Ecole Polytechnique
Promotion 1988

Option Informatique
Directeur: Monsieur Cousot

Rapport d'option

Réalisation du décompilateur d'arbres attribués du système FNC–2

PPAT

(a **P**retty-**P**rinter for **A**ttributed **T**rees)

Jean-Philippe Jouve

Stage effectué du 11 avril 1991 au 1$^{\text{er}}$ juillet 1991

INRIA

Projet Chloé

Directeur de stage: Monsieur Jourdan

Ecole Polytechnique
Promotion 1988

Option Informatique
Directeur: Monsieur Cousot

Rapport d'option

Réalisation du décompilateur d'arbres attribués du système FNC–2

PPAT

(a **P**retty-**P**rinter for **A**ttributed **T**rees)

Jean-Philippe Jouve

Stage effectué du 11 avril 1991 au 1$^{\text{er}}$ juillet 1991

INRIA

Projet Chloé

Directeur de stage: Monsieur Jourdan

Remerciements

Je tiens à remercier Martin Jourdan de m'avoir proposé ce stage, qui m'a permis de découvrir le monde de la recherche en informatique, et d'utiliser les outils puissants que sont les grammaires attribuées.

Merci à Didier Parigot, qui n'a pas compté son temps pour me permettre d'utiliser l'œuvre de sa vie: le système FNC–2.

Merci à toute l'équipe du projet Chloé, et plus spécialement à Olivier Durin, pour leur sympathique accueil au sein de l'INRIA.

Je tiens enfin, à remercier tout particulièrement Anne-Cécile Barrabès, qui a su supporter mes humeurs, soutenir mes efforts et partager ma joie lorsque le *boot-strapping*, (comme elle se plait à le dire) du système PPAT a fonctionné.

Table des matières

Table des figures

Introduction

Depuis que les grammaires attribuées (ou AG) ont été introduites par Knuth [Knu68], elles sont devenues une méthode privilégiée pour décrire et implanter les programmes dirigés par la syntaxe, en particulier les applications de traitement des langages, tels que les compilateurs, les traducteurs, et les éditeurs dirigés par la syntaxe.

De nombreuses recherches ont été effectuées sur les différents aspects des grammaires attribuées, et de nombreux systèmes d'évaluation d'attributs ont été implantés [DJL88].

Parmi ces systèmes, FNC–2 [JP89], développé par le projet Chloé de l'INRIA, est un des plus avancés. Il est en passe de devenir un système de qualité industrielle, et procure une grande puissance d'expression, et une souplesse d'utilisation liée à son caractère fortement modulaire. Son fonctionnement repose sur les derniers résultats relatifs à l'implantation des grammaires attribuées [JPJ90] et l'utilisation de OLGA (**O**uf! un **L**angage pour les **G**rammaires **A**ttribuées), un nouveau langage, conçu spécialement pour la spécification des GAs [JLP90].

L'aspect modulaire a été particulièrement approfondi dans le système FNC–2, permettant ainsi le découpage d'une *grosse* application en plusieurs phases successives. De façon sommaire, nous pouvons dire qu'une phase correspond à un traitement d'un arbre décoré, pouvant conduire à un ou plusieurs arbres décorés. Lors du développement d'une application sous FNC–2, il est souvent souhaitable de *visualiser* ces arbres, soit pour contrôler les étapes intermédiaires (action de *debugging*), soit pour obtenir le résultat final (par exemple le code objet d'un compilateur). Une telle action est appelée *décompilation.*

Actuellement la *décompilation* est une tâche fastidieuse car elle impose l'utilisation d'effets de bords pour spécifier *à la main* le décompilateur, ce qui, comme nous le verrons, est peu souhaitable dans le cadre des grammaires attribuées.

Le but de ce stage est donc de concevoir et de réaliser un constructeur de décompilateurs d'arbres attribués, prenant en entrée la spécification d'un tel arbre et la spécification de la décompilation, afin de produire automatiquement le décompilateur.

Après un rapide survol du système FNC–2 et des différents outils utilisés au cours de ce stage, nous présenterons le langage PPAT, puis l'implantation du système PPAT proprement dit au moyen des grammaires attribuées et du système FNC–2.

Première partie

Le système FNC–2

Résumé

Dans le chapitre qui suit, nous allons présenter les différents outils composant le système FNC–2, puis nous expliquerons le concept d'application dans ce système, enfin nous détaillerons la méthodes de développement d'une telle application dans le cadre du système d'exploitation UNIX.

Chapitre I

Présentation du système FNC–2

I.A Généralités

I.A-1 Buts et qualités du système FNC–2

FNC–2 est un système de traitement des grammaires attribuées dont le but ultime est de fournir un outil de qualité industrielle. Pour ce faire il dispose des qualités suivantes:

- **Efficacité:** les évaluateurs générés, fondés sur le paradigme des séquences de visite [Kas80], sont totalement déterministes. De plus, ce paradigme, permet d'utiliser des techniques d'optimisation de mémoires très efficaces [Kas84, EnJ90, JuP90]. Ces évaluateurs sont de ce fait, aussi compétents en temps et en mémoire, que les programmes écrits à la main et utilisant les arbres comme structures de représentation interne.

- **Puissance d'expression:** Cette efficacité n'est pas réalisée au détriment de la puissance d'expression puisque FNC–2 accepte une très large classe de grammaires attribuées, à savoir les grammaires attribuées fortement non circulaires [CoF82].

- **Facilité d'utilisation:** OLGA, le langage d'entrée de FNC–2, est fortement typé et applicatif, ce qui permet une grande sûreté de programmation.

La puissance du système, repose, comme nous l'avons dit, sur les méthodes d'évaluations. Cependant l'objet de ce stage étant de créer une application dans ce système, nous allons présenter les outils utilisés pour ce développement, et non pas l'implantation de ces outils.

I.A-2 Description d'une application pour FNC–2

Le concept de base de FNC–2 est le suivant: une grammaire attribuée spécifie une application qui transforme une arbre attribué en un autre arbre attribué (en fait zéro, un ou plusieurs).

Ce concept est le fondement de la modularité du système FNC–2. En effet une *grosse* application peut-être découpée en un nombre de passes quelconque, chaque passe prenant comme entrée la représentation intermédiaire générée par la passe précédente, et la transformant en une autre représentation intermédiaire pouvant servir d'entrée à la passe suivante. Chaque passe peut être représentée par une grammaire attribuée, bien sûr, chacune de ces grammaires est beaucoup plus petite et par conséquent plus facilement manipulable que la grammaire attribuée unique représentant la totalité de l'application. Toutes ces passes peuvent être décrites par des grammaires attribuées ou par d'autres outils.

I.B Les Syntaxes Attribuées Abstraites

Pour FNC–2, nous avons vu que, en ce qui concerne les aspects linguistiques, les évaluateurs d'attributs sont considérés comme des applications arbre–arbre, prenant en entrée un arbre décoré et fournissant zéro, un ou plusieurs arbres décorés.

Les arbres décorés qui sont les entrées et les sorties d'un évaluateur d'attributs, sont décrits par des *syntaxes attribuées abstraites.* Ces dernières sont définies dans des unités de compilation séparées, qui jouent le rôle de modules de déclarations vis-à-vis d'une grammaire attribuée donnée, considérée quant-à elle comme un module définissant un évaluateur d'attributs.

La structure syntaxique des arbres sur lesquels opère un évaluateur d'attributs généré par FNC–2 est décrite par une syntaxe abstraite plutôt que par une syntaxe concrète. Ceci a deux avantages principaux:

- la syntaxe abstraite est libérée des contraintes syntaxiques liées à la méthode d'analyse syntaxique (par exemple le problème de la précédence des opérateurs, ou celui du *else* dans des clauses imbriquées), et exprime de façon plus claire les imbrications des structures sémantiques.

- Les arbres abstraits sont en général plus petits que les arbres concrets parce qu'ils sont débarrassés des mots clefs et des autres informations purement textuelles.

Le formalisme de description des arbres abstraits, repose sur la définition de *phyla* et d'*opérateurs*, qui sont respectivement à rapprocher des notions de *non-terminals* et de *productions* des grammaires acontextuelles usuelles.

Nous verrons par la suite comment ces arbres abstraits peuvent être obtenus à partir d'un texte ou comme résultat d'une grammaire attribuée.

Les syntaxes attribuées abstraites peuvent partager, au moyen de clauses d'importation, des phyla, des opérateurs, et des attributs. Ceci est particulièrement intéressant pour décrire une passe dévaluation décorant un arbre sans en modifier la structure.

I.B-1 Opérateurs et Phyla

La syntaxe abstraite d'un langage est faite *d'opérateurs* et de *phyla.*

I.B-1.a Les opérateurs

Les *opérateurs* étiquettent les noeuds des arbres abstraits, et représentent les *caractéristiques* du langage, telles que les opérateurs, les expression conditionnelles, etc.

Les opérateurs sont divisés en deux classes:

- Les opérateurs hétérogènes d'arité fixe, qui peuvent avoir des descendances de types différents, mais en nombre fixe. Parmi eux, les opérateurs d'arité nulle, sont les feuilles de l'arbre et représentent les atomes du langage.

- Les opérateurs homogènes d'arité variable, dont toutes les descendances sont de même type mais dont le nombre peut varier d'une instance à l'autre. Ces opérateurs sont aussi appelés *opérateurs liste* et leur apparition dans un arbre un *nœud liste.*

Le concept de type des descendances est formalisée par la notion de *phylum.*

I.B-1.b Les phyla

Les *phyla* (pluriel de *phylum*) sont des sous-ensembles, non vides et disjoints de l'ensemble des opérateurs, qui représentent les *notions* du langage, par exemple les variables, les expressions, les déclarations, etc.

A chaque position d'une descendance d'un opérateur, est associé un phylum. Ce phylum contient précisément les opérateurs qui peuvent être admis comme opérateur racine du sous-arbre représentant la descendance dans cette position. Ainsi à chaque occurrence dans un arbre abstrait, est associée un phylum qui indique quels opérateurs peuvent apparaître à ce point.

I.B-2 Définition d'une syntaxe attribuée abstraite

Pour définir une syntaxe abstraite, il faut définir les opérateurs, les phyla et les attributs:

- Pour les opérateurs, cela signifie définir leur arité, et les phyla associés à leurs fils.

- Pour les phyla, cela signifie définir les opérateurs qu'ils contiennent.

- Pour les attributs, cela signifie définir les phyla auxquels ils sont attachés et leur type.

Le sous langage d'OLGA utilisé pour définir les syntaxes attribuées abstraites s'appelle *asx.*

GRAMMAIRE	COMMENTAIRES
`grammar boxes-base is`	le nom de la grammaire est `boxes-base`
`root is PPAT ;`	`PPAT` est le phylum contenant l'opérateur représentant la racine de l'arbre
`PPAT = ppat ;`	`PPAT` est un phylum contenant un unique opérateur
`ppat -> BOX ;`	`ppat` est un opérateur d'arité fixe

`BOX = box terminal empty-box ;`
`box -> BOXES ;`
`terminal -> STRING-BOX ;`	

`empty-box -> ;`	`empty-box` est un opérateur d'arité nulle
`BOXES = boxes ;`	
`boxes -> BOX * ;`	`boxes` est un opérateur liste
`STRING-BOX = string-box ;` `string-box -> ;`	
`end grammar ;`	

Figure I.1 : Syntaxe abstraite `boxes-base`

I.B-3 Exemple de définition de syntaxes attribuées abstraites

Pour montrer comment les notions de phyla et d'opérateurs se mettent en œuvre pour définir une syntaxe attribuée abstraite, nous allons donner l'exemple de la syntaxe utilisée par le système PPAT pour effectuer la mise en page.

La syntaxe à définir est celle d'un arbre de boîtes. Nous supposons qu'une boîte est soit une boîte dite *terminale*, c'est-à-dire représentant une chaîne de caractères, soit une liste de boîtes, soit la boîte vide. Nous englobons cette structure dans un phylum appelé PPAT et nous obtenons la grammaire de la figure I.1.

Pour que cette grammaire puisse être utilisée, nous devons définir les attributs permettant de décorer cet arbre; un tel exemple est donné par la grammaire de la figure I.2.

Si nous souhaitons maintenant calculer les coordonnées des boîtes, nous pouvons le faire grâce à une grammaire attribuée prenant en entrée un arbre de type `boxes-in` et fournissant en sortie un arbre décrit par la grammaire `boxes-out` définie à la figure I.3, l'arbre de sortie a la même structure mais nous avons rajouté un attribut représentant les coordonnées des boites aux phyla `BOX` et `PPAT`.

Nous avons vu sur cet exemple le langage de définition des syntaxes abstraites, et nous avons mis en évidence la modularité de ces définitions

GRAMMAIRE	COMMENTAIRES
`grammar boxes-in is`	définition de la grammaire `boxes-in`
`from ppat-predef import all ;`	importation de types complexes
`import grammar boxes-base ;`	importation de tous les phylums et de tous les opérateurs de la grammaire `boxes-base`
`root is PPAT ;`	
`$separator (BOX) : sep-info ;`	le phylum `BOX` est décoré par l'attribut `$separator`
`$string-box (STRING-BOX) : string ;`	le phylum `STRING-BOX` est décoré par l'attribut `$string-box`
`end grammar ;`	

Figure I.2 : Syntaxe abstraite `boxes-in`

GRAMMAIRE	COMMENTAIRES
`grammar boxes-out is`	définition de la grammaire `boxes-out`
`import grammar boxes-in ;`	importation de tous les phylums et de tous les opérateurs de la grammaire `boxes-base`
`root is PPAT ;`	
`$Coord (PPAT , BOX) : coord-box;`	définition d'un attribut supplémentaire sur `PPAT` et `BOX`.
`end grammar ;`	

Figure I.3 : Syntaxe abstraite `boxes-out` grâce à l'utilisation de la clause d'importation.

I.C Le langage de construction des arbres

Nous avons vu que le système FNC–2 utilise des arbres décrits par une syntaxe abstraite. Il est donc nécessaire de disposer d'un langage de construction de ces arbres afin de réaliser le lien entre l'analyseur syntaxique dont nous parlerons au paragraphe I.E et le système FNC–2 proprement dit.

Le langage de construction des arbres du système FNC–2 s'appelle **atc**, il est inspiré de travaux précédents effectués à l'INRIA (le système Metal).

Atc a des liens très étroits avec le langage OLGA, il s'agit presque d'un sous-ensemble des expressions de construction d'arbres de OLGA. Cependant un certain nombre de différences existent:

- *atc* travaille sur une syntaxe concrète et non pas sur une syntaxe abstraite, bien que les arbres générés soient décrits par des syntaxes attribuées abstraites.

- à chaque non-terminal est associé un unique attribut synthétisé, anonyme, représentant un arbre.

- il n'est pas possible de définir de nouveaux attributs dans *atc*, en fait *atc* ne peut manipuler que des éléments de type arbre (pour permettre la construction), ou de type *token* ou *source-index* qui sont les types utilisés pour représenter respectivement les valeurs lexicales des terminaux et la position courante dans le texte d'entrée.

I.C-1 Structure d'une spécification de construction d'arbre

Une spécification de construction d'arbre est composée:

- d'une en-tête permettant de préciser la syntaxe abstraite devant être produite,

- d'éventuelles déclarations de nouveaux phyla et/ou opérateurs, qui seront utilisés au cours de la construction mais qui devront disparaître dans le résultat produit.

- d'éventuelles déclarations de fonctions.

- du corps de la spécification proprement dit, contenant une liste de productions concrètes, auxquelles sont associées des actions de construction d'arbres.

La restriction sur les types manipulés mentionnée plus haut, impose que tous les attributs, de la syntaxe attribuée abstraite de sortie, soient de type *token* ou *source-index*.

I.C-2 Les actions de construction d'arbre

Nous avons vu que le corps d'une spécification *atc* est composé d'un ensemble de productions réelles, auxquelles sont associées des actions de construction d'arbre.

Ces actions ont la forme d'expressions, le langage *atc* étant comme OLGA un langage applicatif, les expressions ne peuvent contenir d'effets de bord. La puissance d'expression d'*atc* est presque identique à celle de OLGA. Les expressions peuvent être composées:

- d'appels de fonctions: les fonctions sont soient des fonctions définies par l'utilisateur, soient des fonctions associées aux opérateurs telles que nous les décrivons au paragraphe I.C-2.a.
- d'expressions conditionnelles (identiques à celles d'OLGA).
- de décorations d'arbre: c'est par ces actions que l'utilisateur peut affecter une valeur à un attribut de la syntaxe attribuée abstraite de sortie.
- de parcourt des *forêts d'arbres* (voir ci-dessous).
- de sélection par la reconnaissance de motifs ("pattern matching"). Nous décrivons ces expressions au paragraphe I.C-2.b.

Nous avons mentionné dans le paragraphe I.B la notion de nœuds liste associés aux opérateurs liste. Soit:

$$\texttt{op}- > \texttt{PH1} * (\texttt{ou} +)$$

un tel noeud. Si l'opérateur `op` appartient au phylum `PH`, on peut considérer qu'un nœud liste associé à `op` est de type `PH`[1], et que chaque fils de ce nœud est de type `PH1`. On parlera alors de *forêt d'arbres* pour faire référence à une structure identifiable à un élément de type `list of PH1`. Nous allons voir au paragraphe I.C-2.b comment un tel élément est obtenu.

I.C-2.a Les fonctions de constructions d'arbres

A chaque opérateur de la syntaxe attribuée abstraite de sortie, est associé une ou plusieurs fonctions. Ces fonctions peuvent être surchargées, la figure I.4 donne le ou les profils des fonctions associées à chaque type d'opérateur.

Dans cette figure, le type `list of PH1` fait référence au type associé à la *forêt d'arbres* dont nous avons expliqué la signification plus haut.

[1]nous verrons par la suite qu'il est presque possible d'identifier un phylum à un type et un opérateur à une fonction. Cette unification n'est pas actuellement totale, mais elle est un des objectifs des développeurs de FNC–2

Phylum	opérateur	profils des fonctions
PH	`op -> PH1 PH2 ...PHn`	`op (PH1 ; PH2 ; ...; PHn) : PH`

PH	`op -> PH1 *`
	`op -> PH1 +`
		`op () : PH`
		`op (PH1) : PH`
		`op-post (PH ; PH1) : PH`
		`op-pre (PH1 ; PH) : PH`
		`op-merge (list of PH1 ; list of PH1) : PH`
		`op-all (list of PH1) : PH`

Figure I.4 : Fonctions de constructions d'arbres associées aux opérateurs

I.C-2.b La reconnaissance de motifs ou le "pattern matching"

Une expression représentant un arbre (construit ailleurs), peut être comparée à un ou plusieurs motifs ("pattern"). Ce mécanisme de comparaison à un motif permet de lier des variables locales à des sous-arbres de l'expression (ou à l'arbre lui-même).

Un motif est soit une constante, soit une variable, soit un constructeur de motif, identique à une fonction, dont les paramètres sont eux-mêmes des motifs.

Pour un nœud d'arité fixe la comparaison ne peut donner lieu à un résultat positif, que pour un motif dont le constructeur correspond à la fonction de construction telle qu'elle a été décrite au paragraphe I.C-2.a.

Pour un nœud liste issu de l'opérateur `op` appartenant au phylum `PH` dont les fils appartiennent au phylum `PH1`, les motifs pouvant conduire à une comparaison positive sont:

- `op ()` : la comparaison sera positive pour la liste vide uniquement.

- `op(M)` : où `M` est un motif pouvant être comparé à un élément appartenant au phylum `PH1`, conduira à une comparaison positive si `op` n'a qu'un fils dont l'élément coïncide avec `M`.

- `op-post(M1,M2)` : où `M1` est un motif pouvant être comparé à un arbre dont la racine est `op`, et `M2` est un motif pouvant être comparé à un élément appartenant au phylum `PH1`, conduira à une comparaison positive si `op` a au moins deux fils dont le plus à droite (le dernier) coïncide avec `M2` et le reste de la liste coïncide avec `M1`.

- `op-pre(M2,M1)` : similaire au précédent mais `M2` doit coïncider avec le premier fils et `M1` avec le reste de la liste.

- `op-all (X)` : où `X` **doit** être une **variable non typée** coïncide toujours et permet de lier `X` à la *forêt d'arbres* associée.

I.D OLGA, le langage de description des grammaires attribuées

Le langage OLGA a été conçu pour décrire les grammaires attribuées. Ceci suppose deux aspects:

- OLGA permet de spécifier tout ce qui est nécessaire pour définir une grammaire attribuée, cela signifie en particulier, définir les attributs, permettre leur calcul dans les règles sémantiques, etc.

- mais OLGA est aussi un langage applicatif à vocation générale.

Nous allons aborder ces deux aspects de OLGA.

I.D-1 OLGA: un langage applicatif à vocation générale

OLGA est un langage purement applicatif, cela signifie qu'il n'existe pas d'effets de bords.

OLGA est un langage fortement typé. Les constructeurs de types sont les énumérations, les sous-ensembles de types scalaires, les enregistrements, les unions, et les listes homogènes. A chaque type est associé une fonction de construction (ou de conversion). De plus, comme nous le verrons au paragraphe I.D-2, dans le cadre des grammaires attribuées *fonctionnelles*, il existe des types arbres et des valeurs arbres.

