aussi bas est intéressant pour la méthode des GA. Le grand principe est de décrire ce qu'on veut calculer sans s'occuper de comment on le calcule: l'introduction de la modularité ne devrait donc porter que sur le côté descriptif et rester au niveau "méta".

III.4 Notre approche

Notre approche des problèmes posés par l'utilisation des GA en tant qu'outil de spécification a été plus ou moins inspirée par les inconvénients rencontrés en utilisant du système FNC–2. Il constitue un outil de choix pour expérimenter les améliorations que l'on peut apporter à l'utilisation des GA.

Comme nous l'avons vu dans la section II.3, ce système, en l'occurrence son langage d'entrée OLGA, contient un certain nombre d'amélioration concernant le problème de la taille.

Ainsi, OLGA dispose d'un générateur de règles de copie qui permet d'éviter d'écrire presque toutes les règles sémantiques qui ne font qu'un transfert d'information. Ce service est déclenché dés qu'il manque la définition d'une occurrence d'attribut dans une production: sa valeur est alors remontée, ou descendue selon qu'il s'agit d'un attribut synthétisé ou hérité.

Mais il manquait, pour aller encore plus loin dans la réduction du nombre de règles à écrire dans une spécification, un service comparable à ce que proposait M. D. TIEMANN, section III.2.2. C'est pourquoi la première modification apportée à OLGA concerne l'ajout de la possibilité de spécifier des règles par défaut pour une classe d'attributs qui ont le même comportement. Les occurrences d'attributs sans définitions dans une production correspondent alors soit à une régle par défaut, soit à une règle de copie.

L'extension d'OLGA, OLGA++, ainsi obtenue contient un service de génération de règles sémantiques manquantes complet. Ce sur-langage et son fonctionnement sont décrits plus en détail dans le chapitre IV.

A partir de cette approche, il est tentant d'élargir le concept pour parvenir à développer la modularité et la réutilisabilité dans la méthode des grammaires attribuées.

En effet, si l'on considère OLGA++ non plus comme un sur-langage d'OLGA, mais comme un langage de spécification de modules par défaut extérieurs à un module OLGA de spécification de GA, cela nous ramène à une solution similaire à celle de G.D.P. DUECK et G.V. CORMACK décrite section III.3.3, avec leur système MAGGIE.

Les modules décrits sont alors des ensembles de règles associées à des filtres, groupés pour une classe d'attributs donnée. L'instanciation se fait ici par l'association d'un ensemble d'attributs à chaque classe.

De même, on aurait pu choisir de réaliser l'évolution décrite par U. KASTENS et W. M. WAITE pour le système LIGA.

En effet, on peut établir un certain nombre de correspondances dans les notions utilisées pour résoudre certains problèmes :

- les structures INCLUDING, CONSTITUENTS et CHAIN qui sont définies pour effectuer l'accès aux attributs distants correspondent à plusieurs notions existant dans OLGA, notamment les attributs globaux, les attributs mixtes et leur comportement par défaut avec la génération de règles sémantiques manquantes.

- Le calcul sur les symboles et l'héritage sont des notions qui transparaissent dans l'idée de classe d'attributs développée dans OLGA++;

- L'attribution cumulée est déjà présente par le fait que l'on peut décrire autant de passes dans une GA qu'il y a de calculs à effectuer sur une même syntaxe.

Ces deux visions amènent bien l'obtention de la modularité et même de la réutilisabilité, mais elle n'est pas complètement satisfaisante pour nous dans la mesure où l'on aimerait parler de généricité. La nuance consiste dans le fait qu'un module réutilisable doit s'appliquer tel quel, moyennant le renommage éventuel qui constitue ici toute l'instanciation. L'instanciation d'un module générique peut être, plus qu'un simple renommage, une particularisation d'un cas décrit de manière générale.

L'approche de C. KLEINHANS permet d'obtenir la modularité dans la structuration d'une application : elle se rapproche en cela des modules LIGA, puisqu'elle aussi permet d'oublier une partie de la syntaxe complète. Mais elle est encore loin de pouvoir permettre une réelle réutilisation de ces modules, trop proches de la structure syntaxique, même en effectuant des renommages.

Quant à l'approche de R. Farrow, T.J. Marlowe et D.M. Yellin avec les Grammaires Attribuées Composables (CAGs), c'est celle qui nous paraît la plus satisfaisante car la vision décrite est plus celle de la généricité que celle de la modularité alliée à la réutilisabilité.

Les remarques faites dans la section III.3.5.5 sont les seuls reproches que nous faisons à cette méthode développée dans le même temps que nous.

Notre approche est donc plus générale et se situe donc plus sur un plan "méta" du génie logiciel pour les GA. C'est ainsi qu'on aboutit à la notion de Grammaire Attribuée Générique (GAG).

Une GAG est une grammaire attribuée qui a la particularité de ne contenir que les éléments essentiels à la résolution d'un problème particulier. Ainsi, un programmeur qui voudrait décrire un algorithme général pour un problème devrait choisir une syntaxe suffisamment abstraite pour se réduire aux seules structures qui ont une réelle utilité dans l'algorithme.

Le but de l'opération est d'instancier une GAG sur une grammaire G pour aboutir à une GA basée sur G et contenant en plus la sémantique de la GAG. Il ne s'agit pas pour nous de produire un évaluateur, mais bien une GA que l'on peut éventuellement retoucher ou compléter. On se place donc simplement au niveau de l'aide à l'écriture d'une GA, ce qui explique que notre méthode essaie de conserver les propriétés que l'on peut avoir pour les GA mises en cause.

Une grammaire attribuée est une spécification déclarative des dépendances entre les calculs attachés aux nœuds d'un arbre.

L'idée d'instanciation d'une GAG sur une grammaire G donnée découle directement de ce concept. Il suffit en effet d'imaginer que pour un arbre de G on veuille appliquer les calculs décrits par une GAG. Il s'agit alors de trouver l'arbre de la GAG qui lui correspond, effectuer les calculs sur cet arbre, puis récupérer ces calculs dans l'arbre de G. L'idée est simple, et les seules choses à préciser sont les points stratégiques où l'on veut récupérer les calculs effectués. Cette précision équivaut à donner les types de nœud de G que l'on peut associer à des nœuds d'un arbre de la GAG.

La sémantique de notre instanciation correspond exactement à ce que fait la méta-composition : appliquer les calculs d'une grammaire sur une autre. C'est pourquoi cette méthode va constituer notre principal outil pour réaliser une instanciation d'une GAG sur une grammaire G.

La description de la façon dont on veut instancier une GAG se fait en établissant des correspondances entre les notions importantes d'un problème, décrites par les symboles de la grammaire de la GAG, et les symboles équivalents contenus dans la grammaire G.

Pour savoir si une GAG peut être instanciée, il suffit de savoir si la description des associations entre les symboles de la GAG et ceux de G permet de définir une fonction de couplage qui associe à chaque arbre de la grammaire G un arbre de la grammaire de la GAG.

Notre stratégie va donc se limiter à définir et construire une fonction de couplage α qui associe à chaque arbre de la grammaire G un arbre de la grammaire de la GAG β, et le résultat $\gamma = \alpha \circ \beta$ correspondra bien à notre idée de l'instanciation.

Il est clair que définir une fonction de couplage automatiquement restreint les chances d'instanciation par rapport à une définition manuelle comme celle effectuée dans une Glue pour les CAGs. Mais ce choix nous permet de rester toujours dans la lignée du descriptif, du déclaratif de la théorie des GA.

Le chapitre V décrit en détail ce mécanisme d'instanciation d'une Grammaire Attribuée Générique en l'appliquant pour le cas simple où la GAG et la grammaire G sont deux grammaires attribuées classiques. Il concerne principalement la définition et la construction automatique de la fonction de couplage.

Le chapitre VI étend le concept à ce qu'on appelle des Grammaires Attribuées Ouvertes (GAO) interfaçables à travers des attributs d'entrée et de sortie, qui constituent les ouvertures avec l'extérieur par lesquelles elle peut communiquer avec d'autres GAO. Il faut alors étudier l'extension de la méthode à l'instanciation de plusieurs GAG qui peuvent être dépendantes entre elles, sur une grammaire qui, elle aussi, peut être une GAO.

Le chapitre VII est une extension de la méthode de construction de la fonction de couplage qui permet d'augmenter les chances de définir cette fonction. On augmente ainsi la puissance d'expression jusqu'au degré qui nous paraît le plus intéressant par rapport à une définition manuelle.

Chapitre IV

Extension d'Olga : spécification des règles sémantiques manquantes

Malgré l'existence d'un service de génération de règles de copie dans FNC–2, la taille des grammaires écrites en OLGA était toujours trop imposante et le nombre de règles identiques ou presque identiques trop important.

Nous avons donc entrepris d'élaborer un outil permettant de spécifier ce type de règles, et de les générer lorsqu'elles sont absentes. Nous avons en cela repris l'idée de M. D. Tiemann décrite sommairement dans la section III.2.2, page 41.

En effet, l'expérience de programmation en OLGA montre qu'il reste encore un trop grand nombre de règles autres que les règles de transfert. Or, nombreuses sont celles qui sont construites suivant un modèle similaire pour un même attribut dans différentes productions. Ces règles répétitives prennent une place "inutile" dans la grammaire, sans compter l'ennui, pour l'utilisateur, d'avoir à les recopier (même avec un éditeur de texte performant!).

Ainsi, considérons un attribut synthétisé $\$a$ d'un opérateur de liste op_1, défini par la production $op_1 \rightarrow PH_1*$, et le même attribut $\$a$ défini sur un autre opérateur de liste op_2, décrit par la production $op_2 \rightarrow PH_2*$.

Si la règle sémantique définissant l'attribut $\$a$ dans le premier cas a la forme:

> **map left** f
>   **value** $init$
>   **other** $\$a(PH_1)$
> **end map**;

où f est une fonction et $init$ une valeur initiale servant de valeur par défaut en cas d'arité nulle, et dans le second cas:

> **map left** f

value *init*
other $\$a(PH_2)$
end map;

il serait plus judicieux de pouvoir écrire une seule règle de la forme:

map left *f*
value *init*
other $\$a(PH)$
end map;

où PH représenterait alors soit PH_1, soit PH_2, selon l'opérateur.

L'utilisateur peut de cette façon définir ses propres règles implicites.

Cela nécessite bien sûr de munir OLGA d'une structure supplémentaire permettant de spécifier ces règles sémantiques et, à partir de celles-ci, de générer les règles manquantes.

Les choix effectués ainsi qu'une description générale du principe adapté à OLGA sont décrits dans la section IV.1. La section IV.2 décrit le mécanisme plus en détail.

IV.1 Présentation

Notre mécanisme est basé sur l'idée de M. D.Tiemann [Tie87]. Elle a du être adaptée au langage OLGA et à ses particularités. OLGA étant basé sur une syntaxe abstraite, un certain nombre de constructions spécifiques ont été introduites, c'est pourquoi un *filtrage par la syntaxe* s'avère nécessaire, comme nous le verrons plus loin, en plus d'un filtrage par la sémantique.

La notion de *classe* décrite dans [Tie87] a gardé tout son intérêt, puisqu'elle permet de regrouper des attributs ayant le même comportement implicite sur un certain nombre de règles.

Dans OLGA, il est possible de définir dans une production des *attributs locaux* qui pourront être utilisés dans les règles sémantiques. Or, il n'est pas rare de vouloir utiliser un même type d'attribut local dans différentes productions, d'où l'utilité de pouvoir le définir une seule fois implicitement pour toutes celles dans lesquelles il est utile.

Ceci nous amène à l'introduction du concept nouveau d'*attribut local par défaut*, qui précéde la description de notre vision de la *classe* d'attributs.

IV.1.1 Le concept d'attribut local par défaut

Un *attribut local par défaut*, comme son nom l'indique, se comporte comme un attribut local, à la différence près que celui-ci peut être défini implicitement pour plusieurs productions.

Il est déclaré par un nom, un type et un ensemble de phyla correspondant aux opérateurs des productions dans lesquelles il joue le rôle d'un attribut local.

Ce concept permet ainsi de rendre plus lisible une spécification. En effet, définir des attributs locaux communs à certaines productions va permettre d'introduire une méthode plus systématique de programmation en GA.

Par exemple, dans le domaine de la compilation, et notamment dans une GA effectuant une vérification de types, on pourra introduire systématiquement un attribut local implicite *correctid* déclaré pour toutes les productions de la grammaire ayant un identificateur en membre droit, et chargé de rendre une valeur booléenne indiquant la correction de son usage.

Une fois cet attribut déclaré, il pourra servir dans les règles sémantiques de toutes les productions concernées.

Un attribut local *déclaré* par défaut peut, ou non, être *défini* dans cette partie.

Dans le premier cas, s'il apparaît dans une règle d'une production où il n'est pas redéfini localement, et si la production appartient à l'ensemble de celles pour lesquelles l'attribut est déclaré, la définition par défaut est appliquée pour générer la définition adéquate.

Dans le second cas, il appartient à l'utilisateur de le définir pour chacune des productions où il sera utilisé.

IV.1.2 Le concept de classe

Une *classe* est conçue ici comme une entité chargée de jouer le rôle des attributs qu'elle représente dans les règles sémantiques des productions sur lesquelles elle est déclarée. Les attributs représentés doivent avoir la même "direction" (*hérité,synthétisé, composé, global*).

On déclare une *classe* en précisant son nom ($\$nom$ différent de tous les autres noms d'attributs et de classes), les attributs qu'elle regroupe, c'est-à-dire leurs noms et les phyla auxquels ils peuvent être attachés, et éventuellement, son type.

Celui-ci peut être, soit celui de tous les attributs, soit indéfini, si les attributs ne sont pas du même type.

Une classe *déclarée* par défaut doit obligatoirement être *définie* dans la spécification des règles implicites.

Une fois cette classe déclarée, elle ne pourra servir que dans les définitions de classes (y compris la sienne).

L'utilisation d'une classe se fait de façon non ambigüe: elle ne peut prendre la place que d'un seul attribut à la fois dans un contexte donné.

IV.1.3 Le filtrage

Lorsque, dans une production, un attribut n'a pas de définition sémantique, il faut alors regarder si l'on ne peut pas en construire une à partir de modèles de règles par défaut. Il faut alors déterminer la classe à laquelle appartient l'attribut, et regarder, parmi les règles proposées, quelle est celle qui convient dans ce cas.

Il est donc nécessaire de pouvoir choisir, et ce choix passe par le filtrage d'une production par une série de descriptions formelles. Vient ensuite le filtrage du contexte à l'aide d'un ensemble de modèles par défaut pouvant s'appliquer pour la description retenue.

Le filtrage par la syntaxe

Il s'agit ici de vérifier que la définition de l'opérateur représentant la production dans laquelle on veut ajouter une règle manquante, est conforme à une description syntaxique donnée.

Cette description est associée à un ensemble de modèles par défaut pouvant s'appliquer pour définir des attributs dans ce contexte.

L'opérateur peut être décrit par une liste comportant des noms d'opérateurs auxquels il peut être comparé, des phyla pouvant le contenir, et des variables symbolisant les opérateurs des phyla auxquels elles se restreignent, et qui peuvent également le contenir.

La partie droite de la production, elle, est décrite uniquement par des variables, représentant un certain nombre de phyla que l'on peut préciser ou non.

Ainsi, il s'agit de comparer les parties gauche et droite de la description syntaxique avec les membres gauche et droit de la production.

Dans le cas où la description peut effectivement s'appliquer à la production, il faut alors s'assurer qu'un (ou plusieurs) des modèles de règles implicites associés est bien applicable dans le contexte correspondant. C'est ici qu'intervient le filtrage par la sémantique.

Dans le cas contraire, on compare la production avec la description suivante.

Le filtrage par la sémantique

Le filtrage par la sémantique est en fait une vérification que la règle par défaut est bien celle dont on a besoin: il faut simplement vérifier que les occurrences d'attributs, de classes ou d'attributs locaux que l'on peut trouver ont un sens dans le contexte, c'est-à-dire pour la production en cause.

Si le modèle ne convient pas, on effectue les mêmes vérifications sur le suivant dans la liste associée à la description filtrant la production.

Si aucun des modèles n'est applicable, il faut alors revenir au niveau supérieur en essayant de filtrer la production par une autre description syntaxique de la classe.

IV.2 Manuel de référence du sur-langage de spécification

Ce langage est fait pour s'insérer dans une phase de grammaire attribuée.

Dans un module de GA, la place d'une spécification de règles implicites doit être après une partie déclaration d'attributs, ceux-ci servant de base à la construction de classes, et avant une partie définition de règles sémantiques, celles-ci pouvant utiliser les attributs locaux par défaut.

IV.2.1 Description syntaxique

La description de la syntaxe concrète est donnée sous sa forme BNF complète en annexe, partie B, page 190.

Il y a deux parties principales: une partie déclaration dans laquelle on doit déclarer les attributs locaux et les classes, et une partie définition dans laquelle sont regroupées les définitions de règles par défaut associées à chaque classe ou attribut local. Ce qui donne:

<*default part*> = "**defaults**" <*dec-part*> <*def-part*>;

IV.2.1.1 La partie déclaration

On peut y faire deux sortes de déclarations: celles des attributs locaux et celles des classes. Ces parties peuvent s'intercaler et ce, autant de fois que l'on veut.

Voici la syntaxe concrète représentant cette partie:

<*dec-part*> = <*dec-part*> <*dec-clause*> ;
<*dec-part*> = <*dec-clause*> ;
<*dec-clause*> = <*local-dec*>;
<*dec-clause*> = <*class-dec*>;

Les attributs locaux

La syntaxe de la partie déclaration d'attributs locaux se présente de cette manière:

<*local-dec*> = "**local**" <*local-list*> ;
<*local-dec*> = ;
<*local-list*> = <*local-list*> <*local-dc*>;

<local-list> = *<local-dc>*;
<local-dc> = *<id>* "(" *<phyla∗>* ")" ":" *<type id>* ";";

L'identificateur *<id>* est le nom de l'attribut et son type *<type id>*, un des types prédéfinis ou déclarés.

La liste de phyla *<phyla∗>* contient les phyla regroupant les opérateurs représentant les productions pour lesquelles l'attribut local est utilisable. Cette liste peut être vide, et dans ce cas, c'est comme si elle représentait tous les phyla de l'arbre abstrait.

Les variantes possibles de déclaration d'attribut local par défaut sont donc :

local

nom1 (liste-phyla) : type1 ;

nom2 () : type2 ;

...

Les classes

La partie déclaration de classes se présente de cette manière:

<class-dec> = "**classes**" *<class-list>*;
<class-dec> = ;
<class-list> = *<class-list>* *<class-dc>* ;
<class-list> = *<class-dc>* ;
<class-dc> = *<attrib name>* "(" *<attrib-list>* ")" *<type-restrict>* ";" ;

Les noms de classes commencent par un $, comme ceux des attributs, et ils doivent être uniques.

Indiquer le type de la classe (*<type-restrict>*), n'est pas obligatoire. S'il est précisé, ce type doit aussi être celui des attributs représentés par la classe, et s'il n'est pas précisé, alors, dans le cas où tous les attributs sont du même type, ce type est aussi le sien, et sinon, elle est de type indéfini.

Ce dernier cas n'est gênant que si les fonctions utilisées dans les règles sémantiques associées à la classe ne sont pas des fonctions polymorphiques (du type *identité* par exemple).

La liste *<attrib-list>* contient les attributs représentés par la classe: elle ne peut en aucun cas être vide. Chaque élément de cette liste est en fait une liste de noms d'attributs suivie d'une liste de phyla. La syntaxe de *<attrib-list>* se symbolise ainsi:

<attrib-list> = *<attrib-list>* "," *<attrib-elem>* ;
<attrib-list> = *<attrib-elem>* ;

<attrib-elem> = *<attr+>* *<phyl-restrict>* ;

Chaque liste *<phyl-restrict>* associée à un ensemble d'attributs représente les phyla auxquels on peut trouver rattachés les attributs lorsqu'ils sont remplacés par la classe.

Cette liste peut être vide, et remplace alors tous les phyla sur lesquels on a défini les attributs.

L'ensemble de toutes les listes *<phyl-restrict>* constitue une liste globale qui, comme pour les attributs locaux, contient les phyla contenant les opérateurs associés aux productions dans lesquelles la classe a une signification. Cette liste, si elle est vide, représente alors tous les phyla connus à cet endroit et correspondant aux attributs.

Voici quelques exemples de déclaration de classe:

classes
$\$C1$ ($\$a$, $\$b$ () ; $\$c$ (PH_1)) : $type1$;
$\$C2$ ($\$c$ (PH_2, PH_3));
$\$C3$ ($\$d$ (); $\$e$ (PH_1, PH_3));
$\$C4$ ($\$f$, $\$g$, $\$h$) : $type4$;
...

La classe $\$C1$ regroupe trois attributs de même type $type1$: $\$a$ et $\$b$ sont restreints à leurs phyla de déclaration (qui sont les mêmes pour l'un comme pour l'autre), et $\$c$ est, lui, restreint à un seul phylum PH_1. L'ensemble des phyla associés à la classe comprend donc l'ensemble des phyla de $\$a$ et $\$b$, plus, s'il n'y est pas déjà, PH_1.

La classe $\$C2$ représente l'attribut $\$c$ de type $type1$: elle est donc de ce même type. De plus, l'ensemble des phyla qui lui est associé est celui de $\$c$: (PH_1, PH_2).

La classe $\$C3$ se décrit de la même façon que $\$C1$, mais ici, les attributs $\$d$ et $\$e$ peuvent être de types différents, auquel cas son type est indéfini. Dans le cas contraire, elle est du même type que les attributs.

La description de la classe $\$C4$ implique qu'elle et ses attributs sont de type $type4$ et que $\$f, \$g$ et $\$h$ sont définis sur le même ensemble de phyla, qui est aussi celui associé à la classe.

IV.2.1.2 La partie définition

Dans cette partie sont définis les attributs locaux et les classes dans un ordre quelconque.

Toutes les classes déclarées doivent être obligatoirement définies, mais, comme nous l'avons déjà vu, les attributs locaux par défaut peuvent ne pas être déclarés à cet endroit, mais dans les productions dans lesquelles ils sont utilisés, de la même façon que les attributs locaux traditionnels.

À chaque définition est associée une liste de *descriptions syntaxiques* de productions, et à chacune d'entre elles correspond une liste de modèles de règles sémantiques.Une définition d'attribut local ou de classe de nom *nom* se présente ainsi:

default *nom* **use**
  where *liste-op* → *liste-var* **use** *liste-reg* **end where** ;
  where *liste-op* → *liste-var* **use** *liste-reg* **end where** ;
  ...
end default;

La description d'une production

La partie *liste-op* → *liste-var* représente la *description* par défaut d'une production, c'est-à-dire, en quelque sorte, son "filtre" syntaxique.

La liste *liste-op* est une liste d'éléments permettant de filtrer l'**opérateur** d'une production.

Un élément de cette liste est:

- soit un opérateur,
- soit un phylum,
- soit une variable qui représente les opérateurs des phyla auxquels elle est restreinte.

  Exemple: $op : PH_1 \mid PH_2 \ldots$
  où *op* est la variable représentant l'un des opérateurs des phyla PH_1 et PH_2 dans les règles implicites qui suivent.

  Il va sans dire que ces phyla ne sont qu'une seconde restriction, la classe ou l'attribut local que l'on définit étant déjà limité(e) à un certain nombre de productions.

  Ceux-ci doivent donc impérativement se trouver dans la liste préalablement donnée dans la déclaration correspondante.

  Si la variable n'est pas restreinte à de nouveaux phyla, ce sont ceux de la première restriction qui sont pris en compte.
- soit l'élément *null*, pouvant, comme une variable se restreindre à un certain nombre de phyla parmi ceux de la classe ou de l'attribut local.

  Cet élément représente alors les *opérateurs nullaires prédéfinis* des phyla.

Exemple: *null* (PH_1, PH_2)

représentant les opérateurs $null - PH_1$ ou $null - PH_2$.

Pour ce qui est de la liste *liste-var*, qui permet de filtrer le **membre droit** d'une production, trois cas peuvent se présenter:

1. cas des opérateurs nullaires prédéfinis:

   Quant la description a pour partie gauche un opérateur de type *null*, il n'y a pas de liste *liste-var*.

2. cas des opérateurs d'arité fixe:

   La liste *liste-var* est une liste d'objets servant à filtrer les phyla du membre droit d'une production. Ces objets sont:

   - soit une variable qui représente les phyla auxquels elle est restreinte.

     Exemple:

     – X où X est une variable pouvant représenter n'importe quel phylum.

     – $X : PH_1 \mid PH_2 \ldots$ où X est une variable et PH_1, PH_2 les phyla qu'elle est susceptible de représenter dans une production.

     On peut également préciser la position absolue dans la production, du phylum représenté:

     Exemple:

     – $X.3 : PH_1 \mid PH_2$ où X est une variable remplacant le phylum PH_1 ou le phylum PH_2 en troisième place dans la production.

     On peut aussi indiquer si la variable peut se rencontrer plusieurs fois:

     Exemple:

     – $X* : PH_1 \mid PH_2$ où X est une variable remplacant un nombre éventuellement nul de phyla de type PH_1 ou PH_2.

     – $X+ : PH_1 \mid PH_2$ où X est une variable remplacant au moins un phylum de type PH_1 ou PH_2.

   - soit un élément du type $*$ ou $+$, qui remplace 0 ou plusieurs phyla dans le premier cas, et 1 ou plusieurs phyla dans le second cas. Ces phyla peuvent être n'importe lesquels parmi ceux auxquels on a restreint la classe ou l'attribut local que l'on définit.

3. cas des opérateurs de liste:

   La liste *liste-var* peut s'exprimer sous les formes suivantes:

   - $(var)+$
   - $(var)*$

- $(var) + *$
- $(var) * +$

Les listes étant homogènes, on n'a besoin que d'une seule variable *var* pour décrire la partie droite d'une production.

Cette variable *var* se comporte comme celles que l'on a dans le cas précédent pour ce qui est de la restriction à certains phyla, mais on ne peut préciser une position ou lui ajouter un élément du type + ou $*$.

Par contre les éléments à l'extérieur des parenthèses symbolisent, pour +, un opérateur de liste d'au moins une opérande, pour $*$, un opérateur de liste pouvant être d'arité nulle, et pour $*+$ ou $+*$ l'une ou l'autre de ces deux possibilités.

Nommer un objet en utilisant une variable permet de l'utiliser dans les règles sémantiques associées à la description.

Voici un exemple de définition d'une classe $\$classe$ d'un type quelconque restreinte aux phyla PH_1, PH_2, PH_3, PH_4:

default $\$classe$ **use**
  where $op : PH_1 \mid PH_3 \rightarrow (X : PH_1 \mid PH_4, Y+ : PH_2, Z* : PH_2 \mid PH_3, X*)$ **use**
    liste-reg **end where** ;
  where *null* **use** *liste-reg* **end where** ;
  where $PH_2, op_1 2 \rightarrow (X : PH_1 \mid PH_2) * +$ **use** *liste-reg* **end where** ;
end default;

Le phylum PH_1 regroupe les opérateurs $(op_1 1, op_1 2, op_1 3)$, le phylum PH_2, $(op_2 1, op_2 2)$, PH_4, $(op_4 1, op_4 2)$ et PH_3 ne représente que l'opérateur op_3.

Les productions filtrées par la première description sont, pour cet exemple:

$op_1 1 \rightarrow PH_1\ PH_2\ PH_2$;
  où $op = op_1 1 \in PH_1$, $X = PH_1.1$, $Y \in (PH_2.2, PH_2.3)$, $Z \in \emptyset$
$op_3 \rightarrow PH_1\ PH_2\ PH_2\ PH_2\ PH_3\ PH_2\ PH_4$;
  où $op = op_3 \in PH_3$, $X \in (PH_1.1, PH_4.7)$, $Y \in (PH_2.2, PH_2.3, PH_2.4)$,
  $Z \in (PH_3.5, PH_2.6)$

On peut remarquer que la variable X employée dans la description est restreinte la première fois à certains phyla, et qu'elle ne l'est pas la seconde fois. Malgré tout, étant donné qu'il s'agit de la même variable, la restriction est valable pour toutes ses occurrences dans la description.

Les productions filtrées par la deuxième description sont:

null-$PH_1 \rightarrow$;

$null\text{-}PH_2 \rightarrow ;$
$null\text{-}PH_3 \rightarrow ;$
$null\text{-}PH_4 \rightarrow ;$

Et enfin, les productions filtrées par la dernière description:

$op_1 2 \rightarrow PH_1* ;$
où $X = PH_1$
$op_2 1 \rightarrow PH_1* ;$
où $X = PH_1$
$op_2 2 \rightarrow PH_2+ ;$
où $X = PH_2$

Les règles sémantiques

La partie *liste-reg* est chargée de donner les règles sémantiques implicites associées à la description de productions donnée.

Elle est composée différemment selon qu'il s'agit de définir
- un attribut local, ou une classe de type synthétisé ou composé-synthétisé,
- une classe de type hérité (global) ou composé-hérité.

- dans le premier cas, il s'agit d'une seule règle sémantique correspondant à la spécification de l'attribut local ou d'un attribut de l'opérateur.
- dans le second cas, il faut spécifier un attribut des fils, et l'on doit alors préciser à quel phylum fils appartient l'attribut que l'on veut définir. On peut ainsi avoir autant de règles sémantiques que de phyla dans le membre droit de la description.

  Pour les distinguer, il faut préciser la variable représentant le phylum dont on veut définir un attribut.

  En reprenant l'exemple ci-dessus en considérant une classe du genre hérité $\$clasher$ au lieu de $\$classe$, définie sur les mêmes phyla PH_1, PH_2, PH_3 et PH_4, on peut avoir:

  default $\$clasher$ **use**
  where $op : PH_1 \mid PH_3 \rightarrow (X : PH_1 \mid PH_4, Y+ : PH_2, Z* : PH_2 \mid PH_3, X*)$
  for X **use** *reg* ;
  for $Y.first$ **use** *reg* ;
  for Y **use** *reg* ;
  for Z **use** *reg* ;

        end where
        where $PH_2, op_1 2 \rightarrow (X : PH_1 \mid PH_2) * + :$ *liste-reg* ;
            for X **use** *reg* ;
        end where
    end default;

On peut ici observer l'emploi d'un qualificateur traditionnellement réservé aux cas d'opérateurs de liste pour désigner un phylum particulier dans une production à arité fixe: dans $Y.first$.

En effet, dans la partie par défaut, on peut utiliser les qualificateurs: $first$, $last$, $left$ et $right$ sur les variables dans les règles sémantiques associées à un opérateur d'arité fixe.

Ceux-ci ont alors une signification particulière:

- $first$ désigne le premier phylum pouvant être représenté par la variable et sur lequel on a défini l'attribut du contexte.
- $last$ désigne le dernier phylum pouvant être représenté par la variable et sur lequel on a défini l'attribut du contexte.
- $left$ désigne le premier phylum à la gauche de celui dont il s'agit, pouvant être représenté par la variable et sur lequel on a défini l'attribut du contexte.
- $right$ désigne le premier phylum à la droite de celui dont il s'agit, pouvant être représenté par la variable et sur lequel on a défini l'attribut du contexte.

Il s'agit donc ici de désigner des *positions relatives* de variables par rapport aux autres du même nom et correspondantes au contexte.

Une règle *reg* est constituée d'expressions OLGA classiques s'appliquant à des occurrences d'attributs normales ou par défaut. Dans ce dernier cas, il est nécessaire, lors du filtrage sémantique, de les instancier, si possible, par des occurrences d'attributs valables dans le contexte (c-à-d la production "filtrée").

Il a également fallu introduire une structure spécifique aux productions d'arité fixe, chargée d'itérer sur les fils et de récupérer des informations, comme la structure *map* le fait pour des opérateurs de liste. La difficulté vient du fait que les opérateurs d'arité fixe sont hétérogènes, et qu'il n'est par conséquent pas question d'utiliser la même structure.

Cette nouvelle expression simulant un *map* se nomme **fmap**: *map* pour les opérateurs d'arité $f$ixe. Elle est présentée dans la section suivante.

Les structures particulières

- **fmap**

  Cette expression n'a de sens que si elle utilisée pour spécifier le calcul d'un attribut *synthétisé* d'un opérateur d'arité fixe.

  Il s'agit en fait de parcourir le corps d'une production d'arité fixe, en prélevant des informations qui ne se trouvent que sur certains phyla.

  Le *fmap* doit pouvoir "sauter" les phyla qui ne l'intéressent pas.

  Il est alors possible, même dans le cas d'une production d'arité non nulle, de n'avoir aucun phylum susceptible de convenir, et dans ce cas, une clause facultative est prévue, servant à donner une valeur par défaut à l'expression au lieu de générer une erreur.

  Une expression *fmap* peut se schématiser ainsi:

  > **fmap** *direction fonction*
  >   **value** *init*
  >   **other** *attributs*
  > **end fmap**;

  La *direction* peut prendre les valeurs *left* ou *right*, selon que le parcours du corps de la production se fait dans le sens gauche-droite ou droite-gauche.

  La *fonction* est la fonction à appliquer sur les occurrences d'attributs décrites dans la clause *other*.

  init est la valeur initiale de l'expression: elle sert lors du premier appel de la fonction, lorsqu'on a trouvé les premières occurrences d'attributs de la clause *other* valides dans le contexte. Lorsque l'on ne peut trouver les occurrences d'attributs adéquates, elle est la valeur par défaut que l'on donne à l'expression.

  La recherche des occurrences adéquates se fait à partir des occurrences d'attributs de la règle implicite. Les "sauts" de phyla dans la production correspondent au fait qu'aucun d'eux ne peut être représenté par la variable qualifiée sur laquelle s'applique l'attribut.

  Reprenons l'exemple précédent en considérant la classe de genre synthétisé *$classyn* définie sur les mêmes phyla PH_1, PH_2, PH_3 et PH_4:

  default *$classyn* **use**
      where $op{:}PH_1 \mid PH_3 \rightarrow (X{:}PH_1 \mid PH_4,\ Y{+}{:}PH_2,\ Z{*}{:}PH_2 \mid PH_3,\ X{*})$ **use**
          fmap left *f*
              value *init*
              other *$classyn(X)*
          end fmap;

end where
where *null* **use** *init* **end where** ;
where PH_2, $op_1 2 \rightarrow$ $(X{:}PH_1 \mid PH_2){*}{+}$ **use**
map left *f*
value *init*
other *$classyn(X)*
end map;
end where
end default;

Plaçons nous dans le cas d'une recherche de définition manquante d'un attribut *$attr* synthétisé déclaré sur les phyla PH_1 et PH_3 et appartenant à la classe *$classyn*.

On va donc chercher dans les règles par défaut de cette classe.

Or, dans le contexte des productions:

$op_1 1 \rightarrow PH_1\ PH_2\ PH_2$;

et

$op_3 \rightarrow PH_1\ PH_2\ PH_2\ PH_2\ PH_3\ PH_2\ PH_4$;

la description syntaxique adéquate est la première:

$op{:}PH_1 \mid PH_3 \rightarrow (X{:}PH_1 \mid PH_4,\ Y{+}{:}PH_2,\ Z{*}{:}PH_2 \mid PH_3,\ X{*})$

L'occurrence par défault *$classyn(X)* de la règle correspondante sera alors instanciée, au cours du filtrage sémantique:

- dans le premier cas, par *$attr(*$PH_1$.1*)*,

- dans le second cas, d'abord par *$attr(*$PH_1$.1*)* puis s'il existe un attribut *$attrib* de la classe *$classyn* déclaré sur PH_4, par *$attrib(*$PH_4$.7*)*.

- **let ... in** se comporte comme un *let ... in* des règles sémantiques normales, à la différence qu'il peut comporter des objets typiques d'une spécification de règles implicites, comme les classes, les attributs locaux par défaut, appliqués à des variables, elles mêmes qualifiées de manière particulière (qualificateurs de position relative dans les productions à arité fixe).

  Exemple d'initialisation d'une valeur selon l'opérateur concerné:

default *$classe* **use**
where $op{:}PH_1 \mid PH_3 \rightarrow (X{:}PH_1 \mid PH_4,\ Y{+}{:}PH_2,\ Z{*}{:}PH_2 \mid PH_3,\ X{*})$ **use**
let *init* : *type* :=
case *op* **is**
$op_1 1$: *val1* ;

```
                    op₃ : val2 ;
                    other : val3 ;
                end case
            in
            fmap left f
                value init
                other $classe(X)
            end fmap;
        end where
    end default;
```

Cela nécessite donc une instanciation lors du filtrage sémantique, au même titre que les règles sémantiques par défaut.

IV.2.2 Implantation dans OLGA

Comme nous l'avons déjà souligné, la spécification des règles sémantiques manquantes doit s'effectuer dans un module OLGA de spécification de GA. C'est pourquoi nous devions l'implanter à l' "intérieur" d'OLGA.

Actuellement, la nouvelle version du compilateur OLGA est une version "bootstrappée"[1] fonctionnant avec un service de génération de règles sémantiques manquantes réduit par rapport à nos objectifs , mais néanmoins puissant. Son développement est toujours en cours et évolue en fonction des modifications apportées à OLGA.

La figure IV.1 montre la place, dans le système FNC–2, des différentes phases amenant la génération de règles sémantiques (partie défaut dans un module de spécification de GA, et générateur).

IV.2.3 Résultats sur un exemple

L'exemple que l'on va utiliser ici est celui d'un traducteur grâce auquel je me suis familiarisée avec FNC–2. Le langage choisi est PL0, qui est un petit langage utilisé par N. Wirth [Wir??] pour exposer les concepts fondamentaux de la compilation des langages de haut niveau.

Cela va permettre de donner un exemple de traducteur construit avec FNC–2(voir SIMPROC [JoP88]), tout en présentant plus concrètement la version d'OLGA incorporant la spécification des règles sémantiques manquantes.