Les fonctions, y compris celles définies par l'utilisateur, peuvent être polymorphes, bien que, à l'heure actuelle, ce polymorphisme soit réduit aux types ensembles et aux types liste. Le nom des fonctions peut-être surchargé, ce qui augmente la lisibilité des spécifications.

OLGA est un langage possédant une structure de bloc, les éléments de base du langage étant proches de ceux de Ada.

Tout comme Ada, OLGA possède la notion de modules, permettant de définir un ensemble d'objets liés entre eux. Cette particularité participe à la grande modularité du système FNC–2. Un module d'OLGA est divisé en deux unités de compilation:

- un *module de déclaration* dans lequel sont regroupés les définitions de types, de constantes et le profil des fonctions; ces objets étant *visibles* de l'extérieur.

- un *module de définition* contenant l'implantation de ces objets et éventuellement des objets *invisibles*.

En plus des éléments de base d'un langage, OLGA possède une grande puissance d'expression grâce, en particulier, à la notion de reconnaissances de motifs ("pattern matching"). Cette notion a déjà été expliquée dans le paragraphe I.C, le *pattern matching* d'OLGA est similaire à celui de *atc*. Cependant la reconnaissance de motifs sur des éléments de type arbre n'est pas encore accessible au sein des règles sémantiques car cela pose un problème au niveau de la dépendance des attributs.

Il existe de plus des violations au caractère applicatif de OLGA, en particulier la faculté de produire des messages d'erreurs et celle d'appeler des fonctions *externes*, donc pouvant produire des effets de bords.

I.D-2 OLGA: un langage de description des grammaires attribuées

OLGA permet de définir des grammaires attribuées. Il existe en fait pour le système FNC–2 deux types de grammaires attribuées:

- les grammaires attribuées *fonctionnelles* : ces grammaires peuvent avoir plusieurs arbres de sortie dont les syntaxes abstraites sont différentes (tant au niveau des attributs, qu'au niveau de la structure même de l'arbre) de la syntaxe abstraite de l'arbre d'entrée.

- les grammaires attribuées *procédurales* : ces grammaires prennent en entrée un arbre décoré et fournissent comme résultat le même arbre (au niveau de la structure), mais décoré de façon différente (possibilité de rajouter des attributs ou d'en supprimer). Ces grammaires réalisent en fait un effet de bord.

Ces deux types de grammaires ont en commun toute la partie qui concerne la déclaration des attributs et la structure générale de la spécification. Nous allons donc dans un premier temps exposer les points commun puis nous verrons les particularités relatives à ces deux types de grammaires.

I.D-2.a Les attributs dans une grammaire attribuée

On distingue trois classes d'attributs:

- les attributs *importés*: ce sont les attributs décorant l'arbre d'entrée de la grammaire.

- les attributs *de travail*: ce sont des attributs locaux à la grammaire, ils servent au calcul mais ne sont pas accrochés à l'arbre de sortie et par conséquent ils sont invisibles pour l'utilisateur (à l'extérieur de la grammaire).

- les attributs *exportés*: ce sont les attributs décorant l'arbre (ou les arbres) de sortie de la grammaire.

Les attributs *importés* sont définis dans la spécification de la syntaxe attribuée abstraite de la grammaire d'entrée, ils sont donc accessibles dans la grammaire attribuée, mais ne peuvent en aucun cas être modifiés.

Les attributs *exportés* quant-à eux sont définis (pour ce qui est de leur type et des phyla auxquels ils sont accrochés) dans la spécification de la syntaxe attribuée abstraite de la grammaire de sortie, mais il faut définir dans la grammaire attribuée leur mode de calcul, c'est-à-dire, comme nous allons le voir leur catégorie.

Il existe quatre catégories pour les attributs *exportés* et les attributs *de travail*. Ils peuvent être:

- *synthétisé*: la valeur d'un tel attribut est calculé au cours de la visite du nœud correspondant au phylum auquel il est accroché. Ces attributs servent donc à propager l'information du *bas* vers le *haut* de l'arbre.
- *hérité*: la valeur d'un tel attribut est transmise au phylum auquel il est accroché, par le père de ce phylum. Ces attributs servent à propager l'information de la racine vers les feuilles.
- *mixte*: un tel attribut peut être considérer comme l'association d'un attribut synthétisé et d'un attribut hérité. Leur intérêt est de pouvoir bénéficier de la génération des règles par défaut (paragraphe I.D-2.b).
- *global*: un attribut de cette catégorie, est accessible dans tout le sous-arbre dont la racine est un opérateur appartenant au phylum auquel il est accroché. L'intérêt d'un tel attribut est de permettre d'avoir accès à une information globale sans pour autant propager un attribut pour transporter cette information.

I.D-2.b Règles de copies par défaut

Le système FNC–2 procure, afin de clarifier les spécifications de grammaires attribuées, un mécanisme de génération de règles sémantiques par défaut. Ces règles par défaut sont générées lorsqu'une valeur doit être affectée à un attribut et que celle-ci n'est pas fournie par l'utilisateur. Voyons l'action de ces règles pour les différentes catégories d'attributs.

- Pour un attribut synthétisé, la valeur pour le père, est celle de son fils le plus à droite parmi ceux sur lesquels l'attribut est défini (si un tel fils n'existe pas cela provoque une erreur).
- Pour la partie synthétisé d'un attribut mixte, la valeur pour le père, est celle de son fils le plus à droite parmi ceux sur lesquels l'attribut est défini, si un tel fils existe, sinon se sera la valeur de la partie hérité de cet attribut.
- Pour un attribut hérité, la valeur de l'attribut pour un fils sera:
  - si il s'agit du fils le plus à gauche, la valeur de cet attribut pour le père si elle existe et une erreur autrement.
  - si il ne s'agit pas du fils le plus à gauche, la valeur de cet attribut pour son frère, immédiatement à gauche, si elle existe et une erreur sinon.
- Pour la partie hérité d'un attribut mixte, la valeur de l'attribut pour un fils sera:
  - si il s'agit du fils le plus à gauche, la valeur de la partie hérité de cet attribut pour le père.
  - si il ne s'agit pas du fils le plus à gauche, la valeur de la partie synthétisée de cet attribut pour son frère immédiatement à gauche si elle existe, et une erreur sinon.

I.D-2.c Productions et règles sémantiques

La partie principale d'une spécification de grammaire attribuée consiste dans la définition des règles sémantiques associées aux diverses productions.

En fait, chaque production correspond à un bloc, dans lequel il est possible de définir des valeurs (aussi appelées *attributs locaux*), des fonctions, etc. Ces définitions sont locales à la production et par conséquent *invisibles* à l'extérieur.

Les règles sémantiques indiquent quand à elles la façon de calculer les diverses occurences d'attributs, et sont donc de la forme d'une affectation d'une expression à une occurence d'attribut. Toutes les expressions classiques de OLGA peuvent être utilisées, il faut y rajouter la possibilité de *parcourir* la liste des fils pour un opérateur liste , la possibilité d'accéder à la position d'un phylum dans une production (en particulier pour un opérateur liste), et la possibilité d'accéder à l'arité d'un opérateur.

Le système FNC–2 détermine l'ordre d'évaluation des attributs en fonction de ces règles sémantiques, et des dépendances qu'elles impliquent au sein de l'ensemble des attributs. Mentionnons que le nom FNC–2 signifie **F**ortement **N**on **C**irculaire, il n'est donc pas possible de créer des dépendances circulaires entre les attributs, nous verrons au cours de l'étude de PPAT, un exemple de restriction apportée par cette définition.

L'évaluateur d'attributs généré par FNC–2 et correspondant à une grammaire attribuée, a pour objectif de calculer tous les attributs de la racine, et uniquement ceux-là. Cela signifie que l'évaluateur ne calcule en fait que les attributs dont les attributs de la racine dépendent, cette restriction peut conduire à forcer des dépendances pour obtenir le résultat souhaité[2]. De plus seuls les attributs de la racine sont visibles à l'extérieur de la grammaire attribuée (c'est-à-dire dans l'interface d'appel en langage C).

I.D-2.d Modularité

Une spécification de grammaire attribuée peut contenir des clauses d'importation de modules, afin d'inclure des définitions de types, de fonctions ou de constantes. Ceci permet de réduire les spécifications de grammaires attribuées à la partie *intéressante* au niveau de la sémantique.

Notons cependant que pour le moment, il n'est pas possible dans un module séparé de faire référence aux types associés à une syntaxe abstraite, cela impose que les fonctions faisant référence à ces types sont présentes dans la spécification de la grammaire attribuée[3].

I.D-2.e Les grammaires attribuées procédurales

Dans une spécification de grammaire attribuée procédurale, l'arbre d'entrée et l'arbre de sortie ont le même format. Il est cependant nécessaire de

[2]Dans le futur il est prévu de pouvoir choisir explicitement les attributs qui doivent être calculés

[3]Cette restriction sera éliminée par la suite, ce qui permettra en outre de simplifier considérablement certaines spécifications de PPAT et d'en augmenter la lisibilité.

spécifier la syntaxe abstraite d'entrée et la syntaxe abstraite de sortie, car les deux arbres ne sont pas décorés de la même façon.

C'est dans le cadre de ces grammaires, que la possibilité d'importer une syntaxe abstraite dans une autre est particulièrement utile. En effet l'utilisateur écrira en général une spécification représentant la syntaxe abstraite sans attribut, et il importera cette syntaxe dans la syntaxe d'entrée et la syntaxe de sortie, de sorte que les deux spécifications correspondante ne contiennent que des définitions d'attributs.

I.D-2.f Les grammaires attribuées fonctionnelles

Le rôle d'une grammaire attribuée fonctionnelle, étant de produire en sortie un ou plusieurs arbres, différents de l'arbre d'entrée, il est nécessaire de disposer d'expressions permettant de construire ces arbres.

Les expressions de construction d'arbres, sont, comme dans le cadre du langage *atc* (voir paragraph I.C) des appels aux fonctions associées aux opérateurs.

De façon pratique, puisque nous avons vu que seuls les attributs de la racine sont calculés et accessibles à l'extérieur de la grammaire, l'utilisateur accroche à la racine de l'arbre d'entrée des attributs représentant les arbres de sortie.

I.E le système SYNTAX

Nous avons vu que le système FNC–2 est un système de traitement d'arbres attribués. Nous avons expliqué le fonctionnement des grammaires attribuées, (qui du point de vu de FNC–2 sont des applications de l'ensemble des syntaxes abstraites attribuées, dans lui-même), et nous avons présenté le langage *atc* permettant la construction d'arbres abstraits attribués.

Nous présentons maintenant brièvement le système SYNTAX qui n'est pas partie intégrante de FNC–2, mais qui est utilisé par *atc* pour effectuer l'analyse syntaxique d'un texte.

En effet nous avons vu au paragraphe I.C que le corps d'une spécification *atc* est composée d'une liste de productions concrètes associées à des actions de constructions d'arbres. Or justement le système SYNTAX (dont on peut comparer le rôle aux utilitaires LEX et YACC de UNIX) construit un analyseur syntaxique à partir de la définition d'une grammaire concrète.

De plus le système SYNTAX construit, à partir d'une description lexicale du langage (écrite dans un langage appelé *lecl*)et des tables issues de l'analyse de la grammaire concrète, un analyseur lexical.

En plus de la définition lexicale du langage et de la grammaire concrète, l'utilisateur peut fournir à SYNTAX des spécifications permettant de résoudre les conflits intervenant lors de la construction de l'analyseur syntaxique ascendant par la méthode LALR(1). Il s'agit du langage *prio* qui est très proche de la partie équivalente de YACC.

L'avantage le plus appréciable de SYNTAX est son traitement des erreurs. Ce traitement se décompose en trois ou quatre phases:

1. détection

2. affichage sur le terminal du message d'erreur et de la position de l'erreur.

3. tentative de correction

4. rattrapage, si la correction a échoué.

Tous ces aspects de traitement de l'erreur sont décrits séparément de la grammaire, dans un langage appelé *recor*.

I.F Création d'une application sous FNC–2

Nous avons exposé les principaux outils permettant de construire une application pour le système FNC–2, nous allons maintenant voir les principes gouvernant l'utilisation de ces outils pour définir une telle application, puis nous exposerons brièvement la mise en œuvre de ces principes sous UNIX.

I.F-1 Qu'est-ce qu'une application pour FNC–2

La structure typique d'une application pour FNC–2 est représentée sur la figure I.5 que nous allons commenter. Tout d'abord les objets entourés par des rectangles représentent des parties du programme global, nous verrons au paragraphe suivant comment ils sont obtenus dans la pratique.

Les parties entourées par une forme ellipsoïdale représentent les fichiers propres à FNC–2, que l'on regroupe sous le terme générique de *spécifications*, la première ligne de l'objet explique le contenu de la spécification et la seconde le langage dans laquelle elle est écrite. Etudions les différents types de spécifications:

lecl il s'agit de la description lexicale du langage dans lequel est écrit le texte source.

atc comme nous l'avons vu cette spécification définie la grammaire concrète du langage du texte source et les actions de constructions d'arbres. Combinée avec la spécification *lecl*, elle permet à SYNTAX de construire les analyseurs correspondant au langage source.

asx ces spécifications permettent de définir les syntaxes abstraites attribuées, les flèches en pointillés indiquent le format de l'arbre produit par une partie de l'application totale.

olga ces spécifications définissent soit des grammaires attribuées, soit des modules (de définition ou de déclaration). La compilation de ces spécifications, par les moyens que nous étudierons au paragraphe suivant, permet de créer les différentes phases de l'application.

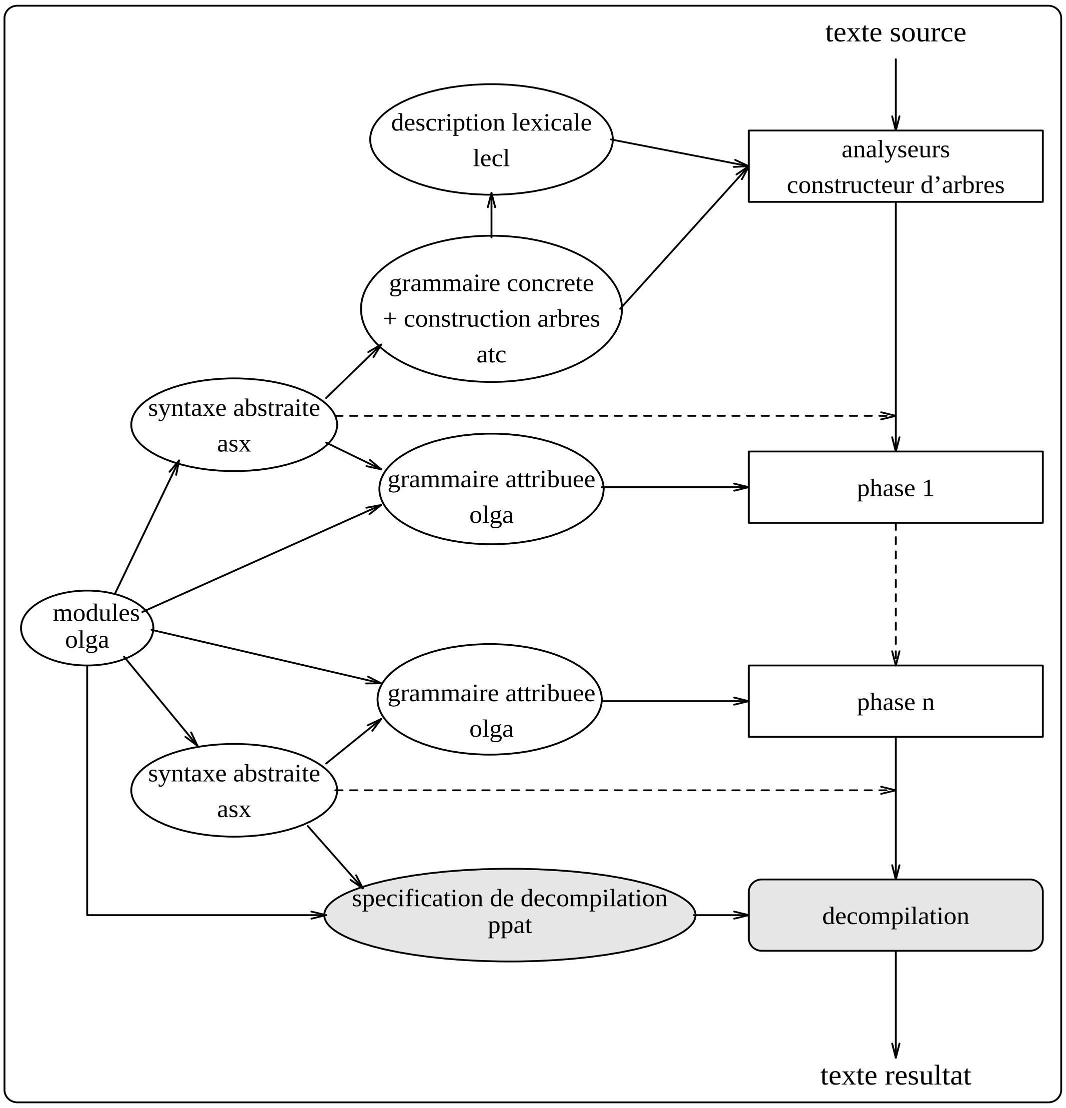

Figure I.5 : Structure d'une application type ppat la spécification *ppat*, ainsi que toute la partie grisée, correspond à l'objet du stage. Cette partie figure sur le schéma de la structure générale d'une application pour exposer les principes définitifs, c'est-à-dire les principes de construction d'une application après la création de PPAT. Actuellement, l'application de décompilation doit être écrite à la main, en général au moyen d'une grammaire attribuée réalisant des effets de bords.

I.F-2 Création d'une application sous UNIX

Nous avons donné au paragraphe précédent la structure générique d'une application du point de vu de FNC–2. Nous allons ici expliquer comment une telle application est effectivement obtenue.

Tout d'abord, nous supposons que nous travaillons sous le système d'exploitation UNIX, et que l'utilisateur dispose de tous les outils de FNC–2, et qu'il souhaite créer l'application de nom `XXX`.

La première étape consiste à créer une structure de répertoires contenant certains fichiers, dont nous allons donner la signification, comme indiqué sur la figure I.6.

- le fichier `XXX/makeXXX` est un *makefile*, c'est-à-dire une description de compilations et de dépendances pour tous les fichiers situés dans le répertoire `XXX/src`. Ce fichier doit être écrit *à la main*.

- Le répertoire `XXX/spec` contient toutes les spécifications nécessaires à FNC–2 pour créer l'application `XXX`.

- Le fichier `XXX/XXX.mkfnc2` est une description, pour FNC–2, de l'application `XXX`. Ce fichier contient la liste de toutes les *spécifications* situées dans le répertoire `XXX/spec`, et pour chacune d'elle, son type (grammaire attribuée, module de déclaration, module de définition, syntaxe abstraite, module *atc*, spécification ppat), et toutes les spécifications dont elle dépend. Ce fichier est *traité* par une application du système FNC–2, appelée `mkfnc2`, pour créer des *makefile*, dont le rôle est d'appeler les outils de FNC–2 dans un ordre respectant les dépendances indiquées.

- Le répertoire `XXX/src`, contient, en plus des fichiers de *sources* crées par l'utilisateur pour fournir, par exemple, le code des fonctions externes, trois fichiers `XXX_main.c` , `XXX_sact.c`, `XXX_smp.c`, dont la présence est obligatoire, qui doivent être écrits par l'utilisateur, et dont les rôles sont:

  – `XXX_main.c` est le fichier contenant la fonction `main` de l'application `XXX`, cette fonction effectue entre autre l'appel à SYNTAX, et lance l'exécution des évaluateurs d'attributs au moyen de fonctions contenues dans le fichier `XXX_smp.c`. C'est dans ce fichier que se fait l'éventuel traitement des arguments d'appel de `XXX`.