La description de PL0 ainsi que la traduction à effectuer sont décrites en annexe dans les parties A.1 et A.2. Viennent ensuite les grammaires attribuées permettant d'effectuer la vérification de types et la génération de code sous leurs formes OLGA

[1]écrite en OLGA

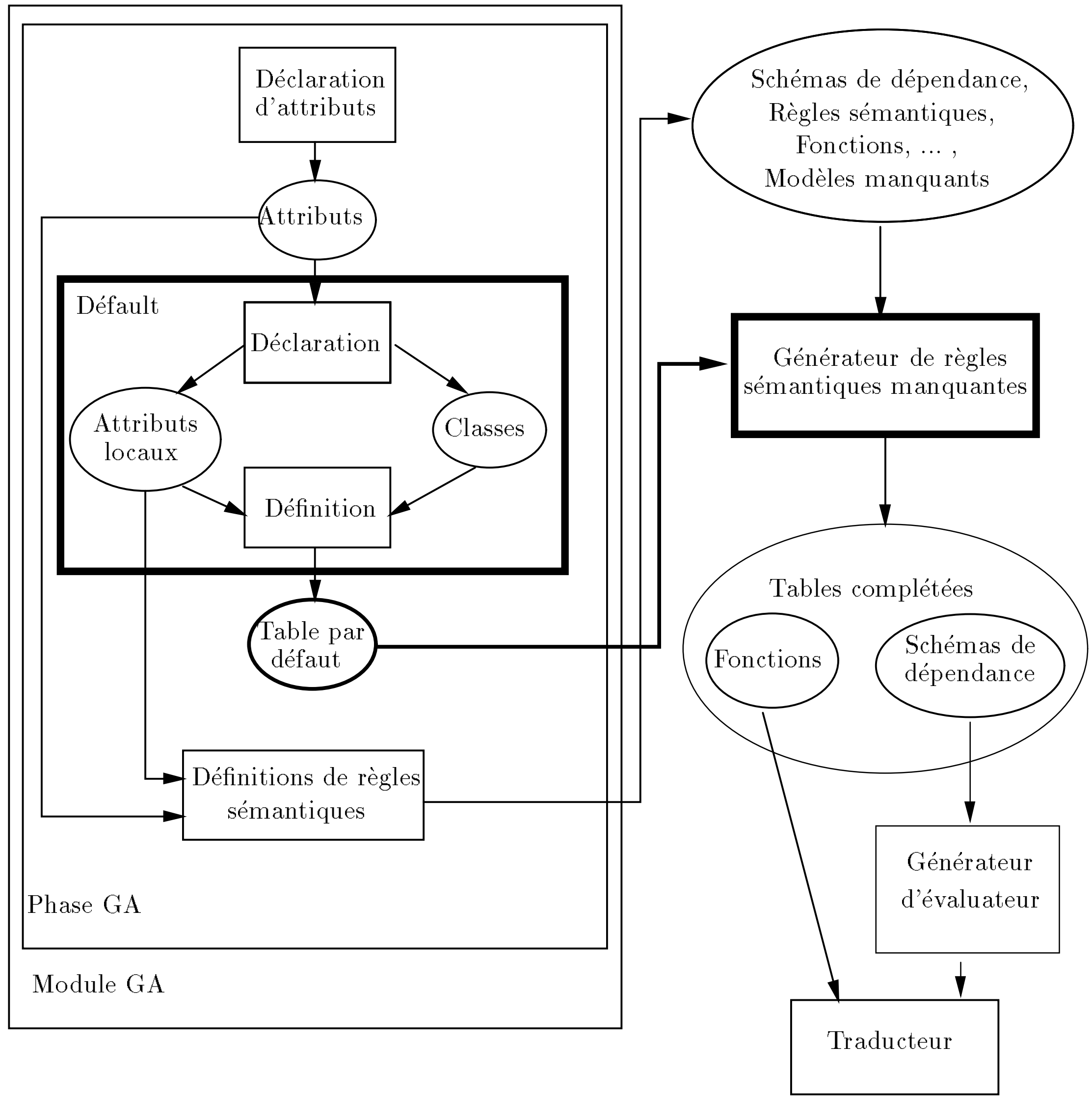


Figure IV.1 : Partie défaut dans FNC–2.

normales (partie A.3.1, pages 159 et 167) puis avec une partie spécification de règles implicites (partie A.3.2, pages 175 et 182) construite avec l'ensemble des possibilités du langage décrit.

En plus d'une diminution de la taille des GA, on s'aperçoit que, lorsqu'on a spécifié des règles par défaut et pris en compte l'existence de règles de copie, le nombre de règles sémantiques a diminué d'environ 60% dans les deux cas par rapport à la version exhaustive initiale.

Mais ce résultat est déjà assez important car il faut tenir compte du fait qu'il s'agit ici de petites grammaires très spécifiques, et donc peu sujettes à de nombreuses répétitions de structures.

Cette considération, et aussi le fait que l'on gagne un temps appréciable, permet de montrer l'intérêt que constitue l'ajout d'un tel service au langage de spécification OLGA.

IV.3 Conclusion

La spécification de règles sémantiques manquantes s'avère donc très utile, tant au point de vue diminution de la taille des GA et du nombre de règles significatives, que sur le plan du confort d'utilisation.

En effet, un autre aspect de ce service est la transformation de la vision traditionnellement syntaxique que l'on a d'une grammaire attribuée en une vision plus sémantique: les règles concernant un attribut donné peuvent être regroupées au lieu d'être réparties dans les spécifications des productions.

C'est ce en quoi nous nous rapprochons de la vision du système LIGA et des MAG décrits dans le chapitre précédent.

Spécifier un problème par une GA revient souvent à reprendre un certain nombre d'informations communes à d'autres problèmes du domaine dans lequel on se situe. Ainsi, pour la compilation, il y a des attributs qui reviennent systématiquement et qui sont, qui plus est, définis de la même manière dans certaines productions (c'est le cas d'un attribut chargé de véhiculer une information de correction dans une phase de vérification de types, ou d'un attribut transportant le code produit dans une GA de génération de code).

On pourrait donc facilement être amené à construire une spécification de règles implicites commune pour un domaine donné (par exemple pour l'analyse de flots de données, en compilation), l'utilisateur n'ayant ensuite qu'à détailler les informations réellement spécifiques à son application.

Ainsi, une expertise du système permettrait de réduire considérablement la taille des spécifications.

Ceci nous amène à la notion de spécialisation d'un langage pour une classe particulière d'applications. Il existe d'ailleurs une expérience intéressante, celle du système GIGAS [CFL88], qui montre qu'il est possible, pour un domaine particulier, en l'occurrence la présentation graphique de structures arborescentes, de concevoir un langage spécialisé et le système correspondant.

C'est ce pas que doit franchir OLGA afin de mieux s'adapter aux problèmes que l'on est amené à résoudre en GA et ainsi faciliter l'usage d'un système tel qu'FNC–2 dans des domaines moins "traditionnels" que celui de la compilation.

Chapitre V

La Généricité et les Grammaires Attribuées

Nous avons développé la notion de Grammaire Attribuée Générique (GAG) pour répondre au manque d'outils de génie logiciel dans l'utilisation des GA comme méthode de spécification.

Une GAG est un module décrivant la résolution d'un problème, spécifiée sous forme d'une grammaire attribuée, que l'on peut appliquer par instanciation sur une autre syntaxe. Cette notion étend le concept de modularité et augmente la facilité d'utilisation d'une grammaire attribuée en tant qu'outil de programmation, au même titre qu'un module générique classique dans un langage conçu pour le développement de grosses applications, tel qu'Ada.

Le but est d'avoir la possibilité, à partir d'une syntaxe "nue", de construire une application, complètement ou en grande partie, en utilisant des instanciations de divers modules génériques sur cette grammaire.

La grammaire d'entrée peut donc être une grammaire indépendante du contexte, ou une grammaire attribuée à laquelle on veut ajouter de la sémantique. Afin de simplifier la description du problème, on ne considère, dans cette partie, que le premier cas.

Ce chapitre décrit le processus d'instanciation d'une GAG sur une grammaire d'entrée, en partant de la notion de grammaire attribuée de la théorie originelle.

V.1 Le concept de Grammaire Attribuée Générique

Une Grammaire Attribuée Générique (GAG) est donc une grammaire attribuée classique dont la syntaxe est suffisamment abstraite pour généraliser le problème qu'elle décrit.

Pour parler d'une GAG à appliquer sur une grammaire donnée, on emploiera le mot **gène**, mettant plus en valeur le fait qu'il s'agit d'un module ayant une sémantique particulière.

L'application d'une GAG se fait sur une grammaire correspondant à un “langage” donné. Il s'agit ici d'une **grammaire indépendante du contexte** (CFG: Context Free Grammar), qui doit servir de base à la **Grammaire Attribuée** produite en sortie.

Avant de décrire dans la section V.1.3 la manière dont les calculs décrits par le gène peuvent s'appliquer à une grammaire d'entrée, nous allons préciser les éléments qui entrent en jeu, à savoir les définitions d'une GA générique, et celle du système de correspondance nécessaire à l'application de la sémantique du gène sur la grammaire d'entrée.

V.1.1 La partie générique

La sémantique associée à un problème donné prend appui sur les éléments sémantiques de base, les attributs définis sur les non-terminaux de la GAG.

Il s'agit en fait, pour décrire ce module, de savoir décrire une GA ne comportant que les éléments importants pour le problème à traiter, c'est-à-dire la CFG minimale et les attributs.

Une GAG $\beta = (G, A, F)$ est une Grammaire Attribuée classique, décrite section II.1.1, en *forme normale* où :

- G est une grammaire indépendante du contexte
  $G = (N, T, P, Z)$ où

  N est un ensemble de symboles non-terminaux

  T est l'ensemble des symboles terminaux significatifs.

  P est un ensemble de productions tel que $P \in N \times N^*$

  Z est l'axiome $Z \in N$
- A est l'ensemble des attributs
- F est l'ensemble de règles sémantiques

Dans la suite, le gène est représenté par $\beta = (G_\beta = (N_\beta, T_\beta, P_\beta, Z_\beta), A_\beta, F_\beta)$ et la grammaire d'entrée par $G = (N, T, P, Z)$

La section suivante permet de définir plus précisément la notion de correspondance, nécessaire à l'instanciation, entre un gène et une grammaire d'entrée.

V.1.2 Le système de correspondance

Afin de permettre la projection de la sémantique d'un gène sur une grammaire, il faut établir les correspondances qui doivent exister entre ces deux modules.

Pour pouvoir appliquer la sémantique du gène sur la grammaire d'entrée G, il faut vérifier que la syntaxe minimale G_β qui constitue sa base peut être instanciée sur G, moyennant la connaissance de repères syntaxiques. A tout arbre de G on doit pouvoir associer un arbre de G_β.

Les notions sémantiques d'un gène qui doivent trouver un équivalent dans la grammaire d'entrée, sont les non-terminaux et les terminaux significatifs, c'est-à-dire conservés à cause de leur influence sur la description du problème.

Il est nécessaire d'associer aux non-terminaux et terminaux du gène appliqué, les non-terminaux et les terminaux sémantiquement correspondants de la grammaire d'entrée. Ceux-ci seront les repères syntaxiques nécessaires à l'instanciation de G_β sur G.

Nous décrivons maintenant la fonction qui permet de définir ces correspondances, d'abord pour les non-terminaux, puis pour les terminaux.

Les non-terminaux

Tout non-terminal de la grammaire G_β du gène doit être associé à au moins un non-terminal de G. Le fait que plusieurs non-terminaux de G puissent correspondre à un même non-terminal du gène signifie que l'on peut reconnaître dans la grammaire G plusieurs notions syntaxiques qui ont le même comportement sémantique pour le problème traité dans ce gène.

De plus, tout non-terminal de G ne peut correspondre qu'à un seul non-terminal de G_β, c'est-à-dire à un seul rôle dans ce contexte.

La fonction, notée $\mathcal{C}or$, permettant de définir les correspondances entre non-terminaux de G_β et non-terminaux de G est décrite de la façon suivante :

Définition V.1.1 : Fonction de correspondance entre non-terminaux

Soit η l'élément auquel correspondent tous les non-terminaux qu'on ne veut pas associer à un non-terminal du gène.

La fonction de correspondance entre non-terminaux, notée $\mathcal{C}or$, est une application surjective définie de la manière suivante :

$$\mathcal{C}or\colon N \longrightarrow N_\beta \cup \{\eta\} \ t.q.$$

$$\forall x \in N_\beta \cup \{\eta\} \ \exists X \in N \quad t.q. \ \ \mathcal{C}or(X) = x \ (surjectivit\acute{e})$$

$$et \ \forall X \in N \ \exists! \ x \in N_\beta \cup \{\eta\} \quad t.q. \ \mathcal{C}or(X) = x$$

♣

Nous allons considérer dans un premier temps que le non-terminal Z_β racine du gène β doit obligatoirement être associé au non-terminal Z racine de G : $\mathcal{C}or(Z) = Z_\beta$

Les non-terminaux de la grammaire d'entrée associés à un non-terminal du gène (et pas η) de cette manière sont appelés *non-terminaux clefs*, et peuvent être définis ainsi.

Définition V.1.2 : Non-terminaux clefs

> *L'ensemble des* non-terminaux clefs *noté N_k est un sous-ensemble des non-terminaux de la grammaire d'entrée, $N_k \subset N$, défini de la manière suivante :*
>
> $$N_k = \bigcup_{x \in N_\beta} \mathcal{C}or^{-1}(x)$$
>
> *avec $\mathcal{C}or^{-1} \;:\; N_\beta \cup \{\eta\} \longrightarrow \mathcal{P}(N)$ la réciproque de $\mathcal{C}or$.*
>
> ♣

On note N_n l'ensemble des non-terminaux de N qu'on a associé à l'élément η pour signaler qu'ils ne sont pas sémantiquement importants. Ils se nomment non-terminaux **neutres** et se définissent ainsi :

$$N_n = \mathcal{C}or^{-1}(\eta) \;\text{ d'où }\; N_k \cap N_n = \emptyset \text{ et } N_k \cup N_n = N$$

Les terminaux

L'association des terminaux de la grammaire d'entrée G aux terminaux de la grammaire G_β du gène se fait selon le même principe que pour les non-terminaux.

On commence donc par définir une fonction de correspondance entre terminaux, $\mathcal{C}or_t$, sur le même schéma que $\mathcal{C}or$.

Définition V.1.3 : Fonction de correspondance entre terminaux

> *Soit η_t l'élément auquel correspondent tous les terminaux qu'on ne veut pas associer à un terminal du gène*
> *La fonction de correspondance entre terminaux, notée $\mathcal{C}or_t$, est une application surjective définie de la manière suivante :*
>
> $$\mathcal{C}or_t \colon T \longrightarrow T_\beta \cup \{\eta_t\} \; t.q.$$
>
> $$\forall \omega \in T_\beta \cup \{\eta_t\} \; \exists \Omega \in T \;\; t.q. \;\; \mathcal{C}or_t(\Omega) = \omega$$
> $$et \; \forall \Omega \in T \; \exists! \; \omega \in T_\beta \cup \{\eta_t\} \;\; t.q. \; \mathcal{C}or_t(\Omega) = \omega$$
>
> ♣

Les terminaux de la grammaire d'entrée associés à un terminal du gène (et pas η_t) de cette manière sont appelés *terminaux clefs*, et peuvent être définis ainsi :

Définition V.1.4 : Terminaux clefs

L'ensemble des terminaux clefs *noté T_k est un sous-ensemble des terminaux de la grammaire d'entrée, $T_k \subset T$, défini de la manière suivante :*

$$T_k = \bigcup_{\omega \in T_\beta} \mathcal{C}or_t^{-1}(\omega)$$

avec $\mathcal{C}or_t^{-1} \ : \ T_\beta \cup \{\eta_t\} \longrightarrow \mathcal{P}(T)$ la réciproque de $\mathcal{C}or_t$

♣

Comme pour les non-terminaux, l'ensemble des terminaux **neutres** noté T_n se définit par :

$$T_n = \mathcal{C}or_t^{-1}(\eta_t) \ \text{ d'où } T_k \cap T_n = \emptyset \text{ et } T_k \cup T_n = T$$

V.1.3 Stratégie d'instanciation d'un gène

Après avoir mis en relation les différentes structures de la grammaire d'entrée G jouant un rôle sémantique important, avec celles de la grammaire G_β du gène $\beta = (G_\beta, A_\beta, F_\beta)$, nous allons décrire le processus qui permet de récupérer les calculs effectués sur un arbre du gène pour les appliquer à un arbre de G.

Cette stratégie d'instanciation de β par la grammaire G se base sur les ensembles de non-terminaux et de terminaux clefs, N_k et T_k, construits par la définition de la correspondance ($\mathcal{C}or$ et $\mathcal{C}or_t$).

L'idée de l'instanciation consiste à associer à tout arbre t de G, un arbre t_β de G_β afin que les règles sémantiques qui s'appliquent sur t_β soient transposées sur t.

Cette idée correspond à décrire une *fonction de couplage* α (section II.2, page 21) telle que : $\alpha : \ G \ \longrightarrow \ G_\beta$. La sémantique de l'instanciation est alors la sémantique du gène appliquée au résultat de cette fonction de couplage.

Nous allons donc regarder si les correspondances établies définissent bien une fonction de couplage (cf section V.2).

Pour ce faire, t est partitionné en *régions* déterminées grâce aux repères que sont les non-terminaux et les terminaux clefs. On définit alors une notion particulière de *connexité* décrite dans la section V.2.1. La construction de ces *régions connexes* est décrite section V.2.2.

Si le partitionnement est effectué de manière correcte, l'étape suivante est l'association de chacune des régions ainsi déterminées avec le morceau (sous-arbre interne) correspondant dans un arbre t_β du gène (section V.2.3).

Dans ce chapitre, l'association est limitée au sous-arbre interne déterminé par une production du gène. Mais il n'est pas exclu de pouvoir l'étendre à à un sous-arbre interne de profondeur supérieure à 1 (cf chapitre VII).

Ainsi, si l'on est parvenu à définir une fonction qui, à tout arbre t (décomposé en régions) associe un arbre t_β, les correspondances définissent bien une fonction de couplage, et donc une instanciation valide. Si cela n'est pas le cas, cela ne veut pas dire qu'une telle fonction n'existe pas, mais simplement que l'on ne peut la définir automatiquement.

Une fois définie, il reste maintenant à construire cette fonction de couplage (cf section V.3) de façon automatique, afin de limiter au maximum le travail du programmeur.

Afin de pouvoir employer la méthode de la méta-composition de grammaires couplées par attributs décrite dans la section II.2.2, page 24, nous allons implanter cette fonction de couplage par une grammaire attribuée respectant certaines contraintes.

Une grammaire attribuée fonctionnelle traduisant un arbre d'une première syntaxe en un arbre d'une seconde correspond à deux grammaires couplées par une fonction de couplage (cf grammaires couplées par attributs : section II.2, page 21). S'il est possible de mettre "bout à bout" deux grammaires attribuées de ce type (la syntaxe résultat de la première sert de base à la seconde), la méta-composition est permet de produire une GA résultat qui ne construit pas l'arbre intermédiaire.

L'emploi de cette méthode pour l'instanciation d'un gène se fera de la manière suivante :

- Une grammaire attribuée fonctionnelle de couplage α est construite automatiquement : pour tout arbre t de G, elle construit un arbre t_β de la grammaire G_β du gène.

  Couplage de G et G_β : $G \xrightarrow{\alpha} G_\beta$

- La projection de la sémantique du gène $\beta = (G_\beta, A_\beta, F_\beta)$ sur la grammaire d'entrée G s'effectue par la composition de α et β, comme le montre le schéma de la figure V.1.

$$\mathrm{G} \xrightarrow{\alpha \text{ couplage}} G_g \xrightarrow{\beta \text{ gène}} \mathrm{G'}$$

$$\mathrm{G} \xrightarrow{\alpha \ \circ \ \beta} \mathrm{G'}$$

Figure V.1 : Application de la sémantique par la composition descriptionnelle.

En effet $\gamma = \alpha \circ \beta$ est une grammaire attribuée basée sur la grammaire G de α et qui a la même sémantique que β. Ceci correspond bien au but que l'on s'était fixé.

Le problème se "résume" donc à construire la grammaire attribuée fonctionnelle de couplage α . Or le fait de vouloir utiliser la méta-composition implique des contraintes pour la construction de α. En effet, la méta-composition telle qu'elle est définie par Ganzinger et Giegerich dans la section II.2, ne s'opère que pour des règles de α portant sur des attributs syntaxiques, c'est à dire appartenant à la grammaire de base G_β de β. Ces règles sont soit l'*Identité*, soit des fonctions de constructions d'arbres de G_β, soit la fonction *If-Then-Else*.

Dans un premier temps nous ne produirons pour α que des règles de type construction d'arbre ou de type transfert (utilisation de l'*Identité*).

De plus, le résultat de la méta-composition $\gamma = \alpha \circ \beta$ où α est purement synthétisée et β quelconque, est dans la plupart des cas de la même classe que β. Afin de pouvoir le plus souvent conserver les propriétés d'un gène β, on décide donc de construire une GA α purement synthétisée.

En utilisant les notations de la section II.2, on doit donc construire une fonction de couplage

$$G(\Sigma) \xrightarrow{\alpha} G_\beta(\Sigma)$$

avec $\Sigma = (S, F)$ la signature sémantique (où S est un ensemble de types sémantiques et F est un ensemble de fonctions sur S) selon les contraintes suivantes :

On a $\alpha = (G(\Sigma), A, F)$ grammaire attribuée et

- $G(\Sigma) = (N, T, Z, P)$ grammaire d'entrée, où :
  - N est un ensemble de non-terminaux ou types syntaxiques : $N \cap T = \emptyset$,
  - T est un ensemble de terminaux ou types sémantiques, $T \subseteq S$,
  - Z est la racine, $Z \in N$,
  - P est l'ensemble des productions ou fonctions syntaxiques : $N \rightarrow [\ (N \cup T) - \{Z\}]^*$,
- Tous les attributs $a \in A$ sont de type syntaxique $type(a)\ \in N_\beta$
- Tous les attributs $a \in A$ sont synthétisés $a \in SYN$
- Les attributs de type syntaxique sont définis dans les règles sémantiques de α, soit par une fonction de transfert, soit par une fonction $f/p_\beta \in P_\beta$ où :

  $f/p_\beta : (X_1, \ldots, X_n) \longrightarrow X_0$

  construit l'arbre de type X_0 avec la production :

  $p_\beta : X_0 \longrightarrow X_1 \ldots X_n$

Les fonctions que l'on veut construire ne travaillent que sur le domaine syntaxique, on n'a donc pas besoin, lors de cette construction, de la signature Σ. Une grammaire $G(\Sigma)$ sera donc notée G dans la suite.

La stratégie d'instanciation se décompose donc en deux étapes. La première permet de vérifier la validité de la fonction de couplage définie par les correspondances établies entre les deux grammaires G et G_β. La seconde étape consiste à construire effectivement la grammaire attribuée de couplage.

V.2 Définition de la fonction de couplage

V.2.1 Notion de connexité

Dans un arbre t de G, les non-terminaux et les terminaux clefs constituent des repères pour la construction de "régions" particulières.

Un arbre est constitué d'une racine, de feuilles et de nœuds internes. La connaissance de ceux-ci définit totalement cet arbre. En donnant des conditions sur ces éléments, on peut donner une définition de la notion de **sous-arbre** que nous allons utiliser.

Définition V.2.1 : Sous-arbre dans un arbre t

Un sous-arbre t' peut être défini dans un arbre t s'il satisfait les conditions suivantes :

- *la racine de t' est :*
  - *soit la racine de t,*
  - *soit un nœud interne de t.*
- *les feuilles de t' sont :*
  - *soit une feuille de t,*
  - *soit un nœud interne de t.*
- *les nœuds internes de t' sont aussi des nœuds internes de t.*

♣

Si l'on impose des conditions plus contreignantes, qui dépendent des non-terminaux et des terminaux clefs, sur les nœuds racine, internes et feuilles d'un tel sous-arbre, on peut alors définir un type de "région" particulier que l'on appelle **région connexe**.

Une région connexe est un sous-arbre t_c, délimité par des non-terminaux clefs ou des terminaux, et ne contenant aucun nœud interne étiqueté par un non-terminal clef.

Définition V.2.2 : Région connexe

Une région connexe dans un arbre syntaxique t donné de la grammaire d'entrée G est un sous-arbre syntaxique t_c tel que :

- *la racine de t_c est un non-terminal clef de N_k*
- *les nœuds feuilles de t_c sont :*
  - *soit une feuille de t, c'est-à-dire un terminal (clef ou non) de t,*
  - *soit un nœud interne de t associé à un des non-terminaux clefs de N_k*
- *il n'existe aucun nœud interne de t_c qui soit associé à un non-terminal clef de N_k.*

♣

Un arbre syntaxique t correspondant à un arbre d'analyse de la grammaire G est ainsi décomposé en un ensemble de régions connexes. On dit alors que l'on construit un **arbre de régions connexes** sur t (voir figure V.2).

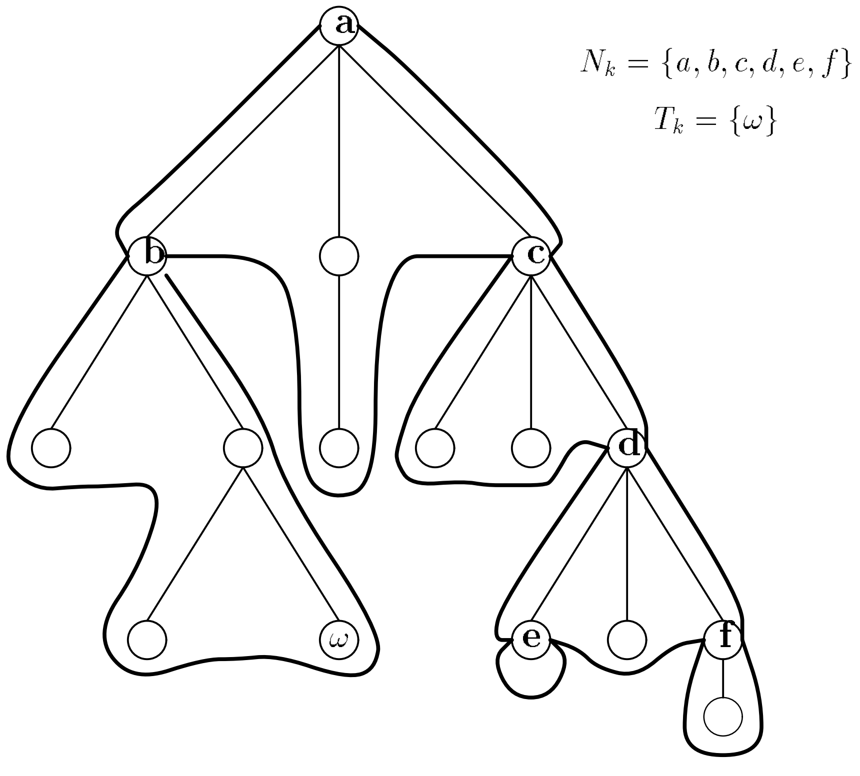


Figure V.2 : Un arbre de régions connexes.

La construction de toutes les régions connexes va permettre d'établir plus facilement des correspondances avec les productions du gène (comme nous le verrons section V.2.3), puisqu'on ne s'occupe que des notions qui ont une correspondance dans le gène.

Pour exprimer la construction de régions connexes de façon plus formelle, nous pouvons utiliser une relation particulière de dérivation qui exprime cette notion de connexité liée aux non-terminaux clefs de N_k.

Cette dérivation est notée $\underset{C}{\Longrightarrow}$. Elle est définie sur $V = T \cup N$ pour une grammaire indépendante du contexte $G = (N, T, P, Z)$, par :

$$\underset{C}{\Longrightarrow} \;=\; \{\, (\alpha A \beta \;,\; \alpha\gamma\beta) \;\mid\; \alpha \in V^*,\ A \notin N_k \;,\; \beta \in (T \cup N_k)^* \;,\; A \rightarrow \gamma \in P\}$$

Tout élément $(\alpha A \beta \;,\; \alpha\gamma\beta)$ de $\underset{C}{\Longrightarrow}$ est noté : $\alpha A \beta \underset{C}{\Longrightarrow} \alpha\gamma\beta$.

Les fermetures transitives sont notées $\overset{*}{\underset{C}{\Longrightarrow}}$ et $\overset{+}{\underset{C}{\Longrightarrow}}$.

La différence avec la relation classique, $\Longrightarrow$, définie sur une telle grammaire est qu'ici $\beta \in (T \cup N_k)^*$ et non à T^* seulement. Ceci indique bien que nous allons nous intéresser aux non-terminaux dérivant vers une chaîne composée de non-terminaux clefs et de terminaux, ce qui coïncide avec la définition d'une région connexe.

Nous allons effectuer une "réduction" de la grammaire G en réduisant son ensemble de production P à un ensemble de productions du type $A \rightarrow \beta$, où $A \in N_k$ et $\beta \in (N_k \cup T_k)^*$ (comparables aux productions du gène).

On peut alors formaliser le problème en disant que ce qui nous intéresse, c'est d'arriver à construire des productions du type :

$X_0 \rightarrow x_1\ X_1\ \ldots\ x_n\ X_n\ x_{n+1}$ avec $X_i \in N_k$, $0 \leq i \leq n$ et $x_i \in T_k$, $1 \leq i \leq n+1$

à partir d'une production

$X_0 \rightarrow \alpha \in P$, avec $\alpha \in (N \cup T)^*$

par application de la dérivation

$\alpha \overset{*}{\underset{C}{\Longrightarrow}} y_i\ X_1 \ldots\ y_i\ X_i\ \ldots\ y_n\ X_n\ y_{n+1}$, avec $y_i \in T^*$

tel que $y_i = \alpha_1\ \beta_1 \ldots \alpha_j\ \beta_j \ldots \alpha_m\ \beta_m\ \alpha_{m+1}$, avec $\alpha_j \in T_n^*$ et $\beta_j \in T_k^*$

d'où $x_i \;=\; \beta_1 \ldots \beta_j \ldots \beta_m$

Ces nouvelles productions $X_0 \rightarrow x_1\ X_1\ \ldots\ x_n\ X_n$ sont dites réduites. Elles sont associées aux sous-arbres qui correspondent aux dérivations que l'on a effectuées avec $\overset{*}{\underset{C}{\Longrightarrow}}$ depuis X_0.

Ces sous-arbres appartenant à la grammaire d'entrée décrivent les régions connexes recherchées.

Il est à noter que deux dérivations $\overset{*}{\underset{C}{\Longrightarrow}}$ différentes peuvent amener à une même production réduite.

De plus, les régions connexes peuvent être de taille non bornée et en nombre infini si un non-terminal neutre peut dériver dans une liste de lui-même sans non-terminal clef au milieu.

Pour décrire ce phénomème, on définit une notion d'accessibilité d'un non-terminal Y depuis un non-terminal $X \notin N_k$ dérivée de la relation $\underset{C}{\overset{*}{\Longrightarrow}}$ et notée : $X \underset{C}{\overset{*}{\Longrightarrow}} Y$.

On a alors :

$$X \underset{C}{\overset{*}{\Longrightarrow}} Y \Longleftrightarrow \left[\begin{array}{l} \exists \alpha, \gamma \in V^*,\ \beta \in (T \cup N_k)^* \text{ t.q.} \\ \\ \alpha X \beta \underset{C}{\overset{*}{\Longrightarrow}} \alpha \gamma \beta \text{ et } Y \in \gamma \end{array} \right.$$

Le cas où l'on peut avoir un non-terminal X récursif, $X \underset{C}{\overset{*}{\Longrightarrow}} X$, peut être une cause d'échec de l'instanciation, car on veut avoir une caractérisation statique des régions pour pouvoir construire statiquement la fonction de couplage, mais dans certains cas il est tout de même possible de le faire. Ce problème est étudié de manière plus complète dans la section suivante.

V.2.2 Construction des régions connexes

Il s'agit ici de décrire la manière dont les arbres de la grammaire d'entrée G peuvent être partitionnés en régions connexes, de manière à ne tenir compte que des non-terminaux et des terminaux clefs.

La construction de régions connexes pour une grammaire G se fait donc par une opération de réduction de cette grammaire en une grammaire dite réduite $G' = (N', T', P', Z')$.

Les sous-arbres correspondant à chacune des dérivations $\underset{C}{\overset{*}{\Longrightarrow}}$ utilisées pour obtenir une production réduite $p \in P'$ sont construits au fur et à mesure de la réduction de la grammaire G vers la grammaire G' ne comportant que des non-terminaux clefs de N_k. A l'issue de ce processus, s'il n'échoue pas, chacun d'eux constitue une région connexe associée à la production réduite obtenue par la dérivation qu'il représente.

Chaque production réduite p doit pouvoir s'identifier, moyennant les correspondances précédemment déterminées, à une production du gène. Celle-ci devra ensuite pouvoir être construite sur une région connexe associée, comme nous le verrons dans la section V.3.

Pour parvenir à ce résultat, nous allons procéder en trois étapes. La première permet d'éliminer les terminaux non significatifs, c'est-à-dire les terminaux neutres, en les remplaçant par la chaîne terminale vide $\mathcal{E}$. On obtient ainsi une nouvelle grammaire $G_1 = (N_1, T_1, P_1, Z_1)$ dans laquelle ne sont présents que les terminaux clefs.

La deuxième étape consiste à déterminer les non-terminaux qui ne peuvent dériver que vers cette chaîne terminale $\mathcal{E}$. Ces non-terminaux non actifs sont appelés des non-terminaux morts et sont également supprimés dans la nouvelle grammaire

$G_2 = (N_2, T_2, P_2, Z_2)$ construite à partir de G_1. On a alors supprimé toutes les branches isues de ces non-terminaux morts, c'est-à-dire les branches mortes.

Le cas exposé précédemment des non-terminaux neutres récursifs ne posent un problème que s'il en reste effectivement à ce stade du processus, c'est-à-dire dans N_2. En effet, cela veut dire qu'ils doivent véhiculer la sémantique du gène. Ils doivent donc être dans les régions connexes que l'on veut caractériser, ce qui pose un problème pour la détermination statique de ces régions. La troisième étape consiste à étudier ce problème et conduit à l'abandon ou la poursuite du processus de définition de la fonction de couplage. Si l'on peut continuer, on a alors une nouvelle grammaire $G_3 = (N_3, T_3, P_3, Z_3)$ à partir de laquelle on pourra déterminer des régions connexes de taille bornée et en nombre fini.

La quatrième et dernière étape consiste à effectuer la réduction, c'est-à-dire obtenir des productions ne contenant que des non-terminaux clefs et des terminaux clefs, tout en construisant les régions connexes.

La réduction est faite par remplacement successif, dans chaque production, de tous les non-terminaux qui ne sont pas clefs par ceux vers lesquels ils dérivent. On obtient ainsi une nouvelle grammaire $G_4 = (N_4, T_4, P_4, Z_4)$ qui peut contenir des productions identiques (comme les grammaires G_1, G_2 et G_3 d'ailleurs), que l'on transformera pour obtenir la vraie grammaire réduite $G' = (N', T', P', Z')$.

Au fur et à mesure de cette réduction, on conserve la trace de l'arbre de dérivation utilisé r(p) associé à chaque production p. Ces arbres sont des sous-arbres (interne) d'arbres de G et lorsque p est une production totalement réduite, r(p) est la région connexe qui a permis de l'obtenir.

Passons à une description plus détaillée de ces quatre étapes de la construction des régions connexes.

Entrée :

On dispose donc de $G = (N, T, P, Z)$ représentant la grammaire d'entrée, ainsi que des ensembles déterminés par les relations $\mathcal{C}or$ et $\mathcal{C}or_t$:

- $N_k \subset N$ et $N_n \subset N$ respectivement ensemble des non-terminaux clefs et ensemble des non-terminaux neutres,
- $T_k \subset T$ et $T_n \subset T$ respectivement ensemble des terminaux clefs et ensemble des terminaux neutres

Étape 1 : Suppression des terminaux neutres

On obtient $G_1 = (N_1, T_1, P_1, Z_1)$ où :

- $N_1 = N$,
- $Z_1 = Z$,

- $T_1 \; = \; T_k$,
- P_1 nouvel ensemble de productions construit à partir de P de la manière suivante :
  Pour toute production p de P
  $p: \; X_0 \rightarrow x_1 \; X_1 \; \ldots \; x_m \; X_m \; x_{m+1} \in P$ où $x_i \in T_n^*$ et $X_i \in N$
  on construit $p_1 \in P_1$:

  $p_1: \; X_0 \rightarrow X_1 \; \ldots \; X_m$

  Remarques :

  - $X_0 \rightarrow \alpha \in P$ avec $\alpha \in T_n^*$ devient $X_0 \rightarrow \mathcal{E} \in P_1$.
  - G_1 n'est pas une vraie grammaire car elle peut contenir des productions identiques, différenciées initialement par les terminaux supprimés.

    On considère néanmoins que P_1 n'est qu'une nouvelle "apparence" de P, et que les productions de P_1 ont le même nom que les productions de P desquelles elles sont tirées.

    Par la suite, on continuera d'appeler grammaires, ces pseudo-grammaires que l'on construit à partir de G.

Étape 2 : Suppression des branches mortes

La suppression des branches mortes de la grammaire G_1 se fait en deux étapes. La première consiste à déterminer les non-terminaux morts, c'est-à-dire qui ne dérive que dans des terminaux neutres, donc ici $\mathcal{E}$, sans passer par un non-terminal clef. La seconde permet de les supprimer et donc de "couper" les branches qui en sont issues.

1. On construit l'ensemble des non-terminaux morts :

$$MORTS = \{X \mid X \in N_n et X \underset{C}{\overset{+}{\Longrightarrow}} \mathcal{E} \text{ et } X \underset{C}{\overset{+}{\not\Longrightarrow}} \alpha Y \beta \text{ où } Y \in N_k\}$$

Algorithme V.2.1 : Construction de l'ensemble $MORTS$

début
$MORTS \leftarrow \{X \mid X \in N_n \; et \; X \rightarrow \mathcal{E} \; \in P_1$
$et \; \nexists X \rightarrow \alpha Y \beta \; \in P \; où \; Y \in N_k\}$;
répéter
$PREC \leftarrow MORTS;$
pour chaque *non-terminal* $X \in N_n$ **faire**
si $\forall X \rightarrow \alpha \in P_1$ *on a* $\alpha \in PREC^*$ **alors**
$MORTS \leftarrow MORTS \cup \{ X \}$
fin si
fin pour
jusqu'à $PREC = MORTS$;
fin

2. La seconde étape consiste à "couper" les branches mortes, c'est-à-dire supprimer les productions ne faisant intervenir que des non-terminaux morts.

   On obtient la grammaire $G_2 = (N_2, T_2, Z_2, P_2)$ telle que :

   - $N_2 = N_1 - MORTS$
   - $T_2 = T_1$
   - $Z_2 = Z_1$
   - P_2 nouvel ensemble de productions issu de P_1 :
     Chaque production p_1 de P_1 de la forme :
     $X_0 \rightarrow \alpha_0 X_1 \alpha_1 \ldots X_i \alpha_i \ldots X_n \alpha_n$
     où $X_i \in MORTS$ et $\alpha_i \in [N - MORTS]^*$
     est remplacée par la production $p_2 \in P_2$:
     $p_2 : \; X_0 \rightarrow \alpha_0 \alpha_1 \ldots \alpha_i \ldots \alpha_n$
     sans ajouter $X_0 \rightarrow \mathcal{E}$.

Étape 3 : Cas des non-terminaux neutres récursifs

Comme nous l'avons déjà remarqué, il est nécessaire de caractériser les régions connexes de manière statique. C'est pourquoi nous allons étudier en premier lieu le cas où l'on peut avoir des non-terminaux neutres $X \in (N_n \cap N_2)$, donc $\notin MORTS$ (c'est-à-dire pouvant produire au moins un symbole clef), et tels que $X \overset{*}{\underset{C}{\Longrightarrow}} X$.

On note REC l'ensemble de ces non-terminaux :

$$REC = \{ \; X \mid X \in (N_n \cap N_2) \; , \; X \overset{*}{\underset{C}{\Longrightarrow}} X \; \}$$

On note Γ la relation qui décrit la récursivité mutuelle entre deux non-terminaux qui appartiennent à REC. On a alors :

$$X \;\Gamma\; Y \Longleftrightarrow (X \overset{*}{\underset{C}{\Longrightarrow}} Y \text{ et } Y \overset{*}{\underset{C}{\Longrightarrow}} X)$$

On peut alors déterminer des classes d'équivalence basées sur cette relation. On note alors pour chaque $X \in REC$:

$$\overline{X} = \{\ Y \mid Y \in REC\ ,\ X \;\Gamma\; Y\ \}$$

Cela signifie en particulier que si $\forall X \in REC$ on renomme X par le nom de sa classe d'équivalence, il existe au moins une production dans P_2 telle que :
$\overline{X} \rightarrow \alpha\overline{X}\beta$ avec $\alpha, \beta \in (N_2 \cup T_2)^*$

A partir de ces productions, on peut considérer deux cas qui amènent soit à la possibilité de poursuivre l'instanciation, soit à l'arrêt pour cause de non validité de la fonction de couplage :

- si les seules productions de ce type sont telles que $\alpha = \emptyset$ et $\beta = \emptyset$, la dérivation éventuellement infinie qui peut être engendrée n'est pas productrice de terminaux clefs ou non-terminaux clefs en plus de ceux vers lesquels dérive le non-terminal représenté par $\overline{X}$. L'ambigüité de G n'influence donc pas l'instanciation du gène.

  On se contente alors de supprimer ces productions de P_2 et de conserver le renommage pour la suite du processus. On ne pourra ensuite plus trouver de production récursive dans une région connexe.

  A la fin de la construction de la fonction de couplage, ces productions $\overline{X} \rightarrow \overline{X}$ seront de nouveau prises en compte sous leur jour réel : $X \rightarrow Y$, avec $Y \in \overline{X}$. Les attributs déclarés sur Y seront forcément les mêmes que ceux déclarés sur X (renommage conservé jusque là) et les règles sémantiques associées à ces productions seront les règles de transfert des valeurs des attributs de Y vers les attributs de X.

- si une des productions de ce type est telle que $\alpha \neq \emptyset$ ou $\beta \neq \emptyset$, avec $\alpha, \beta \in (N_2 \cup T_2)^*$, il y a forcément possibilité de production infinie de terminaux ou de non-terminaux clefs venant de α ou de β. Ceci ne permet pas de déterminer statiquement toutes les régions connexes que l'on peut construire.

  Dans ce cas, on refuse l'instanciation du gène sur la grammaire G : la fonction de couplage déterminée par la description des correspondances n'est pas valide.

Dans le cas où l'on peut continuer le processus d'instanciation, on dispose donc d'une grammaire $G_3 = (N_3, T_3, P_3, Z_3)$ telle que :

- N_3 contient les non-terminaux non récursifs de N_2 et les non-terminaux récursifs renommés par le nom de la classe d'équivalence à laquelle ils appartiennent.
- $T_3 = T_2$
- P_3 contient les productions de P_2 où les non-terminaux récursifs sont renommés et ne contient pas les productions récursives $X \rightarrow X$ où X est un non-terminal renommé et donc un nom de classe.

  Note : on abandonne ici la notation $\overline{X}$ pour désignée une classe d'équivalence, afin d'homogénéiser la notation des éléments de N_3.
- $Z_3 = Z_2$

Avant de passer à la phase de réduction, il faut associer à chaque production $p \in P_3$ un sous-arbre noté $r(p)$ qui représente le sous-arbre de dérivation correspondant dans G. Ici il s'agit de la production p elle-même sous sa forme initiale dans P.

Étape 4 : Phase de réduction

Dans cette phase de réduction il s'agit de remplacer, dans les productions de P_3, toutes les occurrences, dans les membres droits de production, des non-terminaux neutres qui restent dans N_3 ($\notin MORTS$) par les non-terminaux clefs qui en dérivent selon toutes les dérivations possibles. On obtient ainsi des productions dont les membres gauche et droits sont des non-terminaux clefs.

L'ensemble $REMP$ de ces non-terminaux à remplacer est donc :

$REMP = N_n \cap N_3$.

Nous allons remplacer, dans chaque production $p \in P_3$, ces non-terminaux de $REMP$ tout en modifiant l'arbre $r(p)$.

La réduction se fait petit à petit en appliquant l'opération élémentaire qui consiste à réduire une occurrence d'un non-terminal de $REMP$ en appliquant une possibilité de réduction.

Cette opération élémentaire s'appelle un *pas de réduction* et consiste à remplacer une occurrence d'un non-terminal X en partie droite d'une production p, $X \in RHS(p)$, par un membre droit d'une des productions p' où X est membre gauche, $X = LHS(p')$. On obtient ainsi la nouvelle production p'' à la place de p. La production p' est supprimée.

Cette opération de remplacement *Remp* de l'occurrence X_i de X dans p par le membre droit de p' peut s'exprimer ainsi :

$$Remp : \ (\ p,\ X_i,\ p'\ ) \ \longrightarrow \ p''$$

où p'' est le résultat du remplacement de l'occurrence X_i de X par $RHS(p')$ dans p.

Exemple :

Si on a : $p: A \rightarrow B\ B\ C$ et $p': B \rightarrow C\ D$
alors $Remp(p, B_2, p') = p''$
où $p'' : A \rightarrow B\ C\ D\ C$
$\diamondsuit$

Pour obtenir le sous-arbre $r(p'')$ associé à la nouvelle production p'', le sous-arbre $r(p')$ est “accroché” dans le sous-arbre $r(p)$ au nœud correspondant à l'occurrence X_i de X. Pour savoir de quel nœud de $r(p)$ il s'agit, on utilisera la fonction :

$$U : \ (\ r(p),\ X_i\ ) \ \longrightarrow \ u$$

qui, à partir d'un non-terminal X_i de p donne le nœud u étiqueté par le non-terminal X ($label(u) = X$) correspondant dans le sous-arbre $r(p)$ associé.

On peut prendre n'importe quelle notation pour décrire un nœud u, la plus représentative étant celle de Dewey.

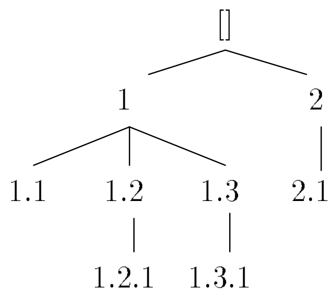


Figure V.3 : Notation de Dewey.

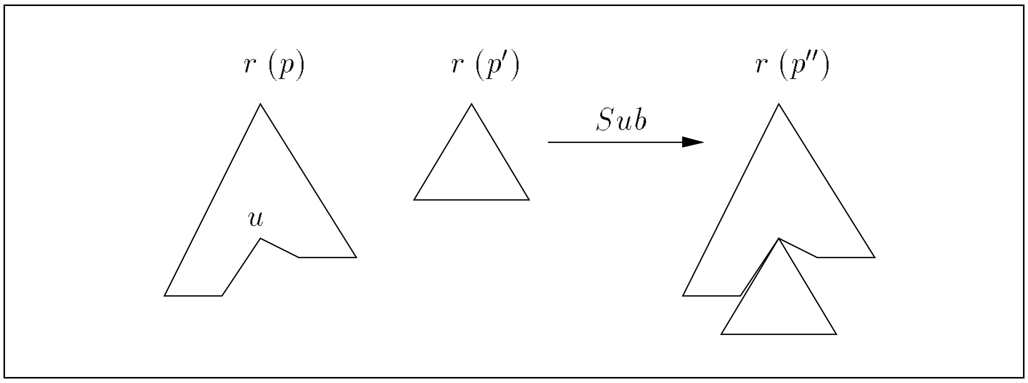
Cette notation, comme le montre la figure V.3, permet d'associer à chaque nœud d'un arbre, un “nom” unique, donnant le chemin d'accès à ce nœud depuis la racine. Ce nom est composé d'une suite d'entiers : chaque nom de nœud comporte le nom de son père, auquel est ajouté son numéro de fils dans la production.

L'opération d'accrochage d'un sous-arbre $r(p')$ dans $r(p)$ correspondant à la réduction de l'occurrence X_i de X dans p est une substitution du nœud feuille $u = U(r(p), X_i)$ de $r(p)$ par le sous-arbre $r(p')$.

On peut décrire une telle opération Sub par :

$$Sub : \ (\ r(p),\ u,\ r(p')\ ) \ \longrightarrow \ r(p'')$$

Ce qui se schématise par :



Un pas de réduction pour X dans une production p se décrit donc ainsi :

Réduction de l'occurrence X_i de X dans p avec p'

$$Red: \ (\ p, \ r(p) \ , X_i, \ p' \ ) \ \longrightarrow \ (\ p'', \ r(p'') \ )$$

où :

- $p'' \ = \ Remp \ (p, X_i, p')$
- $r(p'') \ = \ Sub \ (r(p), U(r(p), X_i), r(p'))$

Chaque sous-arbre $r(p)$ est un sous-arbre de G qui, à terme, devient une région connexe associée à la production p qui a été totalement réduite.

Réduction

La réduction complète est effectuée en prenant chacun des non-terminaux $X \in REMP$ un par un et en effectuant pour chacun le remplacement de toutes ses occurrences dans toutes les productions où il apparaît en partie droite.

Chaque remplacement d'un non-terminal X est, pour chaque production p telle que $X \in RHS(p)$, le résultat de la combinaison de tous les pas de réduction que l'on peut réaliser pour ces occurrences.

On note P_X l'ensemble des productions de P_3 dont X est le membre gauche : $P_X = \{ \ p' \mid p' \in P_3$ et $LHS(p') = X \ \}$.

On note X_i les n occurrences de X dans le membre droit d'une production $p \in P_3$ telle que $X \neq LHS(p)$

$p: \ X_0 \rightarrow \alpha_1 \ X_1 \ldots \alpha_j \ X_j \ldots \alpha_n \ X_n \ \alpha_{n+1}$,

On note u_i le nœud dans le sous-arbre $r(p)$ qui correspond à X_i, tel que $u_i = U(r(p), X_i)$.

La réduction des n occurrences de X de p se fait en utilisant une fonction de combinaison que l'on décrit de la manière suivante :

Soit $P1$ un ensemble de productions $p \in P_3$ dont les $i-1$ premières occurrences de X ont déjà été remplacées, si on appelle la fonction **Combinaisons** :

$$\textbf{Combinaisons} : \ (\ P1 \ , \ i \ , \ P_X \ , \ n \ ) \ \longrightarrow \ P2$$

on exécute :

- si $i > n$ alors STOP : $P1$ contient déjà les productions où les n occurrences de X ont été remplacées.
- sinon :

  1. construction d'un nouvel ensemble New de productions tel que :
     Pour une production $p \in P1$, on construit :
     $New(p) = \{ \ p'' \mid p''$ est le résultat de $Red \ (p, r \ (p), X_i, p')$
     avec une des productions $p' \in P_X \ \}$
     Ce qui nous donne :

     $$New = \bigcup_{p \in P1} New(p)$$

  2. retour d'un nouvel ensemble $P2$, résultat de l'appel de **Combinaisons**(New, $i+1$, P_X, n).

Ces appels récursifs s'arrêtent dés qu'on a atteint $i = n+1$ ce qui signifie que les n occurrences de X dans chacune des productions contenues dans le $P1$ passé en paramètre, ont été remplacées, et ce de toutes les manières possibles que l'on a trouvé dans P_X. Le résultat renvoyé est alors cet ensemble P_1.

L'appel de cette fonction se fera, pour chaque production $p \in P_3$, à partir d'un ensemble P_1 réduit au singleton $\{ \ p \ \}$ et de l'entier $i = 1$, qui indique que l'on a encore remplacé aucune des occurrences de X dans p. Voici la fonction **Combinaisons** décrite plus en détail.

fonction Combinaisons(P: ensemble de productions de P_3;
i: entier;
P_X: ensemble de productions de P_3;
n: entier)
: ensemble de productions de P_3 ;

début
si $i = n + 1$ **alors**

Combinaisons $\leftarrow P$

sinon

$New \leftarrow \emptyset$
pour chaque $p \in P$ **faire**

pour chaque $p' \in P_X$ **faire**
{ *Pas de réduction* }
$Red\ (\ p,\ r\ (p),\ X_i, p') = (\ p'',\ r\ (p'')\ )$
$New \leftarrow New \cup \{\ p''\ \}$

fin pour

fin pour
Combinaisons $\leftarrow$ **Combinaisons**$(New, i+1, P_X, n)$

fin si

fin

Exemple :

On a l'ensemble de productions P_3 suivant :
$P_3 = \{\ p_1 :\ Z \rightarrow A\ B\ ,\ p_2 :\ Z \rightarrow A\ A\ ,\ p_3 :\ A \rightarrow L\ ,$
$p_4 :\ A \rightarrow C\ C\ ,\ p_5 :\ B \rightarrow C\ L\ ,\ p_6 :\ C \rightarrow E\ \}$
où $\{\ Z,\ L,\ E\ \} = N_k$ et $REMP = \{\ A,\ B,\ C\ \}$
pour $X = A \in REMP$, on a : $P_X = \{\ A \rightarrow L\ ,\ A \rightarrow C\ C\ \}$.

Pour la production $p_2 \in P_3$ contenant deux occurrences de A, on appelle la fonction **Combinaisons**($\{\ p_2\ \}$, 1 , P_X ,2) qui exécute :

1. $New(p_2) = \{\ Z \rightarrow L\ A\ ,\ Z \rightarrow C\ C\ A\ \}$ où la première correspond à $Remp\ (p_2, A_1, A \rightarrow L)$ dans $Red\ (p_2, r\ (p_2), A_1, A \rightarrow L)$ et la seconde à $Remp\ (p_2, A_1, A \rightarrow C\ C)$ dans $Red\ (p_2, r\ (p_2), A_1, A \rightarrow C\ C)$.

   ici $New = New(p_2)$

2. Appel de **Combinaisons**(New, 2, P_X)

Combinaisons(New, 2, P_X) exécute ensuite :

1. $New(Z \rightarrow L\ A) = \{\ Z \rightarrow L\ L\ ,\ Z \rightarrow L\ C\ C\ \}$.
   $New(Z \rightarrow C\ C\ A) = \{\ Z \rightarrow C\ C\ L\ ,\ Z \rightarrow C\ C\ C\ C\ \}$.
   ici $New = \{\ Z \rightarrow L\ L\ ,\ Z \rightarrow L\ C\ C\ ,\ Z \rightarrow C\ C\ L\ ,\ Z \rightarrow C\ C\ C\ C\ \}$
2. Appel de **Combinaisons**(New, 3, P_X)

Combinaisons(New, 3, P_X ,2) ne fait rien, ce qui donne à la place de p_2 les productions :

$\{\ Z \rightarrow L\ L\ ,\ Z \rightarrow L\ C\ C\ ,\ Z \rightarrow C\ C\ L\ ,\ Z \rightarrow C\ C\ C\ C\ \}$

$\diamondsuit$

On doit obtenir la nouvelle grammaire $G_4 = (N_4, T_4, Z_4, P_4)$ telle que :

- $Z_4 = Z_3 = Z$
- $N_4 = N_3 - REMP = N_k$
- $T_4 = T_3 = T_k$
- P_4 est le résultat de la réduction présentée par l'algorithme suivant.

Algorithme V.2.2 : Réduction

début
$PROD \leftarrow P_3$
pour chaque *non-terminal* $X \in REMP$ **faire**

$NewPROD \leftarrow \emptyset;$
$P_X = \{\ p_j \mid LHS(p_j) = X\ \}$
pour chaque *production* $p \in PROD$ *telle que* $X \in RHS(p)$ *et* $X \neq LHS(p)$ **faire**

n = *nombre d'occurrences de* X *dans* $RHS(p)$
$NewPROD \leftarrow NewPROD$
$\cup$ **Combinaisons**($\{\ p\ \}$, 1 , P_X , n)

fin pour
$PROD \leftarrow (\ NewPROD$
$\bigcup \{\ pr \mid pr \in PROD\ et\ X \notin RHS(prod)\ \}\ )$
$-\ \{\ p_X \mid p_X \in PROD\ et\ X = LHS(p_X)\ \}$

fin pour
$P_4 \leftarrow PROD$
fin

Cet algorithme traite les remplacements des non-terminaux de $REMP$ de façon linéaire. En effet, on ne prend en compte qu'une seul fois chaque non-terminal $X \in REMP$ pour remplacer ses occurrences dans les membres droits de chacune des productions p telles que $X \in RHS(p)$ et $X \neq LHS(p)$.

Une fois qu'on a fait ces remplacements, les productions p' telles que $X = LHS(p')$ ne nous serviront plus à rien et peuvent être supprimées. Le processus s'arrête donc forcément une fois que l'on a effectué le traitement pour chacun des non-terminaux $\in REMP$, qui sont en nombre fini.

Étant donné que l'on a éliminé les risques de récursion sur un non-terminal neutre, à la fin de cette réduction, les seuls types de productions que l'on trouve sont :

$X_0 \rightarrow \alpha$ avec $X_0 \in N_k$ et $\alpha \in (N_k \cup T_k)^+$

Exemple :

On reprend l'exemple précédent pour lequel on montre le remplacement dans les productions, mais pas la substitution dans l'arbre associé :

$$\begin{aligned} PROD = P_3 = \{\ & p_1 :\ Z \rightarrow A\ B\ ,\ p_2 :\ Z \rightarrow A\ A\ ,\ p_3 :\ A \rightarrow L\ , \\ & p_4 :\ A \rightarrow C\ C\ ,\ p_5 :\ B \rightarrow C\ L\ ,\ p_6 :\ C \rightarrow E\ \} \end{aligned}$$

et pour chacun des $X \in REMP$ on a :

- $X = A$

  $NewPROD = \emptyset$

  $P_X = \{\ A \rightarrow L\ ,\ A \rightarrow C\ C\ \}$

  Pour chaque $p \in PROD$ telle que $A \in RHS(p)$:

  - $Z \rightarrow A\ B$ où $n = 1$

    On obtient par l'appel de **Combinaisons** ($\{\ Z \rightarrow A\ B\ \}$, 1 , P_X , 1) :

    $NewPROD = \{\ Z \rightarrow L\ B\ ,\ Z \rightarrow C\ C\ B\ \}$

  - $Z \rightarrow A\ A$ où $n = 2$

    On obtient par l'appel de **Combinaisons** ($\{\ Z \rightarrow A\ A\ \}$, 1 , P_X , 2), vu dans l'exemple précédent :

    $NewPROD = \{\ Z \rightarrow L\ L\ ,\ Z \rightarrow L\ C\ C\ ,\ Z \rightarrow C\ C\ L\ ,\ Z \rightarrow C\ C\ C\ C\ ,\ Z \rightarrow L\ B\ ,\ Z \rightarrow C\ C\ B\ \}$

  $$\begin{aligned} PROD &= (NewPROD \cup \{\ A \rightarrow L\ ,\ A \rightarrow C\ C\ ,\ B \rightarrow C\ L\ ,\ C \rightarrow E\ \}) - P_X \\ &= \{\ Z \rightarrow L\ L\ ,\ Z \rightarrow L\ C\ C\ ,\ Z \rightarrow C\ C\ L\ ,\ Z \rightarrow C\ C\ C\ C\ , \\ &\quad\ Z \rightarrow L\ B\ ,\ Z \rightarrow C\ C\ B\ ,\ B \rightarrow C\ L\ ,\ C \rightarrow E\ \} \end{aligned}$$

- $X = B$

  $NewPROD = \emptyset$

$P_X = \{ B \rightarrow C\ L \}$

Pour chaque $p \in PROD$ telle que $B \in RHS(p)$:

- $Z \rightarrow L\ B$
  On obtient : $NewPROD = \{ Z \rightarrow L\ C\ L \}$
- $Z \rightarrow C\ C\ B$
  On obtient : $NewPROD = \{ Z \rightarrow L\ C\ L\ ,\ Z \rightarrow C\ C\ C\ L \}$

$$PROD = \{ Z \rightarrow L\ L\ ,\ Z \rightarrow L\ C\ C\ ,\ Z \rightarrow C\ C\ L\ ,\ Z \rightarrow C\ C\ C\ C\ ,\ Z \rightarrow L\ C\ L\ ,\ Z \rightarrow C\ C\ C\ L\ ,\ C \rightarrow E \}$$

- $X = C$

  $NewPROD = \emptyset$

  $P_X = \{ C \rightarrow E \}$ Toutes les productions $p \in PROD - P_X$ sont telles que $C \in RHS(p)$, et on obtient facilement le nouvel ensemble :

  $$PROD = \{ Z \rightarrow L\ L\ ,\ Z \rightarrow L\ E\ E\ ,\ Z \rightarrow E\ E\ L\ ,\ Z \rightarrow E\ E\ E\ E\ ,\ Z \rightarrow L\ E\ L\ ,\ Z \rightarrow E\ E\ E\ L \}$$

On a alors $P_4 = PROD$

$\diamondsuit$

Remarque :

Si les productions posant un problème de branches potentiellement infinies n'avaient pas été supprimées, l'algorithme aurait tout de même terminé et certaines occurrences de non-terminaux neutres récursifs seraient restées dans la partie droite de certaines productions.

Et ceci parce que les remplacements concernant un non-terminal ne se font qu'en une seule fois et ne concerne pas les productions ayant ce non-terminal en partie gauche.

Résultat :

A partir de G_4, qui peut contenir des productions identiques, et des régions connexes associées à chacune de ses productions, on peut construire une grammaire réduite $G' = (N', T', Z', P')$ et l'ensemble R des ensembles de régions associés à chaque production de P', tels que :

- $T' = T_4 = T_k$,
- $N' = N_4 = N_k$,
- $Z' = Z_4 = Z$,
- P' est obtenu en supprimant de P_4 les productions réduites identiques :

  les productions $p_1 \ldots p_i \ldots p_n$ de P_4 telles que $p_1 = \ldots = p_i = \ldots = p_n$ sont remplacées dans P' par la production $p' = p_i$.

- R est l'ensemble des R_p avec $p \in P'$ tels que :

  l'ensemble R_p des régions correspondant à la production réduite p de P' est défini ainsi :

  $R_p \; = \; \{ \; r(p') \mid p' \in P_4 \;$ et $p' = p \; \}$

  On a donc

  $R = \bigcup_{p \in P'} R_p$

L'étape suivante va consister à mettre chaque production réduite p en relation avec la production du gène qui lui correspond. Les ensembles de régions connexes associées à p constituent les parties de la grammaire G sur lesquelles la sémantique de la production du gène doit être projetée. C'est pourquoi l'étape de construction de la fonction de couplage consiste localement à construire cette production du gène à travers les régions connexes associées.

V.2.3 Correspondance avec le gène

On dispose maintenant d'une grammaire réduite $G' = (N', T', Z', P')$ et de l'ensemble R associé. Cette étape permet de trouver les parties du gène correspondantes à chaque production réduite $p \in P'$ et donc aux sous-arbres $r \in R_p$ associés.

Le langage décrit par G' doit, moyennant les relations établies avec les éléments de la grammaire G_β du gène, correspondre au langage engendré par G_β, ou tout du moins en être un sous-ensemble.

La recherche de correspondance commence par une traduction de la production réduite dans le langage du gène.

Ainsi, si l'on a une production réduite :

$A \rightarrow B \; C \; \Omega \; D$ où $A, B, C, D \in N_k$ et $\Omega \in T_k$

et que l'on a également :

$\mathcal{C}or(A) = \mathcal{C}or(C) = a \; , \; \mathcal{C}or(B) = b \; , \; \mathcal{C}or(D) = d$, où $a, b, d \in N_\beta$

et $\mathcal{C}or_t \; (\Omega) = \omega$, où $\omega \in T_\beta$

la question que l'on se pose est celle-ci :

$$a \longrightarrow b \; a \; \omega \; d \;\; \in P_\beta ?$$

Il suffit donc de contrôler que toutes les traductions de productions réduites p' correspondent à une production p_β de G_β et forment les couples (p', p_β).

Si c'est le cas, ces couples (p', p_β), avec $p' \in P'$ et $p_\beta \in P_\beta$, permettent de savoir quelle production du gène p_β doit être construite sur chaque arbre $r \in R_{p'}$.

On construit alors un ensemble de nouveaux couples, *Assoc* :

$$Assoc = \{ \; (r, p_\beta) \mid r \in R_{p'} \text{ et on a le couple } (p', p_\beta) \; \}$$

Si l'on est parvenu à effectuer ces associations, on sait alors que la fonction de couplage définie par la description des correspondances entre les symboles de G et les symboles de G_β est valide. Reste à savoir si sa construction de manière automatique est faisable, ce que décrit la section suivante V.3, auquel cas l'instanciation du gène β par G est réussie.

V.3 Construction de la fonction de couplage

Il s'agit de construire la grammaire attribuée α qui, à partir de la grammaire d'entrée G, va produire en sortie un arbre de la grammaire G_β du gène.

Les productions de G qui doivent à terme supporter une sémantique identique à celle du gène sont celles que l'on a mises en évidence lors de l'étape de partitionnement en régions, c'est-à-dire les productions présentes dans les régions connexes r. La sémantique projetée sur un tel arbre r par la composition de α et du gène β est celle de la production du gène qu'on lui a associée dans le couple $(r, p_\beta) \in Assoc$.

Construire α consiste à produire, pour chaque région connexe r, les règles sémantiques permettant de construire la production p_β du gène.

Sur les non-terminaux de la grammaire d'entrée, on va devoir déclarer des attributs permettant de construire des sous-arbres du type des non-terminaux du gène. Les règles sémantiques de chaque production $p \in P$ vont décrire la manière dont les attributs du père, $LHS(p)$, qui seront donc des sous-arbres du gène, vont être calculés à partir de ceux des fils $\in RHS(p)$.

Une des deux contraintes imposées par le fait qu'α est ensuite composée avec le gène β, est qu'elle ne doit utiliser dans ses règles sémantiques que des fonctions de constructions d'arbres portant sur des attributs syntaxiques ou des fonctions de transfert (c'est-à-dire la fonction *Identité*).

L'autre contrainte que l'on s'impose est que α doit être purement synthétisée.

La définition de cette grammaire attribuée de couplage se fait en deux étapes.

La première va consister simplement à déterminer, pour chacune des régions connexes indépendamment des autres, les différentes possibilités de construction de la production p_β associée à la région. On applique pour chaque région l'algorithme **A1** décrit section V.3.1. Cette étape ne détermine que des possibilités qui satisfont les deux contraintes sur α apportées par l'utilisation de la méta-composition.

Ces possibilités induisent des déclarations d'attributs sur les non-terminaux et aussi sur les terminaux, ce qui constitue une différence avec la théorie initiale. Cela est du à la nécessité dans certains cas de transporter les terminaux du gène associés aux terminaux clefs de la grammaire G pour les prendre en compte à l'endroit où la production du gène est construite.

La deuxième étape, décrite section V.3.2, a alors pour but de vérifier que l'on peut produire une grammaire attribuée. Il faut d'abord vérifier la compatibilité des déclarations d'attributs sur un même non-terminal dans toutes les régions dans lequel il apparait. Ceci permet d'éliminer un certain nombre de possibilités qui ne permettent pas de construire un ensemble d'attribut A_α pour α.

Il faut ensuite vérifier l'existence et l'unicité de la définition de chaque attribut dans une production. On obtient alors les associations de possibilités de constructions de toutes les régions, telles que l'on a une sémantique commune pour chaque production, quelque soit la région où l'on peut la trouver, ce qui constitue des possibilités pour l'ensemble F_α des règles sémantiques de α.

L'algorithme **A2** effectue cette détermination des possibilités pour les ensembles A_α et F_α.

La grammaire α produite doit être de la forme : $\alpha = (G, A_\alpha, F_\alpha)$ où

- $G = (N, T, Z, P)$ grammaire d'entrée,
- A_α est un ensemble d'attributs synthétisés :
  $A_\alpha = SYN_\alpha \cup HER_\alpha$ et $HER_\alpha = \emptyset$, avec
  $A_\alpha = \bigcup_{X \in (N \cup T)} A_\alpha(X)$
  Ces attributs, notés $\$Y(X)$ sont de type syntaxique lorsqu'ils correspondent à un non-terminal du gène, et de type sémantique lorsqu'ils véhiculent des terminaux du gène :
  $\forall X \in (N \cup T)\ ,\ A_\alpha(X) \subseteq \{\ \$Y \mid Y \in (N_\beta \cup T_\beta)\ \}$
- F_α est un ensemble de règles sémantiques telles que :
  $F_\alpha = \bigcup_{p \in P} F_\alpha(p)$ avec
  $\forall p \in P\ ,\ F_\alpha(p) = \{\ f \mid f = f/p_\beta$ où $p_\beta \in P_\beta$, ou $f =$ fonction de transfert $\}$

V.3.1 A1 : Construction de possibilités pour une région connexe.

Il s'agit ici d'associer à chaque sous-arbre r un ensemble de possibilités de construction de la production p_β du gène qui lui est associée dans l'ensemble $Assoc$ construit section V.2.3.

On peut déterminer l'ensemble $P/_r$ des productions de G que l'on trouve dans le sous-arbre r.

L'ensemble des non-terminaux apparaissant en partie gauche ou droite des productions de $P/_r$ est noté $N/_r \subset N$.

De même, on note $T/_r \subset T$, l'ensemble des terminaux des parties droites des productions de $P/_r$.

Le non-terminal associé à la racine [] (Dewey) du sous-arbre r est notée $Z/_r \in N_C$: il s'agit du non-terminal clef de partie gauche de la production réduite associée.

On peut ainsi déterminer la grammaire $G/_r = (N/_r, T/_r, P/_r, Z/_r)$.

Cette grammaire permet de produire des arbres de dérivation parmi lesquels se trouve le sous-arbre r. Il est tout à fait possible que deux sous-arbres r_1 et r_2 différents (et ne correspondant pas forcément à la même production réduite) définissent deux grammaires $G/_{r_1}$ et $G/_{r_2}$ identiques.

La description de p_β que l'on utilise est :

$p_\beta :\ Y_0 \longrightarrow Y_1 \ldots Y_j \ldots Y_m$ avec $Y_j \in (N_\beta \cup T_\beta)$

où Y_j représente le symbole, non-terminal ou terminal, en $j$ème position dans la production p_β. Pour désigner le nom de ce symbole dans N_β ou T_β, on utilisera la notation $\overline{Y_j} \in (N_\beta \cup T_\beta)$. Pour Y_j et $Y_{j'}$, tels que $j \neq j'$, dans la production p_β, on peut avoir $\overline{Y_j} = \overline{Y_{j'}}$ dans $N_\beta \cup T_\beta$. Cela signifie alors que l'on a deux occurrences, en j et en j', d'un même symbole dans p_β.

Exemple : Si on a la production du gène $p_\beta :\ a \rightarrow b\ b\ c$ alors les symboles Y_j sont ici : $Y_1 = b_1$, $Y_2 = b_2$, $Y_3 = c_3$, c'est-à-dire les noms de symboles indicés par la position dans la production. Les noms $\overline{Y_j}$ sont ici : $\overline{Y_1} = b = \overline{Y_2}$ et $\overline{Y_3} = c$.

On parlera donc de $\$\overline{Y_j}(X) \in A_\alpha$ pour désigner un attribut de X du type $\overline{Y_j}$ du symbole en $j$ème position dans p_β.

On définit ce qu'est une possibilité de construction $C = (G/_r, A_C, F_C)$ d'une production p_β pour une région connexe r. L'ensemble de règles sémantiques F_C de celle-ci ne doit contenir qu'une seule fois la règle sémantique f/p_β qui construit la production p_β.

Si $f/p_\beta \in F_C(p)$, avec $p \in P/_r$, alors toutes les autres productions $p' \in P/_r$ ne pourront avoir dans leurs ensembles $F_C(p')$ que des fonctions de transfert des attributs des fils de p'.

Définition V.3.1 : Possibilité de construction

Une possibilité de construction *C d'une production p_β sur une région connexe r est représentée par la grammaire attribuée $(G/_r, A_C, F_C)$ avec :*

- *A_C est l'ensemble des attributs synthétisés déclarés sur les symboles $X \in (N/_r \cup T/_r)$*
  $A_C = \bigcup_{X \in (N/_r \cup T/_r)}\ A_C(X)$ avec

$$A_C(X) \subseteq \{\ \$\overline{Y_j} \mid \overline{Y_j} \in (N_\beta \cup T_\beta)\ \mathit{et}\ Y_j \in p_\beta\ \}$$

- *F_C est l'ensemble des règles sémantiques permettant de calculer ces attributs de façon à obtenir la construction de la production p_β comme valeur de l'attribut synthétisé de la racine $Z/_r$*
  $F_C = \bigcup_{p \in P/_r}\ F_C(p)$ avec

$$\exists p \in P/_r \ telle\ que\ F_C(p) = \{\ f/p_\beta\ \}$$

$$et\ \forall p' \in P/_r\ telle\ que\ p' \neq p\ ,\ F_C(p') = \{\ f \mid f = transfert\}$$

♣

Avant de commencer la construction, on dispose d'éléments qui vont nous permettre de préciser la manière dont nous allons procéder.

Ainsi, on dispose d'informations sur les attributs que l'on doit déclarer sur certains non-terminaux, comme les non-terminaux clefs et les non-terminaux morts de $N/_r$ et sur les terminaux.

Les relations $\mathcal{C}or$ et $\mathcal{C}or_t$ permettent d'établir les déclarations des attributs sur les non-terminaux clefs et les terminaux clefs. Les non-terminaux neutres morts et les terminaux neutres ne peuvent par définition être chargés d'aucune sémantique.

On note A_{C_0} l'ensemble d'attributs prédéclarés :

- $\forall X \in (N/_r \cap N_k)\ ,\ A_{C_0}(X) = \{\ \$\overline{Y_j} \mid \mathcal{C}or(X) = \overline{Y_j}$ et $Y_j \in p_\beta\}$
  et, en particulier : $A_{C_0}(Z/_r) = \{\ \$\overline{Y_0}\ \}$ où $Y_0 = LHS(p_\beta)$.
- $\forall X \in (T/_r \cap T_k)\ ,\ A_{C_0}(X) = \{\ \$\overline{Y_j} \mid \mathcal{C}or_t(X) = \overline{Y_j}$ et $Y_j \in p_\beta\}$
- $\forall X \in (N/_r \cap MORTS)\ ,\ A_{C_0}(X) = \emptyset$
- $\forall X \in (T/_r \cap T_n)\ ,\ A_{C_0}(X) = \emptyset$

Ces attributs sont déterminés une fois pour toute et il n'est pas possible d'étendre ces ensembles initiaux.

Il ne nous reste plus alors qu'à trouver les attributs à déclarer sur les non-terminaux neutres non morts correspondant aux nœuds internes de r. Ces non-terminaux doivent forcément porter des attributs du type des non-terminaux ou des terminaux de p_β (syntaxique ou sémantique) car ils dérivent vers des non-terminaux et des terminaux clefs associés à ces mêmes symboles de p_β.

Ces attributs servent à transmettre les informations des feuilles vers la racine, et on sait qu'au "milieu" une des productions de $P/_r$ doit supporter la fonction f/p_β

On peut ainsi déduire certaines informations sur les règles sémantiques à produire.

En effet, on ne construit qu'une fois la production p_β et cette construction se fait au niveau d'une certaine production $p \in P/_r$ ce qui induit, sur $X = LHS(p)$, un ensemble d'attribut réduit au singleton $\{\ \$\overline{Y_0}(X)\ \}$, avec $Y_0 = LHS(p_\beta)$.

Si l'on considère l'arbre r, on peut dire alors que la valeur de l'unique attribut $\$\overline{Y_0}(X)$ est transférée jusqu'à la racine de l'arbre $Z/_r$, et que tous les non-terminaux neutres $X_i \notin MORTS$ par lesquelles elle est véhiculée doivent avoir ce même ensemble d'attributs $\{ \$\overline{Y_0}(X_i) \}$.

Cela veut dire aussi que les productions apparaissant dans l'arbre r "au dessus" de la production p ne doivent comporter dans leur membre droit qu'un seul non-terminal neutre non mort. En effet, il n'y a qu'un seul attribut à transporter et tous les non-terminaux neutres non morts en transportent obligatoirement, au contraire d'un non-terminal $\in MORTS$ (par définition).

Les productions "en dessous" supportent les transferts des attributs $\$\overline{Y_j}$ correspondant aux éléments Y_j du membre droit de p_β. Un non-terminal neutre $X \notin MORTS$ peut alors porter plusieurs attributs de ce type: $A_C(X) \subseteq \{ \$\overline{Y_j} \mid Y_j \in RHS(p_\beta) \}$.

On peut déterminer deux types de fonctions produisant les fonctions $f \in F_C(p)$ pour une production $p \in P/_r$. Ainsi, si l'on considère la production du gène p_β associée à r et l'ensemble des attributs des membres droits de p noté A_fils tel que:

$A\_fils = \{ \$\overline{Y_j}(X) \mid X \in RHS(p)\ ,\ \$\overline{Y_j} \in A_C(X)\}$.

on a:

$$\textbf{Transfert} :\ (\ A\_fils\ ,\ LHS(p)\ )\ \longrightarrow\ \{\ f\ \}$$

qui donne un ensemble de fonctions f de transfert des attributs de $Attr\_fils$.

On a aussi:

$$\textbf{Construction} :\ (\ A\_fils\ ,\ LHS(p)\ ,\ p_\beta\ )\ \longrightarrow\ f/p_\beta$$

où f/p_β est la règle sémantique de construction de l'arbre de type $LHS(p_\beta)$ à partir des éléments de $RHS(p_\beta)$ représentés par les valeurs des attributs de A_fils, selon la production p_β.

La stratégie que nous avons adoptée consiste à essayer de déterminer des possibilités de construction en appliquant ce principe:

- une possibilité de construction C_1 est constituée par la construction de p_β dés que possible, c'est-à-dire dés que, pour une production $p \in P/_r$, l'ensemble A_fils contient tous les attributs $\$\overline{Y_j}$ des éléments Y_j de $RHS(p_\beta)$.

- les autres possibilités de construction sont produites en retardant la construction de p_β. Retarder la construction d'un pas signifie qu'au lieu d'appliquer la construction de p_β sur la première production p où c'est possible, on va faire des transfert d'attributs pour p et construire p_β sur p' la production directement au dessus de p dans r. Ceci crée une deuxième possibilité C_2. Retarder de deux pas signifie que l'on construit p_β sur la production p'' directement au dessus de p' dans r, ce qui crée une troisième possibilité C_3, etc...

  On crée ainsi n possibilité C_n, où $n-1$ est le nombre de productions au dessus de p dans r.

Ceci nous permet d'avoir le maximum de chances de trouver des ensembles communs d'attributs pour tous les non-terminaux neutres qui apparaissent dans plusieurs régions connexes.

Nous allons maintenant décrire l'algorithme qui construit, selon le principe énoncé précédemment, un ensemble $Cons$ des possibilités de construction de la production p_β, pour chaque arbre r :

$$Cons(r) = \{\ C \mid C = (G/_r, A_C, F_C)\ \}$$

Cette construction se fait par un parcours en profondeur d'abord de l'arbre r.

Chaque possibilité C est construite petit à petit, durant le parcours.

Pour une possibilité C en cours de construction, et pour chaque nœud u de r on a besoin de savoir quels symboles précis de p_β (c'est-à-dire leur position dans p_β) on peut construire. On associe alors à chaque nœud u l'ensemble des "valeurs" des occurrences d'attributs que l'on déclare sur le non-terminal associé $label(u) = X$:

$$Attr_C(u) = \{\ Y_j \mid Y_j \ \text{ symbole en } \ j\text{ème position dans la production } p_\beta\}$$

qui correspond à la déclaration des attributs sur X :

$$A_C(X) = \{\ \$\overline{Y_j} \mid \overline{Y_j} \in (N_\beta \cup T_\beta)\ \}$$

On peut définir au départ la possibilité C_0 qui sert de base à la construction de l'ensemble des possibilités, et qui contient les déclarations d'attributs de A_{C_0}. De la même façon que pour cet ensemble, on sait déjà quels sont les valeurs que l'ont doit avoir sur les nœuds de r étiquetés par des non-terminaux clefs ou des terminaux clels :

$\forall u \in r$ avec $label(u) = X \in (N/_r \cap N_k)$,
$Attr_{C_0}(u) = \{\ Y_j\ \}$ avec Y_j non-terminal en $j$ème position dans la production p_β
et $\mathcal{C}or(X) = Y_j$ }

$\forall u \in r$ avec $label(u) = X \in (T/_r \cap T_k)$,
$Attr_{C_0}(u) = \{\ Y_j\ \}$ avec Y_j terminal en $j$ème position dans la production p_β
et $\mathcal{C}or_t\ (X) = Y_j$ }

Le schéma de l'algorithme est celui-ci :

Algorithme V.3.1 :

début
pour chaque *arbre r associé à la production du gène* p_β; $Y_0 \rightarrow\ Y_1 \ldots Y_j \ldots Y_m$ }
faire

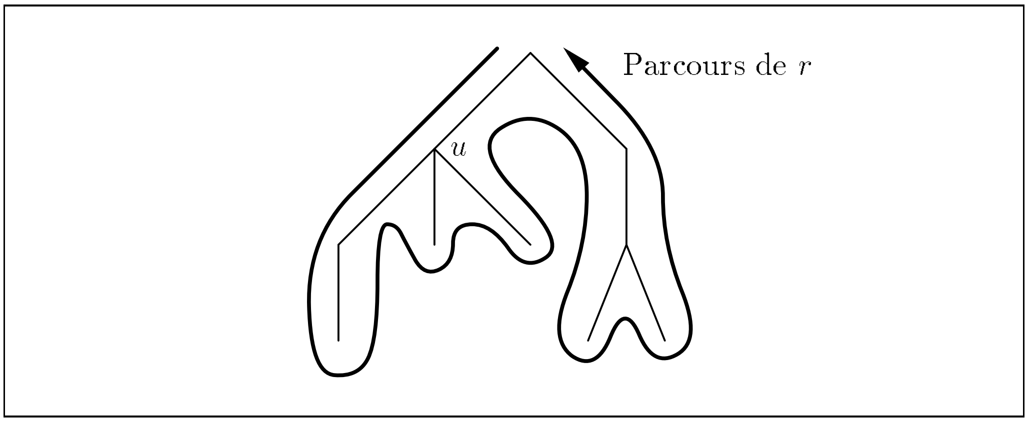
• *Construction de la possibilité de base* $C_0 = (G/_r, A_{C_0}, \emptyset)$ *et des ensembles* $Attr_{C_0}(u)$, $\forall u \in r$ *tel que* $X = label(u) \in ((N/_r \cap N_k) \cup (T/_r \cap T_k))$
• *Initialisation de l'ensemble des possibilités de construction:* $Cons \leftarrow \{ C_0 \}$
• *Appel de la procédure de construction de* $Cons$ *à partir du nœud racine de* r*, que l'on note* [] *avec label* $([]) = Z/_r$
CONSTRUCTION$(r, [], Cons)$
• $Cons$ *contient les possibilités que l'on a réussi à construire ou* $\emptyset$ *sinon*
si $Cons = \emptyset$ **alors**

On n'a réussi à construire aucune possibilité de construction de la production p_β *sur le sous-arbre* r *:*
ECHEC DE L'INSTANCIATION

fin si

fin pour
fin

La procédure **CONSTRUCTION** parcourt les nœuds de l'arbre r selon une stratégie ascendante :



Lorsqu'on est sur un nœud u non feuille de r, $Cons$ contient toutes les possibilités qui résultent des augmentations successives réalisées à partir de la possibilité initiale C_0. Si cet ensemble est vide, ce n'est pas la peine de continuer, la construction automatique de la fonction de couplage n'est pas faisable, l'instanciation a échoué.