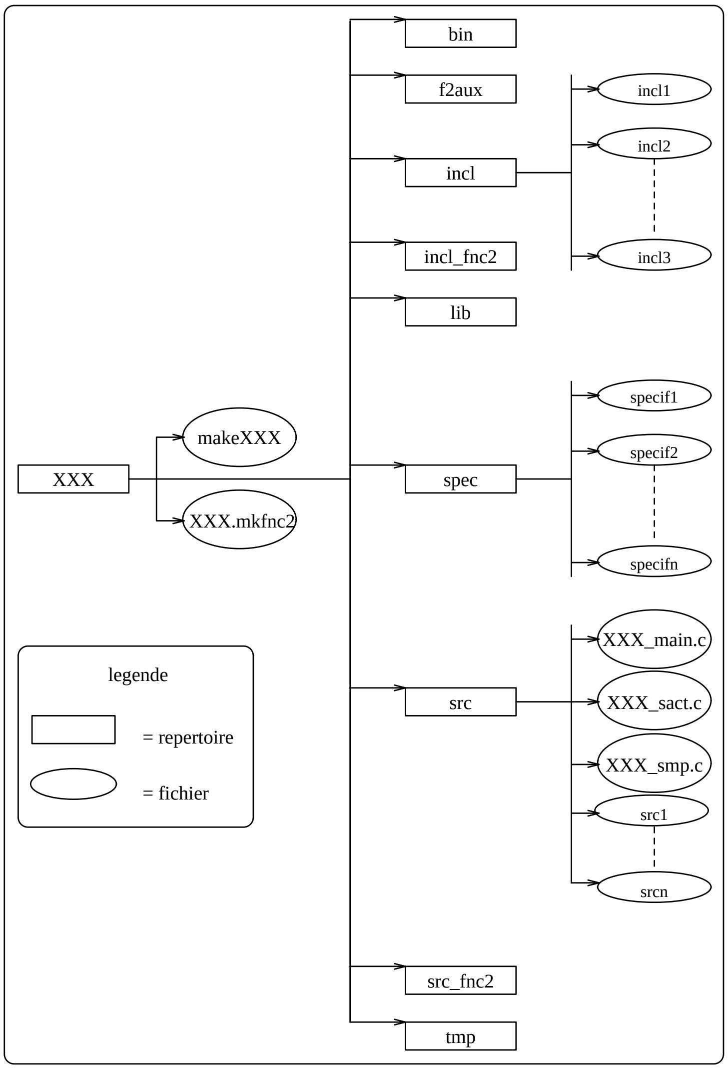

Figure I.6 : Structure du répertoire pour créer l'application `XXX`

- `XXX_smp.c` est le fichier contenant la fonction d'appel des évaluateurs d'attributs correspondant aux grammaires attribuées. C'est donc en écrivant le corps de cette fonction que l'utilisateur réalise l'enchaînement des diverses phases de l'application et qu'il utilise les résultats des évaluateurs d'attributs.
- `XXX_sact.c` est le fichier contenant les actions à effectuer au cours de l'analyse syntaxique (typiquement le traitement des commentaires imbriqués).

• Le répertoire `XXX/incl`, contient les fichiers de définitions associés au fichiers sources de l'utilisateur.

Nous avons décrit tous les éléments nécessaires à la création d'une application sous FNC–2. La compilation de cette application se fait par un appel à l'utilitaire `make` de UNIX, qui effectue les compilations partielles spécifiées dans `XXX/makeXXX`, et dans les *makefile* crées par `mkfnc2`. Cette compilation peut se décomposer en deux étapes:

1. appel des constructeurs FNC–2 sur toutes les spécifications. Tous les fichiers généré par FNC–2 sont placés dans:
   - `XXX/src_fnc2` pour les fichiers *source* (c'est-à-dire les fichiers `.c`).
   - `XXX/incl_fnc2` pour les fichiers de *définition*, (c'est-à-dire les fichiers `.h`), associés aux précédents.
   - `XXX/tmp` et `XXX/f2aux` pour les fichiers temporaires ou les fichiers traçant l'exécution des constructeurs.
2. compilation du langage C produit par ces constructeurs.

Les fichiers objets issus de la compilation des fichiers *source*, sont placés dans le répertoire `XXX/lib`, et l'exécutable `XXX` est placé dans le répertoire `XXX/bin`.

Conclusion

Nous avons présenté les différentes parties de FNC–2 permettant la construction d'une application, et nous avons expliqué comment ces outils étaient appelés pour construire effectivement cette application.

L'existence de l'outil `mkfnc2` permet de simplifier la tâche de l'utilisateur, car celui-ci n'a besoin de créer que les fichiers réellement utiles pour son application, sans se préoccuper de la façon dont ces fichiers seront compilés.

Il faut noter de plus l'existence d'un outil supplémentaire, facilitant encore plus la tâche de l'utilisateur, appelé `FNC-2`, qui évite l'écriture du fichier `XXX.mkfnc2`, et enchaîne, de façon *interactive*, les différentes phases de la compilation, après avoir crée lui-même le fichier `XXX.mkfnc2` (en lançant les constructeurs FNC–2 sur toutes les spécifications présentes dans `XXX/spec`, de façon à extraire les dépendances).

Deuxième partie

Le système PPAT

Résumé

Nous rappelons que le but de ce stage est de créer une application pour le système FNC–2, dans le sens que nous avons décrit au chapitre I.

Dans un premier temps, nous allons définir le langage correspondant à cette application. Ensuite, nous expliquerons comment nous avons implanté l'application PPAT.

Chapitre II

Le langage PPAT

Introduction

Le but du langage PPAT est de fournir à l'utilisateur du système FNC–2, un outil lui permettant d'obtenir des représentations concrètes des arbres qu'il manipule.

Nous avons donc conçu le langage PPAT pour qu'il soit très proche, syntaxiquement et sémantiquement du langage OLGA. Nous avons cherché à conserver au sein de PPAT la puissance d'expression de OLGA, de façon à permettre à l'utilisateur de *décompiler* des arbres complexes avec une souplesse maximale.

Le formalisme de mise en page est inspiré de celui de PPML (Pretty-Printer Meta-Language [Bor88]), qui est un langage de spécification de décompilation d'arbres abstraits (non attribués).

II.A Principes de base du langage PPAT

Un programme PPAT permet de spécifier la manière dont sera décompilé un arbre attribué abstrait d'un certain type. Nous avons vu au chapitre I que le type d'un arbre abstrait était défini dans la spécification d'une syntaxe attribuée abstraite (spécification *asx*). Le principe général de fonctionnement de PPAT est alors schématisé par la figure II.1.

De même que dans une spécification OLGA de grammaire attribuée, on défini pour chaque nœud de l'arbre un ensemble de règles sémantiques, dans une spécification de décompilation PPAT, on défini pour chaque nœud de l'arbre, un *format de d'édition.* Comme dans OLGA, un nœud de l'arbre est symbolisé par la définition de l'opérateur portant sur ce nœud.

Pour décrire le *format d'édition*, on utilise un langage de boîtes, semblable à celui de PPML. Une boîte peut-être de trois types:

- *vide*, qui est une boîte de hauteur et de largeur nulles.
- *simple*, type utilisé pour représenter un objet individuel.
- *composé*, type correspondant à une liste de boîtes, positionnées les unes par rapport aux autres au moyen d'un *séparateur*.

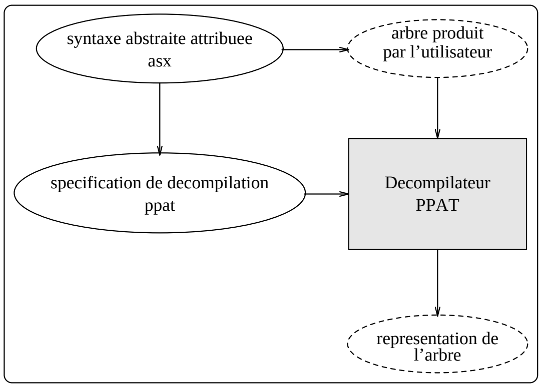

Figure II.1 : Schéma général de l'application PPAT

Contrairement à PPML, PPAT permet de décompiler des arbres attribués. Pour ce faire, à chaque type complexe devant être affiché, l'utilisateur doit faire correspondre un format d'édition. Ce format est décrit de la même façon que celle utilisée pour les nœuds de l'arbre.

Un programme PPAT est donc un ensemble de règles:

$$\begin{array}{lcl} < noeud > & \leadsto & < boite > \\ < type > & \leadsto & < boite > \end{array}$$

A ce principe de base viennent s'ajouter les facilités suivantes:

- des structures de contrôles, semblables à celles d'OLGA, permettent de créer des formats dirigés par la valeur d'attributs, d'expressions, du type d'un arbre[1], etc
- Il est possible de transmettre des informations, spécifiques à la décompilation, dans l'arbre au moyen de *contextes*, qui se comportent comme des attributs hérités.
- la forme des expressions de PPAT, est similaire à celle des expressions OLGA, en particulier il est possible de faire des appels de fonctions définies par l'utilisateur dans des modules externes.

II.B Manuel de référence

Les spécifications lexicales et syntaxiques de PPAT, qui sont données dans l'annexe A de ce rapport, sont exprimées dans les deux méta-langages de SYNTAX, respectivement *lecl* et *bnf* (pour plus de détails, voir le paragraphe I.E).

[1] La sélection en fonction du type d'un arbre est implanté au niveau de PPAT, mais son utilisation demeure impossible en raison de l'absence de cette possibilité dans OLGA (voir le chapitre IV). Cette faculté sera donc disponible dans PPAT, sitôt qu'elle le sera dans OLGA.

II.B-1 Définition lexicale

La définition lexicale de PPAT est fondée sur celle du langage OLGA. les séparateurs et opérateurs utilisés sont:

```
! != !~ % & ( ) * + , - -> .
/ : := ; < <= = > >= [ ] | ~ "
```

les mots réservés sont:

```
$arity $position $scx   and   case   context else elsif
end    first     global if    insert last    left match
then   type      value  where wheretype
```

II.B-2 Description syntaxique et sémantique de PPAT

Nous allons donner dans ce paragraphe la définition syntaxique d'un programme PPAT, telle qu'elle peut-être vue dans la spécification *atc* de PPAT(annexe A), et nous détaillerons parallèlement les différentes notions mises en jeu.

II.B-2.a Structure générale du programme

La grammaire concrète donnant la structure générale d'un programme PPAT est:

```
<ppat prog> = <header>
                <decls>
                <rules>
              end ppat ";" ;
<header> = ppat <ID> "(" <ID> ")" ";" ;
```

Un programme PPAT est donc composé d'une en-tête, de déclarations et de règles. L'en-tête du programme comporte deux identificateurs:

- le premier est le *nom interne* du module PPAT, c'est-à-dire son nom vis-à-vis du système FNC–2.
- le second est le nom (interne) du module contenant la définition de la syntaxe abstraite attribuée à décompiler.

Les déclarations sont définies par:

```
<decls> = <decls> <decls sep> ;
<decls> = <decls> <decls cxt> ;
<decls> = <decls> <import clause> ;
<decls> = ;
```

C'est-à-dire qu'un programme PPAT comporte une liste de déclarations (éventuellement vide), dans lesquelles l'utilisateur peut définir des séparateurs, des contextes, des importations de module.

II.B-2.b Importation d'un module

L'importation d'un module se fait de la même façon qu'en OLGA, à savoir:

```
<import clause> = from <ID> import all ";" ;
```

Par cette clause, l'utilisateur rend accessible, à l'intérieur du module PPAT, tous les objets définis dans le module de nom interne correspondant à l'identificateur.

II.B-2.c Déclaration des séparateurs

Nous avons dit, au paragraphe II.A, que les boîtes composées étaient définies par un séparateur. Ces séparateurs sont semblables à des constantes et doivent être définis avant d'être utilisés. Leur définition correspond à:

```
<decls sep> = separator <decl sep +> ;
<decl sep +> = <decl sep +> <sep def> ;
<decl sep +> = <sep def> ;
<sep def> = "<" <SEP ID> ">" "=" <separator> ";" ;
<separator> = "<" <ID> <sep param *> ">" ;
<sep param *> = <sep param +> ;
<sep param *> = ;
<sep param +> = <sep param +> "," <sep param> ;
<sep param +> = <sep param> ;
<sep param> = <INT> ;
```

La déclaration d'un séparateur correspond à l'affectation d'une description de séparateur à un identificateur : <SEP ID>.

La description d'un séparateur est elle-même composée de deux parties:

1. le type du séparateur : h , v , hv ou hov

2. la liste des paramètres du séparateur (des entiers).

Nous allons décrire les quatre types de séparateurs ainsi que le rôle des paramètres.

II.B-2.c-1 Le séparateur horizontal <h dx>

Deux boîtes B1 et B2 composées par le séparateur <h dx>, sont disposées de telle sorte que la première ligne de B2 est alignée avec la dernière ligne de B1, avec un espacement horizontal de dx caractères (voir figure II.2). L'avantage de cette disposition est d'assurer que le texte produit est analysable, puisque le texte est lu de gauche à droite et de haut en bas, ce qui ne serait pas le cas si nous avions aligné le sommet des deux boîtes.

II.B-2.c-2 Le séparateur vertical <v di , dh>

Deux boîtes B1 et B2 composées par le séparateur <v di , dh>, sont disposées verticalement avec un espacement vertical de dh lignes, et une indentation de di caractères par rapport à la boîte englobante (voir figure II.3).

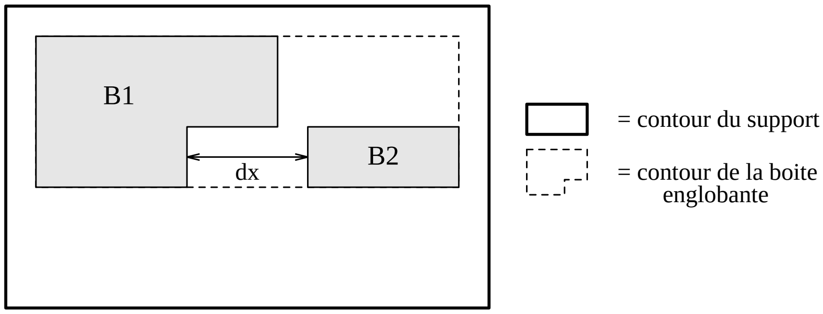

Figure II.2 : Composition de deux boîtes par le séparateur `<h dx>`

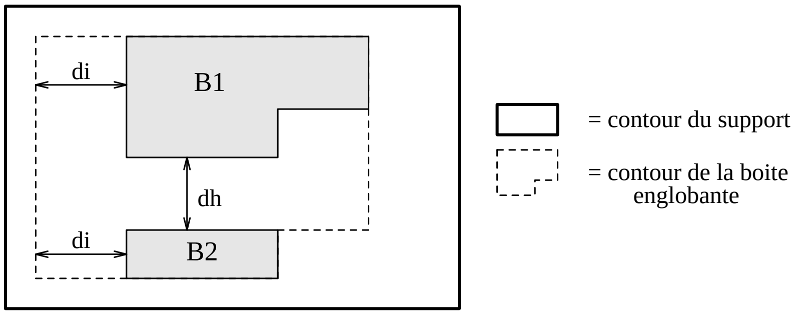

Figure II.3 : Composition de deux boîtes par le séparateur `<v di , dh>`

II.B-2.c-3 Le séparateur horizontal et vertical `<hv dx , di , dh>` Deux boîtes B1 et B2 composées par le séparateur `<hv dx , di , dh>`, sont disposées de telle sorte que:

- si B2 alignée avec la dernière ligne de B1, avec un espacement horizontal de `dx` caractères, tient dans le support d'affichage, alors elles sont disposées de cette façon.
- sinon, B2 est placée `dh` lignes en dessous de B1 avec une indentation de `di` caractères par rapport à la boîte englobante.

La figure II.4 indique le résultat de la composition de trois boîtes par le séparateur `<hv dx , di , dh>` de façon à illustrer les deux possibilités de dispositions.

II.B-2.c-4 Le séparateur horizontal ou vertical `<hov dx , di , dh>` Une liste de boîtes composées par le séparateur `<hov dx , di , dh>` sont disposées:

- de la même façon qu'avec le séparateur `<h dx>` si cette disposition tient dans le support d'affichage
- de la même façon qu'avec le séparateur `<v di , dh>` si la disposition horizontale ne tient pas dans le support d'affichage.

La différence avec le séparateur `hv` est que ici, la décision de mise en page est globale à la liste des boîtes, alors que dans le cas du séparateur `hv`, cette

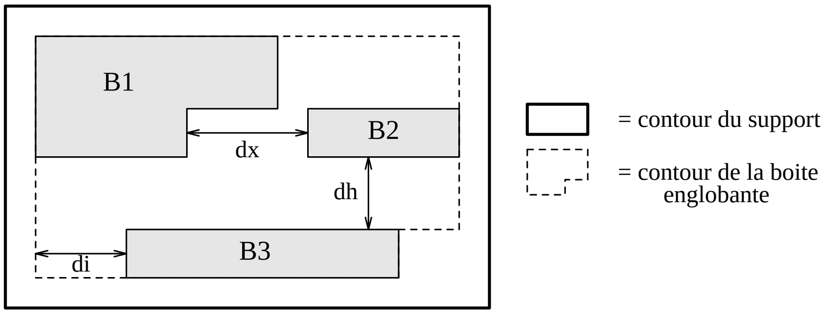

Figure II.4 : Composition de trois boîtes par le séparateur `<hv dx,di,dh>`

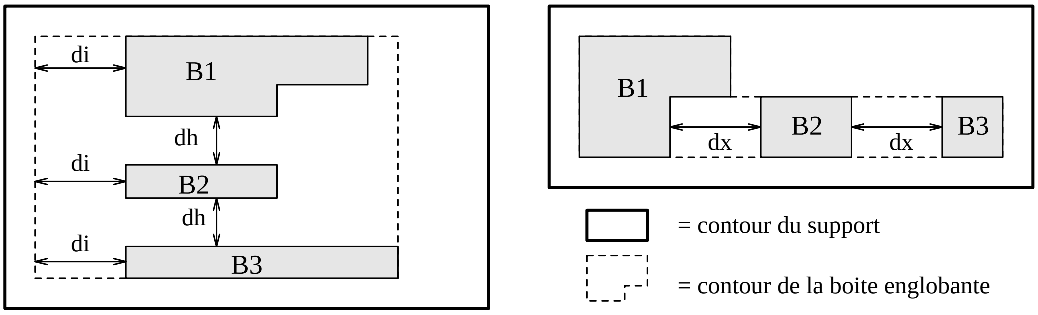

Figure II.5 : Les deux possibilités de composition de trois boîtes par le séparateur `<hov dx , di , dh>`

décision est prise pour chaque boîte. La figure II.5 montre les deux résultats possibles pour la composition de trois boîtes.

Les deux derniers types de séparateurs sont conditionnels, c'est-à-dire que le résultat produit dépend de la largeur de la page du support d'affichage, alors que les deux premiers sont *rigides* et peuvent provoquer un *débordement*.

II.B-2.d Déclaration des contextes

Nous avons mentionné, au paragraph II.A, la possibilité de transmettre de l'information relative à la décompilation par le moyen de *contextes*. Ces *contextes* sont considérés comme des attributs hérités au sens d'OLGA.

La déclaration des contextes est décrite par:

```
<decls cxt> = context <decl cxt +> ;
<decl cxt +> = <decl cxt +> <cxt def> ;
<decl cxt +> = <cxt def> ;
<cxt def> = <attrib name> "(" <phyla, *> ")" ":" <ID> ";" ;
<phyla, *> = <phyla, +> ;
<phyla, *> = ;
<phyla, +> = <phyla, +> "," <phylum> ;
<phyla, +> = <phylum> ;
```

Ces déclarations sont donc similaires aux déclarations d'attributs dans

OLGA, le mot clef `attribute` d'OLGA étant remplacé par le mote clef `context`. Un contexte est défini par la donnée de son nom, de la liste des phylums sur lesquels il est défini, et son type. Contrairement à OLGA, la catégorie du contexte n'est pas spécifiée, car tous les contextes sont par définition hérités.

les contextes peuvent être de type prédéfini ou complexe, dans ce dernier cas, le type correspondant doit être défini dans un module de déclaration importé comme nous l'avons vu au paragraphe II.B-2.b.

II.B-2.e Les règles de formattage sur les opérateurs

Tous les objets nécessaires à la définition d'un module PPAT étant déclarés, nous allons maintenant étudier la partie principale d'une spécification PPAT, à savoir les règles de formattage sur les opérateurs. Une telle règle est définie par:

1. la description de l'opérateur.

2. la description de la boîte associée à cet opérateur.

Une règle de formattage est définie par:

```
<rule op> = where <production> pprint
              <top box>
            end where ";" ;
<production> = <ID> "->" <phylum> "+" ;
<production> = <ID> "->" <phylum> "*" ;
<production> = <ID> "->" <phyla *> ;
<phyla *> = <phyla +> ;
<phyla *> = ;
<phyla +> = <phyla +> <phylum> ;
<phyla +> = <phylum> ;
<top box> = <box> ;
<top box> = ;
<box>  = <compound> ;
<box>  = <string box> ;
<box>  = <rec call> ;
<box>  = <attrib occ> ;
<box>  = <pplist> ;
<box>  = <conditional> ;
<box>  = <case match> ;
<box>  = <case tree> ;
```

La description de l'opérateur est, comme nous l'avons dit au paragraphe II.A, identique à la description des opérateurs dans OLGA.

Pour ce qui est du format d'édition, il s'agit de la description d'une boîte pouvant appartenir à l'un des formats suivants:

- boîte composée

- boîte chaîne de caractères
- appel récursif
- occurence d'attribut
- traitement des opérateurs liste
- boîte conditionnelle
- boîte contrôlée par la coïncidence d'un motif et d'une expression
- boîte contrôlée par la coïncidence d'un motif et d'un arbre

Nous allons maintenant décrire ces différents formats.

II.B-2.e-1 Boîte composée

La boîte composée représente, comme nous l'avons dit au paragraphe II.A, l'action d'un séparateur sur une liste de sous-boîtes. Ce format est décrit par:

```
<compound> = "[" <SEP ID> <box +> "]" ;
<box +> = <box +> <local sep> <box> ;
<box +> = <box> ;
<local sep> = "<" <SEP ID> ">" ;
<local sep> = ;
```

Chaque sous-boîte peut-être d'un format quelconque, parmi ceux cités plus haut. Afin de réduire le nombre de crochets, on peut utiliser des séparateurs locaux, ceux-ci correspondent uniquement à une simplification de l'écriture et on a l'équivalence suivante:

$$\begin{array}{rl} & [\texttt{sep}\ b_1\ \ldots\ b_i < \texttt{seploc} > b_{i+1} \ldots\ b_n] \\ \Leftrightarrow & [\texttt{sep}\ b_1\ \ldots\ [< \texttt{seploc} > b_i\ b_{i+1}]\ \ldots\ b_n] \end{array}$$

II.B-2.e-2 Boîte chaîne de caractères

Ce format de boîte décrit de façon évidente par:

```
<string box> = <STRING> ;
```

représente une boîte de hauteur unité et de largeur égale au nombre de caractères de la chaîne.

II.B-2.e-3 Appel récursif

L'appel récursif représente une référence à un phylum apparaissant dans la définition de l'opérateur. La boîte correspondante est donc la boîte provenant de la décompilation de l'arbre associé au phylum mentionné. Ce format est décrit par:

```
<rec call>  = <symbol> <context list> ;
<symbol> = <ID> ;
<symbol> = <ID> "." <qualifier> ;
<qualifier> = <INT> ;
<context list> = "(" <context *> ")"  ;
<context list> = ;
<context *> = <context +> ;
<context +> = <context +> "," <context>;
<context +> = <context> ;
<context> = <attrib name> ":=" <expr>;
```

De la même façon que dans OLGA, si un phylum est présent plusieurs fois dans la définiton d'un opérateur d'arité fixe, on distingue ces différentes occurences en le qualifiant par un entier représentant son numéro d'ordre.

De façon à permettre la décompilation d'un sous-arbre en fonction d'informations contenues dans le nœud courant, on peut[2] spécifier la valeur des contextes avec lesquels ce sous-arbre doit être décompilé. La valeur d'un contexte est décrite au moyen d'une expression PPAT, nous étudierons ces expressions au paragraphe II.B-2.g.

Il n'est pas obligatoire de faire figurer dans le format d'édition associé à un opérateur tous les phylums de sa définition, ceci permet d'ignorer certains sous-arbres[3].

II.B-2.e-4 Occurence d'attribut

Nous avons dit que la supériorité de PPAT sur PPML reposait sur le fait que PPAT décompile des arbres abstraits attribués. Ce format permet de visualiser la valeur des attributs de l'arbre de la façon suivante:

```
<attrib occ> = <attrib name> ;
<attrib occ> = <attrib name> "(" <symbol> ")" ;
<symbol> = <ID> ;
<symbol> = <ID> "." <qualifier> ;
<qualifier> = <INT> ;
<qualifier> = left ;
<qualifier> = right ;
<qualifier> = first ;
<qualifier> = last ;
```

Comme dans OLGA, les attributs pouvant apparaître dans un format d'édition associé à un opérateur, sont les attributs locaux à la production définissant cet opérateur, c'est-à-dire soit les attributs attachés au phylum auquel l'opérateur appartient, soit les attributs attachés au phylum fils de cet opérateur.

[2] En fait, il est obligatoire de donner une valeur à tous les contextes attachés au phylum représentant ce sous-arbre; il faut en effet se souvenir que les contextes sont en fait des attributs hérités.

[3] Nous verrons, au chapitre IV, pourquoi le fait de ne pas mentionner un phylum entraîne que le sous-arbre correspondant n'est pas décompilé.

Une occurence d'attribut est représentée, comme dans OLGA, par le nom de l'attribut suivi par le nom du phylum concerné entre parenthèses. Ci le symbol est omis, l'attribut est considéré être un attribut du phylum auquel l'opérateur appartient.

Il est possible de mentionner tous les attributs d'un phylum, que ce soit ceux de l'arbre d'entrée, ou même les contextes définis sur ce phylum.

Un attribut est décompilé suivant la règle associée à son type.

II.B-2.e-5 Traitement des opérateurs liste

Nous avons vu, au paragrphe I.B du chapitre I, que les opérateurs d'une syntaxe abstraite pouvaient être d'arité fixe ou bien de type liste. De même que dans OLGA, un traitement particulier existe pour les opérateurs liste. Il est décrit par:

```
<pplist> = pplist  <direction>
             global <SEP ID>
             <insert>
             subbox  <box>
           end  pplist ;
<direction> = left ;
<direction> = right ;
<insert> = insert <STRING> ;
<insert> = ;
```

La boîte correspondant à un tel format est en fait une boîte composée, dont le séparateur est déclaré après le mot clef `global`, et dont la liste de sous-boîte est construite de la façon suivante: à chaque élément de la liste traitée, est associée une boîte définie par:

- si l'élément courant n'est pas le dernier traité, il s'agit de la boîte composée dont le séparateur est `<h 0>`, la première sous-boîte est la boîte associée au format donné après le mot clef `subbox`, et la seconde la boîte chaîne de caractères associée à la chaîne suivant le mot clef `insert` (éventuellement la chaîne vide si le mot clef `insert` est omis).
- si l'élément courant est le dernier traité, il s'agit de la boîte associée au format donné après le mot clef `subbox`.

L'ordre dans lequel la liste traitée est parcourue, est donné par l'un des deux mot clef `left` ou `right`.

Ainsi si on définit le format suivant:

```
where op -> PH * pprint
        pplist left
          global <h1>
          insert ","
          subbox PH
        end pplist
end where ;
```

Si l'on pose:

$$\texttt{sep} - \texttt{nul} =< \texttt{h0} >;$$

alors, lors de la décompilation de l'arbre, la boîte associée à un nœud d'arité n dont la racine est `op` est:

$$[< \texttt{h1} > \texttt{PH.1} < \texttt{sep} - \texttt{nul} > ","\texttt{PH.2} < \texttt{sep} - \texttt{nul} > "," \ldots \texttt{PH.n}]$$

Si on avait remplacé `left` par `right` dans le format, le résultat aurait été:

$$[< \texttt{h1} > \texttt{PH.n} < \texttt{sep} - \texttt{nul} > ","\texttt{PH.n} - 1 < \texttt{sep} - \texttt{nul} > "," \ldots \texttt{PH.1}]$$

II.B-2.e-6 Boîte conditionnelle

Jusquà présent nous n'avons présenté que des formats d'édition *rigides*, il est nécessaire d'introduire des structures de contrôle de façon à pouvoir contrôler la façon dont un arbre est décompilé. La boîte conditionnelle est la forme la plus simple de contrôle de décompilation, elle est définie par:

```
<conditional> = if  <expr>  then
                  <box>
                <elsif part>
                else
                  <box>
                end  if ;
<elsif part> = <elsif part>
               elsif  <expr>  then
                 <box> ;
<elsif part> = ;
```

Cette structure en bien entendue très proche de celle des expressions conditionnelles d'OLGA. Les expressions de conditions sont des expressions PPAT booléenes. Elles peuvent être construite en utilisant tous les éléments visibles à l'intérieur de la règle, c'est-à-dire en particulier les occurences d'attributs locales à cette production, les contextes et des constantes.

II.B-2.e-7 Boîte contrôlée par la coïncidence d'un motif et d'une expression

La deuxième forme de contrôle est celle qui correspond au *pattern matching* d'OLGA sur des expressions, (tel que nous l'avons décrit au paragraphe I.C-2.b du chapitre I). La définition de cette structure de contrôle est donnée par:

```
<case match> = case <expr> is
                <match clauses>
                <match default>
               end  case ;
<match clauses> = <match clauses> <match clause> ;
<match clauses> = <match clause> ;
<match clause> = <case clauses>  ":"
```

```
                    <box>  ";" ;
<case clauses> = <case clauses> ","  <case clause> ;
<case clauses> = <case clause> ;
<match default> = other  ":"
                    <box>  ";" ;
<case clause> = <root pattern> ;
<root pattern> = <pattern cons> ;
<root pattern> = <ID> ;
<root pattern> = <constant manifeste> ;
<root pattern> = const <expr> ;
<pattern cons> = <ID>  "(" <patterns *> ")" ;
<patterns *> = <patterns +> ;
<patterns *> = ;
<patterns +> = <patterns +> ","  <pattern> ;
<patterns +> = <pattern> ;
<constant manifeste> = <INT> ;
<constant manifeste> = %REAL ;
<constant manifeste> = %CHAR ;
<constant manifeste> = %STRING ;
<pattern> = <root pattern> ;
```

Comme dans OLGA, l'utilisation de motifs utilisant des variables permet de lier ces variables à la partie coïncidente de l'expression.

II.B-2.e-8 Boîte contrôlée par la coïncidence d'un motif et d'un arbre

La dernière structure de contrôle, certainement la plus puissante, est celle qui permet de faire du *pattern matching* sur des parties de l'arbre d'entrée. La définition de cette structure est très proche de la précédente:

```
<case tree> = case  tree <symbol>  match
      <match clauses>
      <match default>
  end  case ;
```

Cependant, comme nous l'avons déjà fait remarquer, si cette structure est traitée par PPAT, elle ne peut à l'heure actuelle être utilisée, car elle n'est pas encore implantée dans OLGA, ce qui, comme nous le verrons dans le chapitre IV, empêche sa compilation.

II.B-2.f Les règles de formattage sur les types

Nous avons mentionné, au paragraphe II.B-2.e-4, que la décompilation d'un attribut se fait en utilisant le format associé à son type. La définition d'une règle de formattage pour un type est la suivante:

```
<rule type *> = <rule type *> <rule type> ;
<rule type *> = ;
<rule type> = wheretype <ID> pprint
```

```
               <top box type>
             end where ";" ;
<top box type> = <box type> ;
<top box type> = ;
<box type> = <string boxt> ;
<box type> = <compound boxt> ;
<box type> = <fieldref boxt> ;
<box type> = <pplist boxt> ;
```

Ces règles sont très proches, tant au niveau de la définition qu'au niveau du fonctionnement, des règles de formattage des opérateurs. Comme pour les opérateurs, il existe plusieurs formats:

- boîte chaîne de caractères
- boîte composée
- accès au champ d'un type enregistrement
- traitement des types liste

Les deux premiers formats sont similaires à ceux correspondant pour les opérateurs, nous allons donc décrire les deux derniers.

II.B-2.f-1 Accès au champ d'un type enregistrement

Lors de la décompilation d'un attribut dont le type est un enregistrement, il est possible d'accéder au champs de ce type en indiquant le nom du champ. Ceci est à rapprocher du format de décompilation récursive des opérateurs, bien qu'il n'y ait pas d'équivalent pour les types de la notion de contextes.

II.B-2.f-2 Traitement des types liste

Le principe de traitement des types liste est à rapprocher de celui des opérateurs liste:

```
pplist boxt> = pplist <ID> <direction>
global <SEP ID>
<insert>
subbox <box type>
  end pplist ;
```

Nous avons fait apparaître dans le format de traitement des listes, un identificateur, permettant de faire référence, dans le format de la sous-boîte, à l'élément courant de la liste. Cet identificateur joue donc le rôle du phylum fils d'un opérateur liste.

II.B-2.g Les expressions de PPAT

Nous avons voulu donner à PPAT une grande puissance d'expression, tout en concervant aux spécifications de décompilation une relative simplicité. Pour celà, nous avons permis, en plus des expressions arithmétiques classiques, les appels de fonctions.

Ces fonctions sont définies par l'utilisateur dans des modules externes, donc n'alourdissent pas la spécification de décompilation, et permettent de bénéficier de toute la puissance d'expression de OLGA.

Conclusion

Nous avons dans ce chapitre, présenté les principes du décompilateur d'arbres attribués abstraits PPAT, et nous avons défini le langage permettant de décrire un module de décompilation.

L'objet des chapitres suivants, est de décrire la façon dont nous avons implanté ce décompilateur.

Chapitre III

Généralités sur l'implantation de PPAT

Introduction

Rappelons que les grammaires attribuées ont été crées afin de définir un formalisme permettant de spécifier des transformations sur des arbres décorés. Par ailleurs un analyseur lexico-syntaxique est un outil permettant de transformer un texte en un arbre ayant un format prédéfini. De plus, il est possible d'écrire un programme parcourant un arbre décoré afin d'en restituer une représentation sous la forme d'un texte. Ainsi les grammaires attribuées utilisées avec les deux outils précédents, sont un moyen particulièrement efficace pour créer une application de *traduction*, c'est-à-dire prenant un texte en entrée et fournissant un autre texte en sortie.

Dans ce chapitre, nous allons voir comment l'application PPAT peut être implantée en utilisant intensivement les grammaires attribuées et par conséquent le système FNC–2.

Nous rappelons que PPAT est une application permettant d'obtenir des représentations concrètes d'arbres attribués abstraits. Nous avons expliqué, au chapitre II, que pour réaliser cette opération de décompilation, l'utilisateur devait écrire une spécification, correspondant à la syntaxe abstraite de l'arbre à décompiler, et nous avons vu que cette spécification mettait en jeu une structure de boîtes.

III.A La *phase d'exécution* de PPAT

Une partie du système PPAT devra donc prendre en entrée un arbre abstrait, correspondant à une syntaxe abstraite définie par l'utilisateur, et fournir en sortie une représentation concrète de cet arbre. Nous appellerons cette partie du système PPAT la *phase d'exécution*.

Supposons que nous disposons d'un outil prenant en entrée un arbre abstrait et fournissant en sortie un arbre de boîtes, représentant l'arbre initial. Si nous appelons `O` cet outil, nous sommes en mesure de décrire la *phase d'exécution* de PPAT sous la forme d'une application au sens de FNC–

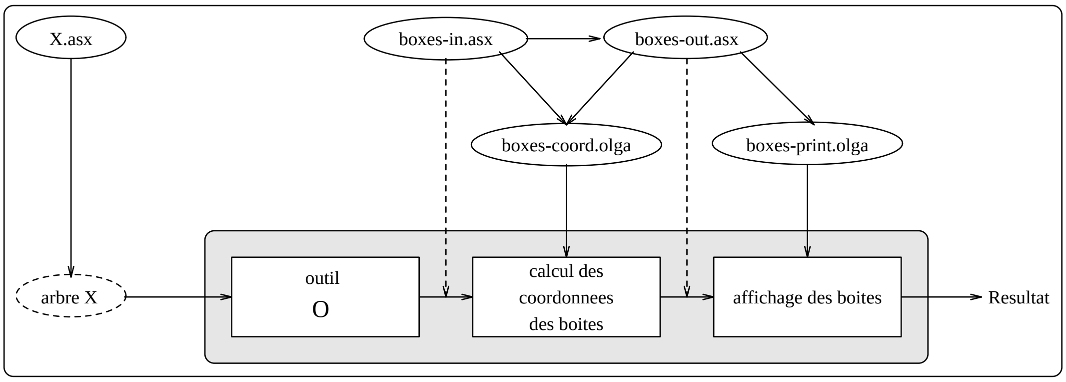

Figure III.1 : Schéma de la *phase d'exécution* de PPAT

2 (voir la figure I.5 du chapitre I), le schéma correspondant à cette phase est décrit par la figure III.1.

Dans ce schéma, les ellipses entourent les spécifications et les rectangles les outils issus de la compilation de ces spécifications. Les spécifications sont:

- `boxes-in.asx`, la syntaxe abstraite attribuée décrivant un arbre de boîtes
- `boxes-out.asx`, la syntaxe abstraite attribuée décrivant un arbre de boîtes décoré avec les coordonnées et la taille des boîtes.
- `boxes-coord.olga` une grammaire attribuée procédurale réalisant le calcul des coordonnées des boîtes.
- `boxes-print.olga` une grammaire attribuée imprimant le contenu des boîtes aux coordonnées précédemment calculées.
- `X.asx` le format de l'arbre à décompiler. Cette partie est écrite par l'utilisateur.

III.B La *phase de compilation* de PPAT

Pour décrire la *phase d'exécution* de PPAT nous avons supposé l'existence de l'outil `O`. Cet outil dépend bien évidemment de la spécification de décompilation, et le rôle de l'autre partie de PPAT, dite *phase de compilation*, est justement de créer cet outil.

Cet outil réalise une transformation sur des arbres, il peut donc être représenté par une grammaire attribuée. La *phase de compilation* de PPAT est donc une phase de traduction, prenant en entrée une spécification de décompilation et fournissant en sortie une spécification de grammaire attribuée. La *phase de compilation* peut-être considérée comme une application au sens de FNC–2, dont le schéma est donné à la figure III.2.

Les spécifications représentées sur ce schéma sont:

- `ppat-base.asx` la syntaxe abstraite non attribuée du langage PPAT.

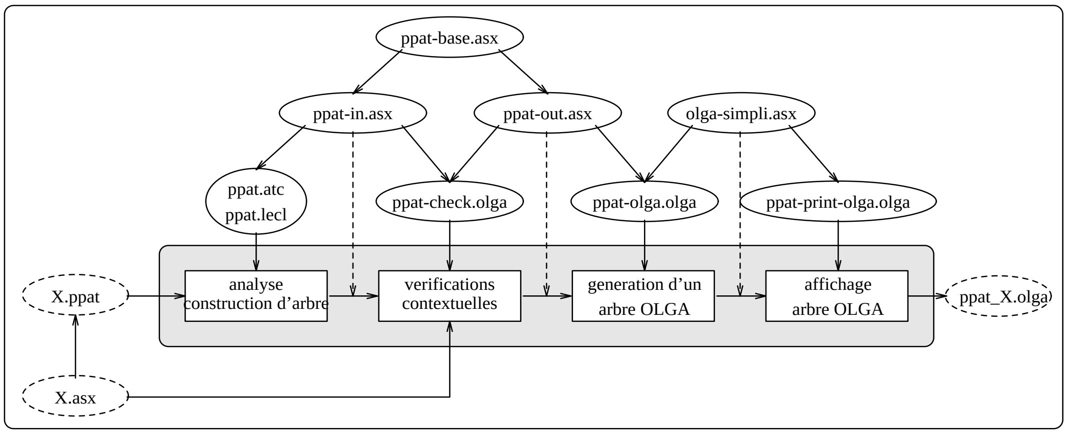

Figure III.2 : Schéma de la *phase de compilation* de PPAT

- `ppat-in.asx` la syntaxe abstraite du langage PPAT, augmentée des attributs issus de l'analyse syntaxique.
- `ppat-out.asx` la syntaxe abstraite du langage PPAT, augmentée des renseignements obtenus au cours de la phase de vérifications contextuelles.
- `ppat.atc` et `ppat.lecl`, les spécifications nécessaires à l'analyse syntaxique et à la construction de l'arbre ppat.
- `olga-simpli.asx` la syntaxe abstraite d'un sous-ensemble du langage OLGA.
- `ppat-check.olga` une grammaire attribuée procédurale effectuant les vérifications contextuelles de la spécification PPAT et décorant l'arbre avec les renseignements obtenus en utilisant les tables des symboles.
- `ppat-olga.olga` une grammaire attribuée fonctionnelle traduisant un arbre PPAT en un arbre OLGA.
- `ppat-print-olga.olga` une grammaire attribuée affichant le texte OLGA correspondant à son arbre d'entrée.

Nous avons représenté en plus les spécifications:

- `X.ppat` qui est la spécification de décompilation et par conséquent l'entrée de la *phase de compilation*,
- `X.asx` qui est la syntaxe abstraite correspondant aux arbres à décompiler.

III.C Le système global

Nous avons dit que la *phase de compilation* de PPAT doit permettre de créer l'outil `O` de la *phase d'exécution*. Nous pouvons donc réunir les deux phases sur un seul schéma. Ce schéma est représenté sur la figure III.3.