Si $Cons$ n'est pas vide et que tous les fils de u ont déjà été parcourus ou sont des feuilles, on peut essayer d'augmenter chaque possibilité $C \in Cons$ pour ce nœud u.

Si aucune des possibilités $C \in Cons$ n'a pu être augmentée sur u alors le nouvel ensemble $Cons$ ne contiendra plus rien. On passe au nœud suivant pour le parcours et le processus recommence.

Augmenter une possibilité C, à partir du nœud u de l'arbre r signifie trouver les ensembles $A_C(X)$ et $F_C(p)$ où $X = label(u)$ et $p = Prod(u)$ avec $Prod(u)$ la production p de G appliquée au nœud u dans l'arbre r.

Comme nous l'avons dit précédemment, seuls deux types d'ensembles $F_C(p)$ peuvent être produits : $\{ f/p_\beta \}$ ou $\{ f \mid f = \text{transfert} \}$.

Il se peut que ces deux solutions d'augmentation soit réalisables pour une possibilité C. Dans ce cas, il suffit de créer une nouvelle possibilité C' dans $Cons$, identique à C pour les calculs précédents mais qui diffère à partir de ce nœud. Augmenter C revient ici à augmenter C d'une manière et la nouvelle possibilité C' d'une autre..

On définit alors les effets que peuvent avoir les deux types d'opérations, construction et transfert, sur l'augmentation d'une possibilité C, à partir de :

- l'arbre r associé à p_β : $Y_0 \to Y_1 \dots Y_j \dots Y_m$,
- le nœud u de r,
- la production $p = Prod(u)$,
- l'ensemble $Attr\_fils = \bigcup_{1 \leq i \leq Nbfils(u)} Attr_C(Fils(u,i))$

  où $Fils(u,i)$ donne le nœud numéro i de u, et $Nbfils(u)$ le nombre total des fils de u,
- et l'ensemble $A\_fils = \bigcup_{X \in RHS(p)} A_C(X)$

On a :

1. Construction de la production p_β associée.
   Ne peut se faire que si l'on dispose de tous les éléments de la partie droite de p_β, c'est-à-dire :

   $\forall Y_j \in RHS(p_\beta)\ ,\ Y_j \in Attr\_fils$

   alors on a :

   $Attr_C(u) \leftarrow \{ \$Y_0 \}$
   $A_C(X) \leftarrow \{ \$\overline{Y_0} \}$
   $F_C(p) \leftarrow \textbf{Construction}(A\_fils, X, p_\beta)$
2. Transfert des valeurs des attributs des fils de u
   $Attr_C(u) \leftarrow Attr\_fils$
   $A_C(X) \leftarrow A\_fils$
   $F_C(p) \leftarrow \textbf{Transfert}(A\_fils, X)$

L'augmentation d'une possibilité C sur un nœud u donné se fait en fonction des ensembles $A_C(label(u))$, $Attr_C(u)$ et $F_C(Prod(u))$.

En effet, si au cours du parcours on tombe sur un nœud u associé au non-terminal X dont l'ensemble $A_C(X)$ est vide, cela signifie que u ne correspond pas à un non-terminal clef (donc en particulier $u \neq []$) et qu'on a jamais rencontré de nœud u'

dans r tel que $label(u') = X$. On n'a donc pas pu non plus rencontrer la production $p = Prod(u)$, car alors, on aurait déjà rencontré $X = LHS(p)$.

Dans ce cas, on regarde si la construction de p_β est possible, c'est-à-dire que l'ensemble $Attr\_fils$ contient bien tous les éléments de $RHS(p_\beta)$. Si elle l'est, on est dans le cas où il faut dupliquer C, car les deux solutions sont envisageables : construction de p_β et retard de la construction.

Dans le cas contraire où l'on ne dispose pas de tous les éléments de $RHS(p_\beta)$ dans $Attr\_fils$, cela peut vouloir dire que la production p_β a déjà été construite et donc $Attr\_fils = \{\ LHS(p_\beta)\ \}$, ou bien que l'on ne peut pas encore construire p_β. Pour ces deux cas, on n'a qu'une seule possibilité pour augmenter C : le transfert des attributs des fils sur le père.

On applique alors l'algorithme suivant lorsque $A_C(X) = \emptyset$:

Algorithme V.3.2 : Cas de u tel que $label(u) = X$ et $A_C(X) = \emptyset$

début
si $\forall Y_j \in RHS(p_\beta)\ ,\ Y_j \in Attr\_fils$ **alors**

$\{$ *Duplication de la possibilité* C $\}$
• *Création de* $C' = (G/_r, A_{C'}, F_{C'}) = C$ *dans* $Cons$, *puis*
• *Ajout dans* C *: solution du* transfert
$Attr_C(u) \leftarrow Attr\_fils$
$A_C(X) \leftarrow A\_fils$
$F_C(p) \leftarrow \textbf{Transfert}(A\_fils, X)$
• *Ajout dans* C' *: solution de la* construction
$Attr_{C'}(u) \leftarrow \{\ Y_0\ \}$
$A_{C'}(X) \leftarrow \{\ \$\overline{Y_0}\ \}$
$F_{C'}(p) \leftarrow \textbf{Construction}(A\_fils, X, p_\beta)$
• *augmenté* $\leftarrow$ *vrai*

sinon

$\{\ Attr\_fils = \{\ Y_0\ \}$ *ou* $\exists Y_j \in p_\beta\ ,\ Y_j \notin Attr\_fils\ \}$
• *Une seule solution :* transfert
$Attr_C(u) \leftarrow Attr\_fils$
$A_C(X) \leftarrow A\_fils$
$F_C(p) \leftarrow \textbf{Transfert}(A\_fils, X)$
• *augmenté* $\leftarrow$ *vrai*

fin si

fin

Le booléen *augmenté* indique si la possibilité courante a pu être augmentée sur le nœud u.

Si au cours du parcours on tombe sur un nœud u associé au non-terminal X dont l'ensemble $A_C(X)$ n'est pas vide, cela peut signifier deux choses: soit $u = []$, et dans ce cas là $A_C(X) = A_C(Z/_r) = \{ \$\overline{Y_0} \}$; soit on est déjà passé par un nœud u' tel que $label(u') = X$ et $A_C(X)$ a déjà été calculé.

Dans le premier cas où $u = []$, la compatibilité avec ce qui existe déjà peut s'exprimer simplement en terme d'égalité entre ce qu'on peut produire et le $Attr_C(u)$ déjà connu comme étant $Attr_{C_0}(u) = \{ Y_0 \}$.

Si $Attr\_fils = \{ Y_0 \}$, cela veut dire que la production a déjà été construite précédemment et qu'on n'a plus qu'à faire un transfert. Sinon, il faut vérifier que la construction de p_β est possible, donc que $\forall Y_j \in RHS(p_\beta)$, $Y_j \in Attr\_fils$.

Dans ce même cas, la production $p = Prod([])$ ne peut exister plusieurs fois dans r. En effet, si on avait aussi $p = Prod(u')$ dans r cela signifierait qu'il existe un nœud interne u' tel que $label(u') = LHS(p) \in N_k$, ce qui est impossible car r est une région connexe. On n'a donc pas de questions à se poser : on crée l'ensemble $F_C(p)$ tel que l'implique la comparaison des ensembles $Attr\_fils$ et $Attr_C(u)$.

Schématiquement, on a alors :

Algorithme V.3.3 : Augmentation de C : cas du nœud racine $u = []$

début
$\{$ *On a* : $u = []$, $Attr_C([]) = \{ Y_0 \}$, $A_C([]) = \{ \$\overline{Y_0} \} \}$
si $Attr\_fils = Attr_C([])$ **alors**

> $Attr_C(u)$ *et* $A_C(X)$ *existent déjà*
> $F_C(p) \leftarrow$ **Transfert**(A_fils, X)
> $augment\acute{e} \leftarrow vrai$

sinon

> $\{ Attr\_fils \neq Attr_C(A_{[]}) \}$
> **si** $\forall Y_j \in RHS(p_\beta)$, $\$Y_j \in Attr\_fils$ **alors**
>
> > $Attr_C(u)$ *et* $A_C(X)$ *existent déjà*
> > $F_C(p) \leftarrow$ **Construction**(A_fils, X)
> > $augment\acute{e} \leftarrow vrai$
>
> **sinon**
>
> > $augment\acute{e} \leftarrow faux$
>
> **fin si**

fin si fin

Dans le cas où l'on tombe sur un nœud u tel que $label(u) = X = label(u')$ d'un nœud u' déjà rencontré, on connait $A_C(X)$ et l'on doit vérifier que ce que l'on peut construire est correct.

L'ensemble $Attr_C(u)$ que l'on produit devra forcément être **correct** vis à vis de $A_C(X)$, c'est-à-dire que l'on a autant d'éléments Y_j correspondant au symbole $\overline{Y_j} = Y$ dans $Attr_C(u)$ que de déclarations d'attributs de type Y dans $A_C(X)$.

On note $A_C(X)/_Y$ l'ensemble des déclarations d'attributs de $A_C(X)$ de type $Y \in (N_\beta \cup T_\beta)$ tel que :

$A_C(X)/_Y = \{ \ \$\overline{Y_j} \mid \$\overline{Y_j} \in A_C(X) \ , \ \overline{Y_j} = Y \ \}$.

On note aussi $Attr_C(u)/_Y$ l'ensemble des éléments Y_j de $Attr_C(u)$ de type $Y \in (N_\beta \cup T_\beta)$ tel que :

$Attr_C(u)/_Y = \{ \ Y_j \mid Y_j \in Attr_C(u) \ , \ \overline{Y_j} = Y \ \}$.

On dit alors que $Attr_C(u)$ est *correct* vis à vis de $A_C(X)$ si et seulement si

$$\forall Y \in N_\beta \ , \ Card(Attr_C(u)/_Y) = Card(A_C(X)/_Y)$$

où $Card(E)$ donne le nombre d'éléments de l'ensemble E.

Ceci correspond à la contrainte de l'unicité des déclarations d'attributs pour un non-terminal, appliquée localement pour l'arbre r.

De même, les fonctions qui permettent de calculer les attributs de $Attr_C(u)$ devront, si $Prod(u) = Prod(u') = p$, avec u' un nœud précédemment parcouru, être les mêmes que celles que l'on a déjà obtenues et qui sont contenues dans $F_C(p)$. Ceci correspond cette fois à la contrainte de l'unicité des définitions des attributs pour une même production, appliquée localement pour le sous-arbre r.

Trois cas peuvent se présenter :

1. La construction de p_β n'est pas possible ou est déjà faite :

   On vérifie que $Attr\_fils$ est correct vis à vis de $A_C(X)$ et si oui, on fait :
   $Attr_C(u) \leftarrow Attr\_fils$
   $F_C(p) \leftarrow \textbf{Transfert}(A\_fils, X)$

2. La construction de p_β est possible et $A_C(X) = \{ \ \$\overline{Y_0} \ \}$:

   On fait la construction :
   $Attr_C(u) \leftarrow \{ \ Y_0 \ \}$
   $F_C(p) \leftarrow \textbf{Construction}(A\_fils, X, p_\beta)$

3. La construction de p_β est possible mais $A_C(X) \neq \{ \ \$\overline{Y_0} \ \}$:

   On vérifie que $Attr\_fils$ est correct vis à vis de $A_C(X)$ et si oui, on fait :
   $Attr_C(u) \leftarrow Attr\_fils$
   $F_C(p) \leftarrow \textbf{Transfert}(A\_fils, X)$

Dans chacun de ces trois cas, lorsque l'on trouve une valeur pour l'ensemble $F_C(p)$ où $p = Prod(u)$, il se peut que $F_C(p)$ existe déjà : il faut alors vérifier que ce que l'on veut y mettre correspond à ce qui y est déjà.

Si pour chaque nœud u de r que l'on atteint on peut trouver un ensemble $Attr_C(u)$ et donc un ensemble $A_C(X)$ et un ensemble $F_C(X)$ pour $X = label(u)$, on dit alors que la possibilité C a été augmentée. Si ce n'est pas le cas, on sait alors que la possibilité C n'est pas valide et elle doit être supprimée de l'ensemble $Cons$ associé à r.

À la fin du parcours de l'arbre r, on obtient un ensemble de possibilités de constructions $Cons$, tel que $\forall C \in Cons$, on a des ensembles $A_C(X)$ d'attributs pour chaque $X \in N/_r$, dont les notations sont dépendantes de la production p_β associée à r.

Afin d'avoir une notation commune des attributs de ces ensembles pour toutes les régions, on définit une fonction de renommage Ren telle que :

$\forall Y \in N_\beta$ on a

$$\left.\begin{array}{r} \$\overline{Y_j}, \$\overline{Y_{j'}} \in A_C(X)/_Y \\ \text{et } j < j' \end{array}\right\} \Longleftrightarrow \left\{\begin{array}{l} Ren(\$\overline{Y_j}) = \$Y_i \\ Ren(\$\overline{Y_{j'}}) = \$Y_{i'} \\ \text{et } 1 \leq i < i' \leq Card(A_C(X)/_Y) \end{array}\right.$$

Appliquée pour tous les ensembles $A_C(X)$ pour toutes les possibilités C de tous les arbres r, cette fonction nous donne des ensembles d'attributs comparables par l'égalité usuelle d'ensembles.

V.3.2 A2 : Vérification de la déclaration et de la définition des attributs

Pour chaque arbre r, on dispose maintenant d'ensembles :
$Cons/_r = \{\ C \mid C = (G/_r, A_C, F_C)\ \}$

On dit que deux possibilités $C \in Cons/_{r_1}$ et $C' \in Cons/_{r_2}$ sont **compatibles** entre elles pour la déclaration des attributs, si :

$$\forall X \in (N/_{r_1} \cap N/_{r_2})\ ,\ A_C(X) = A_{C'}(X)$$

On note $Compat$ un ensemble de possibilités compatibles entre elles, contenant une seule possibilité C_i pour chaque région r_i :

$$Compat\ =\ \{\ C_i \mid C_i \in Cons/_{r_i}\ , 1 \leq i \leq l\ \}$$

où l est le nombre total de régions r_i,
et $\forall X \in N$, pour toutes les régions r_j où $X \in N/_{r_j}$, on a $C_j \in Cons/_{r_j}$ telle que :

$$C_j \in Compat \iff C_j \textbf{ compatible} \text{ avec } C_k\ ,\ \forall C_k \neq C_j \text{ telle que } C_k \in Compat \text{ et } X \in N/_{r_k}$$

On détermine ainsi pour chaque X et pour un ensemble $Compat$:

$$A_{Compat}(X) = A_{C_j}(X)\ ,\ \forall C_j \in Compat \text{ telle que } C_j \in Cons/_{r_j} \text{ et } X \in N/_{r_j}$$

On calcule donc tous ces ensembles $Compat$, qui sont tous des candidats pour constituer la grammaire de couplage.

Mais il ne suffit pas de savoir que l'on peut trouver un ensemble unique d'attributs $A_{Compat}(X)$ pour chaque non-terminal X pour un ensemble $Compat$, il faut aussi garantir que leur définition dans chaque production est identique quel que soit la région dans laquelle se trouve la production.

1. Détermination des productions qui apparaissent dans plusieurs régions :
$$P_{inters} = \{\ p \mid p \in P/_{r_i} \text{ et } \exists r_{i'} \in R \text{ t.q. } p \in P/_{r_{i'}}\ \}$$

2. Pour chaque production $p \in P_{inters}$ on calcule, pour chaque ensemble de possibilités compatibles $Compat$:
$$F_{Compat}(p) = \bigcap_{C_i \in Compat} F_{C_i}(p)$$
où $F_{C_i}(p) = \emptyset$ quand $p \notin P/_{r_i}$

3. Pour chaque production $p \in P_{inters}$ on supprime un ensemble $Compat$ dès que $\exists C_i \in Compat$ t.q. $F_{C_i}(p) \neq \emptyset$ et $F_{C_i}(p) \neq F_{Compat}(p)$.

S'il ne reste aucun ensemble $Compat$ c'est que l'on ne peut construire la fonction de couplage qui aurait permis l'instanciation du gène G_β par la grammaire d'entrée G.

V.4 Génération de la sémantique

Après avoir appliqué les algorithmes **A1** et **A2**, on obtient un certain nombre d'ensembles $Compat$ qui contiennent les possibilités compatibles entre elles pour toutes les régions.

S'il n'existe qu'un seul ensemble $Compat$, pas de problème. S'il en existe au moins deux, le choix de l'un d'eux se fera de façon arbitraire (On pourrait envisager de mettre en place des critères de choix concernant la façon de construire un arbre de chaque solution).

Il faut ensuite effectuer la fusion de ces possibilités $C \in Compat$ et ajouter les productions de G et les règles de transfert associées qui correspondent aux productions récursives mises de côté au moment de la construction des régions connexes section V.2.2. On en tire ainsi la fonction de couplage recherchée.

Définition V.4.1 : Grammaire attribuée de couplage

La grammaire attribuée de couplage $\alpha = (G_\alpha, A_\alpha, F_\alpha)$ est définie comme suit :

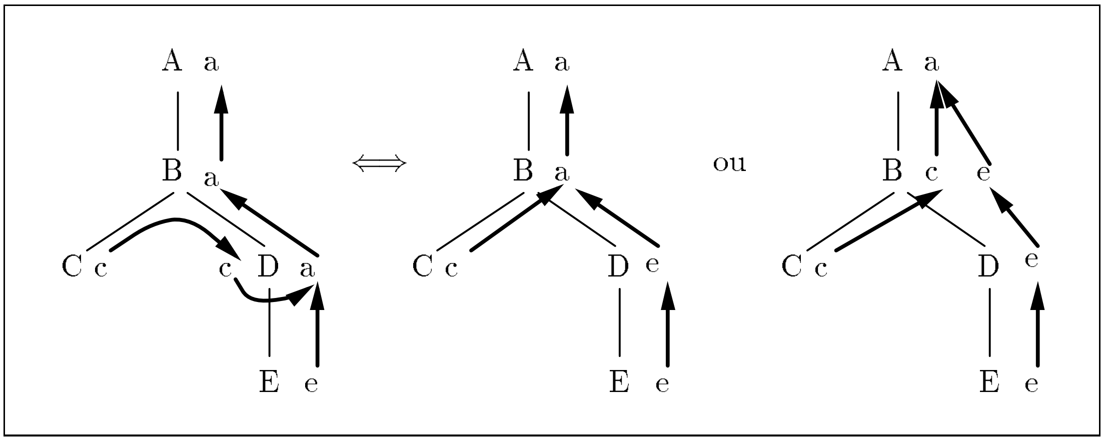
- $G_\alpha = G = (N, T, P, Z)$,
- $A_\alpha = \bigcup_{X \in (N \cup T)} A_\alpha(X)$ *tel que :*
  - *si* $X \in (N \cap MORTS)$ *alors* $A_\alpha(X) = \emptyset$
  - *si* $X \in T_n$ *alors* $A_\alpha(X) = \emptyset$
  - *si* $X \in N_k$ *alors* $A_\alpha(X) = \{\ \$Y\ \}$ *où* $\mathcal{C}or(X) = Y$
  - *si* $X \in T_k$ *alors* $A_\alpha(X) = \{\ \$Y\ \}$ *où* $\mathcal{C}or_t\ (X) = Y$
  - *si X est le nom d'une classe de non-terminaux neutres récursifs alors on le renote* $\overline{X}$ *et* $\forall X_i \in \overline{X}$, $A_\alpha(X_i) = A_{Compat}(\overline{X})$
  - *si* $X \in (N \cap N_n)$ *alors* $A_\alpha(X) = A_{Compat}(X)$
- $F_\alpha = \bigcup_{p \in P}\ F_\alpha(p)$ *tel que :*
  - *si p est une production récursive oubliée lors de la construction des régions connexes, alors on a :*
    $p:\ X_0\ \rightarrow X_1$ *où* X_0 *et* X_1 *appartiennent à la même classe* $\overline{X}$ *de non-terminaux neutres récursifs et* $F_\alpha(p)$ *contient toutes les règles de transfert* $\$Y_i(X_0) = \$Y_i(X_1)$, $\forall\ \$Y_i \in A_{Compat}(\overline{X})$
  - *sinon* $F_\alpha(p) = F_{Compat}(p)$

♣

Il s'agit maintenant de composer la fonction de couplage avec le gène, et on obtient ainsi la sémantique du gène appliquée sur la grammaire d'entrée initiale.

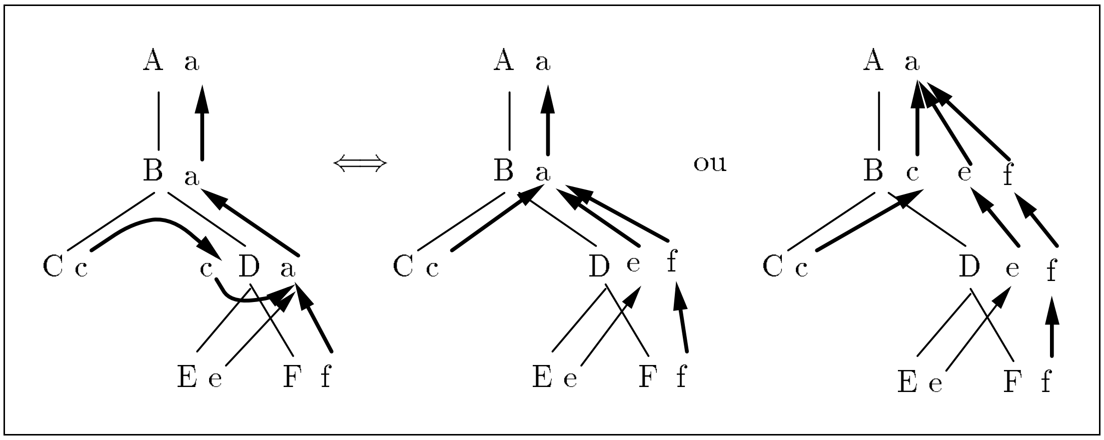
Cette composition se fait en conservant les propriétés du gène lorsque le gène est FNC, ce qui couvre une partie non négligeable des GA. En effet, la grammaire de couplage α est purement synthétisée et sans utilisation multiple d'une occurrence d'attribut dans une règle sémantique. Or on a vu précédemment qu'une méta-composition du type $\alpha o \beta = \gamma$, avec α telle qu'elle est décrite ici, permet à γ d'avoir les même propriétés que β notamment si β est FNC.

Construire α purement synthétisée peut dans certains cas nous priver de certaines possibilités de construction que l'on aurait si l'on autorisait des attributs hérités sur les non-terminaux neutres. Comme pour le cas simple de la figure suivante, on peut toujours se ramener à des attributs uniquement synthétisés à l'intérieur d'une région connexe.



Il est bien entendu toujours interdit de mettre des attributs hérités sur les non-terminaux et terminaux clefs car cela constitue un non sens vis à vis de la description que l'on fait des correspondances entre les symboles d'un gène et certains symboles, les symboles clefs, de la grammaire G.

Mais si l'on s'intéresse à une autre région connexe associée à une production du gène différente, comme la figure suivante nous le montre, on s'aperçoit que dans ce cas précis seule la première solution avec héritage possible sur les non-terminaux neutres permet de décrire une fonction de couplage.



On constate en effet que c'est la seule qui permet d'avoir des déclarations et des définitions d'attributs compatibles, alors que se restreindre aux solutions équivalentes avec uniquement des attributs synthétisés nous oblige à refuser l'instanciation.

Le problème est que construire de façon automatique une telle fonction de couplage en envisageant de pouvoir hériter n'importe quel attribut syntaxique complique sérieusement les choses. La mise en place d'un tel processus peut être très lourde et il n'est pas certain que le gain en puissance d'expression soit suffisant pour en compenser le coût.

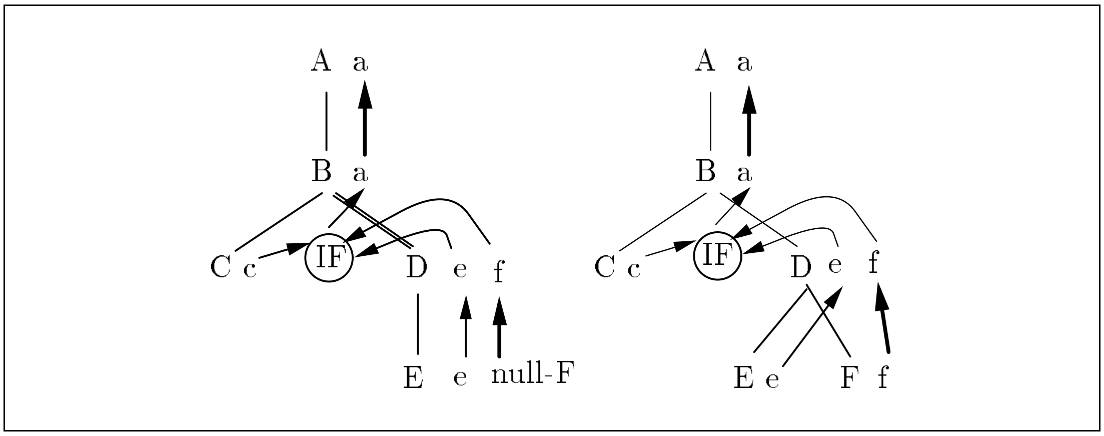
De plus, autoriser l'utilisation d'attributs hérités ne nous permet pas d'utiliser la propriété intéressante concernant le résultat de la méta-composition de la fonction de couplage et du gène : $S \circ C \subset C$.

Une solution pour augmenter la puissance d'expression est d'introduire l'utilisation de la fonction *If-Then-Else*. Néanmoins l'utilisation de fonctions conditionnelles peut poser certains problèmes lors de la méta-composition avec le gène. En effet, la propriété énoncée précédemment n'est pas garantie lorsque les calculs effectués dans les deux branches *Then* et *Else* utilisent le même attribut syntaxique (voir [Rou93]).

On peut néanmoins établir que cette propriété est garantie lorsque l'attribut syntaxique mis en commun n'a pas, dans le gène, d'attributs hérités (démonstration dans [Rou93]).

Même si cette propriété n'est pas forcément applicable, le résultat de la méta-composition est une grammaire attribuée qui hérite de toutes les caractéristiques de la grammaire d'entrée et qui contient la sémantique du module générique que l'on a appliqué. On a de toute façon la garantie que la sémantique du gène générée sur la grammaire source sera correcte.

Dans l'exemple, on a alors pour les deux régions précédentes les possibilités :



La fonction représentée par IF prend pour conditions des valeurs d'attributs sémantiques chargés de décrire le sous-arbre issu de chaque fils. En effet, s'il n'existe pas de sémantique commune pour la production dans les possibilités préalablement sélectionnées, cela veut dire que les sémantiques différentes ne peuvent être induites (par construction) que par les productions issues des non-terminaux en partie droite de la production.

Il s'agit alors de repérer la ou les branches divergentes qui en sont issues, et générer une sémantique conditionnelle tenant compte de ces particularités, d'où l'utilisation de fonctions type *If-Then-Else*. Si on n'arrive pas à trouver de conditions exclusives pour ces différentes possibilités, le gène ne pourra pas être instancié.

Ici la branche divergente est celle qui est issue du non-terminal D sur lequel on a déclaré la totalité des attributs qu'il peut avoir dans toutes les possibilités de toutes les régions dans lesquelles il apparaît ($D.e$ et $D.f$). Cette modification dans le contrôle de la déclaration des attributs est nécessaire si l'on veut aller plus loin et produire des règles sémantiques conditionnelles. En effet, l'absence de déclarations d'attributs communes pour un non-terminal est le symptôme d'une sémantique différente. Il est nécessaire alors de disposer des valeurs syntaxiques nullaires (*null-F*) de chaque attribut syntaxique déclaré pour donner une valeur, dans les productions concernées, aux attributs qui n'en n'ont pas dans ce contexte ($D.f$ = *null-F* dans $D \rightarrow e$).

Cette extension ne sera envisagée que lorsque notre expérience nous aura prouvé qu'elle est réellement nécessaire.

Chapitre VI

Concept étendu de Grammaire Attribuée Générique

Nous avons vu comment nous pouvons récupérer l'expertise sémantique réalisée dans un domaine particulier et contenue dans un gène pour l'instancier sur une application donnée.

Cette méthode a été décrite dans le chapitre précédent pour la théorie classique des GA. Nous allons dans ce chapitre-ci essayer d'augmenter la puissance d'expression du processus en s'intéressant à une notion étendue des GA, les Grammaires Attribuées Ouvertes (GAO). Ces GAO permettent l'utilisation d'attributs en entrée ou en sortie et rendent un interfaçage possible en tous points dans la grammaire. Une définition en est donnée section VI.1

On veut pouvoir partir d'une GAO et construire une application, complètement ou en grande partie, en n'utilisant que des instanciations de divers modules génériques, eux aussi décrits par des GAO.

Dans une application donnée, les modules gènes employés pourront être totalement indépendants, ou avoir des connexions entre eux grâce aux attributs en entrée et en sortie, vus comme les paramètres de la fonction décrite par la GAG. Les cas de dépendance, voir d'inter-dépendance de modules peuvent se produire, soit du fait de leur conception même, car ils correspondent à une distinction naturelle des concepts d'un problème (ainsi, pour construire la table des symboles on a besoin d'informations externes qui font référence dans cet autre module à des éléments à trouver dans la table des symboles) , soit du fait du programmeur lui-même qui les connecte artificiellement, à l'aide d'un ou plusieurs “attributs paramètres” instanciés comme il convient.

Cette vision d'une application comme un jeu de construction faisant intervenir des boîtes qui se juxtaposent ou s'emboîtent, offre une grande souplesse, puisqu'il suffit d'ajouter, d'enlever ou de remplacer une boîte pour changer l'application.

L'instanciation décrite section VI.2 d'un gène sur une GAO s'articule autour de trois axes. Le premier, abordé section VI.2.1, est donné par le fait que cela implique la prise en compte des attributs dans le système de correspondance. Les GAO introduisent également la possibilité d'instancier le gène plusieurs fois sur une même grammaire. La définition et de la construction de la fonction de couplage selon ce nouvel aspect constitue le deuxième axe, section VI.2.2. Le troisième et dernier axe, décrit section VI.2.3, concerne la mise en œuvre de l'application de la sémantique par la méta-composition adaptée à ce nouveau type de fonction de couplage que l'on compose avec un gène de type GAO.

La section VI.3 décrit l'application de plusieurs gènes sur une GAO d'entrée, après leur instanciation individuelle sur la grammaire de base de la GAO d'entrée. Les éléments sémantiques (règles et attributs) de la GAO d'entrée seront pris en compte au même moment que ceux des GAO produites en résultat de la méta-composition.

VI.1 Les Grammaires Attribuées Ouvertes (GAO)

L'intérêt des *Grammaires Attribuées Ouvertes* (GAO) est de pouvoir parler d'attributs d'entrée, dits *importés* et notés in, et de sortie, dits *exportés* et notés out, ce qui rend un interfaçage possible entre plusieurs d'entre elles.

La définition qui nous vient à l'esprit est donc qu'une GAO est une grammaire attribuée classique $\alpha = (G_\alpha, A_\alpha, F_\alpha)$, à laquelle on ajoute la possibilité de déclarer des attributs importés et exportés, tels que :

- L'ensemble des attributs importés est noté $A_{in} \subset A_\alpha$ et l'ensemble des attributs exportés est noté $A_{out} \subset A_\alpha$

  Remarque : rien n'empêche un attribut déclaré in d'être aussi déclaré out. Dans ce cas, sa valeur peut être utilisée mais pas modifiée : elle ne fait que traverser.

- Un attribut in ne doit pas être défini par une fonction de F_α.

La figure VI.1 nous montre un exemple d'arbre de GAO, où les calculs sont représentés par des flèches et les attributs exportés et importés par leurs notations out et in.

On remarque sur ce schéma qu'un attribut exportés peut être utilisé comme argument dans une règle sémantique et servir dans d'autres calculs. On voit bien ici qu'une GAO représente des calculs qui sont répartis dans la grammaire et ne communiquent pas forcément entre eux. Il y a des trous dans la sémantique, qui se décompose alors en bouts de calcul, liés ou non, dont les données sont les attributs importés et dont les résultats sont représentés par les attributs exportés.

Or la théorie des grammaires attribuées nous dit que la valeur sémantique d'une GA est représentée par l'unique attribut synthétisé de la racine. Dans ce cas, comment définir ces attributs importés et exportés ?

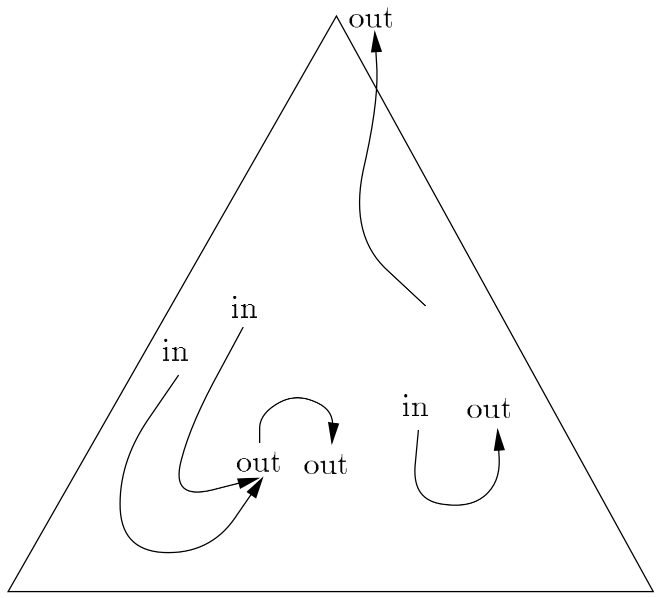


Figure VI.1 : Exemple d'arbre de GAO.

On peut partir alors du principe de la vision des GA couplées décrite par [GaG84], et résumée dans la section II.2, page 21. En effet, ils se placent dans un cadre où l'on peut considérer les attributs sémantiques comme des feuilles dans un arbre, au même titre que les terminaux.

Ces attributs importés ou exportés sont considérés comme des valeurs qui doivent être accrochées sur certains nœuds des arbres d'entrée ou de sortie d'une GA. Contrairement à Ganzinger et Giergerich qui se placent dans le cas général d'une grammaire G sur une signature Σ comprenant aussi les attributs, on va introduire la notion de Syntaxe Attribuée qui sépare les attributs de la syntaxe.

Définition VI.1.1 : Syntaxe Attribuée

On appelle Syntaxe Attribuée (SA) *un couple* (G, A) *où :*

- $G = (N, T, Z, P)$ *est une grammaire indépendante du contexte classique.*
- A *est un ensemble d'attributs associés à certains non-terminaux de* N.

  Ces attributs n'ont pas de direction : ils représentent, dans un arbre de dérivation, les valeurs qui sont accrochées au nœuds étiquetés par ces non-terminaux. Chacun de ces arbres est partiellement décoré.

♣

Une GAO se définit donc en fonction de deux syntaxes attribuées:

Définition VI.1.2 : Grammaire Attribuée Ouverte

Soit $S_1 = (G_1 = (N_1, T_1, Z_1, P_1), A_1)$ *et* $S_2 = (G_2 = (N_2, T_2, Z_2, P_2), A_2)$ *deux syntaxes attribuées.*

On note $\alpha\ :\ S_1 \rightarrow S_2$ *la* Grammaire Attribuée Ouverte *qui à un arbre d'une syntaxe attribuée* S_1 *associe un arbre d'une syntaxe attribuée* S_2.

Soit $\alpha = (G_\alpha, A_\alpha, F_\alpha)$ *une grammaire attribuée ouverte* $S_1 \xrightarrow{\alpha} S_2$ *telle que :*

- $G_\alpha = G_1$
- A_α *est l'ensemble des attributs. Il comprend des attributs qui sont soit de type sémantique* $\in S$, *soit de type syntaxique* $\in N_2$ *et qui peuvent être de trois sortes :*
  - *des attributs dits de* travail, *car ils servent à effectuer les calculs sur chaque nœud d'arbre mais ne sont pas visibles à l'extérieur de la grammaire. Ces attributs sont ceux de la théorie classique.*
  - *les attributs importés qui sont ceux de la SA d'entrée* S_1 *:* $A_{in} = A_1$.
    Ces attributs constituent des données pour α *et se comportent comme des constantes dans les règles sémantiques de chaque production qui en contient une occurrence.*
  - *les attributs exportés qui sont ceux de la SA de sortie* S_2 *:* $A_{out} = A_2$.
    Ces attributs représentent le résultat d'un calcul effectué dans la production qui en contient une occurrence. Savoir dans quelle production doit être la règle sémantique correspondante dépend de la direction qu'on lui donne dans α, *comme pour un attribut de travail.*
- F_α *contient les règles sémantiques qui suivent les contraintes suivantes :*
  - *Les attributs sémantiques de sortie dans une production sont définis par des fonctions* $f_1 \in F$ *ou déclarés en attributs in.*
  - *Les attributs syntaxiques de sortie dans une production sont définis par des fonctions* $f_2 \in P_2$ *ou déclarés en attributs in.*
  - *Les fonctions Identité et IfThenElse peuvent être utilisées dans les règles sémantiques.*
  - *Les occurrences d'attributs syntaxiques ne sont utilisées qu'une seule fois dans les règles sémantiques, excepté pour le cas où*

elles apparaissent dans les branches différentes d'une fonction IfThenElse.

♣

Dans ce cas, la valeur sémantique de la GA est toujours représentée par l'attribut de la racine, qui est un arbre de type S_2 partiellement décoré par les attributs de sortie. De plus, lorsqu'on se replace dans le contexte d'une signature $\Sigma = (S, F)$ où tous les attributs *in* et *out* sont d'un type sémantique $\in S$, on retombe sur la définition des Grammaires Couplées par attributs qui sont une autre "vision" des GA ([GaG84]).

Il se peut que dans certain cas cette vision fonctionnelle des GAO ne nous convienne pas car on ne veut pas construire un arbre en sortie différent de l'arbre d'entrée, mais seulement le décorer différemment. Dans ce cas on parle de GAO Procédurale car on effectue un effet de bord sur l'arbre en modifiant sa décoration.

L'intérêt des GAO Procédurales est qu'elles peuvent être vues comme un moyen d'obtenir en résultat, non pas une unique valeur à la racine, mais plusieurs valeurs qui peuvent être récupérées à n'importe quel endroit dans l'arbre en sortie. C'est d'ailleurs de cette façon que l'on a envie d'utiliser une GAO : interfacer plusieurs GAO Procédurales revient ainsi à séparer les différents concepts d'une GA.

La valeur sémantique d'une GAO Procédurale est alors représentée par l'ensemble des occurrences des attributs *out* déclarés (l'arbre est alors plus vu comme un support qu'un résultat).

Définition VI.1.3 : Grammaire Attribuée Ouverte Procédurale

Soit $S_1 = (G_1 = (N_1, T_1, Z_1, P_1), A_1)$ *une syntaxe attribuée.*

On note $\alpha = (G_\alpha, A_\alpha, F_\alpha)$ *la* Grammaire Attribuée Ouverte Procédurale *telle que l'on a la GAO sous-jacente* $\alpha : S_1 \rightarrow S_3$ *avec :*

- $G_\alpha = G_1$
- S_3 *syntaxe attribuée* $S_3 = (G_1, A_3)$
- A_α *comprend :*
  - *des attributs de* travail.
  - *les attributs importés qui sont ceux de la SA d'entrée* S_1 *:* $A_{in} = A_1$ *(éventuellement vide).*
  - *les attributs exportés qui sont les attributs de la SA de sortie* S_3 *:* $A_{out} = A_3$.
    Chaque attribut de la racine est un attribut out.
- F_α *ne contient aucune règle de construction d'arbres de* G_1

♣

VI.2 Stratégie d'instanciation d'un gène

Nous nous trouvons ici dans le cas où l'on peut avoir des gènes du type GAO, ainsi qu'une grammaire d'entrée qui n'est plus seulement une grammaire mais une GAO également.

Il se peut que la sémantique d'un gène nécessite la connaissance d'éléments préexistants dans la GAO d'entrée : il faut alors envisager d'associer des attributs exportés de la GAO d'entrée avec des attributs importés du gène.

Les attributs importés de la GAO d'entrée peuvent être associés avec des attributs exportés d'un gène, ce qui permet de récupérer des résultats du gène et de les exploiter dans la GAO d'entrée.

Cette section traite dans un premier temps des correspondances entre attributs à établir entre une GAO d'entrée et un gène : section VI.2.1. Il est possible d'envisager la possibilité de spécifier plusieurs instanciations d'un même gène en plusieurs endroits dans un arbre (section VI.2.2). Ceci induit la construction d'une fonction de couplage α particulière, comme nous le verrons section VI.2.3.

VI.2.1 Le système de correspondance

La principale différence avec le cas précédent de GA classiques, est qu'il faut pouvoir associer les attributs importés et exportés de la GAO d'entrée avec ceux du gène. On considère pour le moment que les correspondances établies pour les non-terminaux et les terminaux sont identiques à celles décrites section V.1.2.

Dans la suite, le gène est représenté par $\beta = (G_\beta = (N_\beta, T_\beta, P_\beta, Z_\beta), A_\beta, F_\beta)$ et la GAO d'entrée par $\delta = (G = (N, T, P, Z), A, F)$, où A_β et A contiennent des attributs *in* et *out*.

La sémantique décrite dans un gène peut donc dépendre d'attributs *in* et produire des attributs *out*. On note $A_{\beta_{in}}$ et $A_{\beta_{out}} \subset A_\beta$ ces deux ensembles.

On a alors les contraintes suivantes :

- Chaque attribut *in* d'un gène β doit être associé à un attribut *out* de la GAO d'entrée δ.
- Chaque attribut *out* d'un gène β doit être associé à un attribut *in* de δ.
- Un attribut *out* de δ peut être associé à deux attributs *in* de deux gènes différents.
- Un attribut *out* de δ ne peut être associé qu'à un attribut *in* d'un seul gène.
- Un attribut *in* de δ peut être aussi un attribut *out* de δ. Dans ce cas, il n'existe aucune règle sémantique définissant ses occurrences dans δ.

> Un cas identique dans le gène n'a aucun sens, car cela signifie qu'il n'y a pas de sémantique associée à cet attribut dans ce gène, ce qui est contraire au concept du gène.

Les occurrences d'attributs importés et exportés de la GAO d'entrée δ associés à des attributs exportés et importés d'un gène β constituent l'ensemble des **attributs clefs** de δ pour ce gène.

Les non-terminaux sur lesquels sont déclarés ces attributs clefs doivent être associés aux non-terminaux du gène sur lesquels sont déclarés les attribut du gène correspondants.

L'ensemble des *attributs clefs* A_k est construit lors de l'étape de mise en correspondance pour chaque gène β.

On a : $A_k \subseteq A_{in} \cup A_{out}$

Chacun d'eux est associé à un seul attribut *in* ou *out* du gène (et donc à une signification particulière dans ce contexte d'application). Chaque attribut importé ou exporté du gène doit être associé à un attribut exporté ou importé de δ.

On décrit la relation de correspondance entre attributs $\mathcal{C}or_a$ de la façon suivante :

$$\mathcal{C}or_a \colon A_k \longrightarrow A_{\beta_{in}} \cup A_{\beta_{out}} \text{ t.q.}$$
$$\forall b \in (A_{\beta_{in}} \cup A_{\beta_{out}})\ \exists B \in A_k \ \text{ t.q. } \ \mathcal{C}or_a(B) = b \text{ et}$$
$$\forall B \in A_k\ \exists!\ b \in (A_{\beta_{in}} \cup A_{\beta_{out}}) \ \text{ t.q. } \ \mathcal{C}or_a(B) = b$$
$$\text{et, si } B \text{ est déclaré sur } X \in N, \text{ et } b \text{ est déclaré sur } x \in N_\beta,$$
$$\text{alors } \mathcal{C}or(X) = x \text{ (d'où } \ X \in N_k)$$

VI.2.2 Instanciation multiple

La stratégie d'instanciation d'un gène ne change pas beaucoup dans la mesure où l'on part de la grammaire G de la GAO d'entrée δ, sans s'occuper des attributs *in* et *out* qu'elle possède.

Mais le fait d'avoir introduit les GAO nous permet d'envisager la possibilité de construire plus d'un arbre en sortie, en considérant des attributs exportés de type syntaxique, ce qui modifie un peu notre stratégie.

Ainsi on considère une GAO Procédurale telle qu'elle est décrite section VI.1 mais dont les attributs exportés, constituant sa valeur sémantique, sont de type syntaxique et ne peuvent prendre que des valeurs d'arbres d'une seule et même grammaire.

Soit $S_1 = (G_1 = (N_1, T_1, Z_1, P_1), A_1)$ une syntaxe attribuée et $G_2 = (N_2, T_2, Z_2, P_2)$ une grammaire.

On note $\alpha : \ S_1 \to G_2^+$ une telle GAO Procédurale $\alpha = (G_\alpha, A_\alpha, F_\alpha)$ définie par la GAO sous-jacente $\alpha : S_1 \to S_1'$ telle que :

- $G_\alpha = G_1$
- S'_1 syntaxe attribuée $S'_1 = (G_1, A'_1)$
- A_α comprend :
  - des attributs de *travail*.
  - les attributs *in* qui sont ceux de la SA d'entrée S_1 :
    $A_{in} = A_1$ (éventuellement vide).
  - les attributs *out* qui sont les attributs de la SA de sortie S'_1 :
    $A_{out} = A'_1$ où les attributs de A'_1 sont des attributs syntaxiques du type des éléments de N_2.
- F_α contient les règles sémantiques qui suivent les contraintes suivantes :
  - Les attributs sémantiques de sortie dans une production sont définis par des fonctions $f_1 \in F$ ou déclarés en attributs *in*.
  - Les attributs syntaxiques de sortie dans une production sont définis *uniquement* par des fonctions $f_2 \in P_2$ ou déclarés en attributs *in*.

Cette GAO Procédurale particulière correspond à une fonction qui à un arbre (attribué) d'une grammaire G_1 associe un ou plusieurs sous-arbres d'une grammaire G_2, c'est-à-dire une forêt d'arbres de G_2.

On peut donc appliquer cela à la construction de plus d'un arbre du gène pour la fonction de couplage, ce qui nous donne ici :
$G_1 = G$, $A_1 = \emptyset$, $G_2 = G_\beta$ et A'_1 décrit les attributs exportés supportant des arbres de G_β.

Les implications de ce principe sur les différentes phases de l'instanciation sont décrites ici :

1. Pour le système de correspondance, cela signifie que l'on relaxe la contrainte de l'association obligatoire de l'axiome Z de G avec l'axiome Z_β de la grammaire G_β du gène.

   Simplement, si on les associe : $Z_\beta = \mathcal{C}or(Z)$, alors on a obligatoirement $\mathcal{C}or^{-1}(Z_\beta) = \{ Z \}$ et on retombe sur la même situation, c'est-à-dire l'instanciation unique du gène à la racine.

2. Lors de la construction des régions connexes cela se traduit par le fait qu'à l'issue de la réduction on peut obtenir des productions réduites p telles que $Z = LHS(p)$ avec $Z \in N_n$, axiome de G
   et $\forall X \in RHS(p)$ tel que $X \in N_k$ on a $X \in \mathcal{C}or^{-1}(Z_\beta)$.

   En effet, s'il existe un $X \in RHS(p)$ tel que $X \in N_k$ mais $X \notin \mathcal{C}or^{-1}(Z_\beta)$, cela signifie que les calculs complets du gène ne pourront être projetés sur les dérivations issues de cette production. Dans ce cas, on refuse l'instanciation du gène sur la grammaire G.

Il faut également reconsidérer le cas des non-terminaux neutres récursifs qui dans le processus normal est traité avant la réduction. En effet, s'il existe des productions réduites p telles que $Z = LHS(p)$ avec $Z \in N_n$, les régions connexes qui leur sont associées ne seront pas amenées à supporter de la sémantique (voir point suivant). Le fait d'avoir dans ces régions des non-terminaux neutres récursifs qui n'apparaissent pas ailleurs (éventuellement producteurs de non-terminaux clefs qui ne peuvent être qu'associés avec l'axiome du gène) ne pose donc pas de problèmes.

Or, lors de l'étape 3 du processus décrit section V.2.2, page 93, on repère tous les non-terminaux neutres récursifs, on refuse l'instanciation lorsque ceux-ci sont producteurs de non-terminaux clefs et on supprime les productions récursives pour la suite de la réduction. Il s'agit simplement maintenant de repérer ces non-terminaux et de les renommer par leur nom de classe d'équivalence : la discussion pour la poursuite de l'instanciation est déplacée après la phase de réduction.

Le fait de conserver les productions récursives ne gêne en rien le processus de réduction car l'algorithme est linéaire et termine de toute façon. On obtient alors des productions p dont le membre droit peut contenir une ou plusieurs occurrences de non-terminaux neutres récursifs.

La section C.1,page 198 de l'exemple donné en annexe chapitre C illustre ce cas de figure.

- Si ces non-terminaux apparaissent en partie droite d'une production p telle que $Z = LHS(p)$ avec $Z \in N_n$ et qu'ils n'apparaissent que dans ce type de productions, alors on regarde si les non-terminaux clefs X qui constituent le reste du membre droit sont tels que $X \in \mathcal{C}or^{-1}(Z_\beta)$. Ceci revient à les ignorer et se replacer dans le cas décrit au début.

  Si ces non-terminaux apparaissent dans d'autres productions que celles-ci, on refuse l'instanciation car cela signifie qu'ils seraient porteurs de sémantique alors que l'on ne peut leur en donner dans ce cas là.

- Si on a des occurrences d'autres non-terminaux neutres récursifs dans les membres droits des productions restantes, issues de non-terminaux clefs, alors on effectue le même type de discussion que celle décrite à l'étape 3 de la section V.2.2. Si ces non-terminaux sont producteurs de non-terminaux clefs alors on refuse l'instanciation, sinon on se contente de supprimer les productions récursives ainsi que les productions réduites dans lesquelles ils apparaissent. Le traitement de ces productions est pris en compte normalement à l'issue de la construction de la fonction de couplage.

3. Au moment de la mise en correspondance avec le gène, on ne tient pas compte des régions connexes associées avec les productions réduites issues de l'axiome Z lorsque celui-ci est un non-terminal neutre (ce qui implique une instanciation multiple, d'après ce que l'on a vu précédemment).

4. La fonction de couplage α que l'on construit contient des attributs de sortie qui sont les attributs de type Z_β de chaque $X \in N_k$ tel que $X \notin \mathcal{C}or^{-1}(Z_\beta)$.

   La section VI.2.3 traite de ce cas particulier de α lors de sa méta-composition avec une GAO, ce que peut être le gène β.

VI.2.3 Extension de la méta-composition

La méta-composition telle que décrite section II.2.2 n'est pas prévue pour accepter les GAO. De plus dans la définition qu'on en a donné, on n'accepte pas les GA qui construisent des forêts.

Il faut donc aménager la méta-composition de façon à ce qu'elle puisse effectuer la composition de $\alpha : G_1 \rightarrow G_2^+$ et $\beta : S_2 \rightarrow S_3$.

Les règles permettant la composition de deux GAO sont décrites ainsi :

- Les occurrences d'attributs importés et exportés de β restent, après renommage, des occurrences attributs du même genre (importés ou exportés).

- Soit $X.a.b$ une occurrence d'attribut de γ, si $X.a$ est une occurrence d'attribut de sortie (resp. d'entrée) de α alors si b est synthétisé alors $X.a.b$ est un attribut de sortie (resp. d'entrée) de γ, s'il est hérité $X.a.b$ est un attribut d'entrée (resp. de sortie) de γ.

Dans le cas présent, α n'a pas d'attributs importés et ses attributs exportés sont du type de l'axiome Z_β du gène. Ainsi, les occurrences d'attributs $X.a.b$ de γ où $\mathcal{C}or(X) = Z_\beta$ ne peuvent être que telles que :
$A_\alpha(X) = \{ \; a \; \}$ où $type(a) = Z_\beta$ or $Z_\beta.b \in A_\beta$ ne peut être que synthétisé et déjà exporté dans β puisque les attributs de l'axiome d'une GAO sont exportés par définition.

On peut se contenter ici de l'application de la première des deux règles précédemment citées et A_γ contient donc les mêmes attributs importés et exportés que le gène β, y compris les attributs exportés correspondant aux occurrences d'attributs synthétisés de la racine de chacun des arbres de β construits sur un arbre de α.

Pour ce qui concerne la classe de la GAO γ engendrée, le résultat S$\circ$FNC $\subset$ FNC peut s'appliquer également dans ce cas. En effet, α est toujours purement synthétisée et la méta-composition ne change rien au caractère FNC des dépendances contenues dans la GAO β et n'en ajoute aucune.

VI.3 Stratégie d'instanciation de plusieurs gènes

Le processus d'instanciation de plusieurs gènes sur une GAO d'entrée δ passe par deux étapes. La première consiste à effectuer l'instanciation de chaque gène sur la grammaire G de base de δ suivant le processus décrit dans la section VI.2 précédente. On se retrouve alors avec autant de GAO générées que de gènes.

La caractéristique de ces GAO est qu'elles sont toutes basées sur la grammaire G qui est aussi celle de δ. D'autre part, les attributs importés et exportés de chacune peuvent être associés à des attributs importés et exportés de la GAO d'entrée. En effet ceux des gènes avaient déjà été sujet à une telle association, et, moyennant le renommage effectué par la méta-composition, ceux des GAO générées sont les mêmes. Cette correspondance permet de construire une table contenant ces informations d'interfaçage.

Il faut maintenant générer la sémantique correspondant à la "somme" des sémantiques de chacune des GAO. C'est ce qu'est chargé de faire l'opérateur $\bigoplus$ décrit dans la section suivante.

La figure VI.2 nous donne une idée de la stratégie d'instanciation de trois gènes sur une GAO d'entrée δ pour produire une GA Δ.

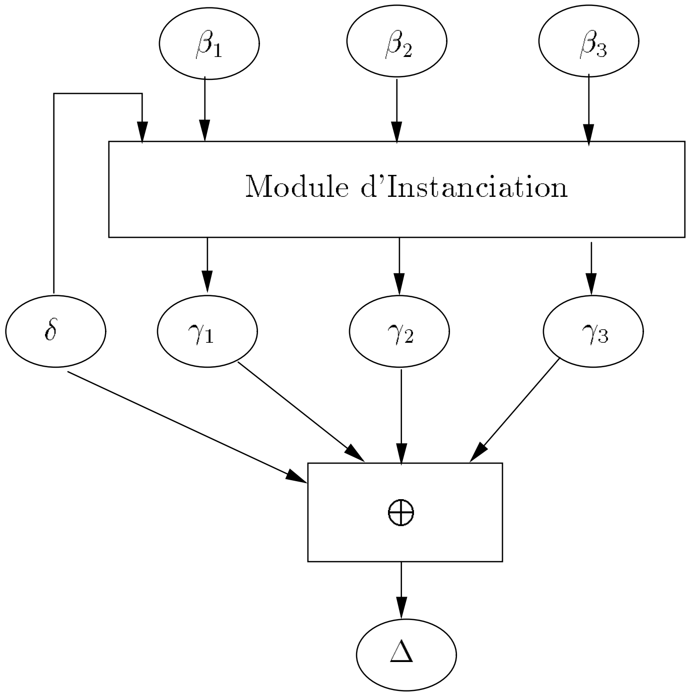


Figure VI.2 : Stratégie d'instanciation de plusieurs gènes.

VI.3.1 L'opérateur $\oplus$

On dispose ici de GAO γ_i qui correspondent à l'instanciation de n gènes β_i sur la grammaire G de la GAO d'entrée δ. On dispose également d'une table de correspondances entre les attributs importés et exportés de chaque GAO générées γ_i et la GAO d'entrée δ.

Le fait que toutes ces GAO soient basées sur la même grammaire de base G nous donne une idée simple de l'opération à effectuer pour n'obtenir qu'une seule GA en sortie. Cette opération effectuant la somme des sémantiques est notées $\bigoplus$.

Cet opérateur ne fait qu'une projection de chacune des GAO générées γ_i sur la GAO d'entrée δ :

- les attributs importés et exportés de chaque γ_i sont renommés suivant la table des correspondances. Ils peuvent éventuellement être reliés par ce nom dans δ à d'autres GAO γ_j
- pour chaque production de G, les règles sémantiques de chaque γ_i sont ajoutées à celles existant déjà dans δ.

La figure VI.3 permet de visualiser le résultat de cette opération sur des arbres de façon simple.

On remarque que tous les attributs importés et exportés de δ dans cet exemple sont associés à des attributs exportés et importés des deux GAO γ_1 et γ_2. Le résultat obtenu alors ne contient plus d'attributs ni importés, ni exportés. Mais ceci n'est pas une règle, et le résultat de l'opérateur $\bigoplus$ peut très bien en comporter.

L'opérateur $\bigoplus$ fournit en résultat une GAO Δ :

$$(\delta , \gamma_1 , \dots , \gamma_i , \dots , \gamma_n) \xrightarrow{\bigoplus} \Delta$$

Il est peut-être intéressant de déterminer les propriétés que peut bien avoir cette GAO résultat. C'est ce que l'on va essayer de caractériser dans la section suivante.

VI.3.2 Propriétés

Afin de pouvoir déterminer les caractéristiques de la GAO résultat Δ il est nécessaire de se baser sur ce que l'on connait, c'est-à-dire la GAO d'entrée δ et les GAO γ_i produites.

Or le problème qui se pose concerne l'interface entre ces GAO. En effet, elles peuvent être liées entre elles par des attributs exportés et importés ce qui implique des dépendances supplémentaires et peut-être même des circularités.

Afin de caractériser cela, nous avons besoin de définir une notion de graphe de dépendance appropriée à notre situation.

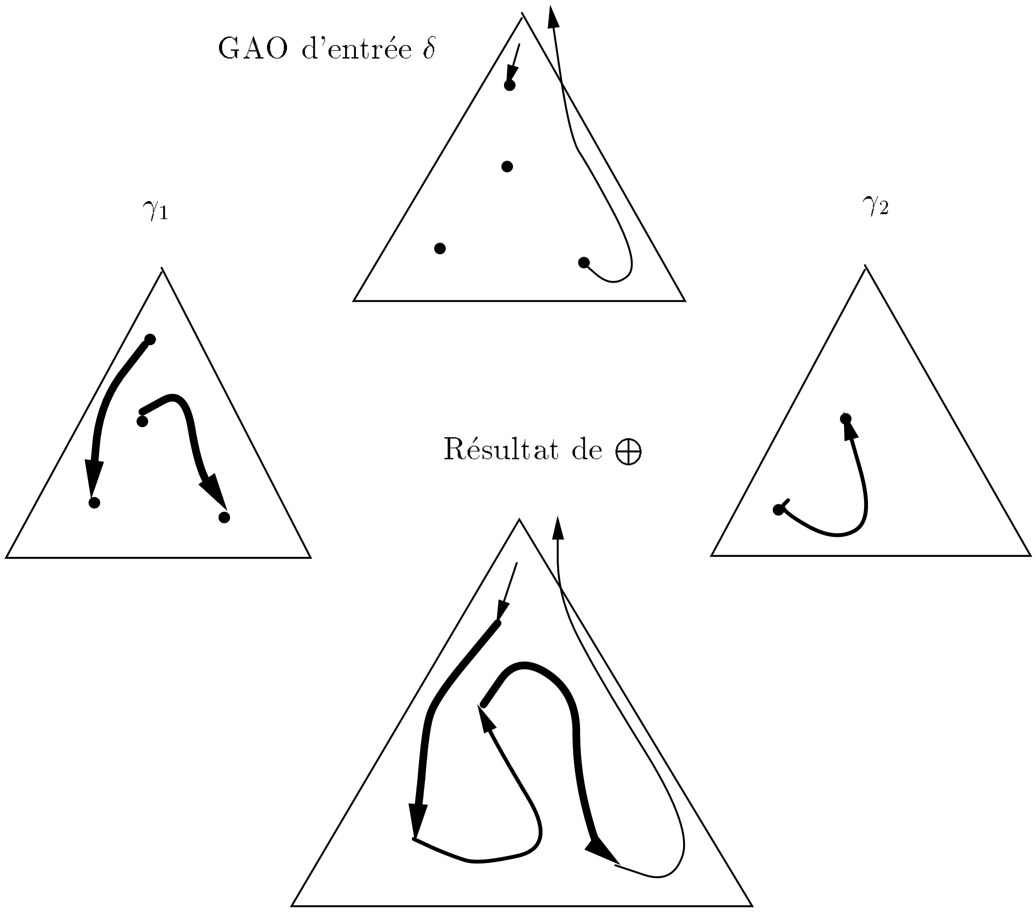


Figure VI.3 : Schéma d'application de $\bigoplus$ sur des arbres.

Le contexte ici est une GAO α où A_{in} est l'ensemble de ses attributs importés et A_{out} l'ensemble de ses attributs exportés.

La notion de graphe de dépendances local à chaque production p, qui schématise les relations entre les attributs, est introduite section II.1.1.2, page 13 :

Le graphe de dépendance local noté $D(p)$ est celui de la relation $\underset{p}{\rightarrow}$ dont les sommets sont les éléments de $W(p)$, l'ensembles des occurrences d'attributs $X.a$ d'une production p.

Pour chaque occurrence d'attribut de sortie de p, $X.a \in OW(p)$, on définit alors l'ensemble $Dep/_p(X.a)$ des occurrences d'attributs qui permettent de la définir :

$Dep/_p(X.a) = \{\ Y.b \mid Y.b \in IW(p)\ ,\ Y.b \underset{p}{\rightarrow} X.a\ \}$

De la même façon, on aimerait pouvoir obtenir pour chaque attribut exporté dans une GAO l'ensemble des attributs importés dont il dépend.

On sait par définition que : $Dep/_p(X.a) = \emptyset$ pour chaque production p où X

apparaît lorsque $X.a$ est une occurrence d'attribut *in*.

On sait aussi que : $\nexists Y.b$ t.q. $X.a \in Dep/_p(Y.b)$ dans aucune production p où X apparaît lorsque $X.a$ est une occurrence d'attribut *out*.

À partir de ces données, on est capable de calculer pour chaque attribut *out* d'une GAO l'ensemble des attributs *in* qui peuvent participer à son calcul.

On note $Dep(X.a)$ cet ensemble, avec $X.a \in A_{out}$, et on l'obtient avec l'algorithme suivant :

Algorithme VI.3.1 :

> **début**
> **pour chaque** $X.a \in A_{out}$ **faire**
>> $Dep(X.a) \leftarrow \emptyset$
>> *convergence* $\leftarrow$ *false ;*
>> **tant que** *non convergence* **faire**
>>> *convergence* $\leftarrow$ *true ;*
>>> **pour chaque** *production p t.q.* $X.a \in OW(p)$ **faire**
>>>> **pour chaque** $Y.b \in Dep/_p(X.a)$ **faire**
>>>>> **pour chaque** *production* p' *t.q.*
>>>>> $Y.b \in OW(p')$ **faire**
>>>>>> **si** $Dep/_{p'}(Y.b) \not\subseteq Dep(X.a)$ **alors**
>>>>>>> $Dep(X.a) \leftarrow Dep(X.a) \cup Dep/_{p'}(Y.b)$ *;*
>>>>>>> *convergence* $\leftarrow$ *false*
>>>>>> **fin si**
>>>>> **fin pour**
>>>> **fin pour**
>>> **fin pour**
>> **fin tant que**
> **fin pour**
> **fin**

On obtient pour chaque occurrence d'attribut $X.a \in A_{out}$ un ensemble d'occurrences d'attributs de A_{in}. A partir de ces ensembles, on définit la notion de graphe de dépendance particulière dont on a besoin : un **Graphe d'I/O**, noté ainsi car il est basé sur les attributs *in* et *out*.

Définition VI.3.1 : Graphe d'I/O

On appelle **Graphe d'I/O** *d'une GAO* α*, le graphe dont les sommets* *sont les occurrences d'attributs importés et exportés de α et tel que :*

$$Y.b \in Dep(X.a) \iff Y.b \to X.a$$

♣

On peut construire un tel graphe pour chacune des GAO que l'on a. Il est d'ailleurs à noter que le graphe I/O des γ_i, moyennant les renommages de la méta-composition, est le même que celui des gènes β_i correspondants, et ce toujours grâce au fait que la fonction de couplage est purement synthétisée et n'introduit aucune dépendance.

Ainsi, en mettant ces graphes en relation par le renommage des attributs exportés et importés selon la table des correspondances, on obtient celui de la GAO résultante Δ. La classe de Δ peut dans certains cas être déduite de ce nouveau graphe I/O et des classes des GAO initiales.

Proposition VI.3.1 :

- *Si chaque GAO est NC et s'il n'existe pas de circularité dans le graphe I/O de la GAO résultat, alors la GAO résultante est NC.*
- *Si chaque GAO est FNC et s'il n'existe pas de circularité dans le graphe I/O de la GAO résultat, alors la GAO résultante est FNC.*

Cette proposition est démontrée dans [Rou93]. Elle donne une condition suffisante mais pas nécessaire pour que la classe de la GAO résultante soit NC ou FNC. Il peut donc se trouver que la GAO Δ obtenue soit NC (resp. FNC) sans que le graphe I/O résultat de l'union des graphes I/O des GAO γ_i et δ soient non circulaires.

La connaissance de ce nouveau graphe I/O permet néanmoins de détecter les cas de non-circularité où l'on peut appliquer les GAO les unes après les autres, et dans ce cas, la méta-composition de ces GAO selon cet ordre est équivalente à l'opérateur $\oplus$.

VI.4 Conclusion

On dispose ici d'une méthode d'instanciation de plusieurs gènes sur une GAO. Cette méthode permet d'obtenir une GAO unique en résultat pour laquelle on construit ensuite un évaluateur.

Il est cependant possible d'envisager une compilation séparée totale si l'on construit les évaluateurs de chacune des GAO produites (que l'on peut déduire des évaluateurs des gènes) pour les composer ensuite selon les interfaces définies.

Cette optique ne nous paraît cependant pas correspondre avec la vision très "méta" que nous avons adopté jusqu'à maintenant. C'est pourquoi nous n'avons pas, dans ce mémoire, poussé l'étude de la séparation des concepts jusqu'aux évaluateurs.

Chapitre VII

Extension des possibilités d'instanciation d'une GA Générique

Ce chapitre concerne l'augmentation de puissance d'expression de l'instanciation d'un gène. En effet, il s'agit d'aller plus loin dans la définition de la fonction de couplage.

Cela concerne un point précis du processus de définition : la correspondance avec le gène. Il faut associer chaque production réduite non plus seulement avec une production du gène, mais avec un sous-arbre interne du gène. Le processus modifié est expliqué dans le section VII.1 suivante.

Cette association différente nécessite bien sûr une adaptation de la construction de la fonction de couplage puisque le but est maintenant de construire un arbre du gène sur un arbre de la grammaire d'entrée G. La section VII.2 décrit ce processus.

VII.1 Définition de la fonction de couplage

Nous ne reprendrons pas ici l'étape de description des correspondances entre les symboles et attributs de la grammaire d'entrée G et les symboles et attributs de β, qui aboutissent à la description de symboles et d'attributs clefs dans G. En effet, celle-ci n'est pas modifiée, de même que l'étape de construction des régions connexes qui en découle.

A ce point du processus on dispose d'une grammaire réduite $G' = (N', T', Z', P')$ et de l'ensemble des régions connexes R associé. L'étape décrite maintenant permet de trouver les parties du gène correspondantes à chaque production réduite $p \in P'$ et donc aux sous-arbres $r \in R_p$ associés.

VII.1.1 Correspondance avec le gène

L'étape de traduction d'une production réduite dans le langage du gène β se fait de la même façon que dans la section V.2.3, mais, si l'on a une production réduite :

$A \rightarrow B\ C\ \Omega\ D$ où $A, B, C, D \in N_k$ et $\Omega \in T_k$

et que l'on a également :

$\mathcal{C}or(A) = \mathcal{C}or(C) = a$, $\mathcal{C}or(B) = b$, $\mathcal{C}or(D) = d$, où $a, b, d \in N_\beta$

et $\mathcal{C}or_t\ (\Omega) = \omega$, où $\omega \in T_\beta$

la question que l'on se pose n'est plus :

$$a \longrightarrow b\ a\ \omega\ d\ \ \in P_\beta ?$$

mais celle-ci :

$$a \stackrel{+}{\Longrightarrow} b\ a\ \omega\ d\ ?$$

Il faut donc déterminer le sous-arbre de dérivation permettant d'obtenir une chaîne $b\ a\ \omega\ d$, à partir de a dans β.

On veut donc construire des couples (p', r_β), $p' \in P'$ et r_β un sous-arbre interne de du gène.

Le problème se ramène à la recherche d'une phrase dans un langage. En effet, en considérant un "sous-langage" du gène issu du membre gauche de la question, le membre droit peut se ramener à une phrase de ce langage.

Ce sous-langage peut se décrire par la grammaire du gène augmentée d'un nouvel axiome qui dérive dans le membre gauche de la "question", et de nouveaux terminaux associés à chaque non-terminal de la partie droite.

Ainsi, dans l'exemple, on ajouterait à la syntaxe du gène :

$Z \rightarrow a$
$a \rightarrow T_a$
$b \rightarrow T_b$
$d \rightarrow T_d$

où $Z \in N_\beta$ et $T_a, T_b, T_d \in T_\beta$

Ce qui induit une nouvelle question :

$Z\ \stackrel{+}{\longrightarrow}\ T_b\ T_a\ \omega\ T_d$?

Afin de trouver la plus petite dérivation possible, il faut également supprimer de la syntaxe du gène les règles dont le membre droit est $\mathcal{E}$, et celles du type $X \rightarrow X$ avec $X \in N_\beta$ qui produisent des branches infinies.

Cette nouvelle grammaire, une fois rendue propre, définit bien le langage que l'on veut obtenir et peut servir de base à la construction d'un automate (par un analyseur syntaxique quelconque), qui permet de savoir si la nouvelle phrase appartient à ce langage, et surtout de déterminer quelles sont les dérivations utilisées.

On connaît d'avance une partie de la dérivation que l'on veut obtenir. Dans l'exemple précédent, on doit avoir :

$Z \rightarrow \ a \ \stackrel{+}{\Longrightarrow} \ b \ a \ \omega \ d \ \rightarrow \ b \ a \ \omega \ T_d \ \rightarrow \ b \ T_a \ \omega \ T_d \ \rightarrow \ T_b \ T_a \ \omega \ T_d$

La dérivation qui nous intéresse est donc extraite de celle fournie par l'analyseur, en supprimant les productions que l'on avait ajoutées au préalable.

Ce processus doit être appliqué pour toutes les productions réduites, ce qui implique la création d'un analyseur à chaque fois.

Or, il est plus réaliste d'envisager d'effectuer un traitement préalable sur la grammaire du gène, afin de construire un unique analyseur capable de répondre à toutes les questions que l'on pourrait être amené à lui poser.

VII.1.2 Transformation de la grammaire du gène

Ainsi, il faut transformer la grammaire du gène de manière à ce que l'on puisse éventuellement obtenir une dérivation de n'importe lequel des non-terminaux du gène et que cette dérivation soit la "plus petite" possible.

On doit alors :

- ajouter un nouvel axiome, et le faire dériver directement dans tous les non-terminaux du gène,
- supprimer toutes les productions qui dérivent directement dans $\mathcal{E}$.
- supprimer toutes les productions du type $X \rightarrow X$ où $X \in N_\beta$.

Si on laisse la nouvelle grammaire dans cet état, elle ne peut qu'engendrer des ambigüités. En effet, la grammaire étant propre au départ, tous les non-terminaux étaient accessibles depuis l'ancien axiome : il y a donc ici toujours deux chemins possibles pour accèder à un non-terminal différent de l'ex-axiome, celui qui passe par cet ex-axiome, et celui, plus direct, qui contient la nouvelle production qui, du nouvel axiome, dérive dans ce non-terminal.

Afin d'éviter ce type de problème, on va introduire un concept de point d'entrée, c'est-à-dire que l'on va introduire des terminaux particuliers permettant de déterminer de manière exclusive quelle sera la première production utilisée.

Ainsi, dans l'exemple précédent, il s'agira d'ajouter à la grammaire du gène :

$Z \rightarrow z$ avec z l'ancien axiome
$Z \rightarrow a$
$Z \rightarrow b$
$Z \rightarrow d$
$a \rightarrow T_a$
$b \rightarrow T_b$
$d \rightarrow T_d$

avec $N'_\beta = \{\ z, a, b, d\ \} \cup \{\ Z\ \}$ et $T'_\beta = \{\ T_z, T_a, T_b, T_d\ \}$ et les points d'entrée $Pt = \{\ \#z, \#a, \#b, \#d\ \}$

Ce qui induit une nouvelle question correspondant à la production réduite : $A \rightarrow B\ C\ \Omega\ D$:

$$\begin{array}{l} Z \overset{+}{\Longrightarrow} \#a \quad T_b\ T_a\ \omega\ T_d\ ? \\ \qquad\ \ \uparrow \\ \text{point d'entrée} \end{array}$$

Le fait de mettre des points d'entrée dans cette grammaire fait que l'on ne considère à chaque fois qu'une partie de l'ancienne grammaire (au plus, la grammaire en entier si le point d'entrée correspond à l'ancien axiome).

Intuitivement on peut penser que chacun des analyseurs engendrés pour ces sous-grammaires sera au pire de la même classe que l'analyseur de la grammaire initiale, ce qui peut amener à caractériser le nouvel analyseur en fonction de l'analyseur de la grammaire initiale.

Nous ne nous intéresserons pas ici à ce sujet, car il nous suffit de savoir que l'on a un analyseur qui va nous permettre d'obtenir toutes les dérivations correspondant aux productions réduites, si elles existent.

Le cas où l'on a plusieurs dérivations possibles associées à une même production réduite indique que plusieurs sémantiques peuvent être appliquées sur la région connexe associée $r \in R_{p'}$. Si ces sémantiques sont toutes équivalentes, cela ne constitue pas un problème, mais si ce n'est pas le cas, alors cela signifie que la fonction de couplage n'est pas unique. Dans ces deux cas, on décide de ne pas poursuivre le processus d'instanciation car on ne peut déterminer une fonction de couplage unique, ce qui peut révéler une erreur dans la mise en correspondance effectuée au départ entre la grammaire G et le gène β.

Si pour une production réduite $p' \in P'$ on obtient la dérivation r_β par l'analyseur, alors on a le couple (p', r_β).

On construit alors un ensemble de nouveaux couples, *Assoc* :

$$Assoc = \{\ (r, r_\beta) \mid r \in R_{p'} \text{ et on a le couple } (p', r_\beta)\ \}$$

Si l'on est parvenu à effectuer ces associations, on sait alors que la fonction de couplage définie par la description des correspondances entre les symboles de G et les symboles de G_β est valide et unique.

VII.2 Construction de la fonction de couplage

Il s'agit toujours ici de produire une grammaire attribuée de couplage α qui à partir de la grammaire d'entrée G produit un arbre de la grammaire G_β du gène, ou plusieurs arbres dans le cas du chapitre VI.

Le but est maintenant d'associer à chaque sous-arbre r un ensemble de possibilités de construction de l'arbre r_β du gène qui lui est associée dans l'ensemble *Assoc* construit section VII.1.

VII.2.1 Notations et définitions

La région connexe r, sous-arbre interne de $G = (N, T, P, Z)$, est décrite par ses nœuds. Chaque nœud u de r est étiqueté par un symbole X qui est soit un non-terminal de N soit un terminal de T pour certains nœuds feuilles.

L'arbre du gène associé r_β est lui aussi décrit par ses nœuds notés u_β. Chaque nœud u_β de r_β est étiqueté par un symbole Y qui est soit un non-terminal de N_β soit un terminal de T_β pour certains nœuds feuilles.

Comme pour le cas simple de la section V.3.1, on note $G/_r = (N/_r, T/_r, P/_r, Z/_r)$ la grammaire induite par la région connexe r. On définit également la grammaire induite par r_β selon la même méthode, et on la note $G/_\beta = (N/_\beta, T/_\beta, P/_\beta, Z/_\beta)$

On définit ce que représente une possibilités de construction de l'arbre r_β du gène sur r.

Définition VII.2.1 : Possibilité de construction

> *Une* possibilité de construction C *d'un arbre* r_β *sur une région connexe* r *est représentée par la grammaire attribuée* $(G/_r, A_C, F_C)$ *avec :*
>
> - A_C *est l'ensemble des attributs synthétisés déclarés sur les symboles* $X \in (N/_r \cup T/_r)$
> $A_C = \bigcup_{X \in (N/_r \cup T/_r)} \; A_C(X)$ *avec*
> $$A_C(X) \subseteq \{ \; \$Y \mid Y \in (N/_\beta \cup T/_\beta) \; \}$$
> - F_C *est l'ensemble des règles sémantiques permettant de calculer ces attributs de façon à obtenir la construction de l'arbre* r_β *comme valeur de l'attribut synthétisé de la racine* $Z/_r$
> $F_C = \bigcup_{p \in P/_r} \; F_C(p)$ *avec*
> $$F_C(p) = \{ \; f \mid f = transfert \; ou \; f = f/p_\beta \; avec \; p_\beta \in P/_\beta \}$$
>
> ♣

Les eléments dont on dispose avant de commencer la construction sont donnés par les relations $\mathcal{C}or$ et $\mathcal{C}or_t$ et permettent de connaître les attributs que l'on a sur certains symboles de $G/_r$.