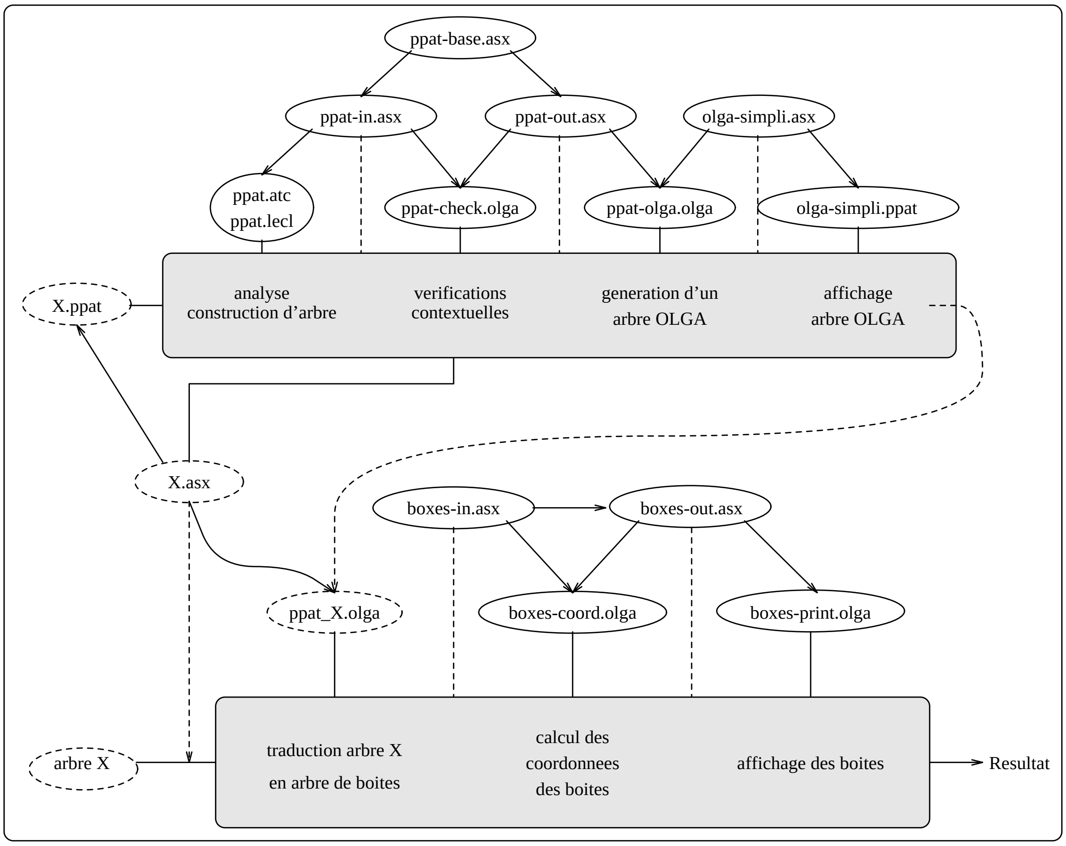

Figure III.3 : Schéma du système PPAT

Sur ce schéma, nous avons non seulement indiqué les liens entre la *phase de compilation* et la *phase d'exécution*, mais nous avons aussi décrit le système dans son état final, c'est-à-dire que nous avons remplacé la grammaire attribuée d'impression d'un arbre OLGA par le module de décompilation d'un arbre OLGA : `olga-simpli.ppat`. Le système est donc *auto-amorcé*, ce qui est un phénomène courant dans le cadre de la compilation.

Sur ce schéma, les spécifications entourées de pointillés sont écrites par l'utilisateur du système PPAT.

Conclusion

Dans ce chapitre, nous avons expliqué comment le système PPAT peut-être implanté au moyen de grammaires attribuées, en différentiant une phase de *compilation* et une phase de *exécution*.

Dans les deux chapitres suivants, nous expliquons comment nous avons réalisé les spécifications mentionnées sur la figure III.3.

Chapitre IV

Implantation de la phase de compilation

Introduction

Dans ce chapitre, nous allons voir en détails le contenu des spécifications de la *phase de compilation* telle qu'elle a été définie au chapitre III.

Les spécifications relatives à l'analyse syntaxique et à la construction de l'arbre associé à une spécification PPAT ayant déjà été étudiées au chapitre II, nous ne revenons pas dessus, leur contenu est fourni dans l'annexe A.

La syntaxe abstraite correspondant à un arbre PPAT et décrite par la spécification `ppat-base.asx` a été conçu de façon à être la plus proche possible d'une spécification de grammaire attribuée dans le langage OLGA, au détriment parfois de la simplicité. Nous verrons les raisons de ce choix dans les paragraphes suivants.

Nous étudions maintenant les trois grammaires de la *phase de compilation*.

IV.A Implantation des vérifications sémantiques

La première étape de la compilation d'une spécification PPAT consiste à vérifier la cohérence de cette spécification.

Le principe général de cette vérification est d'attacher à tous les phylums de la syntaxe abstraite de PPAT, un attribut synthétisé booléen, appelé `$correct`. La cohérence de la spécification est alors vérifiée sur chaque nœud, un arbre étant correct si tous ses sous-arbres le sont et si les vérifications de la cohérence au niveau de sa racine sont positives. La vérification de la cohérence au niveau de chaque sous-arbre, nécessite de véhiculer des informations le long de l'arbre (typiquement des tables de symboles) au moyen d'attributs.

Nous avons dit que nous avons choisi la structure de l'arbre PPAT aussi proche que possible de celle d'un arbre OLGA. Ceci nous permet en effet de rendre un certain nombre de vérifications de l'arbre PPAT, semblables aux vérifications d'un arbre OLGA, et d'utiliser les modules du système FNC–2 traitant ces vérifications. Les vérifications effectuées de façon similaires aux vérifications d'OLGA sont:

- l'existence des modules référencés, il s'agit du module *asx* et des modules intervenant dans les clauses d'importation.
- la cohérence de la définition des contextes: il faut vérifier l'unicité de leur définition, l'existence des phyla auxquels ils sont accrochés, l'existence de leur type.
- la cohérence de l'utilisation des opérateurs, dans les règles de formattage, vis-à-vis de leur définition dans le module *asx*.
- la cohérence des occurences d'attributs dans les règles de formattage des opérateurs.
- l'existence des types utilisés dans les règles de formattage.
- de façon générale, tout ce qui concerne le typage des expressions (affectation d'une valeur à un contexte, expression booléennes dans les conditionnelles, cohérence des motifs dans les `case ... match` et les `case tree`)

Un certain nombre de vérifications particulières à PPAT sont aussi nécessaires, il faut en effet:

- vérifier la cohérence des définitions de séparateurs: il faut s'assurer de l'unicité, de l'existence du type, de la cohérence entre le type et les paramètres du séparateur.
- vérifier dans les règles de formattage que les séparateurs ont bien été définis.
- vérifier que les `pplist` sont bien utilisées pour des opérateurs liste ou des types liste
- vérifier que l'accès à un champ se fait uniquement pour un type enregistrement.
- vérifier que l'accès à un symbole dans un appel récursif se fait uniquement si ce symbole est un fils de l'opérateur.

Ces vérifications particulières à PPAT, nous ont conduit à élaborer un module contenant des fonctions , des types et des constantes utiles pour ces vérifications. Ce module est représenté par les spécifications:

- `dec_ppat-types.olga`
- `def_ppat-types.olga`

dont le contenu figure dans l'annexe A.

Par ailleurs, cette phase de vérifications réalisant un parcours complet de l'arbre PPAT, elle est utilisée pour réunir des informations qui seront utiles par la suite. Il s'agit:

- du nom interne du module PPAT,
- de la table des types,
- de la liste des séparateurs et de leurs caractéristiques,
- de la liste des types devant être affichés (nous verrons au paragraphe suivant une explication de cette liste),
- du type de chaque attribut devant être décompilé,
- des informations relatives au type de chaque boîte apparaissant dans une règle de formattage de type,
- d'un booléen permettant de décider si un opérateur appartient au phylum racine de l'arbre.

Ces informations sont transmises à la seconde phase de traduction de l'arbre par des attributs exportés.

Ces vérifications peuvent bien entendu aboutir à la détection d'erreurs, auquel cas des messages d'erreurs sont produits par effets de bords, la fonction permettant cet effet de bord est la même que celle utilisée dans la phase de vérification du système FNC–2, nous avons dû rajouter des messages particuliers à PPAT dans le module définissant cette fonction.

Si la spécification n'est pas correcte, cela a pour effet de donner une valeur fausse à l'attribut `$correct` de la racine de l'arbre PPAT, or nous avons dit, au chapitre I, que les attributs de la racine d'un arbre étaient accessibles dans l'interface en langage C, ceci permet de produire un message et de ne pas passer aux phases suivantes.

La grammaire attribuée `ag_ppat-check.olga`, réalisant les vérifications sémantiques, figure dans l'annexe A.

IV.B Implantation de la traduction en OLGA

Après avoir vérifé la spécification PPAT, nous sommes en mesure de traduire cette dernière en un arbre OLGA. C'est dans cette phase que la similitude de la syntaxe abstraite de PPAT et de celle d'OLGA est la plus avantageuse.

Nous allons d'abord expliquer le principe général de la traduction d'un arbre PPAT en un arbre OLGA avant d'approfondir quelques points de cette traduction.

IV.B-1 Principe général de la traduction

L'idée directrice est la suivante: nous avons deux syntaxes abstraites, pour transformer l'une en l'autre, nous faisons correspondre ces deux syntaxes en dégageant le sens sémantique de chaque sous arbre de l'une et l'autre et en rapprochant les sous arbres dont les sens sémantiques sont voisins.

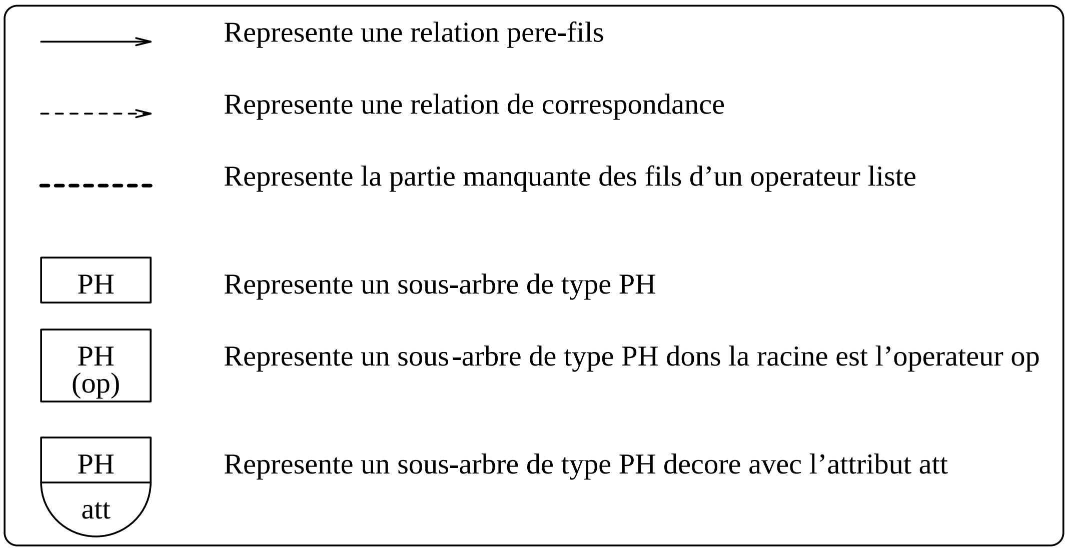

Figure IV.1 : Légende des schémas de rapprochements et de traductions

Ainsi au niveau le plus élevé, il est naturel de faire correspondre la racine des deux arbres. La figure IV.1 donne la légende des schémas qui vont suivre, et la figure IV.2 indique schématiquement ces rapprochements pour les grandes structures des deux arbres.

Si le rapprochement est facile pour les grandes structures, il devient plus complexe à représenter dès qu'un phylum possède plusieurs opérateurs ou qu'une même structure de PPAT doit être rapprochée de plusieurs structures d'OLGA. Nous avons cependant montré sur la figure IV.3 comment la structure contenant les règles de formattage doit être rapprochée des structures d'OLGA.

Le principe fondamental de traduction de la spécification PPAT en une spécification OLGA est alors le suivant:

Une règle de formattage d'un opérateur est transformé en une production, des déclarations, des règles sémantiques.
Les régles de formattage des types sont transformées en un ensemble de définitions de fonctions récursives entre elles.
La grammaire produite devant être une grammaire attribuée fonctionnelle donnant en sortie un arbre correspondant à la syntaxe abstraite `boxes-in.asx`*, Les règles sémantiques produite doivent conduire à la création d'un arbre de boîtes, donc calculer pour chaque nœud de l'arbre à décompiler un attribut représentant une boîte (sauf pour le phylum racine dont l'attribut doit représenter l'arbre tout entier).*

A ce principe fondamental, viennent s'ajouter la traduction des autres structures de l'arbre PPAT, en particulier les déclarations qui sont naturellement transformées en déclarations OLGA.

Après avoir expliqué la méthode avec laquelle nous traduirons l'arbre PPAT en arbre OLGA, nous donnerons de plus amples détails sur la traduc-

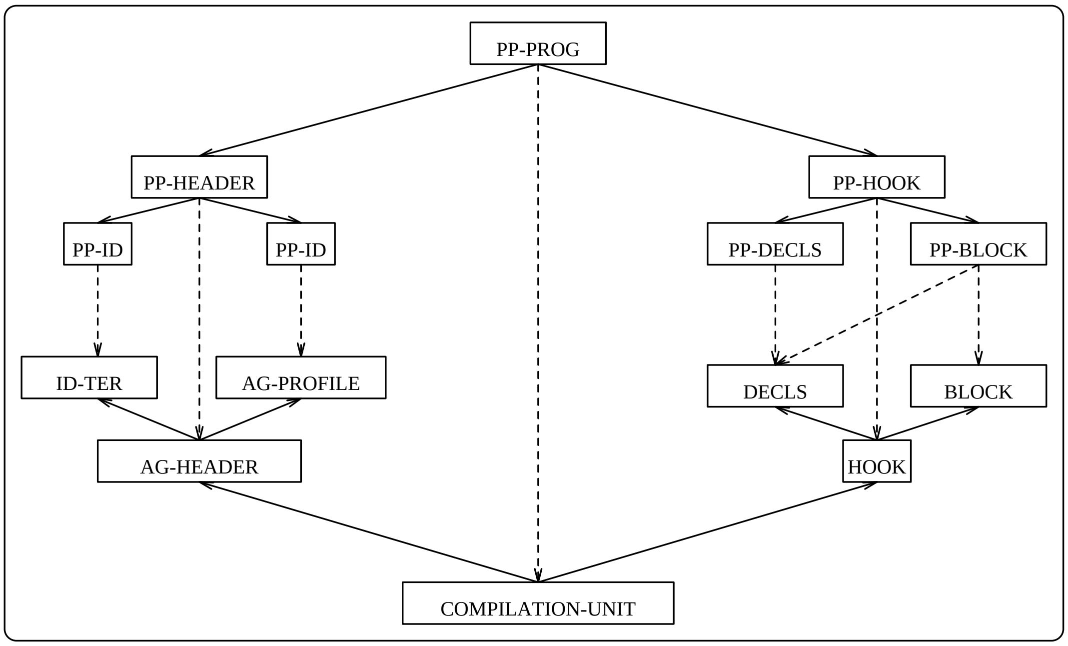

Figure IV.2 : Rapprochement des grandes structures de PPAT (en haut) et d'OLGA (en bas)

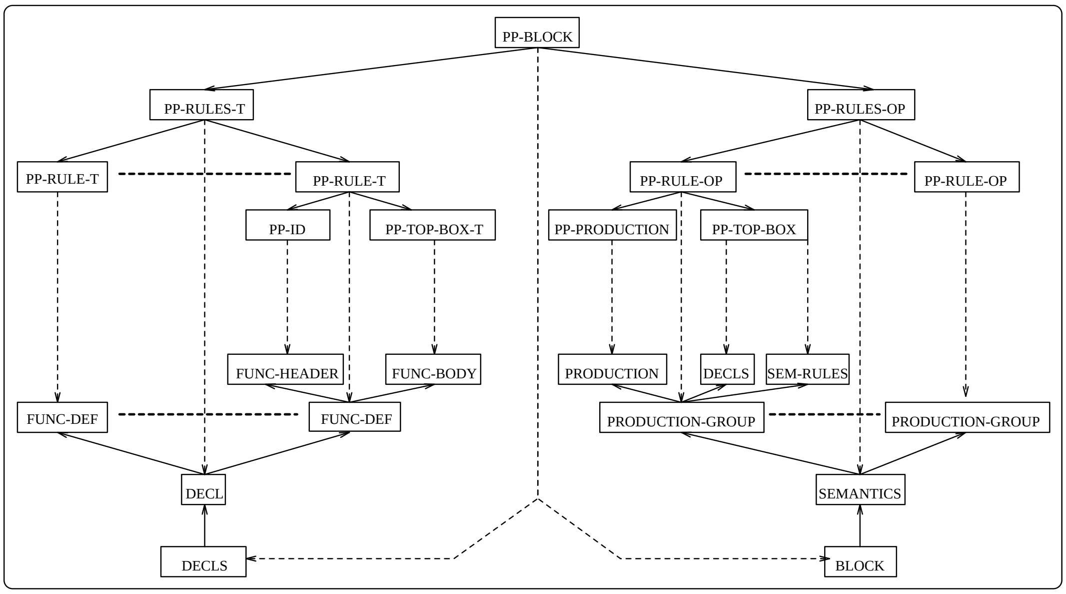

Figure IV.3 : Rapprochement des règles de formattage de PPAT (en haut) et des structures d'OLGA (en bas) tions des déclarations, des règles de formattage des opérateurs et des règles de formattage des types.

IV.B-2 Méthode générale de la traduction

La grammaire `ppat-olga`, réalisant la traduction, est une grammaire fonctionnelle donc, d'après le paragraphe I.D-2.f du chapitre I, la traduction se fait par construction de l'arbre au moyen des fonctions associées aux opérateurs.

Le paragraphe précédent a montré que l'on peut associer à un sous-arbre PPAT un, ou plusieurs, sous-arbres OLGA.

La méthode de traduction est alors la suivante:

> *Soit* `PP-PH` *un phylum de PPAT représentant un sous-arbre que l'on peut* rapprocher *(au sens du paragraphe précédent) des sous-arbres OLGA correspondant aux phyla* `PH1`, `PH2 ...PHn`.
> *On accroche à* `PP-PH` *les attributs synthétisés* `$Ph1`, `...$Phn` *qui sont respectivement de type* `PH1`, `...PHn`.
> *Ces attributs sont calculés grâce aux fonctions associées aux opérateurs de la syntaxe abstraite d'OLGA.*

Pour illustrer cette méthode, donnons l'exemple de la racine de l'arbre PPAT, qui comme indiqué sur la figure IV.2 est à rapprocher de la racine de l'arbre OLGA. La déclaration de l'attribut (représentant l'arbre OLGA traduction de l'arbre PPAT) est:

```
synthesized  $Compilation-Unit (PP-PROG) : COMPILATION-UNIT ;
```

et la règle sémantique de calcul de cet attribut est:

```
where pp-prog -> PP-HEADER PP-HOOK use
  $Compilation-Unit := simple-AG(
                         $Ag-Header (PP-HEADER) ,
                         $Hook (PP-HOOK) ) ;
end where ;
```

Cette méthode est la méthode générale de traduction, cependant, comme nous allons le voir par la suite, il peut s'avérer nécessaire d'introduire des attributs supplémentaires pour effectuer des calculs plus complexes.

IV.B-3 Traduction des déclarations

Nous avons dit que, de façon naturelle, les déclarations de PPAT sont traduites par des déclarations de OLGA.

De façon plus précise, la traduction des trois types de déclarations de PPAT est schématisée sur la figure IV.4. La traduction d'une déclaration de séparateur en une déclaration de constante permettra dans la *phase d'exécution* de passer le séparateur comme un paramètre à une fonction. Ces constantes sont de type `sep-info` défini par:

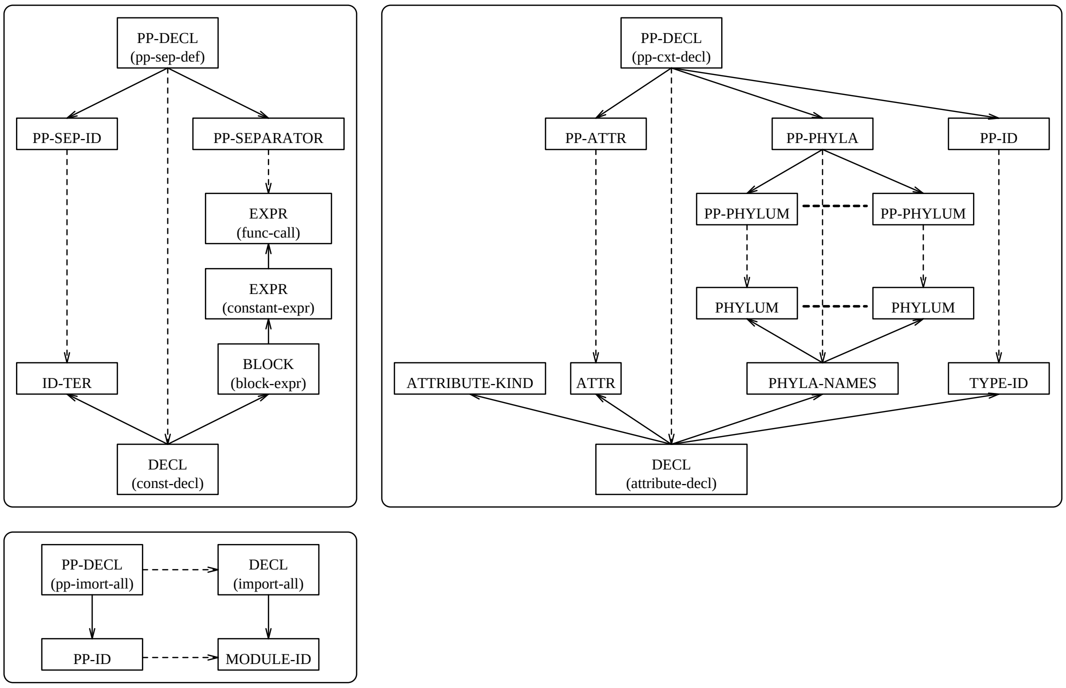

Figure IV.4 : Traductions des déclarations de PPAT

```
sep-param = list of int ;

sep-kind = ( h , v , hv , hov , error ) ;

sep-info = record
                kind :        sep-kind ;
                parametres :  sep-param ;
           end record ;
```

IV.B-4 Traduction des règles de formattage des opérateurs

Nous avons dit dans le paragraphe IV.B-1 qu'à une règle de formattage associée à une opérateur correspondait:

- une production,
- des déclarations,
- des règles sémantiques,

Nous ne revenons pas sur la traduction de la production, car elle est naturelle. En revanche, nous allons détailler les arbres générés par la traduction de la règle de formattage proprement dite.