Ainsi, on a A_{C_0} l'ensemble d'attributs tels que :

- $\forall X \in (N/_r \cap N_k)$, $A_{C_0}(X) = \{ \; \$Y | \; \mathcal{C}or(X) = Y$ et $Y \in N/_\beta \; \}$
  et, en particulier : $A_{C_0}(Z/_r) = \{ \; \$Z/_\beta \; \}$

- $\forall X \in (T/_r \cap T_k)$, $A_{C_0}(X) = \{ \ \$Y \mid \mathcal{C}or_t(X) = Y$ et $Y \in T/_\beta \ \}$
- $\forall X \in (N/_r \cap MORTS)$, $A_{C_0}(X) = \emptyset$
- $\forall X \in (T/_r \cap T_n)$, $A_{C_0}(X) = \emptyset$

Les seuls attributs à trouver sont ceux des non-terminaux neutres qui étiquettent les nœuds internes de r.

VII.2.2 Construction des possibilités pour une région connexe

Nous allons maintenant décrire l'algorithme qui construit un ensemble $Cons$ des possibilités de construction de l'arbre r_β, pour chaque arbre r :

$$Cons(r) = \{ \ C \mid C = (G/_r, A_C, F_C) \ \}$$

Le principe est un peu différent de celui énoncé section V.3.1. En effet, dans une possibilité C l'arbre r_β n'est pas construit sur une seule production de $P/_r$: chacune des fonctions f/p_β qui construisent les productions $p_\beta \in P/_\beta$ est utilisée une seule fois comme règle sémantique ou partie de règle sémantique dans une production $p \in P/_r$.

Les deux opérations décrites précédemment de transfert et de construction d'une production p_β ne sont plus exclusives : on peut avoir pour une production $p \in P/_r$ à la fois des règles sémantiques de transfert de certains attributs des symboles de $RHS(p)$ et des règles sémantiques de construction d'une production $p_\beta \in P/_\beta$.

La construction de cet ensemble $Cons$ des possibilités se fait toujours par un parcours en profondeur d'abord de l'arbre r.

Toute possibilité de construction $C = (G/_r, A_C, F_C)$ est construite par les extensions successives de la possibilité initiale $C_0 = (G/_r, A_{C_0}, \emptyset)$.

Afin de savoir quelles productions de $P/_\beta$ on peut construire sur un nœud de r pour une possibilité C, on utilise un ensemble de *nœuds attributs* associé à chaque nœud u de r et qui contient les nœuds de r_β que l'on a déjà construits dans le sous-arbre issu de u.

On le note

$$Attr_C(u) = \{ \ u^i_\beta \mid u^i_\beta \in r_\beta \ \}$$

qui correspond à la déclaration des attributs sur X :

$$A_C(X) = \{ \ \$Y \mid Y \in (N_\beta \cup T_\beta) \text{ avec } label(u^i_\beta) = Y \ \}$$

De la même façon que pour les ensembles $A_{C_0}(X)$ pour certains $X \in N/_r$, on peut définir des ensembles $Attr_{C_0}(u)$ pour certains nœuds u particuliers :

$\forall u \in r$ avec $label(u) = X \in (N/_r \cap N_k)$,

$Attr_{C_0}(u) = \{\ u^i/_\beta\ \}$ avec $\mathcal{C}or(X) = Y\ \}$ et $u^i/_\beta \in r/_\beta$ tel que $label(u^i/_\beta) = Y$

$\forall u \in r$ avec $label(u) = X \in (T/_r \cap T_k)$,
$Attr_{C_0}(u) = \{\ u^i/_\beta\ \}$avec $\mathcal{C}or_t(X) = Y\ \}$ et $u^i/_\beta \in r/_\beta$ tel que $label(u^i/_\beta) = Y$

Le schéma de l'algorithme de construction des possibilités est le même que celui décrit page 113, ainsi que la procédure **CONSTRUCTION** qui suit une stratégie ascendante de calcul.

Seule l'étape d'augmentation d'une possibilité C à partir du nœud u de l'arbre r se fait de façon différente. Il existe plus de deux types de possibilités d'augmentation. Lorsque plus d'une d'entre elles est applicable, il faut dupliquer la possibilité C en autant de possibilités nécessaires et augmenter chacune d'entre elles différemment.

Plaçons nous, lors du parcours de r sur un nœud u et pour une possibilité C de construction de r_β. On dispose alors des éléments suivants :

- la production $p = Prod(u)$,
- l'ensemble $Attr\_fils = \bigcup_{1 \leq i \leq Nbfils(u)} Attr_C(Fils(u,i))$
  où $Fils(u,i)$ donne le nœud numéro i de u, et $Nbfils(u)$ le nombre total des fils de u,
- et l'ensemble $A\_fils = \bigcup_{X \in RHS(p)} A_C(X)$

Il s'agit maintenant de construire les ensembles $Attr_C(u)$ possibles à partir de $Attr\_fils$ puis de vérifier qu'ils sont compatibles avec l'ensemble $A_C(label(u))$ s'il existe déjà.

A partir de cet ensemble $Attr\_Fils$, on doit construire un ensemble de possibilités de nœuds attributs pour le père u.

La première étape consiste à construire tous les nœuds que l'on peut obtenir à partir des nœuds attributs d'un $Attr\_Fils$ donné, que l'on appelle *ancêtres*.

Cet ensemble, ANC, est d'abord initialisé avec l'ensemble $Attr\_Fils$, ce qui correspond à dire que le transfert de tous les nœuds attributs est une des possibilités envisageables.

A cet ensemble, on va devoir ajouter tous les autres ancêtres que l'on peut obtenir à partir de ces nœuds.

L'algorithme décrit ci-dessous consiste à déterminer si tous les fils d'un nœud sont dans l'ensemble ANC. Si c'est le cas, on l'ajoute à cet ensemble. Ceci est fait d'après les noms des nœuds exprimés avec la notation de Dewey. On définit alors la fonction $long(u)$ qui donne le nombre d'entiers compris dans le nom de u, c'est à dire l'étage où se situe u dans l'arbre depuis la racine.

fonction Calcul_ANC (Attr_Fils : Ensemble de nœuds de r_β) :
Ensemble de nœuds de r_β ;

début
{ *Initialisation* }
$Anc \leftarrow Attr\_Fils$
$Max \leftarrow long\_max(Anc)$ { *longueur maximale des noms des nœuds* u' *de* ANC}
$Cour \leftarrow Max$ { *longueur maximale des nœuds non traités* }
tant que $Cour \neq 0$ **faire**

pour chaque $u' \in Anc$ *tel que* $(long(u') = Cour)$
et ($\neg$ *traité*(u')) **faire**

$in \leftarrow 0$
pour i **de** *1* **à** $Nbfils(Pere(u'))$ **faire**
si $Fils(Pere(u'), i) \in Anc$ **alors**
traité(u') $\leftarrow$ *true*
$in \leftarrow in + 1$
fin si
fin pour
si $in = Nbfils(Pere(u'))$ **alors**
$Anc \leftarrow Anc \cup \{Pere(u')\}$
fin si

fin pour
$Cour \leftarrow Cour - 1$

fin tant que
Calcul_ANC $\leftarrow Anc$
fin

Pour faire les choses correctement, c'est-à-dire en n'oubliant aucun nœud de l'ensemble, on commence par les nœuds dont la longueur l du nom est la plus grande (ce qui correspond à l'arbre le plus profond), on ajoute, si possible, les nœuds pères (dont le nom est de longueur $l-1$), et on poursuit ainsi la remontée vers les nœuds les plus “hauts”(les nœuds déjà utilisés sont marqués par un booléen : *traité*).

En une passe, on a ainsi généré tous les ancêtres possibles d'un ensemble de possibilités $Attr\_Fils$.

Il faut maintenant structurer cet ensemble ANC de manière à reconnaître, parmi tous ces nœuds, ceux qui ne peuvent être produits ensemble. Ainsi, on ne peut obtenir à la fois un nœud et un de ses fils.

On obtient alors pour le nœud père u l'ensemble $POSS\_Pere$ des $Attr^i_C(u)$ possibles tels que :

$$Attr^i_C(u) = \{\ u_\beta \mid u_\beta \in r_\beta \text{ et } \forall u'_\beta \in Attr^i_C(u) \text{ tel que } u'_\beta \neq u_\beta, \\ \text{on a } \neg Fils?(u'_\beta, u_\beta) \text{ et } \neg Fils?(u_\beta, u'_\beta)\ \}$$

où $Fils?(u, u')$ est une fonction qui rend vrai si u' est fils de u :

$Fils?(u, u') \ = \ (Pere(u') = u)$.

Le principe est le suivant : à partir de chaque élément de ANC, on essaye de construire tous les sur-ensembles qui ont ces propriétés.

Il s'agit de ne pas construire deux fois le même ensemble, c'est pourquoi il est nécessaire de numéroter les Nb_anc éléments de ANC à partir de 1 (la taille de ANC est au maximum le nombre de nœuds de l'arbre du gène en question), et d'examiner les éléments dans cet ordre.

Pour obtenir un élément précis de cet ensemble, on définit la fonction *Élem(E,i)*, qui donne l'élément numéro i d'un "ensemble" numéroté E.

La fonction récursive *Surens* qui effectue ce traitement est décrite ci-dessous. Elle doit, pour construire toutes les parties de ANC, être appelée avec *Surens(Ø,ANC, 0)*. Ces parties de ANC sont des ensembles $Attr^i_C(u)$ si elles ne sont pas déjà inclues dans une partie plus grande.

```
fonction Surens (E , ANC: Ensemble de nœud de r_β ; i : entier) : booléen ;

    début
    si i ≠ 0 alors

        Élément ← Élem(i,ANC)
        NouvE ← E ∪ Élement

    sinon

        NouvE ← E

    fin si
    si ∃Attr^i_C(u) ∈ POSS_pere
     tel que NouvE ⊂ Attr^i_C(u) et NouvE ≠ Ø alors

        Surens ← true

    sinon

        Possible ← true
        pour chaque élt ∈ NouvE tel que élt ≠ Élement faire

            Possible ← Possible et ((¬ Fils?(élt,Élement))
                                    et (¬ Fils?(Élement,élt)))

        fin pour
        si Possible alors

            Test ← false
            pour k de i+1 à Nbanc faire
                Test ← Test ou Surens(NouvE,ANC,k)
            fin pour
            si ¬ Test alors
                POSS_père ← POSS_pere ∪ { NouvE }
            fin si
            Surens ← true

        sinon

            Surens ← false

        fin si

    fin si
    fin
```

La phase suivante consiste à vérifier que les types des nœuds attributs contenus dans un ensemble $Attr^i_C(u)$ de $POSS\_pere$ sont compatibles avec les attributs éventuellement déjà déterminés par une précédente analyse (sur un autre nœud étiqueté par le même non-terminal que le nœud d'arbre u que l'on traite).

Ces attributs, s'ils existent, sont dans l'ensemble $A_C(label(u))$.

La "correction" d'un ensemble $Attr^i_C(u)$ vis à vis de l'ensemble $A_C(label(u))$ se définit par le fait que l'on a autant d'éléments u' dans $Attr^i_C(u)$ tels que $label(u') = Y \in (N/_\beta \cup T/_\beta)$ que d'attributs $\$Y$ dans $A_C(label(u))$.

- Si u est la racine de r alors on connaît déjà un ensemble $Attr_C(u) = Attr_{C_0}(u)$ et seul l'ensemble $Attr^i_C(u) = Attr_{C_0}(u)$ est valide. La possibilité C est alors augmentée par l'ajout, dans $F_C(Prod(u))$ des fonctions de transfert et de construction d'arbres qui correspondent à l'obtention de ce $Attr^i_C(u)$ à partir de l'ensemble $Attr\_Fils$.

- Si u n'est pas la racine :

  - s'il existe déjà un ensemble $A_C(label(u))$, on ne conserve dans $POSS\_Pere$ que les ensembles $Attr^i_C(u)$ corrects vis-à-vis de $A_C(label(u))$. Il se peut également que l'on ait déjà un ensemble $F_C(Prod(u))$: on ne conserve alors dans ce nouveau $POSS\_Pere$ que les ensembles $Attr^i_C(u)$ obtenus à partir de $Attr\_Fils$ par les mêmes règles sémantiques que celles de $F_C(Prod(u))$.
  - s'il n'existe pas d'ensemble $A_C(label(u))$, alors tous les ensembles de $POSS\_Pere$ sont considérés comme des possibilités valides.

  La possibilité de construction C de r_β sur r est ensuite dupliquée en n exemplaires, avec n le nombre d'ensembles $Attr^i_C(u) \in POSS\_Pere$. Chacune de ces nouvelles possibilités C_i est augmentée en fonction de l'ensemble $Attr^i_C(u)$ qu'on lui associe :

  - $Attr_{C_i}(u) = Attr^i_C(u)$
  - $A_{C_i}(label(u) = \{\ \$Y \mid Y = label(u_\beta)\ ,\ u_\beta \in Attr^i_C(u)\ \}$
  - $F_{C_i}(Prod(u))$ contient les fonctions de transfert et de constructions d'arbres utilisées pour passer de $Attr\_Fils$ à $Attr^i_C(u)$

  La possibilité C est alors remplacée dans $Cons$ par les n possibilités C_i.

À la fin du parcours de l'arbre r, on obtient un ensemble de possibilités de constructions $Cons$, tel que $\forall C \in Cons$, on a des ensembles $A_C(X)$ d'attributs pour chaque $X \in N/_r$.

La correction de la déclaration et de la définition des attributs se fait de la même manière que celle décrite section V.3.2, page 118.

De la même façon, on obtient un ou plusieurs ensembles *Compat* de possibilités de constructions compatibles entre elles pour toutes les régions connexes. S'il n'en reste pas, c'est que l'on ne peut obtenir une construction automatique de la fonction de couplage.

La génération de la sémantique se fait exactement de la même manière que dans la section V.4, page 119.

Le chapitre C de l'annexe, page 193, nous donne un exemple d'instanciation de deux gènes GAO sur une GAO d'entrée en tenant compte de cette extension dans l'instanciation d'un gène.

VII.3 Conclusion

On est parvenu ici à augmenter le champ d'utilisation d'un gène, en permettant son instanciation sur des grammaires qui sont encore plus éloignées syntaxiquement de la grammaire du gène.

Cela permet d'étendre le principe d'instanciation en l'éloignant du simple renommage pratiqué dans la plupart des cas lorsqu'il s'agit de faire de la réutilisation de module. Il semble ici que l'on soit allé suffisamment loin dans le processus automatique et que l'on ait atteint la limite entre une adaptation poussée et une adaptation à tout prix.

Il existe cependant une autre possibilité pour construire une fonction de couplage toujours synthétisée mais poussant l'adaption un peu plus loin, c'est d'aller jusqu'à produire des règles sémantiques conditionnelles. Cela permet de faire du cas par cas pour chaque production lorsqu'on ne trouve pas de sémantique commune. Cette solution, comme on l'a vu précédemment (section V.4 et [Rou93]), peut induire le fait que l'on n'a plus la garantie de la propriété : $S \circ FNC \subset FNC$.

Chapitre VIII

Conclusion

Nous avons discuté d'une solution permettant aux GA de devenir un outil de spécification d'applications à la fois performant et facile d'emploi. On constate que la théorie originelle de Knuth [Knu68] est la base de nombreux travaux permettant à cette méthode de devenir un outil très complet.

Les travaux réalisés au sein du projet ChLoE vont tous dans ce sens. Tous les aspects de cette méthode ont été ou sont abordés : que ce soit le côté production d'évaluateurs optimisés, parallèles ou incrémentaux ou bien le côté utilisation et écriture de GA (Olga et ses mécanismes de modularité, Olga++), ou encore l'augmentation de la puissance d'expression avec le traitement de la sémantique dynamique et l'intégration de Fnc–2 à Centaur.

Il ne nous restait qu'à nous pencher sur l'aide à la conception d'applications par la méthode des GA. C'est maintenant chose faite dans la mesure où l'on dispose d'un outil permettant d'appliquer la sémantique d'une ou plusieurs GA sur une autre. On peut envisager deux manières d'exploiter ce mécanisme. On peut le voir comme un simple outil de modularité et l'utiliser en écrivant les différents modules utiles sur une syntaxe réduite, mais directement issue de la syntaxe de base de l'application (ce qui se rapproche de la vision de [Kle91]). Mais on peut l'utiliser, et c'est là son principal intérêt, pour décrire la conception d'une application comme un jeu de construction. Cela ne peut être applicable que si l'on dispose déjà d'un ensemble de ces différents modules génériques afin de choisir parmi eux ceux que l'on veut instancier. Ce qui nous amène à parler de la construction de ces modules gènes. Là encore deux solutions s'offrent à nous : la construction directe de chacun de ces modules après expertise de leurs domaines d'application, ou bien la récupération d'une expertise déjà réalisée sur une application donnée. Il s'agit alors simplement d'extraire la sémantique à partir de la spécification des éléments (symboles et attributs) utiles à la résolution du problème.

Le mécanisme décrit ici a été développé surtout dans l'optique de construire des GA. On a affaire à tous les stades du processus, couplage, méta-composition, opérateur $\oplus$, à des GA que l'on peut obtenir sous une forme lisible et éventuellement modifiable. Cette façon de faire nous a bien entendu un peu limité dans la mesure où travailler sur des GA implique certaines contraintes.

Ainsi la fonction de couplage décrite par une GA induit des restrictions au niveau de son existence en tant que GA ainsi que dans sa construction. La puissance d'expression est forcément plus limitée dans la mesure où l'instanciation est moins adaptable que celle décrite dans [FMY92] par exemple. Il est néanmoins possible d'envisager d'aller un peu plus loin dans la souplesse fournie en autorisant l'emploi de fonctions conditionnelles dans la construction de la fonction de couplage. La nécessité d'introduire cette extension dépendra des résultats obtenus sans. Nous manquons encore d'expérience pour nous avancer sur ce sujet.

Notre but maintenant n'est pas seulement d'offrir un service permettant de produire une GA (ou une GAO) à partir de plusieurs gènes et d'une GAO d'entrée afin qu'elle puisse être regardée voir modifiée et donner un certain nombre d'indications sur ces propriétés si on le peut. Il consiste aussi à introduire le moyen de basculer à tout moment vers une production d'évaluateurs.

En effet, les travaux décrits par Gilles ROUSSEL dans [Rou93] permettent de produire un évaluateur à séquences de visites basé sur les évaluateurs à séquences de visites des grammaires que l'on compose. Le résultat est un évaluateur différent de celui que l'on aurait obtenu directement d'après le résultat de la méta-composition des grammaires, lorsque celle-ci est possible. On est maintenant capable également de produire l'évaluateur du résultat de l'opérateur $\bigoplus$, uniquement à partir des évaluateurs de chacune des grammaires dont on fait la “somme”. Ces évaluateurs sont appelés à la demande de l'évaluateur maître produit. Tout ceci nous permet d'obtenir une compilation séparée réelle.

Annexe A

Application à un exemple: Pl0

A.1 Description informelle du langage

PL0 a un seul type de données: les *entiers*. C'est un langage à structure de *blocs*.

Chaque *bloc*, ou unité de compilation, est formé d'une liste (éventuellement vide) de déclarations suivie d'une liste d'instructions.

Il est possible de déclarer des constantes et des variables de type *entier*.

La définition d'une procédure n'est constituée que de son nom et de son corps: elle ne peut avoir aucun paramètre.

Cette présence de procédures introduit le concept de localité des objets (constantes, variables, procédures): chaque objet est considéré comme local à la procédure, ou, plus exactement au bloc dans lequel il est déclaré une seule fois, et sa portée s'étend aux blocs englobés, à moins qu'il n'y soit redéfini.
Les procédures peuvent être récursives. Les instructions comprennent l'affectation d'entiers, une conditionnelle simplifiée (*if-then-end if*), une boucle (*while*), et un appel de procédure (*call*), basées sur les concepts classiques d'exécution séquentielle, conditionnelle et répétitive.

Les expressions manipulées par ces instructions sont formées de variables (dont on prend alors la valeur), de constantes entières, des opérateurs arithmétiques et relationnels conventionnels, ainsi que des parenthèses de désambigüation.

La précédence des opérateurs est ainsi décrite:
$1: \; * \, , \; /$
$2: \; + \, , \; -$
$3: \; < \, , \; > \, , \; \leq \, , \; \geq \, , \; = \, , \; \neq$

Tous les identificateurs utilisés dans un bloc doivent y être connus statiquement selon les règles de visibilité, et leur emploi doit correspondre à leur déclaration.

Il est bien évident que PL0 n'est pas un langage puissant, mais il permet d'avoir un exemple complet d'un traducteur de langage structuré vers un code machine donné.

Voici, pour mieux se rendre compte de la forme du langage PL0, un exemple de programme simple qui donnera lieu à une traduction ultérieure en code PL0:

```
const m=7,n = 85;
var x,y,z,q,r ;
procedure multiply;
    var a,b ;
    begin
    a := x; b := y; z := 0;
    while b > 0 do
        begin
        if odd b then z := z+a;
        a := 2*a; b := b/2
        end
    end;
procedure divide ;
    var w ;
    procedure rage ;
        var c,d ;
        begin
        c := x; d := w
        end;
    begin
    r := x; q := 0; w := y;
    while w ≤ r do w := 2*w ;
    while w>y do
        begin
        q := 2 ∗ q; w := w / 2;
        if w ≤ r then
            begin
            r := r − w; call rage ; q := q + 1
            end
        end
    end ;
procedure gcd;
    var f,g ;
    begin
    f := x; g := y;
    while f ≠ g do
        begin
        if f < g then g := g − f;
        if g < f then f := f − g
        end ;
    z := f
    end;
begin
x := m; y := n; call multiply;
x := 25; y := 3; call divide;
x := 84; y := 36; call gcd
```

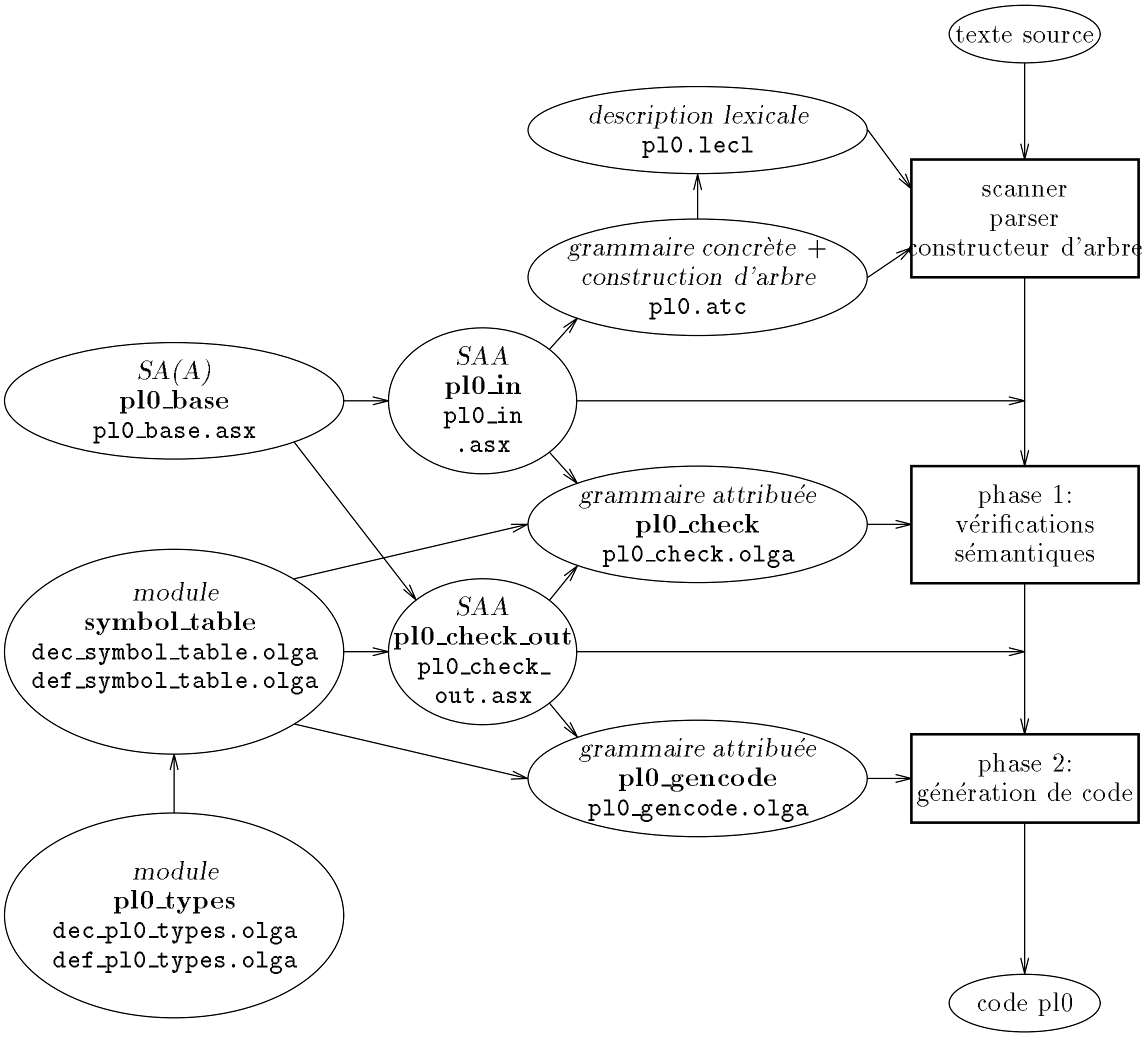


Figure A.1 : Graphe de dépendance des modules pour PL0

end.

A.2 Description de la traduction à effectuer

Notre exemple correspond à une première phase de compilateur, chargée de traduire un texte source en un code correspondant au langage source, et interprétable par la "machine PL0".

La figure 6 nous permet d'avoir une vue d'ensemble de l'enchaînement des différents modules nécéssaires pour effectuer cette traduction.

A.2.1 La machine PL0

Nous allons donc ici décrire brièvement cette machine qui prend en entrée du code PL0.

Il s'agit en fait d'un interprète (l'algorithme mis en œuvre interprète des instructions PL0 séquentiellement). Son architecture peut être décrite comme suit: un registre d'instruction contenant l'instruction en cours d'interprétation, et trois registres d'adresse permettant d'accéder aux données organisées en pile. Cette structure de pile correspond parfaitement à la stratégie d'activation de procédure, la stratégie Last–In/First–Out.

A.2.2 Les vérifications

Il s'agit, dans un premier temps, de vérifier que le texte à traduire appartient bien au langage décrit, du point de vue lexico–syntaxique d'une part, et du point de vue de la sémantique statique d'autre part.

La vérification de la correction lexico–syntaxique étant effectuée par SYNTAX[1], il reste à vérifier la correction de la sémantique statique. Cette vérification est effectuée par une première GA (cf ...).

Les règles appliquées à cette occasion sont:

- On ne déclare pas deux fois le même identificateur dans un même bloc.
- Tout identificateur utilisé est connu dans le bloc courant.
- L'usage de chaque identificateur est conforme à sa déclaration: variable simple, constante ou identificateur de procédure.

En revanche, on n'a pas à s'occuper du typage des expressions et des variables car toutes les valeurs manipulées dans PL0 sont entières (sauf implicitement dans la conditionnelle).

A.2.3 La traduction

Une fois toutes ces vérifications effectuées, il faut générer le code correspondant au texte source correct. C'est la tâche de la seconde GA (cf ...).

Il s'agit ici de transcrire chaque instruction PL0 en son équivalent dans le **code** accepté par la machine PL0 décrite ci–dessus.

Le code que l'on doit fournir comporte les instructions suivantes:

1 Charger un nombre depuis la pile: *LIT*.

[1]Analyseur lexical et syntaxique utilisant la méthode LR

2 Aller chercher une variable dans la pile: *LOD*.

3 Stocker dans la pile (affectation): *STO*.

4 Activer une sous–routine (appel de procédure): *CAL*.

5 Allouer un emplacement dans la pile en incrémentant le pointeur de pile: *INT*.

6 Transfert de contrôle inconditionnel (while) ou conditionnel (if): *JMP*, *JPC*.

7 Ensemble d'opérateurs arithmétiques et relationnels: *OPR*.

Les *adresses* des données dans la pile sont représentées par des couples *(l,a)* où *l* représente la différence de niveau statique, et *a* le déplacement relatif dans un segment de données.

L'utilisation des ces diverses instructions se fait comme suit:

LIT 0,a : charger la constante entière *a*.

LOD l,a : charger la variable d'adresse *(l,a)*.

STO l,a : stocker la variable d'adresse *(l,a)*.

CAL l,a : appeler la procédure du niveau *l* et d'adresse *(0,a)* dans ce niveau.

INT 0,a : réserver *a* emplacements dans la pile.

JMP 0,a : saut à la ligne *a*.

JPC 0,a : saut conditionnel à la ligne *a*.

OPR 0,a : exécuter l'opération *a*.

Signification de *OPR 0,a* selon la valeur de *a*:

a	**operateur**	a	**operateur**
0 :	*Return*	7 :	*Erreur*
1 :	$-$ (*unaire*)	8 :	$=$
2 :	$+$ (*binaire*)	9 :	$\neq$
3 :	$-$ (*binaire*)	10 :	$<$
4 :	$*$ (*binaire*)	11 :	$\geq$
5 :	*div* (*binaire*)	12 :	$>$
6 :	*odd* (*unaire*)	13 :	$\leq$

Ainsi, le code généré correspondant au programme exemple donné précédemment a cette forme:

```
0    JMP  0,107
1    INT  0,5
2    LOD  1,3
3    STO  0,3
4    LOD  1,3
5    STO  0,4
6    LIT  0,0
7    STO  1,5
8    LOD  0,4
9    LIT  0,0
10   OPR  0,12
11   JPC  0,28
12   LOD  0,4
13   OPR  0,7
14   JPC  0,19
15   LOD  1,5
16   LOD  0,3
17   OPR  0,2
18   STO  1,5
19   LIT  0,2
20   LOD  0,3
21   OPR  0,4
22   STO  0,3
23   LOD  0,4
24   LIT  0,2
25   OPR  0,5
26   STO  0,4
27   JMP  0,8
28   OPR  0,0
29   INT  0,5
30   LOD  2,3
31   STO  0,3
32   LOD  1,3
33   STO  0,4
34   OPR  0,0
35   INT  0,4
36   LOD  1,3
37   STO  1,7
38   LIT  0,0
39   STO  1,6
40   LOD  1,4
41   STO  0,3
42   LOD  0,3
43   LOD  1,7
44   OPR  0,13
45   JPC  0,51
46   LIT  0,2
47   LOD  0,3
48   OPR  0,4
49   STO  0,3
50   JMP  0,42
51   LOD  0,3
52   LOD  1,4
53   OPR  0,12
54   JPC  0,77
55   LIT  0,2
56   LOD  1,6
57   OPR  0,4
58   STO  1,6
59   LOD  0,3
60   LIT  0,2
61   OPR  0,5
62   STO  0,3
63   LOD  0,3
64   LOD  1,7
65   OPR  0,13
66   JPC  0,76
67   LOD  1,7
68   LOD  0,3
69   OPR  0,3
70   STO  1,7
71   CAL  0,4
72   LOD  1,6
73   LIT  0,1
74   OPR  0,2
75   STO  1,6
76   JMP  0,51
77   OPR  0,0
78   INT  0,5
79   LOD  1,3
80   STO  0,3
81   LOD  1,4
82   STO  0,4
83   LOD  0,3
84   LOD  0,4
85   OPR  0,9
86   JPC  0,104
87   LOD  0,3
88   LOD  0,4
89   OPR  0,10
90   JPC  0,95
91   LOD  0,4
92   LOD  0,3
93   OPR  0,3
94   STO  0,4
95   LOD  0,4
96   LOD  0,3
97   OPR  0,10
98   JPC  0,103
99   LOD  0,3
100  LOD  0,4
101  OPR  0,3
102  STO  0,3
103  JMP  0,83
104  LOD  0,3
105  STO  1,5
106  OPR  0,0
107  INT  0,8
108  LIT  0,7
109  STO  0,3
110  LIT  0,85
111  STO  0,4
112  CAL  0,8
113  LIT  0,25
114  STO  0,3
115  LIT  0,3
116  STO  0,4
117  CAL  0,9
118  LIT  0,84
119  STO  0,3
120  LIT  0,36
121  STO  0,4
122  CAL  0,10
```

A.3 Les grammaires attribuées

A.3.1 Les grammaires attribuées exhaustives

- La première version de la grammaire attribuée de **vérification de types**, c'est-à-dire sans la spécification des règles par défaut, est écrite ainsi :

```
attribute grammar pl0 (pl0_in, pl0_check_out) : is
        from symbol_table import all ;


    function typinfo(infov:info) : int is
        if infov ĩnfo-error then
            0
        else
        case infov is
            info(X:info-const)    : 1 ;
            info(X:info-var)      : 2 ;
            info(X:info-proc)     : 3 ;
            other                 : 0 ;
        end case
        end if
    end function;

attribute

    { working attributes }

    compound $h.temp-st and $s.temp-st (BLOCK,DECL,CONST,VAR-DC,
                    VAR-DECL, PROC-LIST, DC-CONST, PROC)
                    : symbol-table ;
    synthesized $correct1 (BLOCK,DECL, CONST,VAR-DC, VAR-DECL,
                    PROC-LIST, DC-CONST, PROC) : bool ;
    synthesized $correct2 (BLOCK, DECL, PROC-LIST, PROC ,STMT,COND,
                    EXPR, EXPR-S, TERM,FACT ,VAR,INT) : bool ;
    synthesized $op (OP) : operator ;
    synthesized $op1 (SIGN) : operator1 ;
    synthesized $sign (SIG) : signe ;
    synthesized $comp (COMP) : comparator ;

    { kinds of result attributes }
    synthesized $value () ;
    global $symtab () ;
    synthesized $correct () ;


{ let's go for semantic rules }

{ PROGRAM = program }

where program -> BLOCK use
    $correct := $correct1(BLOCK) & $correct2(BLOCK) ;
    $h.temp-st(BLOCK) := empty-table ;
    $symtab(BLOCK) := printsymtab($scx(BLOCK),$s.temp-st(BLOCK),$s.temp-st(BLOCK)) ;
end where ;

where null-PROGRAM -> use
    $correct := false ;
end where ;


{ BLOCK = block }
```

```
where block -> DECL STMT use
      $correct1 := $correct1(DECL) ;
      $correct2 := $correct2(DECL) & $correct2(STMT) ;
      $s.temp-st := $s.temp-st(DECL) ;
end where ;

where null-BLOCK -> use
      $s.temp-st := $h.temp-st ;
      $correct1 := false ;
      $correct2 := false ;
end where ;


{ DECL = decl }

where decl -> CONST VAR-DC PROC-LIST use
      $correct1 := $correct1(CONST) & $correct1(VAR-DC)
                  & $correct1(PROC-LIST) ;
      $correct2 := $correct2(PROC-LIST) ;
      $s.temp-st := $s.temp-st(PROC-LIST) ;
      $h.temp-st(CONST) := $h.temp-st ;
      $h.temp-st(VAR-DC) := $s.temp-st(CONST) ;
      $h.temp-st(PROC-LIST) := $s.temp-st(VAR-DC) ;
end where ;

where null-DECL -> use
      $s.temp-st := $h.temp-st ;
      $correct1 := false ;
      $correct2 := false ;
end where ;

{ CONST = const-list }

where const-list -> DC-CONST* use
      $correct1 := map left &
                        value true
                        other $correct1(DC-CONST)
                  end map ;
      $s.temp-st := case arity is
                        0     : $h.temp-st ;
                        other $s.temp-st(DC-CONST.last) ;
                  end case ;
      $h.temp-st(DC-CONST) := case position is
                        first : $h.temp-st ;
                        other $s.temp-st(DC-CONST.left) ;
                  end case ;
end where ;


where null-CONST -> use
      $s.temp-st := $h.temp-st ;
      $correct1 := false ;
end where ;


{ DC-CONST = const-decl }

where null-DC-CONST -> use
      $s.temp-st := $h.temp-st ;
      $correct1 := false ;
end where ;

where const-decl -> ID INT
declare
```

```
      value
      correct : bool := (if $id(ID) = error-token then
                        false
                  elsif lookup-in-block ($id(ID), $h.temp-st) then
                        message "Identifier already declared."
                        position ID
                        severity 2
                        value false
                  else
                        true
                  end if) & $correct2(INT) ;
use
      $correct1 := correct ;
      $s.temp-st := if correct then
                        insert ($id(ID),info(info-const($value(INT))), $h.temp-st)
                  else
                        $h.temp-st
                  end if ;
end where ;


{ VAR-DC = var-list }

where null-VAR-DC -> use
      $s.temp-st := $h.temp-st ;
      $correct1 := false ;
end where ;

where var-list -> VAR-DECL*
declare
      value
      correctbool := map left &
                              value true
                              other $correct1(VAR-DECL)
                        end map ;
use
      $correct1 := correct ;
      $s.temp-st := case arity is
                        0 :         $h.temp-st ;
                        other :     if correct then
                                          $s.temp-st(VAR-DECL.last)
                                    else
                                          $h.temp-st
                                    end if ;
                  end case ;
      $h.temp-st(VAR-DECL) := case position is
                              first : $h.temp-st ;
                              other : $s.temp-st(VAR-DECL.left) ;
                        end case ;
end where ;


{ VAR-DECL = var-decl }

where null-VAR-DECL -> use
      $s.temp-st := $h.temp-st ;
      $correct1 := false ;
end where ;

where var-decl -> ID
declare
      value
      correct : bool := if $id(ID) = error-token then
                        false
                  elsif lookup-in-block ($id(ID), $h.temp-st) then
```

```
                    message "Identifier already declared."
                         position ID
                         severity 2
                         value false
               else
                    true
               end if ;
use
     $correct1 := correct ;
     $s.temp-st := if correct then
                    insert ($id(ID),info(info-var(0,0)), $h.temp-st)
               else
                    $h.temp-st
               end if ;
end where ;


{ PROC-LIST = proc-list }

where null-PROC-LIST -> use
     $s.temp-st := $h.temp-st ;
     $correct1 := false ;
     $correct2 := false ;
end where ;

where proc-list -> PROC* use
     $correct1 := map left &
                    value true
                    other $correct1(PROC)
               end map ;
     $correct2 := map left &
                    value true
                    other $correct2(PROC)
               end map ;
     $s.temp-st := case arity is
                    0         : $h.temp-st ;
                    other     : $s.temp-st(PROC.last) ;
               end case ;
     $h.temp-st(PROC) := case position is
                    first     : $h.temp-st ;
                    other     : $s.temp-st(PROC.left) ;
               end case ;
end where ;


{ PROC = proc }

where null-PROC -> use
     $s.temp-st := $h.temp-st ;
     $correct1 := false ;
     $correct2 := false ;
end where ;

where proc -> ID BLOCK
declare
     value
     correct : bool := (if $id(ID.1) = error-token then
                    false
               elsif lookup-in-block ($id(ID), $h.temp-st) then
                    message "Identifier already declared."
                         position ID
                         severity 2
                         value false
               else
                    true
```

```
            end if) ;
use
    $correct1 := correct ;
    $correct2 := $correct1(BLOCK) & $correct2(BLOCK) ;
    $s.temp-st := if correct then
                      insert ($id(ID),info(info-proc(0,0)), $h.temp-st)
                  else
                      $h.temp-st
                  end if ;
    $h.temp-st(BLOCK) := enter-block ($h.temp-st) ;
    $symtab(BLOCK) := printsymtab($scx(BLOCK),$s.temp-st(BLOCK),$s.temp-st(BLOCK)) ;
end where ;


{ STMT = assign , call , ifthen , while , stmt-list }

where null-STMT -> use
    $correct2 := false ;
end where ;

where assign -> VAR EXPR use
    $correct2 := $correct2(VAR) & $correct2(EXPR) ;
end where ;

where call -> ID use
    $correct2 := (if $id(ID) = error-token then
                      false
                  else
                      if typinfo(lookup ($id(ID), $symtab(BLOCK.up)))
                                              != 3 then
                          message "Not a procedure."
                              position ID
                              severity 2
                              value false
                      else
                      true
                      end if
                  end if) ;
end where ;

where ifthen -> COND STMT use
    $correct2 := $correct2(COND) & $correct2(STMT) ;
end where ;

where while -> COND STMT use
    $correct2 := $correct2(COND) & $correct2(STMT) ;
end where ;

where stmt-list -> STMT+ use
    $correct2 := map left &
                  value true
                  other $correct2(STMT)
                  end map ;
end where ;


{ COND = odd comp }

where null-COND -> use
    $correct2 := false ;
end where ;

where odd -> EXPR use
    $correct2 := $correct2(EXPR) ;
end where ;
```

```
where comp -> EXPR COMP EXPR use
      $correct2 := $correct2(EXPR.1) & $correct2(EXPR.2)
                   & $comp(COMP) != wrong-comp ;
end where ;


{ COMP = egal diff , inf , infeg , sup , supeg }

where egal -> use
      $comp := eq ;
end where ;

where diff -> use
      $comp := dif ;
end where ;

where inf -> use
      $comp := inf ;
end where ;

where infeg -> use
      $comp := infeq ;
end where ;

where sup -> use
      $comp := sup ;
end where ;

where supeg -> use
      $comp := supeq ;
end where ;

where null-COMP -> use
      $comp := wrong-comp ;
end where ;


{ VAR = simple-var }

where null-VAR -> use
      $correct2 := false ;
end where ;

where simple-var -> ID use
      $correct2 := if $id(ID) = error-token then
                   false
                   else
                   let mode :int := typinfo(lookup ($id(ID), $symtab(BLOCK.up)))
                   in if mode = 2 then
                         true
                         else
                         message "Not a variable."
                               position ID
                               severity 2
                               value false
                         end if
                   end if ;
end where ;


{ EXPR = expr-comp , expr }

where null-EXPR -> use
      $correct2 := false ;
```

```
end where ;

where expr-comp -> SIG TERM EXPR-S use
      $correct2 := $correct2(TERM) & $correct2(EXPR-S) & $sign(SIG) !=
      wrong-sign ;
end where ;

where expr -> SIG TERM use
      $correct2 := $correct2(TERM) & $sign(SIG) != wrong-sign ;
end where ;


{ EXPR-S = expr-sign-comp , expr-sign }

where null-EXPR-S -> use
      $correct2 := false ;
end where ;

where expr-sign-comp -> SIGN TERM EXPR-S use
      $correct2 := $correct2(TERM) & $correct2(EXPR-S) & $op1(SIGN) !=
      wrong-op1 ;
end where ;

where expr-sign -> SIGN TERM use
      $correct2 := $correct2(TERM) & $op1(SIGN) != wrong-op1 ;
end where ;


{ SIG = nonsign , splus , sminus }

where null-SIG -> use
      $sign := wrong-sign ;
end where ;

where nonsign -> use
      $sign := nothg ;
end where ;

where splus -> use
      $sign := plus ;
end where ;

where sminus -> use
      $sign := minus ;
end where ;


{ SIGN = plus , minus }

where null-SIGN -> use
      $op1 := wrong-op1 ;
end where ;

where plus -> use
      $op1 := addition ;
end where ;

where minus -> use
      $op1 := subtraction ;
end where ;


{ TERM = term-fact , term-comp }

where null-TERM -> use
```

```
      $correct2 := false ;
end where ;

where term-fact -> FACT use
      $correct2 := $correct2(FACT) ;
end where ;

where term-comp -> FACT OP FACT use
      $correct2 := $correct2(FACT.1) & $correct2(FACT.2) & $op(OP) !=
      wrong-op ;
end where ;


{ OP = mul , div }

where null-OP -> use
      $op := wrong-op ;
end where ;

where mul -> use
      $op := multiplication ;
end where ;

where div -> use
      $op := division ;
end where ;


{ FACT = fact-id , fact-int , fact-expr }

where null-FACT -> use
      $correct2 := false ;
end where ;

where fact-id -> ID use
      $correct2 := if $id(ID) = error-token then
                         false
                   else
                         let mode :int := typinfo(lookup ($id(ID), $symtab(BLOCK.up)))
                         in    if (mode = 1) or( mode = 2 ) then
                                     true
                               else
                                     message "Not a factor."
                                           position ID
                                           severity 2
                                           value false
                               end if
                   end if ;
end where ;

where fact-int -> INT use
      $correct2 := $correct2(INT) ;
end where ;

where fact-expr -> EXPR use
      $correct2 := $correct2(EXPR) ;
end where ;


{ INT = int }

where null-INT -> use
      $correct2 := false;
      $value := 0;
end where ;
```

```
where int -> use
    $correct2 := $text != error-token ;
    $value := if $text = error-token then
            0
        else
            int (string ($text))
        end if ;
end where ;


{ ID = id }

where null-ID -> use
end where ;

where id -> use
end where ;
end grammar ;
```

- La grammaire attribuée de **traduction** a pour forme complète:

```
attribute grammar pl0_gencode (pl0_check_out) : (pl0_base) is
    from symbol_table import all ;
    function level_offset_define_var(infovar :info-var;level:int;offset:int):bool is
            external ;

    function level_offset_define_proc(infoproc :info-proc;level:int;offset:int):bool is
            external ;

    function level_offset_define(info:info;level:int;offset:int):bool is
    case info is
        info(X:info-var) : level_offset_define_var(X,level,offset) ;
        info(X:info-proc) : level_offset_define_proc(X,level,offset) ;
        other : false ;
    end case
    end function ;

    function look-var ( id : token; st : symbol-table ) : info-var is
    let info : info := lookup (id,st)
    in case info is
        info(X:info-var) : X ;
        other : info-var-undef ;
    end case
    end function ;

    function look-proc ( id : token; st : symbol-table ) : info-proc is
    let info : info := lookup (id,st)
    in case info is
        info(X:info-proc) : X ;
        other : info-proc-undef ;
    end case
    end function ;

    attribute
    synthesized $code ( BLOCK, DECL, VAR, PROC-LIST, PROC, STMT, COND,
            EXPR, SIG, TERM, FACT):
            code ;
    compound $h.ligne and $s.ligne ( BLOCK, DECL, VAR, PROC-LIST, PROC,
                    STMT, COND, EXPR, SIG, TERM, FACT ):
            int ;
        { $h.ligne : no de ligne a utiliser }
        { $s.ligne : no de la derniere ligne utilisee }
    compound $h.offset and $s.offset ( VAR-DC, VAR-DECL, PROC-LIST, PROC ):
```

```
            int ;
        { $s.offset : derniere valeur d'offset utilisee }
        { $h.offset : valeur d'offset a utiliser }
    inherited $level (BLOCK, DECL, VAR-DC, PROC-LIST, VAR, VAR-DECL, PROC,
            STMT, COND, EXPR, SIG, TERM, FACT ):
            int ;
    synthesized $nbvar ( VAR-DC ) : int ;
    synthesized $act ( COMP, SIGN, OP ) : int ;
    synthesized $correct-ins( VAR-DECL, PROC ) : bool ;
    synthesized $codegen ( PROGRAM ) : bool ;
    synthesized $ligneprog ( BLOCK ) : int ;
    global $tabconsult ( BLOCK ) : symbol-table ;

{ PROGRAM = program }
where program -> BLOCK use
    $codegen := let codeblock : code := code(ligne-code[[
                                    code(ligne(0,JMP,0,$ligneprog(BLOCK))) ,
                                $code(BLOCK) ]])
                in
                print_code("CODE",$scx(BLOCK),codeblock) ;
    $level(BLOCK) := 0 ;
    $h.ligne(BLOCK) := 1 ;
    $tabconsult(BLOCK) := $symtab (BLOCK);
end where ;

where null-PROGRAM -> use
    $codegen := false ;
end where ;

{ BLOCK = block }
where block -> DECL STMT use
    $level(DECL) := $level ;
    $code := code(ligne-code[[
                $code(DECL) ,
                $code(STMT) ]]) ;
    $h.ligne(DECL) := $h.ligne ;
    $h.ligne(STMT) := $s.ligne(DECL) + 1 ;
    $s.ligne := $s.ligne(STMT) ;
    $ligneprog := $s.ligne(DECL) ;
end where ;

where null-BLOCK -> use
    $code := empty-code ;
    $ligneprog := 0 ;
end where ;

{ DECL = decl }
where decl -> CONST VAR-DC PROC-LIST use
    $level(VAR-DC) := $level ;
    $level(PROC-LIST) := $level ;
    $code := code(ligne-code[[
                $code(PROC-LIST) ,
                code(ligne($s.ligne(PROC-LIST) + 1,INT,0,(3+$nbvar(VAR-DC)))) ]]) ;
    $h.ligne(PROC-LIST) := $h.ligne ;
    $s.ligne := $s.ligne(PROC-LIST) +1;
    $h.offset(VAR-DC) := 3 ; { 0 : offset du premier registre,1 du second,2 du dernier }
    $h.offset(PROC-LIST) := $s.offset(VAR-DC) + 1 ;
end where ;

where null-DECL -> use
    $code := empty-code ;
    $s.ligne := 0;
end where ;

{ CONST = const-list }
```

```
where const-list -> DC-CONST* use
end where ;

where null-CONST -> use
end where ;

{ DC-CONST = const-decl }
where null-DC-CONST -> use
end where ;

where const-decl -> ID INT use
end where ;

{ VAR-DC = var-list }
where null-VAR-DC -> use
      $nbvar := 0 ;
      $s.offset := 0 ;
end where ;

where var-list -> VAR-DECL* use
      $nbvar := $arity ;
      $level(VAR-DECL) := $level ;
      $s.offset := case arity is
                        0 : $h.offset - 1 ;
                        other : $s.offset(VAR-DECL.last) ;
                   end case ;
      $h.offset(VAR-DECL) := case position is
                                    first : $h.offset ;
                                    other : $s.offset(VAR-DECL.left) +1 ;
                              end case ;
end where ;

{ VAR-DECL = var-decl }
where null-VAR-DECL -> use
      $correct-ins := false ;
      $s.offset := 0 ;
end where ;

where var-decl -> ID use
      $correct-ins := let recup : info := lookup ($id(ID),$tabconsult(BLOCK.up))
                        in
                        level_offset_define(recup,$level,$h.offset);
      $s.offset := $h.offset ;
end where ;

{ PROC-LIST = proc-list }
where null-PROC-LIST -> use
      $code := empty-code ;
      $s.ligne := 0;
end where ;

where proc-list -> PROC* declare
      value
      ligne-code : ligne-code := map left ligne-code
                                    value ligne-code ()
                                    other $code(PROC)
                              end map ;
      use
      $code := code(ligne-code) ;
      $h.ligne(PROC) := case position is
                              first : $h.ligne ;
                              other : $s.ligne(PROC.left) + 1 ;
                        end case ;
      $s.ligne := case arity is
                        0 : $h.ligne - 1;
```

```
                    other : $s.ligne(PROC.last) ;
                end case ;
    $level(PROC) := $level ;
    $s.offset := case arity is
                    0 : $h.offset - 1 ;
                    other : $s.offset(PROC.last) ;
                end case ;
    $h.offset(PROC) := case position is
                        first : $h.offset ;
                        other : $s.offset(PROC.left) +1 ;
                    end case ;
end where ;

{ PROC = proc }
where null-PROC -> use
    $code := empty-code ;
    $s.ligne := 0;
    $correct-ins := false ;
end where ;

where proc -> ID BLOCK use
    $tabconsult(BLOCK) := $symtab(BLOCK) ;
    $level(BLOCK) := $level + 1 ;
    $correct-ins := let recup : info := lookup ($id(ID),$tabconsult(BLOCK.up))
                    in
                    level_offset_define(recup,$level,$h.offset);
    $h.ligne(BLOCK) := $h.ligne ;
    $s.ligne := $s.ligne (BLOCK) + 1 ;
    $s.offset := $h.offset ;
    $code := code(ligne-code[[
                    $code(BLOCK) ,
                    code(ligne($s.ligne(BLOCK)+1,OPR,0,0)) ]]) ;
end where ;

{ STMT = assign , call , ifthen , while , stmt-list }
where null-STMT -> use
    $code := empty-code ;
    $s.ligne := 0;
end where ;

where assign -> VAR EXPR use
    $s.ligne := $s.ligne(VAR) ;
    $h.ligne(VAR) := $s.ligne(EXPR) + 1 ;
    $h.ligne(EXPR) := $h.ligne ;
    $code := code(ligne-code[[
                $code(EXPR) ,
                $code(VAR) ]]) ;
    $level(VAR) := $level ;
    $level(EXPR) := $level ;
end where ;

where call -> ID
    declare
    value
    recup : info-proc := look-proc ($id(ID),$tabconsult(BLOCK.up)) ;
    correc : bool := if recup = info-proc-undef then
                        false
                    else
                        true
                    end if ;
    use
    $code := if correc then
                    code(ligne($h.ligne,CAL,($level - recup.level),recup.offset))
                else
                    code(ligne($h.ligne,NIL,0,0))
```

```
            end if ;
    $s.ligne := $h.ligne ;
end where ;

where ifthen -> COND STMT use
    $s.ligne := $s.ligne(STMT) ;
    $code := code(ligne-code[[
                $code(COND) ,
                code(ligne($s.ligne(COND)+1,JPC,0,$s.ligne(STMT) +1 )) ,
                $code(STMT) ]]) ;
    $h.ligne(COND) := $h.ligne ;
    $h.ligne(STMT) := $s.ligne(COND) + 2 ;
    $level(COND) := $level ;
    $level(STMT) := $level ;
end where ;

where while -> COND STMT use
    $s.ligne := $s.ligne(STMT) + 1 ;
    $code := code(ligne-code[[
            $code(COND),
            code(ligne($s.ligne(COND)+1,JPC,0,$s.ligne(STMT) + 2)) ,
            $code(STMT),
            code(ligne($s.ligne(STMT)+1,JMP,0,$h.ligne)) ]]) ;
    $h.ligne(COND) := $h.ligne ;
    $h.ligne(STMT) := $s.ligne(COND) + 2 ;
    $level(COND) := $level ;
    $level(STMT) := $level ;
end where ;

where stmt-list -> STMT+ declare
    value
    ligne-code : ligne-code := map left ligne-code
                                    value ligne-code ()
                                    other $code(STMT)
                                end map ;
    use
    $s.ligne := $s.ligne(STMT.last) ;
    $code := code(ligne-code);
    $h.ligne(STMT) := case position is
                                first : $h.ligne ;
                                other : $s.ligne(STMT.left) + 1 ;
                            end case ;
end where ;

{ COND = odd , comp }
where null-COND -> use
    $code := empty-code ;
    $s.ligne := 0;
end where ;

where odd -> EXPR use
    $code := code(ligne-code[[
                $code(EXPR) ,
                code(ligne($s.ligne(EXPR) + 1,OPR,0,7)) ]]) ;
    $s.ligne := $s.ligne(EXPR)+1;
    $h.ligne(EXPR) := $h.ligne ;
    $level(EXPR) := $level ;
end where ;

where comp -> EXPR COMP EXPR use
    $code := code(ligne-code[[
                $code(EXPR.1) ,
                $code(EXPR.2) ,
                code(ligne($s.ligne(EXPR.2) + 1,OPR,0,$act(COMP))) ]]) ;
    $s.ligne := $s.ligne(EXPR.2) + 1 ;
```

```
      $h.ligne(EXPR.1) := $h.ligne ;
      $h.ligne(EXPR.2) := $s.ligne(EXPR.1) + 1 ;
      $level(EXPR.1) := $level ;
      $level(EXPR.2) := $level ;
end where ;

{ COMP = egal , diff , inf , infeg , sup , supeg }
where egal -> use
      $act := 8 ;
end where ;

where diff -> use
      $act := 9 ;
end where ;

where inf -> use
      $act := 10 ;
end where ;

where infeg -> use
      $act := 13 ;
end where ;

where sup -> use
      $act := 12 ;
end where ;

where supeg -> use
      $act := 11 ;
end where ;

where null-COMP -> use
      $act := 6 ;
end where ;

{ VAR = simple-var }
where null-VAR -> use
      $code := empty-code ;
      $s.ligne := 0;
end where ;

where simple-var -> ID
      declare
      value
      recup : info-var := look-var ($id(ID),$tabconsult(BLOCK.up)) ;
      correc : bool := if recup = info-var-undef then
                              value false
                        else
                              true
                        end if ;
      use
      $code := if correc then
                        code(ligne($h.ligne,STO,$level - recup.level,recup.offset))
                  else
                        code(ligne($h.ligne,NIL,0,0))
                  end if ;
      $s.ligne := $h.ligne ;
end where ;

{ EXPR = expr-comp , expr }
where null-EXPR -> use
      $code := empty-code ;
      $s.ligne := 0;
end where ;
```

```
where expr-comp -> SIG TERM EXPR-S use
      $s.ligne := $s.ligne(EXPR-S) ;
      $code := code(ligne-code [[
                        $code(TERM) ,
                        $code(SIG) ,
                        $code(EXPR-S) ]]) ;
      $h.ligne(SIG) := $s.ligne(TERM) +1 ;
      $h.ligne(TERM) := $h.ligne ;
      $h.ligne(EXPR-S) := $s.ligne(SIG) +1 ;
      $level(TERM) := $level ;
      $level(EXPR-S) := $level ;
end where ;

where expr -> SIG TERM use
      $code := code(ligne-code [[
                        $code(TERM) ,
                        $code(SIG) ]]) ;
      $s.ligne := $s.ligne(SIG) ;
      $h.ligne(SIG) := $s.ligne(TERM) + 1;
      $h.ligne(TERM) := $h.ligne ;
      $level(TERM) := $level ;
end where ;

{ EXPR-S = expr-sign-comp , expr-sign }
where null-EXPR-S -> use
      $code := empty-code ;
      $s.ligne := 0;
end where ;

where expr-sign-comp -> SIGN TERM EXPR-S use
      $code := code(ligne-code [[
                        $code(TERM) ,
                        code(ligne($s.ligne(TERM)+1,OPR,0,$act(SIGN))) ,
                        $code(EXPR-S) ]]) ;
      $s.ligne := $s.ligne(EXPR-S) ;
      $h.ligne(TERM) := $h.ligne ;
      $h.ligne(EXPR-S) := $s.ligne(TERM) + 2 ;
      $level(TERM) := $level ;
      $level(EXPR-S) := $level ;
end where ;

where expr-sign -> SIGN TERM use
      $code := code(ligne-code [[
                        $code(TERM) ,
                        code(ligne($s.ligne(TERM)+1,OPR,0,$act(SIGN))) ]]) ;
      $s.ligne := $s.ligne(TERM) +1 ;
      $h.ligne(TERM) := $h.ligne ;
      $level(TERM) := $level ;
end where ;

{ SIG = nonsign , splus , sminus }
where null-SIG -> use
      $code := empty-code ;
      $s.ligne := 0;
end where ;

where nonsign -> use
      $code := empty-code;
      $s.ligne := $h.ligne -1 ;
end where ;

where splus -> use
      $code := empty-code;
      $s.ligne := $h.ligne -1 ;
end where ;
```

```
where sminus -> use
      $code := code(ligne($h.ligne,OPR,0,1)) ;
      $s.ligne := $h.ligne ;
end where ;

{ SIGN = plus , minus }
where null-SIGN -> use
      $act := 6 ;
end where ;

where plus -> use
      $act := 2;
end where ;

where minus -> use
      $act := 3;
end where ;

{TERM = term-fact , term-comp }
where null-TERM -> use
      $code := empty-code ;
      $s.ligne := 0;
end where ;

where term-fact -> FACT use
      $code := $code(FACT);
      $s.ligne := $s.ligne(FACT) ;
      $h.ligne(FACT) := $h.ligne;
      $level(FACT) := $level ;
end where ;

where term-comp -> FACT OP FACT use
      $code :=code(ligne-code [[
                        $code(FACT.1) ,
                        $code(FACT.2) ,
                        code(ligne($s.ligne(FACT.2)+1,OPR,0,$act(OP))) ]]);
      $s.ligne := $s.ligne(FACT.2) + 1 ;
      $h.ligne(FACT.1) := $h.ligne ;
      $h.ligne(FACT.2) := $s.ligne(FACT.1) + 1 ;
      $level(FACT.1) := $level ;
      $level(FACT.2) := $level ;
end where ;

{ OP = mul , div }
where null-OP -> use
      $act := 6;
end where ;

where mul -> use
      $act := 4;
end where ;

where div -> use
      $act := 5;
end where ;

{ FACt = fact-id , fact-int , fact-expr }
where null-FACT -> use
      $code := empty-code ;
      $s.ligne := 0;
end where ;

where fact-id -> ID
      declare
```

```
      value
      recup : info := lookup ($id(ID),$tabconsult(BLOCK.up)) ;
      correc : bool := if recup   info-error then
                          false
                     else
                          true
                     end if ;
      use
      $code := if correc then
                     case recup is
                          info(X:info-var):
                               code(ligne($h.ligne,LOD,$level - X.level,X.offset)) ;
                          info(X:info-const):
                               code(ligne($h.ligne,LIT,0,X.val)) ;
                          other : code(ligne($h.ligne,NIL,0,0)) ;
                     end case
                else
                     code(ligne(0,NIL,0,0))
                end if ;
      $s.ligne := $h.ligne ;
end where ;

where fact-int -> INT use
      $code := code(ligne($h.ligne,LIT,0,$value(INT))) ;
      $s.ligne := $h.ligne ;
end where ;

where fact-expr -> EXPR use
      $code := $code(EXPR) ;
      $s.ligne := $s.ligne(EXPR) ;
      $h.ligne(EXPR) := $h.ligne ;
end where ;

{ INT = int }
where null-INT -> use
end where ;

where int -> use
end where ;

{ ID = id }
where null-ID -> use
end where ;

where id -> use
end where ;

end grammar ;
```

A.3.2 Les grammaires attribuées incluant une partie défaut

- La grammaire attribuée de **vérification de types**, mais en précisant les règles par défaut, a cette forme :

```
attribute grammar pl0 (pl0_in, pl0_check_out) : is
        from symbol_table import all ;
        function typinfo(infov:info) : int is
             if infov   info-error then
                  0
             else
```

```
            case infov is
                  info(X:info-const)      : 1 ;
                  info(X:info-var)        : 2 ;
                  info(X:info-proc)       : 3 ;
                  other                   : 0 ;
            end case
            end if
      end function;

attribute

      { working attributes }

      compound $h.temp-st and $s.temp-st (BLOCK,DECL,CONST,VAR-DC,
                              VAR-DECL, PROC-LIST, DC-CONST, PROC)
                              : symbol-table ;
      synthesized $correct1 (BLOCK,DECL, CONST,VAR-DC, VAR-DECL,
                        PROC-LIST, DC-CONST, PROC) : bool ;
      synthesized $correct2 (BLOCK, DECL, PROC-LIST, PROC ,STMT,COND,
                        EXPR, EXPR-S, TERM,FACT ,VAR,INT) : bool ;
      synthesized $op (OP) : operator ;
      synthesized $op1 (SIGN) : operator1 ;
      synthesized $sign (SIG) : signe ;
      synthesized $comp (COMP) : comparator ;
      { kinds of result attributes }
      synthesized $value () ;
      global $symtab () ;
      synthesized $correct () ;

{ ——————————————————————————————————————————-
                        PARTIE DECLARATION PAR DEFAUT
——————————————————————————————————————————- }

local
      correctid (PROC, DC-CONST, VAR-DECL) : bool ;
      correct (PROC, DC-CONST, VAR-DECL, CONST, VAR-DC, PROC-LIST) : bool ;
      cor2 () : bool ;
classes
      $correct ($correct1,$correct2) : bool ;
      $cor1 ($correct1) : bool ;
      $cor2 ($correct2) : bool ;

{ ——————————————————————————————————————————-
                        PARTIE DEFINITION PAR DEFAUT
——————————————————————————————————————————- }

default correctid use
      where -> (X:ID, Y*) use
            if $id(X) = error-token then
                  false
            elsif lookup-in-block ($id(X), $h.temp-st) then
                  message "Identifier already declared."
                  position X
                  severity 2
                  value false
            else
                  true
            end if ;
      end where
end default ;

default correct use
      where op -> (X:ID, Y*) use case op is
                  var-decl, proc : correctid ;
                  other : correctid & $correct2(Y) ;
```

```
                end case ;
        end where
        where -> (X)+* use map left &
                value true
                other $correct(X)
            end map ;
        end where
end default ;

default cor2 use
        where -> (X*) use fmap left &
                value true
                other $correct2(X)
            end fmap ;
        end where
end default ;

default $correct use
        where null use false where ;
        where -> (X)+* use correct end where ;
end default ;

default $cor1 use
        where -> (X*) use correct end where ;
end default ;

default $cor2 use
        where -> (X*) use cor2 end where ;
end default ;

{ ————————————————————————————————————————————————————-
                              REGLES SEMANTIQUES
————————————————————————————————————————————————————- }

{ PROGRAM = program }

where program -> BLOCK use
        $correct   := $correct1(BLOCK) & $correct2(BLOCK) ;
        $h.temp-st(BLOCK)       := empty-table ;
        $symtab(BLOCK) := printsymtab($scx(BLOCK),$s.temp-st(BLOCK),$s.temp-st(BLOCK)) ;
end where ;

where null-PROGRAM -> use
        $correct   := false ;
end where ;

{ BLOCK = block }

where block -> DECL STMT use
        $correct1  := $correct1(DECL) ;
        $s.temp-st := $s.temp-st(DECL) ;
end where ;
where null-BLOCK -> use
end where ;

{ DECL = decl }

where decl -> CONST VAR-DC PROC-LIST use
        $correct1  := $correct1(CONST) & $correct1(VAR-DC)
                & $correct1(PROC-LIST) ;
end where ;

where null-DECL -> use
end where ;
```

```
{ CONST = const-list }

where const-list -> DC-CONST* use
end where ;

where null-CONST -> use
end where ;

{ DC-CONST = const-decl }

where null-DC-CONST -> use
end where ;

where const-decl -> ID INT use
        $s.temp-st := if correct then
                        insert ($id(ID),info(info-const($value(INT))), $h.temp-st)
                        else
                        $h.temp-st
                        end if ;
end where ;

{ VAR-DC = var-list }

where null-VAR-DC -> use
end where ;

where var-list -> VAR-DECL* use
end where ;

{ VAR-DECL = var-decl }

where null-VAR-DECL -> use
end where ;

where var-decl -> ID use
        $s.temp-st := if correct then
                        insert ($id(ID),info(info-var(0,0)), $h.temp-st)
                        else
                        $h.temp-st
                        end if ;
end where ;

{ PROC-LIST = proc-list }

where null-PROC-LIST -> use
end where ;

where proc-list -> PROC* use
end where ;

{ PROC = proc }

where null-PROC -> use
end where ;

where proc -> ID BLOCK use
        $s.temp-st := if correctid then
                        insert ($id(ID),info(info-proc(0,0)), $h.temp-st)
                        else
                        $h.temp-st
                        end if ;
        $h.temp-st(BLOCK) := enter-block ($h.temp-st) ;
        $symtab(BLOCK) := printsymtab($scx(BLOCK),$s.temp-st(BLOCK),$s.temp-st(BLOCK)) ;
end where ;
```

```
{ STMT = assign , call, ifthen, while , stmt-list }

where null-STMT -> use
end where ;

where assign -> VAR EXPR use
end where ;

where call -> ID use
        $correct2 := (if $id(ID) = error-token then
                                      false
                                else
                                      if typinfo(lookup ($id(ID), $symtab(BLOCK.up)))
                                                     != 3 then
                                      message "Not a procedure."
                                            position ID
                                            severity 2
                                            value false
                                      else
                                      true
                                      end if
                                end if) ;
end where ;

where ifthen -> COND STMT use
end where ;

where while -> COND STMT use
end where ;

where stmt-list -> STMT+ use
end where ;

{ COND = odd comp }

where null-COND -> use
end where ;

where odd -> EXPR use
end where ;

where comp -> EXPR COMP EXPR use
        $correct2 := cor2 & $comp(COMP) != wrong-comp ;
end where ;

{ COMP = egal , diff , inf , infeg , sup , supeg }

where egal -> use
        $comp     := eq ;
end where ;

where diff -> use
        $comp     := dif ;
end where ;

where inf -> use
        $comp     := inf ;
end where ;

where infeg -> use
        $comp     := infeq ;
end where ;

where sup -> use
        $comp     := sup ;
```

```
end where ;

where supeg -> use
        $comp     := supeq ;
end where ;

where null-COMP -> use
        $comp     := wrong-comp ;
end where ;

{ VAR = simple-var }

where null-VAR -> use
end where ;

where simple-var -> ID use
        $correct2  := if $id(ID) = error-token then
                          false
                          else
                          let mode :int := typinfo(lookup ($id(ID), $symtab(BLOCK.up)))
                          in if mode = 2 then
                                true
                                else
                                message "Not a variable."
                                      position ID
                                      severity 2
                                      value false
                                end if
                          end if ;
end where ;

{ EXPR = expr-comp , expr }

where null-EXPR -> use
end where ;

where expr-comp -> SIG TERM EXPR-S use
        $correct2  :=cor2 & $sign(SIG) != wrong-sign ;
end where ;

where expr -> SIG TERM use
        $correct2  := cor2 & $sign(SIG) != wrong-sign ;
end where ;

{ EXPR-S = expr-sign-comp , expr-sign }

where null-EXPR-S -> use
end where ;

where expr-sign-comp -> SIGN TERM EXPR-S use
        $correct2  := cor2 & $op1(SIGN) != wrong-op1 ;
end where ;

where expr-sign -> SIGN TERM use
        $correct2  := cor2 & $op1(SIGN) != wrong-op1 ;
end where ;

{ SIG = nonsign , splus , sminus }

where null-SIG -> use
        $sign := wrong-sign ;
end where ;

where nonsign -> use
        $sign := nothg ;
```

```
end where ;

where splus -> use
        $sign := plus ;
end where ;

where sminus -> use
        $sign := minus ;
end where ;

{ SIGN = plus , minus }

where null-SIGN -> use
        $op1 := wrong-op1 ;
end where ;

where plus -> use
        $op1 := addition ;
end where ;

where minus -> use
        $op1 := subtraction ;
end where ;

{ TERM = term-fact , term-comp }

where null-TERM -> use
end where ;

where term-fact -> FACT use
end where ;

where term-comp -> FACT OP FACT use
        $correct2  := cor2 & $op(OP) != wrong-op ;
end where ;

{ OP = mul , div }

where null-OP -> use
        $op   := wrong-op ;
end where ;

where mul -> use
        $op   := multiplication ;
end where ;

where div -> use
        $op   := division ;
end where ;

{ FACT = fact-id , fact-int , fact-expr }

where null-FACT -> use
end where ;

where fact-id -> ID use
        $correct2  := if $id(ID) = error-token then
                      false
                      else
                      let mode :int := typinfo(lookup ($id(ID), $symtab(BLOCK.up)))
                      in if (mode = 1) or( mode = 2 ) then
                         true
                         else
                         message "Not a factor."
                            position ID
```

```
                                    severity 2
                                    value false
                              end if
                        end if ;
end where ;

where fact-int -> INT use
end where ;

where fact-expr -> EXPR use
end where ;

{ INT = int }

where null-INT -> use
        $value      := 0;
end where ;

where int -> use
        $correct2  := $text != error-token ;
        $value      := if $text = error-token then
                       0
                       else
                       int (string ($text))
                       end if ;
end where ;

{ ID = id }

where null-ID -> use
end where ;

where id -> use
end where ;

end grammar ;
```

- La grammaire attribuée de **traduction** à laquelle on a rajoutée une partie défaut se présente ainsi:

```
attribute grammar pl0_gencode (pl0_check_out) : (pl0_base) is
      from symbol_table import all ;
      function level_offset_define_var(infovar :info-var;level:int;offset:int):bool is
                  external ;

      function level_offset_define_proc(infoproc :info-proc;level:int;offset:int):bool is
                  external ;

      function level_offset_define(info:info;level:int;offset:int):bool is
      case info is
            info(X:info-var) : level_offset_define_var(X,level,offset) ;
            info(X:info-proc) : level_offset_define_proc(X,level,offset) ;
            other : false ;
      end case
      end function ;

      function look-var ( id : token; st : symbol-table ) : info-var is
      let info : info := lookup (id,st)
      in case info is
            info(X:info-var) : X ;
            other : info-var-undef ;
      end case
      end function ;
```

```
function look-proc ( id : token; st : symbol-table ) : info-proc is
let info : info := lookup (id,st)
in case info is
      info(X:info-proc) : X ;
      other : info-proc-undef ;
end case
end function ;

attribute
synthesized $code ( BLOCK, DECL, VAR, PROC-LIST, PROC, STMT, COND,
            EXPR, SIG, TERM, FACT):
            code ;
compound $h.ligne and $s.ligne ( BLOCK, DECL, VAR, PROC-LIST, PROC,
                  STMT, COND, EXPR, SIG, TERM, FACT ):
            int ;
      { $h.ligne : no de ligne a utiliser }
      { $s.ligne : no de la derniere ligne utilisee }
compound $h.offset and $s.offset ( VAR-DC, VAR-DECL, PROC-LIST, PROC ):
            int ;
      { $s.offset : derniere valeur d'offset utilisee }
      { $h.offset : valeur d'offset a utiliser }
inherited $level (BLOCK, DECL, VAR-DC, PROC-LIST, VAR, VAR-DECL, PROC,
            STMT, COND, EXPR, SIG, TERM, FACT ):
            int ;
synthesized $nbvar ( VAR-DC ) : int ;
synthesized $act ( COMP, SIGN, OP ) : int ;
synthesized $correct-ins( VAR-DECL, PROC ) : bool ;
synthesized $codegen ( PROGRAM ) : bool ;
synthesized $ligneprog ( BLOCK ) : int ;
global $tabconsult ( BLOCK ) : symbol-table ;

{ ------------------------------------------------------------
                  PARTIE DECLARATION PAR DEFAUT
------------------------------------------------------------ }
local
   correct (TERM, VAR) : bool ;
   recup-proc (TERM) : info-proc ;  { déclaré mais pas défini }
   recup-var (VAR) : info-var ;  { déclaré mais pas défini }
   recup (FACT, PROC, VAR-DECL) : info ;
classes
   $class-code ($code) : code;
   $class-her ($h.ligne(); $h.offset()) : int ;
   $class-syn ($s.ligne(); $s.offset()) : int ;
   $correct ($correct-ins) : bool ;
{ ------------------------------------------------------------
                  PARTIE DEFINITION PAR DEFAUT
------------------------------------------------------------ }
default correct use
   where op -> (X:ID) use
            case op is
                  call : (recup-proc !~ info-proc-undef) ;
                  other : ( recup-var !~ info-var-undef) ;
            end case ;
   end where ;
end default :

default recup use
   where -> (X:ID,Y*) use lookup($id(X),&tabconsult(BLOCK.up)) end where ;
end default :

default $correct use
   where -> (X:ID,Y*) use level_offset_define(recup,$level,$class-her) end where ;
   where null use false end where ;
end default :
```

```
default $class-code use
    where null use empty-code end where ;
    where -> () use empty-code end where ;
    where -> (X)+* use
            let
            ligne-code : ligne-code := map left ligne-code
                                        value ligne-code ()
                                        other $class-code(X)
                                    end map
            in
            code(ligne-code);
    end where ;
    where op (TERM|VAR) -> (X:ID) use
            if correct then
                case op is
                    call : code(ligne($h.ligne,CAL,($level - recup-proc.level),recup-proc.offset)) ;
                    other : code(ligne($h.ligne,STO,($level - recup-var.level),recup-var.offset)) ;
                end case ;
            else
                code(ligne($h.ligne,NIL,0,0))
            end if ;
    end where ;
    where -> (X+) use
            let
            ligne-code : ligne-code := fmap right ligne-code
                                        value ligne-code ()
                                        other $class-code(X)
                                    end fmap
            in
            code(ligne-code);
    end where ;
end default :

default $class-syn use
    where -> (X)*+ use
        case arity is
            0     : $class-her -1 ;
            other $class-syn(X.last) ;
        end case ;
    end where ;
    where null use 0 end where ;
end default :

default $class-her use
    where -> (X)*+ use X :=
        case position is
            first : $class-her ;
            other $class-syn(X.left) + 1;
        end case ;
    end where ;
end default :

{ ————————————————————————————————————————————————————-
                        REGLES SEMANTIQUES
————————————————————————————————————————————————————- }

{ PROGRAM = program }
where program -> BLOCK use
    $codegen := let codeblock : code := code(ligne-code[[
                                    code(ligne(0,JMP,0,$ligneprog(BLOCK))) ,
                                $code(BLOCK) ]])
                in
                print_code("CODE",$scx(BLOCK),codeblock) ;
    $level(BLOCK) := 0 ;
    $h.ligne(BLOCK) := 1 ;
```

```
     $tabconsult(BLOCK) := $symtab (BLOCK);
end where ;

where null-PROGRAM -> use
     $codegen := false ;
end where ;

{ BLOCK = block }
where block -> DECL STMT use
     $code := code(ligne-code[[
                    $code(DECL) ,
                    $code(STMT) ]]) ;
     $h.ligne(STMT) := $s.ligne(DECL) + 1 ;
     $ligneprog := $s.ligne(DECL) ;
end where ;

where null-BLOCK -> use
     $ligneprog := 0 ;
end where ;

{ DECL = decl }
where decl -> CONST VAR-DC PROC-LIST use
     $code := code(ligne-code[[
                    $code(PROC-LIST) ,
                    code(ligne($s.ligne(PROC-LIST) + 1,INT,0,(3+$nbvar(VAR-DC)))) ]]) ;
     $s.ligne := $s.ligne(PROC-LIST) +1;
     $h.offset(VAR-DC) := 3 ; { 0 : offset du premier registre,1 du second,2 du dernier }
     $h.offset(PROC-LIST) := $s.offset(VAR-DC) + 1 ;
end where ;

where null-DECL -> use
end where ;

{ CONST = const-list }
where const-list -> DC-CONST* use
end where ;

where null-CONST -> use
end where ;

{ DC-CONST = const-decl }
where null-DC-CONST -> use
end where ;

where const-decl -> ID INT use
end where ;

{ VAR-DC = var-list }
where null-VAR-DC -> use
     $nbvar := 0 ;
end where ;

where var-list -> VAR-DECL* use
     $nbvar := $arity ;
end where ;

{ VAR-DECL = var-decl }
where null-VAR-DECL -> use
end where ;

where var-decl -> ID use
end where ;

{ PROC-LIST = proc-list }
where null-PROC-LIST -> use
```

```
end where ;

where proc-list -> PROC* use
end where ;

{ PROC = proc }
where null-PROC -> use
end where ;

where proc -> ID BLOCK use
      $tabconsult(BLOCK) := $symtab(BLOCK) ;
      $s.ligne := $s.ligne (BLOCK) + 1 ;
      $code := code(ligne-code[[
                  $code(BLOCK) ,
                  code(ligne($s.ligne(BLOCK)+1,OPR,0,0)) ]]) ;
end where ;

{ STMT = assign , call , ifthen , while , stmt-list }
where null-STMT -> use
end where ;

where assign -> VAR EXPR use
      $s.ligne := $s.ligne(VAR) ;
      $h.ligne(VAR) := $s.ligne(EXPR) + 1 ;
      $h.ligne(EXPR) := $h.ligne ;
end where ;

where call -> ID declare
value
      recup-proc : info-proc := look-proc ($id(ID),$tabconsult(BLOCK.up)) use
end where ;

where ifthen -> COND STMT use
      $code := code(ligne-code[[
                  $code(COND) ,
                  code(ligne($s.ligne(COND)+1,JPC,0,$s.ligne(STMT) +1 )) ,
                  $code(STMT) ]]) ;
      $h.ligne(STMT) := $s.ligne(COND) + 2 ;
end where ;

where while -> COND STMT use
      $s.ligne := $s.ligne(STMT) + 1 ;
      $code := code(ligne-code[[
              $code(COND),
              code(ligne($s.ligne(COND)+1,JPC,0,$s.ligne(STMT) + 2)) ,
              $code(STMT),
              code(ligne($s.ligne(STMT)+1,JMP,0,$h.ligne)) ]]) ;
      $h.ligne(STMT) := $s.ligne(COND) + 2 ;
end where ;

where stmt-list -> STMT+ declare
end where ;

{ COND = odd , comp }
where null-COND -> use
end where ;

where odd -> EXPR use
      $code := code(ligne-code[[
                  $code(EXPR) ,
                  code(ligne($s.ligne(EXPR) + 1,OPR,0,7)) ]]) ;
      $s.ligne := $s.ligne(EXPR)+1;
end where ;

where comp -> EXPR COMP EXPR use
```

```
     $code := code(ligne-code[[
                    $code(EXPR.1) ,
                    $code(EXPR.2) ,
                    code(ligne($s.ligne(EXPR.2) + 1,OPR,0,$act(COMP))) ]]) ;
     $s.ligne := $s.ligne(EXPR.2) + 1 ;
     $h.ligne(EXPR.2) := $s.ligne(EXPR.1) + 1 ;
end where ;

{ COMP = egal , diff , inf , infeg , sup , supeg }
where egal -> use
     $act := 8 ;
end where ;

where diff -> use
     $act := 9 ;
end where ;

where inf -> use
     $act := 10 ;
end where ;

where infeg -> use
     $act := 13 ;
end where ;

where sup -> use
     $act := 12 ;
end where ;

where supeg -> use
     $act := 11 ;
end where ;

where null-COMP -> use
     $act := 6 ;
end where ;

{ VAR = simple-var }
where null-VAR -> use
end where ;

where simple-var -> ID declare
value
     recup-var : info-var := look-var ($id(ID),$tabconsult(BLOCK.up)) use
end where ;

{ EXPR = expr-comp , expr }
where null-EXPR -> use
end where ;

where expr-comp -> SIG TERM EXPR-S use
     $code := code(ligne-code [[
                    $code(TERM) ,
                    $code(SIG) ,
                    $code(EXPR-S) ]]) ;
     $h.ligne(SIG) := $s.ligne(TERM) +1 ;
     $h.ligne(TERM) := $h.ligne ;
     $h.ligne(EXPR-S) := $s.ligne(SIG) +1 ;
end where ;

where expr -> SIG TERM use
     $s.ligne := $s.ligne(SIG) ;
     $h.ligne(SIG) := $s.ligne(TERM) + 1;
     $h.ligne(TERM) := $h.ligne ;
end where ;
```

```
{ EXPR-S = expr-sign-comp , expr-sign }
where null-EXPR-S -> use
end where ;

where expr-sign-comp -> SIGN TERM EXPR-S use
      $code := code(ligne-code [[
                  $code(TERM) ,
                  code(ligne($s.ligne(TERM)+1,OPR,0,$act(SIGN))) ,
                  $code(EXPR-S) ]]) ;
      $h.ligne(TERM) := $h.ligne ;
      $h.ligne(EXPR-S) := $s.ligne(TERM) + 2 ;
end where ;

where expr-sign -> SIGN TERM use
      $code := code(ligne-code [[
                  $code(TERM) ,
                  code(ligne($s.ligne(TERM)+1,OPR,0,$act(SIGN))) ]]) ;
      $s.ligne := $s.ligne(TERM) +1 ;
end where ;

{ SIG = nonsign , splus , sminus }
where null-SIG -> use
end where ;

where nonsign -> use
      $s.ligne := $h.ligne -1 ;
end where ;

where splus -> use
      $s.ligne := $h.ligne -1 ;
end where ;

where sminus -> use
      $code := code(ligne($h.ligne,OPR,0,1)) ;
end where ;

{ SIGN = plus , minus }
where null-SIGN -> use
      $act := 6 ;
end where ;

where plus -> use
      $act := 2;
end where ;

where minus -> use
      $act := 3;
end where ;

{TERM = term-fact , term-comp }
where null-TERM -> use
end where ;

where term-fact -> FACT use
end where ;

where term-comp -> FACT OP FACT use
      $code :=code(ligne-code [[
                  $code(FACT.1) ,
                  $code(FACT.2) ,
                  code(ligne($s.ligne(FACT.2)+1,OPR,0,$act(OP))) ]]);
      $s.ligne := $s.ligne(FACT.2) + 1 ;
      $h.ligne(FACT.2) := $s.ligne(FACT.1) + 1 ;
end where ;
```

```
{ OP = mul , div }
where null-OP -> use
      $act := 6;
end where ;

where mul -> use
      $act := 4;
end where ;

where div -> use
      $act := 5;
end where ;

{ FACt = fact-id , fact-int , fact-expr }
where null-FACT -> use
end where ;

where fact-id -> ID
      declare
      value
      recup : info := lookup ($id(ID),$tabconsult(BLOCK.up)) ;
      correc : bool := if recup ~ info-error then
                              false
                        else
                              true
                        end if ;
      use
      $code := if correc then
                        case recup is
                              info(X:info-var):
                                    code(ligne($h.ligne,LOD,$level - X.level,X.offset)) ;
                              info(X:info-const):
                                    code(ligne($h.ligne,LIT,0,X.val)) ;
                              other : code(ligne($h.ligne,NIL,0,0)) ;
                        end case
                  else
                        code(ligne(0,NIL,0,0))
                  end if ;
end where ;

where fact-int -> INT use
      $code := code(ligne($h.ligne,LIT,0,$value(INT))) ;
end where ;

where fact-expr -> EXPR use
end where ;

{ INT = int }
where null-INT -> use
end where ;

where int -> use
end where ;

{ ID = id }
where null-ID -> use
end where ;

where id -> use
end where ;

end grammar ;
```