La comparaison d'une spécification PPAT et d'une spécification OLGA, conduit à penser qu'une règle de formattage associée à un opérateur doit être traduite par une règle sémantique. En fait ce serait le cas si les formats de

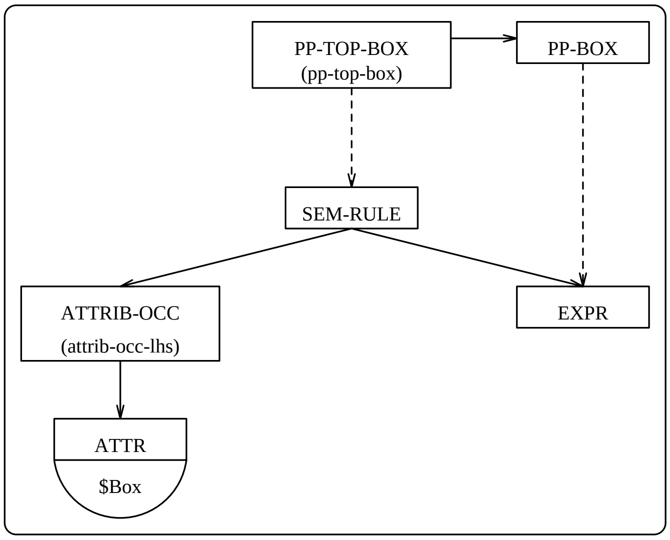

Figure IV.5 : Principe de traduction simplifiée des règles de formattage des opérateurs

décompilation récursive et les formats de traitement des listes n'existaient pas. Dans le cadre de cette hypothèse, et d'après le principe énnoncé au paragraphe IV.B-1, il suffirait de produire pour chaque règle de formattage une règle sémantique unique du type:

`$Box` := *expression correspondant au format*

où l'attribut `$Box` est un attribut du type `BOX` dont on peut voir la définition dans la spécification `boxes-in.asx` (annexe B). Toujours dans le cadre de cette hypothèse, nous aurions alors le principe de traduction représenté sur la figure IV.5

Dans le cadre de cette hypothèse, on traduit une spécification de boîte (`PP-BOX`) par une expression OLGA de type `BOX`.

Revenons au cas réel. Le principe de traduction reste valable, si bien qu'à chaque spécification de boîte doit correspondre une expression OLGA de type `BOX`, calculée, conformément au paragraphe IV.B-2 dans l'attribut `$Expr` de type `EXPR` accroché au phylum `PP-BOX`.

Nous allons voir comment la présence des deux formats écartés dans le cadre de l'hypothèse simplificatrice modifie la traduction, puis nous expliquerons la traduction des différents formats.

IV.B-4.a Problèmes liés aux formats listes

L'idée naturelle pour traduire une spécification de boîte liste, en une expression OLGA de type `BOX` est de traduire le texte PPAT:

```
pplist      direction
            global séparateur
            subbox spécification de la sous-boîte
end pplist
```

par le texte OLGA:

`box(`
    `map` direction `boxes-post`
    `value boxes ()`
    `other` *expression correspondant à la sous-boîte*
    `end pplist)`
    `with $separator :=` *constante associée à* séparateur `end with`

Mais cette traduction n'est pas possible à cause d'une restriction du langage OLGA, qui fait que les expressions `map` ne peuvent pas être des sous expressions.

Cela nous conduit donc à traduire une spécification de boîte associée à un opérateur liste, par deux sous arbres:

- l'un contenant la déclaration d'une valeur, locale à la production, dans laquelle le `map` est effectué,
- l'autre contenant une expression OLGA de type `BOX` utilisant la valeur locale.

Nous devons aussi tenir le compte des décompilations de liste utilisées au sein d'une même production, de façon à pouvoir générer des noms de valeur uniques; ce compte est tenu par l'attribut mixte `$num-pplist`.

IV.B-4.b Problèmes liés aux décompilations récursives

Nous avons dit que le format de décompilation récursive permettait de mentionner, dans la règle de formattage associée à un opérateur, la boîte associée à un de ses fils, tout en transmettant à ce dernier des contextes, que nous avons déjà identifiés à des attributs hérités d'OLGA (paragraphe IV.B-3).

Nous devons donc, à l'intérieur de la production associée à un opérateur créer des règles sémantiques permettant de calculer la valeur de tous les attributs hérités associés à tous les contextes mentionnés dans la règle de formattage.

Une première approche, consiste à décrire la traduction associée à une boîte de décompilation récursive de la façon représentée sur la figure IV.6. Si cette première approche n'est pas exacte, il n'en reste pas moins qu'elle fournie le principe de la traduction, c'est-à-dire:

- que l'expression correspondante telle qu'elle est calculée dans l'attribut `$Expr` correspond à l'occurence de l'attribut `$Box` du fils sur lequel porte la décompilation récursive,
- qu'il faut faire correspondre à chaque définition de contexte une règle sémantique permettant son calcul.

Nous allons voir que l'utilisation conjointe de décompilations récursives et de décompilations conditionnelles impose des modifications sur la traduction.

Soit en effet l'extrait de spécification PPAT suivante:

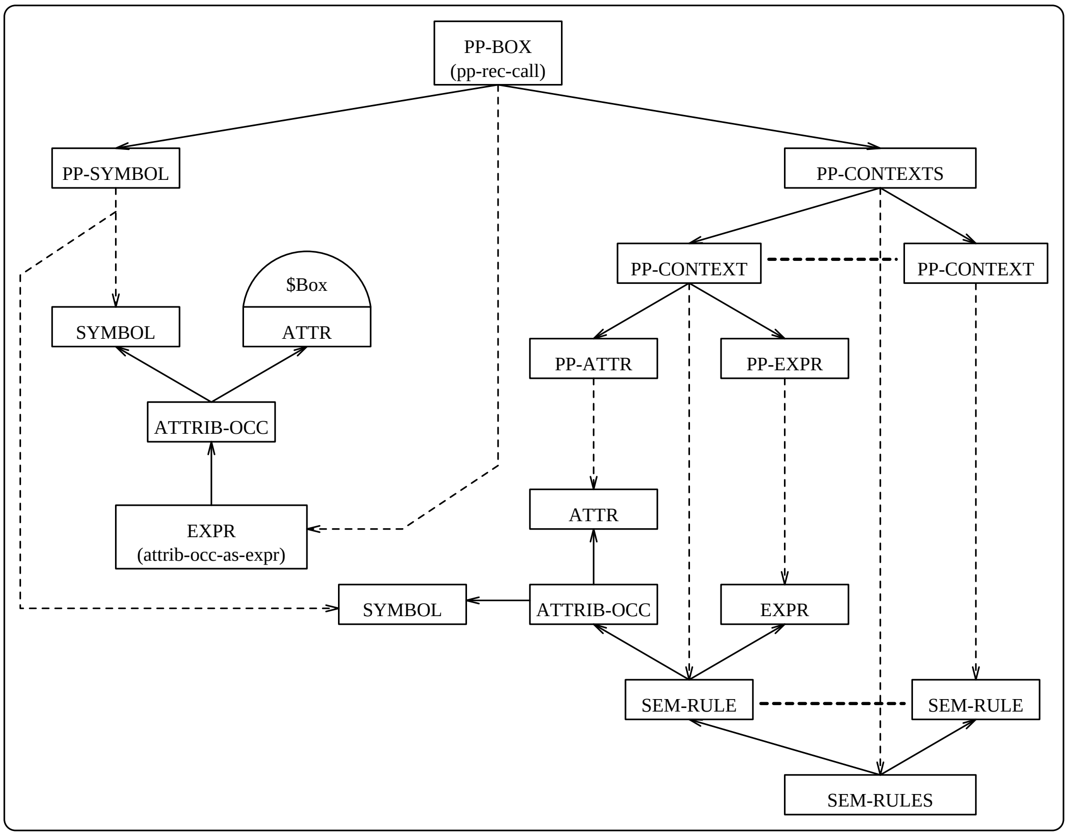

Figure IV.6 : Schéma de traduction simplifiée des décompilations récursives

```
where op -> PH1 PH2 pprint
  if ($att = 1) then
    PH1 ($att1 := 1)
  else
    PH2 ($att2 := 2)
  end if
end where ;
```

Les principes exposés, au paragraphe IV.B-1, concernant la traduction des spécifications de boîtes permet de dire que, même si nous n'avons pas encore vu la traduction des boîtes conditionnelles, nous aurons dans la traduction OLGA, la règle sémantique suivante:

```
  $Box := if ($att = 1) then
            $Box (PH1)
          else
            $Box (PH2)
          end if ;
```

Il faut maintenant traduire les contextes, le problème réside dans le fait que dans le cas présent, la règle sémantique permettant de calculer l'attribut `$att1` du phylum `PH1` doit être une conditionnelle, et que si la condition `($att = 1)` est fausse, nous n'avons pas de valeur à fournir à `$att1`, nous dirons que cette valeur est *manquante*.

Il est facile de se convaincre que le problème est identique pour les formats fondés sur la reconnaissance de motifs, et qu'il faut trouver une solution pour détourner le problème des valeurs *manquantes*.

Nous avons adopté la solution suivante:

A chaque spécification de boîte, nous accrochons:

- *un attribut* `$Expr` *représentant une expression OLGA de type* `BOX`*.*
- *un attribut* `$L-Sem-Rule` *représentant la liste des règles sémantiques issues de la définition de contextes. Cet attribut n'est pas un arbre OLGA, mais une représentation interne.*

L'attribut `$L-Sem-Rule` *de la boîte englobante est transformé en un arbre OLGA de type* `SEM-RULES`*.*
Pour calculer une représentation de règle sémantique associée à un contexte dans une spécification de boîte contrôlée (conditionnelle ou reconnaissance de motif):

- *on utilise la structure de contrôle déduite de la spécification, en conservant toutes les conditions.*
- *Si pour une ou plusieurs de ces conditions la valeur du contexte est* manquante*, on utilise à la place une expression* factice *choisie de façon à conserver les propriétés de typage.*

Cette solution nécessite d'être justifiée, en effet on est en droit de se demander si l'utilisation d'une expression *factice*, ne risque pas de nuire à la cohérence de la spécification produite. La justification repose sur les trois remarques suivantes:

1. La spécification sera correcte vis-à-vis du typage car nous avons précisé que l'expression *factice* était choisie de façon à conserver les propriétés de typage.

2. La spécification sera correcte au niveau logique car nous avons précisé que nous conservions la structure de contrôle, c'est-à-dire par exemple que pour une conditionnelle, toutes les expressions booléennes de conditions sont conservées, dans le même ordre.

3. La spécification sera correcte au niveau du résultat produit (l'arbre de boîte), car nous n'utilisons des expressions *factices* que lorsque la valeur d'un contexte n'est pas définie, or ceci ne peut se passer que si nous sommes dans une partie de la structure de contrôle où l'arbre correspondant au contexte n'est pas décompilé[1], or si le sous-arbre n'est pas décompilé, la valeur donnée au contexte n'a pas d'influence[2] car l'arbre ne sera pas parcouru.

Nous sommes maintenant en mesure de traduire les différents formats.

IV.B-4.c Traduction des boîtes chaîne de caractères

Cette boîte est la plus simple à traduire. En effet elle ne fait pas intervenir de contextes, ni de décompilation de listes.

Donc le seul attribut significatif est `$Expr` qui contient une expression OLGA représentant une boîte `terminal`.

`$Decls` représente une liste vide de déclarations et `$L-Sem-Rule` contient une liste vide de représentations de règles sémantiques.

IV.B-4.d Traduction des occurences d'attributs

Tout comme la précédente, ce format ne fait pas intervenir de contextes, ni de décompilation de listes. Nous avons vu, au chapitre II, qu'un attribut est décompilé grâce au format associée à son type.

Les règles de type étant traduite par des définitions de fonctions, le seul attribut significatif est `$Expr` qui contient une expression OLGA représentant un appel à la fonction associé au type de l'attribut (nous voyons ici l'intérêt d'avoir déterminé le type des attributs devant être affichés au cours de la phase de vérifications).

[1] L'omission de la définition d'un contexte lors de la décompilation d'un arbre est un problème différent qui est traité par la phase de vérification

[2] Nous supposons que l'utilisateur respecte le caractère applicatif de PPAT, donc qu'il ne crée pas d'effet de bord.

IV.B-4.e Traduction des appels récursifs

Nous appliquons ici ce que nous avons expliqué au paragraphe IV.B-4.b.

L'attribut `$Expr` contient une expression OLGA représentant une occurence de l'attribut `$Box` du symbole décompilé.

`$Decls` représente une liste vide de déclarations.

L'attribut `$L-Sem-Rule` contient la liste des représentations de toutes les règles sémantiques associées aux contextes définis sur le symbole.

IV.B-4.f Traduction des boîtes composées

L'attribut `$Expr` représente une expression de construction d'un arbre de boîtes, on utilise la fonction associée à l'opérateur `box` avec pour paramètre, l'expression contenu dans l'attribut `$Expr` du phylum `PP-BOXES` et qui représente un élément de type `BOXES`.

L'attribut `$Decls` contient toutes les déclarations issues des formats associés aux sous-boîtes, et l'attribut `$L-Sem-Rule` la liste de toutes les représentations des règles sémantiques utilisées dans les sous-boîtes.

IV.B-4.g Traduction des décompilations de listes

Nous avons abordé cette traduction au paragraphe IV.B-4.a, où nous avons vu le contenu des attributs `$Decls` et `$Expr`.

L'attribut `$L-Sem-Rule` contient la liste des représentations des règles sémantiques utilisée dans la sous-boîte.

IV.B-4.h Traduction des formats conditionnels

Nous avons vu au paragraphe IV.B-4.b le contenu de l'attribut `$Expr`, il est obtenu grâce à l'attribut `$Elsif-Part(PP-ELSIF-PART)`.

La construction de l'attribut `$L-Sem-Rule` nécessite un double parcours du sous-arbre PPAT représentant le format conditionnel:

1. le premier parcours construit la liste de tous contextes, (dans l'attribut `$L-Attr-Occ`), mentionnés dans les formats intervenant dans le format conditionnel. Cette construction se fait par l'intermédiaire de la fonction:

   $$\mathtt{add-attr-occ(L-ATO, LSR : l-sem-rule) : l-sem-rule}$$

   dont le résultat est son premier argument augmenté des nouvelles représentation de règles sémantiques contenues dans son second argument.

2. le second parcours de l'arbre se fait en utilisant l'attribut `$L-Attr-Occ` pour construire l'attribut `$L-Sem-Rule`. Cette construction se fait à trois niveaux:

(a) au niveau de `PP-ELSIF-CLAUSE`, on utilise la fonction

`construct − ei − clause`

pour construire la liste des représentations de la clause conditionnelle de la façon suivante: pour chaque élément de `$L-Attr-Occ`:

- si cet élément est présent dans la clause, on construit la représentation de la clause de façon naturelle,
- si il est absent, on construit la représentation de la clause en utilisant comme expression *factice* l'expression associée à ce contexte dans `$L-Attr-Occ`.

(b) au niveau de `PP-ELSIF-PART`, on utilise la fonction

`construct − ei − part`

pour construire la liste des représentations de parties *sinon-si* à partir de la liste des listes de représentations de clauses conditionnelles.

(c) au niveau de `PP-BOX`, on utilise la fonction

`construct − conditional`

pour construire la liste des représentations d'expressions conditionnelles, à partir de:

- la liste des représentations de parties *sinon-si*,
- la liste des représentations de règles sémantiques de chacune des deux sous-boîtes correspondant à la partie *vraie* et à la partie *par défaut* de la conditionnelle.

Le fonctionnement de cette fonction est similaire à celui de la fonction `construct-ei-clause`, en ce sens qu'elle construit une représentation d'expression conditionnelle, pour tous les éléments de `$L-Attr-Occ`, en utilisant une expression *factice* lorsque celà est nécessaire.

On peut se convaincre facilement que ce procédé satisfait bien les principes du paragraphe IV.B-4.b. Les structures utilisées peuvent paraître trop complexes, et on peut se demander pourquoi nous avons utilisé des représentations internes de sous-arbres d'OLGA plutôt que ces sous-arbre eux-mêmes, ceci est liée à l'impossibilité, à l'heure actuelle, de faire du *pattern-matching* sur des arbres ASX dans OLGA. Lorsque ceci sera possible, la spécification de la traduction de PPAT vers OLGA sera nettement simplifiable.

IV.B-4.i Traduction des formats contrôlés par des motifs

Le principe de la traduction des formats contrôlés par des motifs est tout à fait similaire à celui de la traduction des formats conditionnels, nous ne revenons donc pas dessus.

IV.B-5 Traduction des règles de formattage des types

Nous avons dit au paragraphe IV.B-1 qu'aux règles de formattage sur les types correspondait une déclaration de fonctions récursives entre elles.

Ainsi chaque règle de type est traduite par une définition de fonction, le nom de cette fonction est donné par le nom du type préfixé par :

`ppat_affichage_`

Ces fonctions ont toutes deux arguments:

- le premier, `object` est du type à décompiler
- le second, `Result` est de type `BOXES`, c'est un auxilliaire de calcul permettant de rendre ces fonctions récursives terminales, car le système FNC–2 repère les fonctions récursives terminales et les traduit par des boucles.

Ces fonctions sont toutes de type `BOX`, ce qui est compatible avec l'usage que nous en avons fait au cours de la traduction des occurences d'attributs.

Nous avons vu que le corps de la fonction était donné par le format de décompilation, nous avons donc l'attribut `$Expr` de type `EXPR` accroché au phylum `PP-BOX-T`, représentant une expression OLGA de type `BOX`.

Nous ne revenons pas sur les traductions des formats chaîne et boîte composée, qui sont en tout point semblables à celles effectuées pour les règles de formattage des opérateurs.

IV.B-5.a Traduction associée à `pp-field-ref`

La traduction associée à l'opérateur `pp-field-ref` a ceci de particulier que, nous avons regroupé sous cet opérateur deux règles de formattage ayant des significations sémantiques relativement différentes. En effet cet opérateur peut représenter:

- l'accès à un champ d'un type enregistrement
- l'accès à la variable locale d'une décompilation de type liste.

La différence entre ces deux significations est faîte au cours de la phase de vérifications et le résultat est passé à la phase de traduction au moyen de l'attribut `$is_var_loc`.

Dans les deux cas, l'expression représente un appel à une fonction de décompilation associée à un type. Dans le cas de l'accès à un champ, il s'agit de la fonction associée au type de ce champ. Dans le cas de l'accès à la variable locale d'une décompilation du type liste `list of T`, il s'agit de la fonction associée à `T`.

IV.B-5.b Traduction des formats associés aux types liste

L'attribut `$Expr` contient la représentation d'une expression OLGA effectuant un choix suivant le nombre déléments de son argument `object` (qui, rappelons le, doit-être de type liste):

- si `object` contient au moins deux éléments, la valeur retournée est celle de l'appel de la fonction avec:

    1. comme premier argument `object` privé de son premier élément (ou de son dernier, suivant le sens de parcours de la liste)
    2. comme second argument `Result` auquel on a rajouté la boîte issue de la composition avec le séparateur `<h 0>` de:
        - la sous-boîte (dont le format est spécifié aprè le mot clef `subbox`)
        - de l'éventuelle boîte chaîne de caractères à insérer

- si `object` contient un seul élément, la valeur retournée est celle de l'appel de la fonction avec:

    1. comme premier argument la liste vide,
    2. comme second argument `Result` auquel on a rajouté la boîte issue de la composition avec le séparateur `<h 0>` de la sous-boîte seule

- si `object` est vide, la valeur retournée est celle correspondant à la boîte issue de la composition de `Result` par le séparateur mentionné après le mot clef `global`

L'utilité du deuxième argument des fonctions de décompilation des attributs apparaît donc ici, puisque nous voyons qu'il permet de rendre terminale la recursion des fonctions associées aux types liste.

Conclusion

Nous avons expliqué la manière dont les principales structures de PPAT sont traduites en OLGA. La totalité de la grammaire `ag_ppat-olga.olga` réalisant cette traduction est présentée dans l'annexe A

IV.C Impression de l'arbre OLGA

Nous disposons à ce stade de la *phase de compilation* de PPAT d'un arbre abstrait attribué représentant une grammaire attribuée. Nous allons présenter la façon dont nous imprimons cet arbre, tout d'abord par le biais d'une grammaire attribuée écrivant dans un fichier, puis nous parlerons de la possibilité d'auto-amorcer le système PPAT.

IV.C-1 Implantation *manuelle*

Nous avons écrit la grammaire attribuée `ppat-print-olga.olga` qui prend un arbre de type `olga-simpli` en entrée et écrit dans un fichier une représentation concrète de cet arbre.

Les actions de manipulation de fichier réalisent bien entendu des effets de bords. Or nous avons dit qu'une des particularités de la programmation avec les grammaires attribuées, est que l'on ne maitrise pas, à priori, l'ordre d'évaluation des attributs. Donc pour que le résultat de l'impression soit correct, il faut réaliser celle-ci en forçant les dépendances entre les attributs de façon à forcer l'ordre d'évaluation.

Pour ce faire, nous utiliserons un attribut mixte: `$code`, de type `File`, défini sur tous les phyla de la syntaxe abstraite `olga-simpli`.

Le principe d'écriture de la grammaire est très semblable à celui de génération des règles par défaut telles que nous l'avons présenté au paragraphe I.D-2.b du chapitre I.

Donnons deux exemples de productions pour illustrer notre propos, le premier pour illustrer le traitement des opérateurs d'arité fixe:

```
where AG-header ->  ID-TER  AG-PROFILE  use
  $name := $ident (ID-TER) ;
  $h.code(AG-PROFILE) := write(write ($h.code ,
                                      "attribute grammar "),
                               $ident(ID-TER)) ;
  $s.code := write ($s.code(AG-PROFILE) , "is\n") ;
end where ;
```

le second pour illustrer le traitement des opérateurs liste:

```
where phyla-names ->  PHYLUM * use
  $h.code (PHYLUM) := case position is
                        first : $h.code ;
                        other : write($s.code(PHYLUM.left) ,
                                      " , ") ;
                      end case ;
end where ;
```

IV.C-2 Auto-amorçage

Le système PPAT ayant pour rôle de produire une représentation concrète d'arbres attribués abstraits, nous avons écrit la spécification PPAT de décompilation des arbres du type `olga-simpli`.

Chronologiquement, nous avons d'abord dû implanter tout le système PPAT en utilisant la grammaire `ppat-print-olga` pour imprimer la traduction de PPAT en OLGA, puis ce système étant à même de fonctionner, nous avons remplaçé cette grammaire, par la grammaire produite en compilant avec ce système la spécification `olga-simpli.ppat`.

Nous donnons les deux parties de la spécification PPAT correspondant aux deux exemples du paragraphe précédent:

```
where AG-header ->  ID-TER  AG-PROFILE  pprint
        [hv120 "attribute grammar" ID-TER AG-PROFILE "is"]
end where ;
```

et:

```
where phyla-names ->  PHYLUM * pprint
        pplist left
                global hov100
                insert ","
                subbox PHYLUM
        end pplist
end where ;
```

Un exemple montrant l'utilisation de contextes et de formats conditionnels est donné à la figure IV.7.

La totalité de la spécification `olga-simpli.ppat`, et la grammaire associée, (telle qu'elle a été produite par PPAT) figurent dans l'annexe C.

Conclusion

Dans ce chapitre, nous avons expliqué comment nous avons implanté la *phase de compilation* de PPAT au moyen de spécifications de grammaires attribuées.

Le traitement de ces spécifications par FNC–2, permet d'obtenir une application traduisant une spécification PPAT de décompilation, en une spécification OLGA de grammaire attribuée.

L'objet du chapitre suivant est d'expliquer comment nous avons implanté les spécifications permettant d'imprimer l'arbre de boîtes crée par cette grammaire.

```
 where func-call -> ID-TER ACTUALS pprint
    if ($id(ID-TER) = token("*"))
       |($id(ID-TER) = token("/"))
       |($id(ID-TER) = token("%"))
       |($id(ID-TER) = token("="))
       |($id(ID-TER) = token("!="))
       |($id(ID-TER) = token("~"))
       |($id(ID-TER) = token("<"))
       |($id(ID-TER) = token(">"))
       |($id(ID-TER) = token("<="))
       |($id(ID-TER) = token(">="))
       |($id(ID-TER) = token("&"))
       |($id(ID-TER) = token("|")) then
        [hv120 "("
               ACTUALS($sep := string($id(ID-TER)) ,
                       $is-bin := 2)
               ")"]
    elsif ($id(ID-TER) = token("+"))
          |($id(ID-TER) = token("-"))
          |($id(ID-TER) = token("!")) then
        if ($arity(ACTUALS) = 2) then
            [hv120 "("
                ACTUALS($sep := string($id(ID-TER)) ,
                        $is-bin := 2)
                ")" ]
        else
            [hv120 "("
              ID-TER
              ACTUALS($sep := string($id(ID-TER)) ,
                      $is-bin := 1)
              ")" ]
        end if
    else
        [hv120 ID-TER
               "("
               ACTUALS($sep := ",", $is-bin := 0)
               ")"]
    end if
end where ;
```

Figure IV.7 : Extrait de la spécification `olga-simpli.ppat`

Chapitre V

Implantation de la phase d'éxécution

Introduction

Nous disposons, à ce stade de l'implantation du système PPAT, d'un traducteur de PPAT vers OLGA, qui fourni une grammaire attribuée fonctionnelle prenant en entrée un arbre spécifié par l'utilisateur, et fournissant en sortie un arbre de boîtes décrit par la syntaxe abstraite `boxes-in.asx` (voir annexe refspecif-exec).

Afin de fournir une représentation concrète de l'arbre décompilé, nous devons dans un premier temps calculer les caractéristiques des boîtes, c'est-à-dire leurs coordonnées et leurs dimensions, puis nous devrons afficher cet arbre.

V.A Calcul des coordonnées des boîtes

La structure d'arbre de boîtes décrite dans la spécification `boxes-in` définit une hiérarchie parmi les boîtes, fournit des renseignements quand-à leur dispositions les unes par rapport aux autres (attribut `$separator`), et le contenu des boîtes simple, dite `terminal` (grâce à l'attribut `$string-box`).

A partir de ces renseignements, nous devons calculer la position absolue de ces boîtes par rapport à l'origine située au coin supérieur gauche de la boîte englobante, dans le système de coordonnées habituel de l'informatique (axe des abscisses orienté de la gauche ver la droite et axe des ordonnées du bas vers le haut).

Nous faisons alors les hypothèses suivantes:

- *un caractère a pour hauteur 1 et largeur 1*
- *une chaîne de n caractères a pour hauteur 1 et pour largeur n*
- *nous appellerons* `largeur-page` *la largeur (en nombre de caractères) du support de sortie.*

Nous allons dans un premier temps montrer que le problème n'est pas soluble par une grammaire attribuée traitée par FNC–2, puis nous proposerons une solution résolvant le problème de façon approchée.

V.A-1 Circularité du problème

Dans la suite, nous noterons pour une boîte B:

- $\mathtt{XY}(B)$ ses coordonnées
- $\mathtt{DIM}(B)$ ses dimensions

Nous écrirons une boîte B résultat de la composition de n boites par un séparateur `sep` :

$$B = (sep, B_i, n)$$

pour spécifier que B_i est elle-même le résultat d'une composition nous écrirons:

$$B_i = (sep_i, B_{ij}, n_i)$$

Le problème consiste à déterminer, pour chaque boîte B, le couple

$$\mathtt{INFO}(B) = (\mathtt{XY}(B), \mathtt{DIM}(B))$$

Pour une boîte terminale B, $\mathtt{INFO}(B)$ est donné par les hypothèses de départ.

Considérons le cas de la boite

$$B = (sep, B_i, n)$$

Supposons connu l'ensemble des $\mathtt{DIM}(B_i)$ et $\mathtt{XY}(B)$. Il existe alors une fonction f telle que:

$$\forall i \in [1 \ldots n-1] : \mathtt{XY}(B_{i+1}) = f(sep, \mathtt{INFO}(B_i), \mathtt{XY}(B))$$

ceci nous permettrait de calculer $\mathtt{INFO}(B_i)$ pour toutes les sous-boîtes et par conséquent, $\mathtt{INFO}B$.

Nous avons vu au chapitre définissant les séparateurs, que la disposition relative des boîtes dépend de la largeur du support de sortie dans le cas d'un séparateur de type `hv` ou `hov`. Donc si on a:

$$B_i = (sep_i, B_{ij}, n_i)$$

où sep est de type `hv` ou `hov`,alors $\mathtt{DIM}(B_i)$ dépend de $\mathtt{XY}(B_i)$. Mais nous avons vu que $\mathtt{XY}(B_i)$ dépend de $\mathtt{XY}(B)$ qui dépend lui-même de $\mathtt{DIM}(B_i)$.

Le problème est donc circulaire, et ne peut pas être traité par le système FNC–2 sans approximations.

V.A-2 Définition et résolution d'un problème approché

Nous avons vu que le problème du calcul des coordonnées est circulaire, cependant, nous souhaitons l'implanter par une grammaire attribuée, donc nous sommes dans l'obligation de trouver une solution approchée.

Après avoir défini le problème approché, et montré en quoi il est approché, nous expliquerons la méthode d'implantation.

V.A-2.a Simplification du problème

Nous avons vu que le caractère circulaire du problème est lié aux séparateurs `hv` et `hov`, et plus précisément, à la nécessité de *décider* de la position d'une boîte en fonction de ses dimensions qui dépendent de sa position.

L'idée de la simplification est de choisir un critère de décision de la position d'une boîte indépendant de ses dimensions. On a vu par ailleurs que le problème serait soluble si tous les séparateurs étaient de type `h` et `v`, c'est-à-dire si le support de sortie était de largeur infinie. Ceci nous a conduit à choisir comme critère de décision, la largeur des boîtes en supposant le support de sortie infini.

Nous appelons *dimensions naturelles* d'une boîte B, et nous notons `DIM-NAT`(B), les dimensionsde la boîte B calculées en supposant le support de sortie infini (c'est-à-dire en considérant les séparateurs `hv` et `hov` comme des séparateurs `h`).

Le problème approché consiste alors à remplacer les définitions des séparateurs `hv` et `hov` du chapitre II par:

- *Deux boîtes B1 et B2 composées par le séparateur* `<hv dx , di , dh>`*, sont disposées de telle sorte que:*
  - *si B2,* **considérée avec ses dimensions naturelles***, alignée avec la dernière ligne de B1, avec un espacement horizontal de* `dx` *caractères, tient dans le support d'affichage, alors elles sont disposées de cette façon.*
  - *sinon, B2 est placé* `dh` *lignes en dessous de B1 avec une indentation de* `di` *caractères par rapport à la boîte englobante.*
- *Une liste de boîtes composées par le séparateur* `<hov dx , di , dh>` *sont disposées:*
  - *de la même façon qu'avec le séparateur* `<h dx>` *si cette disposition* **considérée avec les dimensions naturelles***, tient dans le support d'affichage*
  - *de la même façon qu'avec le séparateur* `<v di , dh>` *sinon.*

Montrons sur un exemple que le problème approché peut ne pas conduire à la même mise en page que le problème exact:

Soit B1, B2, B3, B4, quatre boîtes terminales. La figure V.1 montre alors la différence entre les deux mises en page pour la composition:

$$[< \texttt{hov} > \texttt{B1}[< \texttt{hv} > \texttt{B2B3B4}]]$$

V.A-2.b Résolution du problème approché

Le problème approché n'étant plus circulaire, nous pouvons le résoudre par une grammaire attribuée traitable par FNC–2.

Le principe de la résolution est d'effectuer deux parcours de l'arbre de boîte:

1. au cours du premier parcours, les dimensions naturelles de toutes les boîtes sont calculées dans l'attribut `$DIM-nat` au moyen de la fonction `DIM-NAT`.
2. au cours du second parcours, les coordonnées des boîtes sont calculées par la fonction `XY`, ce qui permet d'en déduire les dimensions réelles.

Il faut faire la distinction entre la phase de décision et la phase de calcul, en effet pour une sous-boîte d'une boîte composée, le calcul des coordonnées se fait en fonction de ses dimensions naturelles (phase de décision) et des dimensions réelles de la boîte qui la précède (phase de calcul), ceci est obligatoire pour que la mise en page corresponde avec la définition approchée des séparateurs.

L'implantation de ces calculs est réalisée dans les spécifications suivantes:

- `ag_boxes-coord.olga` grammaire attribuée réalisant les deux parcours de l'arbre
- `dec_calcul-coord-box.olga` et `def_calcul-coord-box.olga` modules contenant les fonctions nécessaires aux calculs.

La totalité de ces spécifications se trouve dans l'annexe B.

V.B Impression du contenu des boîtes

L'impression de l'arbre de boîtes se fait suivant le même principe que l'impression de l'arbre OLGA telle qu'elle a été décrite au paragraphe IV.C-1 du chapitre IV, c'est-à-dire par impression dans un fichier.

L'impression se fait par la fonction `write-at` que nous avons rajoutée au module `dec_file.olga`, elle permet l'impression d'une chaîne de caractères à une position donnée, en lui fournissant les coordonnées de la boîte, et les coordonnées courantes du *curseur*:

```
File write__at (f,old_x,old_y,actual_x,actual_y,contents)
File f ;
int old_x,old_y,actual_x,actual_y ;
```

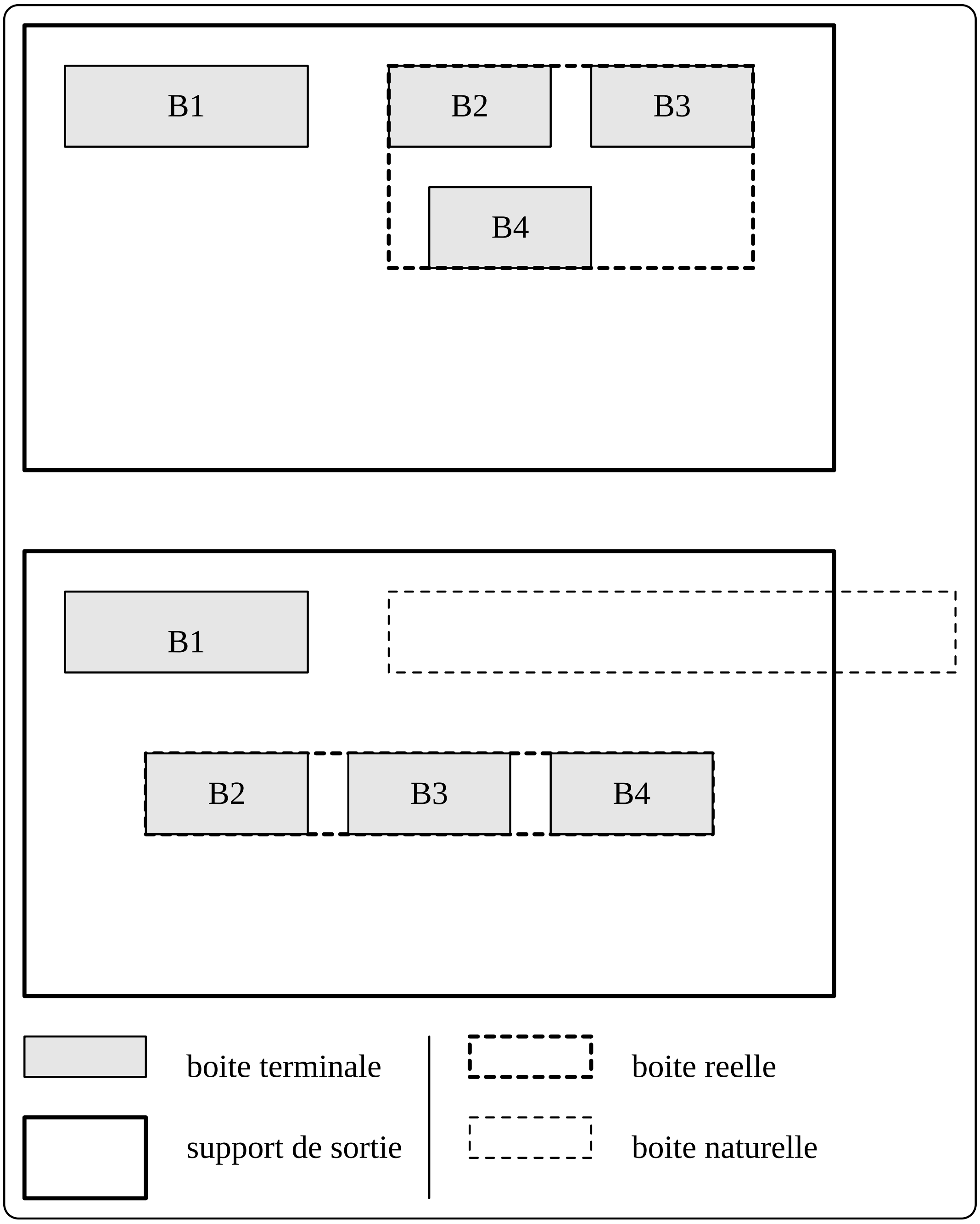

Figure V.1 : Mise en page exacte (en haut) et approchée (en bas)

```
char *contents ;
{
  int i , dx , dy ;
  dy = actual_y - old_y ;
  dx = (dy != 0) ? actual_x : actual_x - old_x ;
  for (i = 0 ; i < dy ; i++)
    fprintf(f,"\n") ;
  for (i = 0 ; i < dx ; i++)
    fprintf(f," ") ;
  fprintf(f,"%s",contents) ;

  return f;
}
```

Les coordonnées courantes du curseur sont véhiculées dans l'arbre par l'attribut mixte `$curr-coord`. Comme pour l'impression de l'arbre OLGA, l'attribut mixte `$sortie` permet de contrôler l'ordre dévaluation des attributs.

La seule production *intéressante* de la grammaire

$$\mathtt{ag\_boxes} - \mathtt{print} - \mathtt{ascii.olga}$$

réalisant l'affichage, est la production associée à une boîte terminale:

```
where terminal ->  STRING-BOX  use
  $s.sortie := write-at(
                 $h.sortie ,
                 $h.curr-coord.X ,
                 $h.curr-coord.Y ,
                 $Coord.COORD.X ,
                 $Coord.COORD.Y ,
                 $string-box(STRING-BOX)) ;
  $s.curr-coord := xy-box($Coord.COORD.X + $Coord.DIM.W ,
                          $Coord.COORD.Y) ;
end where ;
```

Conclusion

Le langage PPAT permet, grâce à un formalisme puissant fondé sur la notion de boîte, de spécifier des représentations concrètes d'arbres abstraits, et son apprentissage par l'utilisateur d'OLGA est grandement facilité par le fait que ces deux langages sont voisins.

La solution d'implantation réalisée au cours de stage, utilise pleinement le principe de modularit'e du système FNC–2. Ceci est particulièrement intéressant dans le cadre d'une *interface* telle que PPAT. En effet, l'utilisateur désirant *personaliser* son décompilateur d'arbres abstraits, (par exemple: en rajoutant de nouveaux séparateurs, ou en modifiant les principes de mise en page, ou encore en choisissant de visualiser les arbres sous le format LaTeX), n'a besoin de modifier que certaines spécifications, ce qui est beaucoup plus aisé que de modifier une spécification qui représenterait la totalité du système.

Outre, l'intérêt de réaliser un outil, la conception de PPAT, a permis de tester, et parfois d'améliorer, le système FNC–2. En effet si on ne prend pas en compte FNC–2 lui-même[1], PPAT est la première application écrite en utilisant FNC–2, et sa réalisation a soulevée des problèmes jusque là presque ignorés (en particulier tout ce qui concerne les grammaires attribuées fonctionnelles).

Les futures réalisations du projet Chlo'e[2], devraient permettre d'améliorer le système PPAT. En particulier le coupleur de grammaires attribuées[3] et l'évaluateur incrémental d'attributs[4], permettrons de rendre plus interactive la génération d'un décompilateur d'arbres attribués, en permettant de voir *instantanément* l'effet de la modification d'une règle de formatage sur la représentation concrète de l'arbre.

Enfin nous espérons que PPAT sera un compagnon agréable et efficace pour les utilisateurs de FNC–2, à la fois pour obtenir de *beaux* listings, mais aussi pour mettre au point d'autres applications en utilisant la possibilité de visualiser des attributs de types complexes.

[1] FNC–2 étant presque totalement auto-amorcé (*bootstrappé*), c'est la première (et la seule avant PPAT) application du système FNC–2

[2] ainsi que l'implantation des facultés d'OLGA non disponibles pour le moment et dont nous avons montré au cours de ce rapport leur intérêt pour PPAT

[3] en cours de réalisation par Gilles Roussel

[4] en cours de réalisation par Olivier Durin

Bibliographie

[Knu68] D.E. Knuth, "Semantics of Context-free Languages," *Math. Systems Theory* **2,** 2 (June 1968), 127–145. Correction: *Math. Systems Theory* **5,** 1 (Mar. 1971), 95–96.

[JP89] M. Jourdan & D. Parigot, "The FNC–2 System User's Guide and Reference Manual" Release 0.4—February 1989.

[DJL88] P. Deransart, M. Jourdan & B. Lorho, *Attribute Grammars: Definitions, Systems and Bibliography,* Lect. Notes in Comp. Sci. #**323**, Springer-Verlag, New York–Heideilberg–Berlin, Aug. 1988.