Annexe B

Description syntaxique du sur-langage de spécification

<default part> = "defaults" <dec-part> <def-part>;

<dec-part> = <dec-part> <dec-clause> ;
<dec-part> = <dec-clause> ;
<dec-clause> = <local-dec>;
<dec-clause> = <class-dec>;

<local-dec> = "local" <local-list> ;
<local-dec> = ;
<local-list> = <local-list> <local-dc>;
<local-list> = <local-dc>;
<local-dc> = <ID> "(" <phyla*> ")" ":" <type ID> ";";

<class-dec> = "classes" <class-list>;
<class-dec> = ;
<class-list> = <class-list> <class-dc> ;
<class-list> = <class-dc> ;
<class-dc> = <attrib name> "(" <attrib-list> ")" <type-restrict> ";" ;

<attrib-list> = <attrib-list> "," <attrib-elem> ;
<attrib-list> = <attrib-elem> ;
<attrib-elem> = <attr+> <phyl-restrict> ;

<attr+> = <attr+> "," <attrib name> ;
<attr+> = <attrib name> ;

<phyl-restrict> = "(" <phyla+> ")";
<phyl-restrict> = ;

<type-restrict> = ":" <type ID> ;
<type-restrict> = ;

<def-part> = <def-part> <def-clause> ;

```
<def-part>          = ;
<def-clause>        = ”default” <classop> ”use” <match-part> ”end” ”default” ”;” ;

<classop>           = <ID> ;
<classop>           = <attrib name> ;
<match-part>        = <match-part> <match-line> ;
<match-part>        = ;
<match-line>        = ”where” <patt> <semreg> ”end” ”where” ”;” ;

<patt>              = <op> ”->” <rhs> ;
<patt>              = <op> ;

<op>                = ”\” ”(” <list-nop> ”)” ;
<op>                = <list-op> ;

<rhs>               = ”(” <list-var> ”)” <indic> ;
<rhs>               = ;

<indic>             = <starplus> ;
<indic>             = <starplus> <starplus> ;
<indic>             = ;

<semreg>            = ”match” <expr-list> ;
<semreg>            = ”use” <sem rule def> ;
<semreg>            = <regher-list> ;
<sem rule def>      = <expression> ;
<expr-list>         = <expr-list> ”;” <expression> ;
<expr-list>         = <expression> ;
<regher-list>       = <regher-list> ”;” <reg> ;
<regher-list>       = <reg> ;
<reg>               = ”for” <symbol> ”use” <sem rule def> ;

<list-op>           = <listop> ;
<list-op>           = ;
<listop>            = <listop> ”,” <op-ph> ;
<listop>            = <op-ph> ;

<op-ph>             = <ID> <restr-op> ;
<op-ph>             = ”null” <null-pattern> ;

<list-nop>          = <list-nop> ”,” <ID> ;
<list-nop>          = <ID> ;

<null-pattern>      = ”(” <phyla*> ”)”;
<null-pattern>      = ;

<restr-op>          = ”(” <or-phyl> ”)” ;
<restr-op>          = ;

<list-var>          = <listvar> ;
<list-var>          = ;
```

```
<listvar>          = <listvar> "," <var> ;
<listvar>          = <var> ;

<starplus>         = "+";
<starplus>         = "*";

<var>              = <var-desc> <var-restrict> ;
<var-restrict>     = ;
<var-restrict>     = ":" <or-phyl> ;

<var-desc>         = <ID> <starplus> ;
<var-desc>         = <starplus> ;
<var-desc>         = <symbol> ;

<or-phyl>          = <phylum> "|" <or-phyl> ;
<or-phyl>          = <phylum> ;

   ;
```

Annexe C

Exemple d'instanciation de plusieurs gènes

Dans cette section nous allons prendre un exemple permettant de mettre en évidence les comcepts que nous avons présentés jusqu'à maintenant. Cet exemple est celui de l'instanciation de deux gènes sur une grammaire et la génération de la grammaire résultante.

La grammaire d'entrée G est décrite de la manière suivante :

1 : goal → expr
2 : expr → term
3 : expr → expr addop term
4 : term → factor
5 : term → term mulop factor
6 : factor → num
7 : factor → (expr)
8 : num → int
9 : int → digit
10 : int → int digit
11 : digit → 0
12 : digit → 1
13 : addop → +
14 : mulop → $*$

Cette grammaire décrit les expressions portant sur des nombres écrits en binaire.

On peut donc vouloir appliquer un gène d'évaluation de nombres binaires, qui correspond à la grammaire attribuée donnée en exemple dans le chapitre des rappels II.

Ce gène se nomme *nbbin* et est décrit par :

```
OUT : z.val : integer;
IN : b.bin : integer;

1 : z → l
    z.val = l.val
    l.scale = 0
2 : l → l b
    l0.val = l1.val + b.val
    l1.scale = l0.scale + 1
    b.scale = l0.scale
3 : l → b
    l.val = b.val
    b.scale = l.scale
4 : b → 1
    b.val = if (b.bin = 1) then 2^b.scale else b.val = 0
```

L'attribut exporté $z.val$ donne le résultat du calcul complet de la valeur d'un nombre binaire à partir des valeurs des occurrences de l'attribut importé $b.bin$.

La relation $\mathcal{C}or$ se décrit de cette manière :
$\mathcal{C}or^{-1}(z) = \{num\}$
$\mathcal{C}or^{-1}(l) = \{int\}$
$\mathcal{C}or^{-1}(b) = \{digit\}$
Ce qui induit l'ensemble des non-terminaux clefs pour ce gène : $N_k = \{num, int, digit\}$

L'attribut exporté du gène $z.val$ doit correspondre à un attribut importé de la GAO δ que l'on construit sur la grammaire d'entrée G décrite précédemment. De même l'attribut importé $b.bin$ de $nbbin$ doit avoir un attribut exporté correspondant dans δ.

La GAO δ sur laquelle on veut instancier $nbbin$ devient donc :

IN : num.val : integer;
OUT : digit.bin : integer;

1 : goal → expr
2 : expr → term
3 : expr → expr addop term
4 : term → factor
5 : term → term mulop factor
6 : factor → num
7 : factor → (expr)
8 : num → int
9 : int → digit
10 : int → int digit
11 : digit → 0
    digit.bin = 0
12 : digit → 1
    digit.bin = 1
13 : addop → +
14 : mulop → *

On a pour ce gène, les correspondances entre attributs suivantes : $\mathcal{C}or_a(num.val) = z.val$ et $\mathcal{C}or_a(digit.bin) = b.bin$.

Si l'on veut obtenir une GA Δ qui reconnaît et évalue les expressions sur les nombres binaires, on va utiliser en plus un gène appelé *binexpr* qui décrit l'évaluation des expressions.

Ce gène est décrit par :

```
IN : Num.val : integer;
IN : Op.operator : typeop;
OUT : Goal.val : integer;

1 : Goal → Expr
    Goal.val = Expr.val
2 : Expr → Operand
    Expr.val = Operand.val
3 : Expr → Expr Op Expr
    Expr.val = Callop(op.operator, expr1.val,
                      expr3.val)
4 : Operand → Expr
    Operand.val = Expr.val
5 : Operand → num
    Operand.val = Num.val
6 : Num →
7 : Op →
```

On établit alors les correspondances suivantes :
$\mathcal{C}or^{-1}(Goal) = \{goal\}$
$\mathcal{C}or^{-1}(Expr) = \{expr\}$
$\mathcal{C}or^{-1}(Operand) = \{term, factor\}$
$\mathcal{C}or^{-1}(Num) = \{num\}$
$\mathcal{C}or^{-1}(Op) = \{addop, mulop\}$
ce qui induit l'ensemble des non-terminaux clefs pour ce gène :
$N_k = \{goal, expr, term, factor, num, addop, mulop\}$

De la même façon que pour *nbbin* il est nécessaire d'associer des attributs de δ aux attributs importés et exportés de *binexpr*. La GAO δ doit être de la forme suivante (en partant de la forme obtenue précédemment pour le gène *nbbin*) :

IN / OUT : goal.val : integer;
IN / OUT : num.val : integer;
OUT : digit.bin : integer;
OUT : addop.operator : typeop;
OUT : mulop.operator : typeop;

1 : goal → expr
2 : expr → term
3 : expr → expr addop term
4 : term → factor
5 : term → term mulop factor
6 : factor → num
7 : factor → (expr)
8 : num → int
9 : int → digit
10 : int → int digit
11 : digit → 0
  digit.bin = 0
12 : digit → 1
  digit.bin = 1
13 : addop → +
  addop.operator = add
14 : mulop → ∗
  mulop.operator = mul

Figure C.1 : GAO d'entrée δ

L'association des attributs pour ce gène est alors :
$\mathcal{C}or_a(num.val) = Num.val$
$\mathcal{C}or_a(goal.val) = Goal.val$
$\mathcal{C}or_a(addop.operator) = Op.operator$
$\mathcal{C}or_a(mulop.operator) = Op.operator$.

On remarque que l'attribut $num.val$ de δ doit être déclaré à la fois IN et OUT pour permettre d'établir la correspondance entre les deux gènes. L'attribut $goal.val$ doit lui aussi être déclaré IN par la correspondance établie avec le gène $binexpr$ et déclaré OUT en tant qu'attribut de l'axiome.

La première étape de la définition de la fonction de couplage est la construction des régions connexes par la réduction de la grammaire de base de δ, G.

C.1 Définition et construction de la fonction de couplage

Dans cet exemple, la réduction de G pour les deux gènes donne :

- pour le gène *nbbin* :

  La réduction s'effectue suivant le principe de l'instanciation multiple décrit section VI.2.2, page 130.

  L'étape 3 modifiée sert donc à repérer les non-terminaux neutres récursifs qui ici appartiennent tous à la même classe d'équivalence : $E = \{\ expr, term, factor\ \}$

  On obtient alors la grammaire à réduire, sans les terminaux neutres et les non-terminaux morts, et avec le renommage :

  1 : goal → E
  2 : E → E
  3 : E → E addop E
  4 : E → E
  5 : E → E mulop E
  6 : E → num
  7 : E → E
  8 : num → int
  9 : int → digit
  10 : int → int digit
  11 : digit →
  12 : digit →

  A la fin de l'algorithme de réduction, on obtient, pour les productions :

  goal → E
  goal → E addop E
  goal → E mulop E
  goal → num
  num → int
  int → digit
  int → int digit
  digit →
  digit →

  Le non-terminal E représentant les seuls non-terminaux neutres récursifs apparaissent tous en partie droite de productions issues de l'axiome neutre *goal*.

L'instanciation est valide et on obtient, en supprimant les productions réduites inutiles :

goal → num
num → int
int → digit
int → int digit
digit →

La région connexe associée à la production *goal* → *num* n'est pas prise en compte lors de l'étape suivante de mise en correspondance avec le gène.

- pour le gène *binexpr* :

  Il n'y a dans cette réduction, pas de difficultés particulières, et l'on obtient la grammaire réduite :

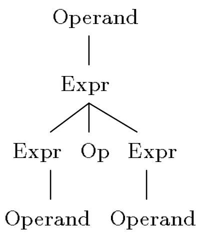
goal → expr
expr → term
expr → expr addop term
term → factor
term → term mulop factor
factor → num
factor → expr
num →
addop →
mulop →

Presque toutes les productions réduites de ces deux grammaires réduites sont associées à des productions du gène, ce qui ne nous posent pas de problème. Toutes sauf une.

Pour le gène *binexpr*, la production réduite : *term* → *term mulop factor* est associée, par les correspondances, à la question suivante, en terme de non-terminaux du gène :

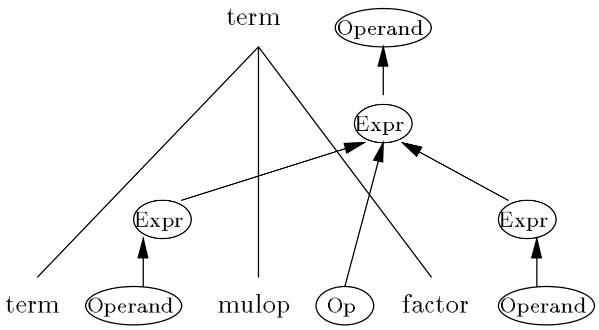
$$Operand \stackrel{*}{\Longrightarrow} Operand\ Op\ Operand\ ?$$

Après le passage dans l'analyseur, on obtient l'unique solution :



Nous devons donc construire cet arbre sur la région connexe associée à cette production réduite, qui s'avère être l'arbre de dérivation constitué de l'unique production numéro 5 dans la grammaire d'entrée :
5 : $term \rightarrow termmulopfactor$

L'unique possibilité pour construire cet arbre est ici :



Pour les autres régions connexes des deux gènes, on n'a pas de problème pour définir et construire correctement la fonction de couplage, et on obtient pour le gène *nbbin*, la GA de couplage décrite figure C.2, et pour le gène *binexpr*, la GA décrite figure C.3

1 :	goal → expr	9 :	int → digit
2 :	expr → term		int.l = digit.b
3 :	expr → expr addop term	10 :	int → int digit
4 :	term → factor		int.l = l(int.l, digit.b)
5 :	term → term mulop factor	11 :	digit → 0
6 :	factor → num		digit.b = b()
	factor.z = z(int.l)	12 :	digit → 1
7 :	factor → (expr)		digit.b = b()
8 :	num → int	13 :	addop → +
	num.z = z(int.l)	14 :	mulop → *

Figure C.2 : Fonction de couplage pour le gène *nbbin*

1 : goal → expr
  goal.Goal = Goal(expr.Expr)
2 : expr → term
  expr.Expr = Expr(term.Operand)
3 : expr → expr addop term
  expr.Expr = Expr(expr.Expr, addop.Op, Expr(term.Operand))
4 : term → factor
  term.Operand = factor.Operand
5 : term → term mulop factor
  term.Operand = Operand(Expr(Expr(term.Operand),
              mulop.Op, Expr(factor.Operand)))
6 : factor → num
  factor.Operand = Operand(num.Num)
7 : factor → (expr)
  factor.Operand = Operand(expr.Expr)
8 : num → int
  num.Num = Num()
9 : int → digit
10 : int → int digit
11 : digit → 0
12 : digit → 1
13 : addop → +
  addop.Op = Op()
14 : mulop → *
  addop.Op = Op()

Figure C.3 : Fonction de couplage pour le gène *binexpr*

C.2 Application de la méta-composition

La figure C.4 montre la GAO résultat de la méta-composition de la fonction de couplage obtenue précédemment et le gène *nbbin*.

OUT : num.z.val : integer;
IN : digit.b.bin : integer;

1 : goal → expr
2 : expr → term
3 : expr → expr addop term
4 : term → factor
5 : term → term mulop factor
6 : factor → num
7 : factor → (expr)
8 : num → int
num.z.val = int.l.val
int.l.scale = 0
9 : int → digit
int.l.val = digit.b.val
digit.b.scale = int.l.scale
10 : int → int digit
int_0.l.val = int_1.l.val + digit.l.val
int_1.l.scale = int_0.l.scale + 1
digit.b.scale = int_0.l.scale
11 : digit → 0
digit.b.val = if (digit.b.bin = 1) then $2^{\text{digit.b.scale}}$ else 0
12 : digit → 1
digit.b.val = if (digit.b.bin = 1) then $2^{\text{digit.b.scale}}$ else 0
13 : addop → +
addop.operator = add
14 : mulop → ∗
mulop.operator = mul

Figure C.4 : Résultat de la méta-composition pour le gène *nbbin*

Tous les attributs du gène sont renommés, mais la GAO produite contient exactement les mêmes attributs importés et exportés que dans le gène.

La figure C.5 montre la GAO résultat de la méta-composition de la fonction de couplage obtenue précédemment et du gène *binexpr*.

IN : num.Num.val : integer;
IN : addop.Op.operator : typeop;
IN : mulop.Op.operator : typeop;
OUT : goal.Goal.val : integer;

1 : goal → expr
goal.Goal.val = expr.Expr.val
2 : expr → term
expr.Expr.val = term.Operand.val
3 : expr → expr addop term
expr.Expr.val = Callop(addop.Op.operator, expr.Expr.val, term.Operand.val)
4 : term → factor
term.Operand.val = factor.Operand.val
5 : term → term mulop factor
term.Operand.val = Callop(mulop.Op.operator, term.Operand.val,
factor.Operand.val)
6 : factor → num
factor.Operand.val = num.Num.val
7 : factor → (expr)
factor.Operand.val = expr.Expr.val
8 : num → int
9 : int → digit
10 : int → int digit
11 : digit → 0
12 : digit → 1
13 : addop → +
14 : mulop → *

Figure C.5 : Résultat de la méta-composition pour le gène *binexpr*

C.3 Application de l'opérateur $\oplus$

La figure C.6 montre le résultat de l'application de l'opérateur $\oplus$ pour les GAO décrites dans le sections précédentes et la GAO d'entrée décrite dans la figure C.1.

OUT : goal.val : integer;

1 : goal → expr
goal.val = expr.Expr.val
2 : expr → term
expr.Expr.val = term.Operand.val
3 : expr → expr addop term
expr.Expr.val = Callop(addop.operator, expr.Expr.val, term.Operand.val)
4 : term → factor
term.Operand.val = factor.Operand.val
5 : term → term mulop factor
term.Operand.val = Callop(mulop.operator, term.Operand.val,
factor.Operand.val)
6 : factor → num
factor.Operand.val = num.val
7 : factor → (expr)
factor.Operand.val = expr.Expr.val
8 : num → int
num.val = int.l.val
int.l.scale = 0
9 : int → digit
int.l.val = digit.b.val
digit.b.scale = int.l.scale
10 : int → int digit
$\text{int.l}_0\text{.val} = \text{int.l}_1\text{.val} + \text{digit.b.val}$
$\text{int.l}_1\text{.scale} = \text{int.l}_0\text{.scale} + 1$
$\text{digit.b.scale} = \text{int.l}_0\text{.scale}$
11 : digit → 0
$\text{digit.b.val} = \text{if (digit.bin = 1) then } 2^{\text{digit.b.scale}} \text{ else } 0$
digit.bin = 0
12 : digit → 1
$\text{digit.b.val} = \text{if (digit.bin = 1) then } 2^{\text{digit.b.scale}} \text{ else } 0$
digit.bin = 1
13 : addop → +
addop.operator = add
14 : mulop → *
mulop.operator = mul

Figure C.6 : Résultat de l'opérateur $\oplus$

Tous les attributs importés et exportés ont été instanciés par des attributs importés et exportés des deux gènes. Ils sont d'ailleurs les seuls dans la grammaire à porter le nom défini dans la GAO d'entrée. Cela concerne les attributs : *goal.val*, *addop.operator*, *mulop.operator*, *num.val* et *digit.bin*.

Seul l'attribut *goal.val* reste exporté en tant qu'attribut de la racine.

Bibliographie

[Alb89] H. Alblas: "*Iteration of Transformation Passes over Attributed Program Trees*", Acta Informatica 27 (1989).

[BaJ78] W. A. Babich, M. Jazayeri: "*The Methods of Attributes for Data Flow Analysis*".

[Bar82] K.Barbar: "*Étude Comparatives des Différentes classes de grammaire d'attributs Ordonnées*" Thèse de troisième cycle, Université de Bordeaux I (1982).

[Bar84] K.Barbar: "*Classification des grammaires d'attributs Ordonnées*", Rapport 8412, Université de Bordeaux I (1984).

[BCD87] P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang & V. Pascual: "*CENTAUR: the System*", rapport RR 777, INRIA, Sophia-Antipolis (Décembre 1987).

[BDL87] P. Boullier, Ph. Deschamp & B. Lorho: "*The SYNTAX Reference Manual*", INRIA, Rocquencourt (Novembre 1987).

[BHR90] D. Binkley, S. Horwitz & T. Reps: "*Interprocedural Slicing Using Dependence Graph*".

[BMC89] C. W. Butler, T. J. Mc Cabe: "*Design Complexity Measurement and Testing*".

[Boc76] G.V. Bochmann: "*Semantic Evaluation from Left to Right*", Comm. of the ACM **19** (2), PP. 55-62 (Février 1976).

[CaP] A. Carle, L. Pollock: "*Incremental Evaluation for Modular Attribute Grammars*".

[CaR87] M. D. Carroll, B. G. Ryder: "*Incremental Data Flow Analysis via Attributes*".

[CaR??] M. D. Carroll, B. G. Ryder: "*Incremental Data Flow Analysis via Dominator and Attribute Updates*".

[CFL88] B. Chabrier, P. Franchi-Zannettacci & V, Lextrait: "*GIGAS: A Graphical Interface Generator from Attribute Specifications*", Université de Nice (1988).

[CFZ82] B.Courcelle, P. Franchi-Zannettacci: "*Attribute Grammars and Recursive Program Schemes*". Théorical Computer Science, 17 (1984).

[ChM79] L. M. Chirica, D. F. Martin: "*An Order-Algebraic Definition of Knuthian Semantics*".

[CNS??] M. L. Christ-Neumann, H. W. Schmidt: "*ASDL: An Object-Oriented Specification Language for Syntax-Directed Environments*".

[CoH79] R. Cohen, E. Harry "*Automatic Generation of Near-Optimal Linear-Time Translators for Non-Circular Attribute Grammars*", 6th ACM POPL, San Antonio, TX, pp 121-134, (1979).

[Der84] P. Deransart : "*Validation des Grammaires d'attributs*" , Thèse d'État, Université de Bordeaux I (Octobre 1984).

[DJL84] P. Deransart, M. Jourdan & B. Lorho: "*Speeding Up Circularity Tests for Attribute Grammars*" , Acta informatica 21, pp 375–391 (1984).

[DRT81] A. Demer, T. Reps & T. Teitelbaum: "*Incremental Evaluation for Attribute Grammars with Application to Syntax-Directed Editors*".

[DuC90] G.D.P. Dueck, G.V. Cormack: "*Modular Attribute Grammars*", Computer J. 33 (1990), 164-172.

[Dur89] O. Durin: "*Génération d'évaluateurs* OLGA *en* LE_LISP", Rapport de stage de D.E.A. (Septembre 1989).

[DVG??] P. L. Della Vigna, C. Guezzy: "*Data Structures and Graph Grammars*".

[EFR89] O. Edelstein, N. Francez, M. Rodeh & S. Sagiv: "*Resolving Circularity in Attribute Grammars with Applications to Data Flow Analysis*".

[EnF82] J. Engelfriet G. Filè : "*Simple Multi-Visit Attribute Grammars*", Memorandum 314, Onderafdeling der informatika, Technische Hogscool (1982).

[Eng84] J. Engelfriet : "*Attribute Grammars: Attribute Evaluation Methods*", Methods and Tools for Compiler Construction (B. Lorho ed.), INRIA-CEC Course, Cambridge University Press, (1984), pp 103-135.

[Fan72] I. Fang "*FOLDS, a Declarative Formal Language Definition System*", PhD thesis, report STAN-CS-72-329, Stanford University (Déc 1972).

[Far83] R. Farrow: "*Automatic Generation of Fixed-Point-Finding Evaluators for Circular, but Well-Defined, Attribute Grammars*" .

[Far86] R. Farrow: "*Automatic Generation of Fixed-Point-Finding Evaluators for Circular, but Well-Defined, Attribute Grammars*" .

[Fil83] G. Filè: "*Interprétation and Réduction of Attribute Grammars*", Acta Informatica 19, 115-150, 1983.

[Fil86] G. Filè: "*Classical and Incremental Evaluators for Attribute Grammars*", 11th Colloquium on Trees in Algebra and Programming (CAAP'86), Nice, LNCS 214, Springer Verlag, 112-126, Mars 1986.

[FMY92] R. Farrow, T. J. Marlowe & D. M. Yellin: "*Composable Attribute Grammars: Support for Modularity in Translator Design and Implementation*", 19th ACM Symposium on Principles of Programming Languages, Albuquerque, NM, 223-234, Jan. 1992.

[Fra88] P. Franchi-Zannettacci: "*Attribute Specifications for Graphical Interface Generation*".

[GHL92] R. W. Gray, V. P. Heuring, S. P. Levi, A. M. Sloane & W. M. Waite: "*Eli : A Complete, Flexible Compiler Construction System*", Communications of the ACM **35**, 121-131, Feb. 1992.

[GaG84] H. Ganzinger & R. Giegerich: "*Attribute Coupled Grammars*", ACM SIGPLAN'84 Symp. on Compiler Construction, Montréal, numéro spécial de ACM SIGPLAN Notices 19, 6 (Juin 1984), 157-170.

[Gan??] H. Ganzinger: "*Transforming Denotational Semantics into Practical Attribute Grammars*".

[Gan??] H. Ganzinger: "*Modular Compiler Descriptions Based on Abstract Semantic Data Types*".

[Gie79] R. Giegerich: "*Introduction to the Compiler Generating System MUG2*", Bericht TUM-INFO-7913, Institut für Informatik, Technische Universität München (Mai 1979).

[Gie86] R. Giegerich: "*On the Relation between Descriptional Composition and Evaluation of Attribute Coupled Grammars*", Forschungbericht 221, University of Dortmund, Jul. 86.

[Gra89] J. O. Graver: "*Type Checking and Type Inference for Object-Oriented Programming Languages*".

[GrW76] S. L. Graham, M. Wegman: "*A Fast and Usually Linear Algorithm for Global Flow Analysis*".

[Gue89] I. Guessarian: "*Improving Fixpoint Tools for Computer Science*".

[HeU73] M. S. Hecht, J. D. Ullman: "*Analysis of a Simple Algorithm for Global Flow Analysis*".

[HoT86] S. Horwitz, T. Teitelbaum: "*Generating Editing Environments Based on Relations and Attributes*".

[JaW75] M. Jazayeri et K. G. Walter : "*Alternating Semantic Evaluator*", ACM 1975 Annual Conf. Minneapolis, MN, Pp 230-234 (1975).

[JLP90] M. Jourdan, C. Le Bellec & D. Parigot, : "*The Olga Attribute Grammar Description Language: Design, Implementation and Evaluation*", Attribute Grammars and their Applications (WAGA), Paris, P. Deransart & M. Jourdan, réds., 222-237, Lect. Notes in Comp. Sci. 461, Springer-Verlag, New York-Heidelberg-Berlin, Sept. 1990.

[JoP88] M. Jourdan, D. Parigot: "*The* FNC–2 *System User's Guide and Reference Manual*",version 0.4 (Février 1989).

[JOR75] Mehdi Jazayeri, W. F. Ogden et W. C. Rounds : "*The Intrinsically Exponential Complexity of the Circularity Problem for Attribute Grammars*" Commun. ACM 18 (1975).

[Jou84] M. Jourdan: "*Les Grammaires Attribuées: Implantation, Applications, Optimisations*", Thèse de docteur-ingénieur, Université de Paris VII, Paris (Mai 1984).

[Jou92] M. Jourdan: "*Des bienfaits de l'analyse statique sur la mise en œuvre des grammaires attribuées*", Thèse d'habilitation (Avril 1992).

[JPJ90] M. Jourdan, D. Parigot, C. Julie, O. Durin, et C. Le Bellec : "*Design, Implementation and Evaluation of the* FNC-2 *Attribute Grammar System*", *SIGPLAN Notices* **25**(6), ACM SIGPLAN '90 Conf. on Programming Languages Design and Implementation,pp. 209–222 (1990).

[Jul89] C. Julie : "*Optimisation de l'espace mémoire pour l'évaluation des grammaires attribuées*", Thèse, Dépt d'Informatique, Univ. Orléans, Sept. 1989.

[JuP90] D. Parigot & C. Julie : "*Space Optimization in the* FNC-2 *Attribute Grammar System*", Attribute Grammars and their Applications (WAGA), Paris, P. Deransart & M. Jourdan, réds., 29-45, Lect. Notes in Comp. Sci. 461, Springer-Verlag, New York-Heidelberg-Berlin, Sept. 1990.

[Kas80] U. Kastens : "*Ordered Attribute Grammars*", Acta Informatica 13, 3, (1980), pp 229-256.

[KaW92] U. Kastens & W. M. Waite : "*Modularity and Reusability in Attribute Grammars*", Fachbereich 102, Univ. of Paderborn, jul. 1992.

[Kas92] U. Kastens : "*LIDO - Short Reference*", Univ. of Paderborn, FRG, Documentation of the LIGA System, 1992.

[Ken??] K. kennedy: "*A survey of Data Flow Analysis Techniques*".

[KeR79] Ken Kennedy et J. Ramanathan : "*A Deterministic Attribute Grammar Evaluator based on Dynamic Sequencing*", ACM Trans. Prog. Lang. Sys. (july 1979).

[KHZ82] U. Kastens, B. Hutt & E. Zimmermann: "*GAG: A Practical Compiler Générator*", LNCS 141, Springer-Verlag, Berlin (1982).

[Kil73] G. A. kildall: "*A Unified Approach to Global Program Optimisations*"1st ACM Symp. on Principles of Progr. Languages, 194-206, Jan. 1973.

[KLM83] G. Kahn, B. Lang, B. Mélèse & E. Marcos: "*Metal: a Formalism to Specify Formalisms*", Sience of Computer Programming **3**, pp. 151-188 (1983).

[Kle91] C. Kleinhans: "*Introduction de la Modularité dans les Grammaires Attribuées*", Thèse, Dépt d'Informatique, Univ. Nice, Feb. 1991.

[Knu68] D.E. Knuth: "*Semantics of Context-Free Languages*", Mathematical Systems Theory **2** (2), pp. 127-145 (Juin 1968).

[Knu90] D.E. Knuth: "*The Genesis of Attribute Grammars*", Attribute Grammars and their Applications (WAGA), Paris, P. Deransart & M. Jourdan, réds., 1-12, Lect. Notes in Comp. Sci. 461, Springer-Verlag, New York-Heidelberg-Berlin, Sept. 1990.

[KoR83] K. Koskimies et K.J. Räihä : "*Modelling of Space-efficient One-pass Translation using Attribute Grammars*", Softrware Practice and Experience 13,2 (1983).

[Kou77] L. T. Kou: "*On Live-Dead Analysis for Global Flow Problems*".

[KRS??] J. Knoop, O. Rüthing & B. Steffen: "*The Value Flow Graph: A Program Representation for Optimal Program Transformations*".

[Lan86] B. Lang: "*The Virtual Tree Processor: Rationale*",Esprit project 348, Third Review Report (Septembre 1986).

[Lor74] B. Lorho "*De la définition à la traduction des langages de programmation: méthode des attributs sémantiques*", Thèse d'État, Université Paul Sabatier, Toulouse-France (1974).

[Lor77] B. Lorho:"*Semantic Attributes Processing in the System DELTA, Methods of Algorithmic Language Implementation*", LNCS**47**, pp 21-40, Springer-Verlag (1977).

[LRS74] M. Lewis, D. J. Rosenkrantz et R. E. Stearns : "*Attributed Translations*, JCSS 9,4 (1974).

[Mug85] K. A. Mughal: "*Control Flow Aspects of Generating Runtime Facilities for Language-Based Programming Environments*".

[NAM74] D. Néel, M. Amirchahy et M. Mazaud : "*Optimisation of Generated Code by Means of Attributes : Local Elimination of Common Redundant Subexpression*, GI4. Jahrestagung, Berlin Ed., Siefks LNCS 26, Springer-Verlag (1974).

[OlO86] K. M. Olender, L. J. Osterweil: "*Specification and Static Evaluation of Sequencing Constraints in Software*".

[OlW89] R. A. Olsson, G. R. Whitehead: "*A Simple Technique for Automatic Recompilation in Modular Programming Languages*".

[PaR86] M. C. Paull, B. G. Ryder: "*Elimination Algorithms for Data Flow Analysis*".

[Par88] Didier Parigot: "*Transformations, Évaluation incrémentale et optimisations des grammaires attribuées : le système* FNC–2", Thèse, Université Paris XI Orsay (Mai 1988).

[Pen90] J. Penjam: "*Computational and Attribute Models of Formal Languages*".

[RaR89] G. Ramalingam, T.Reps: "*Semantics of Program Representation Graphs*".

[RaU81] Kari-Jouko Räihä & Esko Ukkonen : "*Minimizing the Number of Evaluation Passes for Attribute Grammars*", SIAM Jour. on Computing 10 (1981).

[Rei83] S. P. Reiss: "*Generation of Compiler Symbol Processing Mechanisms from Specifications*".

[Rei87] S. P. Reiss: "*Automatic Compiler Production: the Front End*".

[Rep82] T. Reps: "*Generating Language-based Environments*", thèse de PhD, rapport TR 82-514, Dept. of Computer Science, Cornell University, Ithaca, NY (1982).

[Rep84] T. Reps: "*Generating Language-based Environments*", MIT Press, Cambridge, MA, 1984.

[Ros77] B. K. Rosen: "*High-Level Data Flow Analysis*".

[Ros80] B. K. Rosen: "*Monoïds for Rapid Data Flow Analysis*".

[Rou93] G. Roussel: "*Vers une plus grande modularité dans les Grammaires Attribuées*", Thèse non encore soutenue, INRIA-Rocquencourt (93).

[RSS83] K.-J. Räihä, M. Saarinen, M. Sarjakovsky, S. Sippu, E. Soisalon-Soininen & M. Tienari: "*Revised Report on the Compiler Writing System HLP78*", report A-1983-1, DCS, University of Helsinky (Janvier 1983).

[SoW90] M. L. Soffa, D. Whitfield: "*An approach to Ordering Optimizing Transformations*".

[Tar81a] R. Tarjan: "*Fast Algorithms for Solving Path Problems*".

[Tar81b] R. Tarjan: "*A Unified Approach to Path Problems*".

[ThW89] W. Thome, R. Wilhelm: "*Simulating Circular Attribute Grammars trough Attribute Reevaluation*".

[Tie87] M. D. Tiemann: "*Removing Redundancy in Attribute Grammars*", Technical Report, Microœlectronics and Computer Technology Corporation, Austin, TX 78759 (Juillet 87).

[VoM82] A. O. Vooglaid & M. B. Méristé: "*Abstract Attribute Grammars*", Programming and Computer Software **8** (5), pp. 242-251 (Septembre 1982).

[Wal88] J. Walz: "*Incremental Evaluation for a General Class of Circular Attribute Grammars*".

[Wal89] J. Walz: "*Extending Attribute Grammar and Type Inference Algorithms*".

[Wil??] R. Wilhelm: "*Global Flow Analysis and Optimization in the* MUG2 *Compiler generating System*".

[Wir??] N. Wirth: "*Algorithms + Data Structures = Programs*".