[JPJ90] M.Jourdan, D. Parigot, C. Julié, O. Durin & C. Le Bellec, "Design, Implementation and Evaluation of the FNC–2 Attribute Grammar System," in *ACM SIGPLAN '90 Conf. on Programming Languages Design and Implementation*, White Plains, NY, published as *ACM SIGPLAN Notices* **25**, 6 (June 1990), 209–222.

[JLP90] M. Jourdan, C. Le Bellec & D. Parigot, "The Olga Attribute Grammar Description Language: Design, Implementation and Evaluation," in *Attribute Grammars and their Applications* (WAGA), Paris, P.Deransart & M.Jourdan, eds., 222–237, Lect. Notes in Comp. Sci., Springer-Verlag, New York–Heideilberg–Berlin, Sept. 1990.

[Kas80] U. Kastens, "Ordered Attribute Grammars," *Acta Inform.* **13**, 3 (1980), 229–256.

[Kas84] U. Kastens, "The GAG-System—A Tool for Compiler Construction," in *Methods and Tools for Compiler Construction,* B. Lorho, ed. 165–182, Cambridge Univ. Press, Cambridge, 1984.

[EnJ90] J. Engelfriet & W. de Jong, "Attribute Storage Optimization by Stacks," *Acta Inform.* (1990).

[JuP90] C. Julié & D. Parigot, "Space Optimization in the FNC–2 Attribute Grammar System," in *Attribute Grammars and their Applications (WAGA)*, Paris, P. Deransart & M. Jourdan, eds, 29–45, Lect. Notes in Comp. Sci., Springer-Verlag, New York–Heideilberg–Berlin, Sept. 1990.

[CoF82] B. Courcelle & P. Franchi–Zannettacci, “Attribute Grammars and Recursive Program Schemes,” *Theoret. Comput. Sci.* **17**, 2 and 3 (1982), 163–191 and 235–257.

[Bor88] P. Borras “Reference Manual & Compiler Implementation”, SEMA-METRA, (jan. 1988).

Troisième partie

Annexes

Annexe A

Spécifications de la phase de compilation

A.A Grammaires abstraites et concrètes

A.A-1 ppat-base.asx

```
 { -------------------------------------------------------
   -  ASX de base du langage PPAT                        -
   ------------------------------------------------------- }

grammar ppat-base is

root is PP-PROG ;

PP-PROG = pp-prog ;
pp-prog -> PP-HEADER  PP-HOOK ;

  PP-HEADER = pp-header ;
  pp-header -> PP-ID       { decompilator's name }
               PP-ID ;       { input asx }

  PP-HOOK = pp-hook ;
  pp-hook -> PP-DECLS        { contexts and separators declaractions }
             PP-BLOCK ;        { decompilation rules }

    PP-DECLS = pp-decls ;
    pp-decls -> PP-DECL * ;

      PP-DECL = pp-sep-def pp-cxt-decl pp-import-all ;
      { declaration of a separator }
      pp-sep-def -> PP-SEP-ID               { separator's name }
                    PP-SEPARATOR ;        { separator's definition }
        PP-SEPARATOR = pp-separator ;
        pp-separator -> PP-ID               { separator's type }
                        PP-SEP-PARAM ;        { separator's parameters }
          PP-SEP-PARAM = pp-sep-param ;
          pp-sep-param -> PP-INT * ;

      { declaration of a context }
      pp-cxt-decl -> PP-ATTR               { attribut's name }
                     PP-PHYLA               { list of the phylum where it is defined }
                     PP-ID ;               { type of the context }
        PP-PHYLA = pp-phyla;
        pp-phyla -> PP-PHYLUM * ;

      { declaration of a module }
```

```
  pp-import-all -> PP-ID ;

PP-BLOCK = pp-block ;
pp-block -> PP-RULES-T               { rules for complex types decompilation }
            PP-RULES-OP ;               { rules for operator decompilation }
  PP-RULES-T = pp-rules-t ;
  pp-rules-t -> PP-RULE-T * ;
    PP-RULE-T = pp-rule-t ;
    pp-rule-t -> PP-ID               { name of the type }
                 PP-TOP-BOX-T ;       { associated box }
      PP-TOP-BOX-T = pp-empty-box-t pp-top-box-t ;
      pp-empty-box-t -> ;
      pp-top-box-t -> PP-BOX-T ;
        PP-BOX-T = pp-string-box-t pp-compound-box-t pp-field-ref pp-pplist-box-t ;
          pp-string-box-t -> PP-STRING ;
          pp-compound-box-t -> PP-SEP-ID       { name of the separator to be used }
                               PP-BOXES-T ;       { list of the boxes to be compound }
            PP-BOXES-T = pp-boxes-t ;
            pp-boxes-t -> PP-BOX-T * ;

          pp-field-ref -> PP-ID ;               { name of a field }
          pp-pplist-box-t -> PP-ID               { variable representing current elt }
                             PP-DIR               { direction for the transversal of the list }
                             PP-SEP-ID       { name of the separator to be used }
                             PP-INSERT       { string to be inserted between two boxes }
                             PP-BOX-T ;       { kind of the box to be used }

  PP-RULES-OP = pp-rules-op ;               { rules for an operator }
  pp-rules-op -> PP-RULE-OP * ;
    PP-RULE-OP = pp-rule-op ;
    pp-rule-op -> PP-PRODUCTION               { the production which is concerned }
                  PP-TOP-BOX ;              { the box to be used }
      PP-PRODUCTION = pp-ne-list-prod pp-pe-list-prod pp-fixed-prod ;
      pp-ne-list-prod -> PP-ID PP-PHYLUM ;       { never empty list {+} }
      pp-pe-list-prod -> PP-ID PP-PHYLUM ;       { perhaps empty list {*} }
      pp-fixed-prod ->  PP-ID PP-PROD-RHS ;       { fixed arity operator }
        PP-PROD-RHS = pp-prod-rhs ;
        pp-prod-rhs -> PP-PHYLUM * ;
      PP-TOP-BOX = pp-empty-box pp-top-box ;
      pp-empty-box  -> ;
      pp-top-box -> PP-BOX ;
        PP-BOX = pp-string-box pp-attrib pp-rec-call
                 pp-conditional pp-compound-box pp-case-match
                 pp-case-tree pp-pplist-box ;
          pp-string-box -> PP-STRING;
          pp-attrib -> PP-ATTRIB-OCC;
          pp-rec-call -> PP-SYMBOL               { phylum's name }
                         PP-CONTEXTS ;       { decompilation contexts }
            PP-CONTEXTS = pp-contexts;
            pp-contexts -> PP-CONTEXT *;
              PP-CONTEXT  = pp-context;
              pp-context  -> PP-ATTR       { attribut's name }
                             PP-EXPR ;       { value }
          pp-conditional -> PP-EXPR               { first condition }
                            PP-BOX               { true part }
                            PP-ELSIF-PART       { eventually other conditions }
                            PP-BOX ;       { default part }
            PP-ELSIF-PART = pp-elsif-part ;
            pp-elsif-part -> PP-ELSIF-CLAUSE * ;
              PP-ELSIF-CLAUSE = pp-elsif-clause ;
              pp-elsif-clause -> PP-EXPR PP-BOX ;
          pp-compound-box -> PP-SEP-ID       { name of the separator to be used }
                             PP-BOXES;       { list of the boxes to be compound }
            PP-BOXES = pp-boxes;
            pp-boxes -> PP-BOX *;
          pp-case-tree -> PP-SYMBOL               { selector }
```

```
                              PP-MATCH-CLAUSES        { pattern matching clauses }
                              PP-OTHER-CLAUSE ;        { default clause }
              pp-case-match -> PP-EXPR
                               PP-MATCH-CLAUSES
                               PP-OTHER-CLAUSE ;
                PP-MATCH-CLAUSES = pp-match-clauses;
                pp-match-clauses  -> PP-MATCH-CLAUSE + ;
                  PP-MATCH-CLAUSE  = pp-match-clause ;
                  pp-match-clause  -> PP-PATTERNS PP-BOX ;
                    PP-PATTERNS  = pp-patterns ;
                    pp-patterns  -> PP-PATTERN * ;
                      PP-PATTERN  = pp-pattern-cons pp-pattern-var-or-cons
                                    pp-pattern-const-manif  pp-single-case-const ;
                      pp-pattern-cons  -> PP-ID PP-PATTERNS ;
                      pp-pattern-var-or-cons -> PP-ID ;
                      pp-pattern-const-manif -> PP-EXPR ;
                      pp-single-case-const -> PP-EXPR ;
                PP-OTHER-CLAUSE  = pp-match-clause-other;
                pp-match-clause-other -> PP-BOX ;
              pp-pplist-box -> PP-DIR   PP-SEP-ID PP-INSERT PP-BOX ;

PP-DIR = pp-left-dir pp-right-dir ;
pp-left-dir -> ;
pp-right-dir -> ;

PP-INSERT = pp-no-insert pp-insert;
pp-no-insert -> ;
pp-insert -> PP-STRING;

PP-EXPR = pp-bool-and pp-bool-or pp-func-call pp-attrib-occ-as-expr
          pp-int-const  pp-char-const pp-string-const  pp-real-const
          pp-applied-id ;
  pp-bool-and -> PP-EXPR PP-EXPR ;
  pp-bool-or -> PP-EXPR PP-EXPR ;
  pp-func-call -> PP-ID PP-ACTUALS ;
  pp-attrib-occ-as-expr -> PP-ATTRIB-OCC ;
  pp-int-const -> PP-INT;
  pp-char-const -> PP-CHAR;
  pp-string-const -> PP-STRING;
  pp-real-const -> PP-REAL;
  pp-applied-id -> PP-ID;
  PP-ACTUALS = pp-actuals ;
    pp-actuals -> PP-ACTUAL * ;
    PP-ACTUAL = pp-actual ;
    pp-actual -> PP-EXPR ;
PP-ATTRIB-OCC = pp-attrib-occ-lhs pp-attrib-occ ;
              pp-attrib-occ-lhs -> PP-ATTR ;
              pp-attrib-occ -> PP-ATTR PP-SYMBOL ;

PP-SYMBOL = pp-unqual-symb pp-qual-symb ;
pp-unqual-symb -> PP-ID ;
pp-qual-symb -> PP-ID PP-QUALIFIER ;
  PP-QUALIFIER = pp-int-qual pp-left-qual pp-right-qual pp-first-qual pp-last-qual ;
  pp-int-qual -> PP-INT ;
  pp-left-qual -> ;
  pp-right-qual -> ;
  pp-first-qual -> ;
  pp-last-qual  -> ;

PP-ID = pp-id pp-op-id;
pp-id -> ;
pp-op-id -> ;

PP-SEP-ID = pp-sep-id;
pp-sep-id -> ;
```

```
PP-INT = pp-int ;
pp-int -> ;

PP-CHAR = pp-char ;
pp-char -> ;

PP-REAL = pp-real ;
pp-real -> ;

PP-STRING = pp-string ;
pp-string -> ;

PP-PHYLUM = pp-phylum ;
pp-phylum ->  ;

PP-ATTR = pp-attr ;
pp-attr -> ;

end grammar ;
```

A.A-2 ppat-in.asx

```
{  -------------------------------------------------------------
   -  ASX du langage PPAT, resultant de l'ATC et servant de      -
   -  syntaxe d'entree a la grammaire attribuee ppat-check       -
   ------------------------------------------------------------- }

grammar ppat-in is

import grammar ppat-base ;

root is PP-PROG ;
{---        declaration des attributs de type token        ---}
 $pp-id          (PP-ID)          : token ;
 $pp-sep-id      (PP-SEP-ID)      : token ;
 $pp-phylum      (PP-PHYLUM)      : token ;
 $pp-attr        (PP-ATTR)        : token ;
 $pp-int         (PP-INT)         : token ;
 $pp-string      (PP-STRING)      : token ;
 $pp-real        (PP-REAL)        : token ;
 $pp-char        (PP-CHAR)        : token ;
end grammar ;
```

A.A-3 ppat-out.asx

```
{  -------------------------------------------------------------
   -  ASX du langage PPAT, resultant de ppat-check et servant   -
   -  de syntaxe d'entree a la grammaire attribuee ppat-olga    -
   ------------------------------------------------------------- }

grammar ppat-out is

import grammar ppat-base ;
from ppat-types import all ;

root is PP-PROG ;

{--- nom interne du module ppat ---}
 $nom         (PP-PROG)                : string ;
{---           table des types         ---}
 $S_TYPE      (PP-PROG)                : symbol_table_type ;
{--- liste des separateurs ---}
 $S_SEP       (PP-PROG)                : st-sep ;
 $sep-info    (PP-SEPARATOR)           : sep-info ;
{--- Types To Be Printed  ---}
```

```
 $S_TTBP      (PP-PROG)                : st_type_olga ;
{--- types des att-occ devant etre affichees ---}
 $type        (PP-ATTRIB-OCC)          : type_olga ;
 $s.type      (PP-TOP-BOX-T , PP-BOX-T) : type_olga ;
{--- identification de la racine ---}
 $h_root      (PP-TOP-BOX)             : bool ;
{--- variable locale dans une pplist de type ---}
 $is_var_loc  (PP-BOX-T)               : bool ;
{--- declaration des attributs de type token ---}
 $pp-id       (PP-ID)                  : token ;
 $pp-sep-id   (PP-SEP-ID)              : token ;
 $pp-phylum   (PP-PHYLUM)              : token ;
 $pp-attr     (PP-ATTR)                : token ;
 $pp-int      (PP-INT)                 : token ;
 $pp-string   (PP-STRING)              : token ;
 $pp-real     (PP-REAL)                : token ;
 $pp-char     (PP-CHAR)                : token ;
end grammar ;
```

A.A-4 ppat.atc

```
{  ---------------------------------------------------------
   -     Grammaire concrete et construction d'arbres      -
   -                   pour PPAT                          -
   --------------------------------------------------------- }

import grammar ppat-in ;

PP-SEP-ID       += no-sep-id ;
no-sep-id       -> ;

function bin-func-call (op : PP-ID ; expr1, expr2 : PP-EXPR) : PP-EXPR is
{ Constructs a function call with two arguments from a binary expression,
  or a short-circuit expression, according to the given operator. }
    if op = pp-id () with $pp-id := "and" end with then
        pp-bool-and (expr1, expr2)
    elsif op = pp-id () with $pp-id := "or" end with then
        pp-bool-or (expr1, expr2)
    else
        pp-func-call(
          op ,
          pp-actuals-pre (pp-actual (expr1), pp-actuals (pp-actual (expr2))))
    end if
 end function ;

function unary-func-call (op : PP-ID ; expr : PP-EXPR) : PP-EXPR is
{ Same as "bin-func-call" above but for unary functions. }
    pp-func-call (op, pp-actuals (pp-actual (expr)))
end function ;

function boxes-const(boxlist:PP-BOXES;b:PP-BOX;sep:PP-SEP-ID):PP-BOXES is
{ Rajoute la boite "b" a la liste "boxlist", mais si il y a }
{ un separateur local , on construit un boite composee      }
{ avec le dernier element de "boxlist" et "b".              }
{ Cette boite remplace le dernier element de "boxlist".     }
  case sep match
    no-sep-id () :
      pp-boxes-post(boxlist,b);                { pas de separateur local }
    other :
      case boxlist match                       { separateur local }
        pp-boxes(b2) :
          pp-boxes(pp-compound-box(sep,pp-boxes-pre(b2,pp-boxes(b))));
        pp-boxes-post(boxlist2,b2) :
          pp-boxes-post(boxlist2,pp-compound-box(sep,pp-boxes-pre(b2,pp-boxes(b))));
      end case;
  end case
```

```
end function;

function boxes-const-t(boxlist:PP-BOXES-T;b:PP-BOX-T;sep:PP-SEP-ID):PP-BOXES-T is
{ Meme fonction pour les boites de decompilation des types }
  case sep match
    no-sep-id() : { pas de separateur local }
      pp-boxes-t-post(boxlist,b);
    other : { separateur local }
      case boxlist match
        pp-boxes-t(b2) :
          pp-boxes-t(pp-compound-box-t(sep,pp-boxes-t-pre(b2,pp-boxes-t(b))));
        pp-boxes-t-post(boxlist2,b2) :
          pp-boxes-t-post(
            boxlist2,
            pp-compound-box-t(sep,pp-boxes-t-pre(b2,pp-boxes-t(b))));
      end case;
  end case
end function;

*** programme ppat ***
<ppat prog>        = <header> <decls> <rules> end ppat ";" ;
  pp-prog(<header> , pp-hook(<decls> , <rules>))
*** header : nom du programme et type des l'arbres a traiter ***
<header>       = ppat <ID> "(" <ID> ")" ";" ;
  pp-header(<ID>.1 , <ID>.2)
*** declarations ***
<decls>               = <decls> <decls sep> ;
  match <decls> with pp-decls-all (L1) ,
        <decls sep> with pp-decls-all (L2) in
                pp-decls-merge(L1 , L2)
<decls>               = <decls> <decls cxt> ;
  match <decls> with pp-decls-all (L1) ,
        <decls cxt> with pp-decls-all (L2) in
                pp-decls-merge(L1 , L2)
<decls>               = <decls> <import clause> ;
  pp-decls-post(<decls> , <import clause>)
<decls>               = ;
  pp-decls ()
*** importation ***
<import clause>       = from <ID> import all ";" ;
  pp-import-all(<ID>)
*** declarations des separateurs ***
<decls sep>       = separator <decl sep +> ;
<decl sep +>       = <decl sep +> <sep def> ;
  pp-decls-post(<decl sep +> , <sep def>)
<decl sep +>       = <sep def> ;
  pp-decls(<sep def>)
<sep def>       = "<" <SEP ID> ">" "=" <separator> ";" ;
  pp-sep-def(<SEP ID> , <separator>)
<separator>       = "<" <ID> <sep param *> ">" ;
  pp-separator(<ID> , <sep param *>)
<sep param *>       = <sep param +> ;
<sep param *>       = ;
  pp-sep-param ()
<sep param +>       = <sep param +> "," <sep param> ;
  pp-sep-param-post(<sep param +> , <sep param>)
<sep param +>       = <sep param> ;
  pp-sep-param(<sep param>)
<sep param>       = <INT> ;
*** declarations des contextes ***
<decls cxt>       = context <decl cxt +> ;
<decl cxt +>       = <decl cxt +> <cxt def> ;
  pp-decls-post(<decl cxt +> , <cxt def>)
<decl cxt +>       = <cxt def> ;
  pp-decls(<cxt def>)
<cxt def>       = <attrib name> "(" <phyla, *> ")" ":" <ID> ";" ;
```

```
  pp-cxt-decl(<attrib name> , <phyla, *> , <ID>)
<phyla, *>        = <phyla, +> ;
<phyla, *>        = ;
  pp-phyla ()
<phyla, +>        = <phyla, +> "," <phylum> ;
  pp-phyla-post(<phyla, +> , <phylum>)
<phyla, +>        = <phylum> ;
  pp-phyla(<phylum>)
*** specifications des regles  ***
<rules>                = <rule type *> <rule op *> ;
  pp-block(<rule type *> , <rule op *>)
*** specifications des decompilation d'attributs ***
<rule type *>        = <rule type *> <rule type> ;
  pp-rules-t-post(<rule type *> , <rule type>)
<rule type *>        = ;
  pp-rules-t ()
<rule type>        = wheretype <ID> pprint
                        <top box type>
                  end where ";" ;
  pp-rule-t(<ID> , <top box type>)
<top box type>        = <box type> ;
  pp-top-box-t(<box type>)
<top box type>        = ;
  pp-empty-box-t ()
<box type>        = <string boxt> ;
<box type>        = <compound boxt> ;
<box type>        = <fieldref boxt> ;
<box type>        = <pplist boxt> ;
<string boxt>        = <STRING> ;
  pp-string-box-t(<STRING>)
<compound boxt>        = "[" <SEP ID> <boxes type> "]" ;
  pp-compound-box-t(<SEP ID> , <boxes type>)
<boxes type>        = <boxes type> <local sep> <box type> ;
  boxes-const-t(<boxes type> , <box type> , <local sep>)
<boxes type>        = <box type> ;
  pp-boxes-t-post(pp-boxes-t () , <box type>)
<fieldref boxt>        = <ID> ;
  pp-field-ref(<ID>)
<pplist boxt>        = pplist <ID> <direction>
                        global <SEP ID>
                        <insert>
                        subbox <box type>
                  end pplist ;
  pp-pplist-box-t(<ID> , <direction> , <SEP ID> , <insert> , <box type>)
*** specifications des decompilation d'operateurs ***
<rule op *>        = <rule op *> <rule op> ;
  pp-rules-op-post(<rule op *> , <rule op>)
<rule op *>        = ;
  pp-rules-op ()
<rule op>        = where <production> pprint
                        <top box>
                  end where ";" ;
  pp-rule-op(<production> , <top box>)
<production>        = <ID> "->" <phylum> "+" ;
  pp-ne-list-prod(<ID> , <phylum>)
<production>        = <ID> "->" <phylum> "*" ;
  pp-pe-list-prod(<ID> , <phylum>)
<production>        = <ID> "->" <phyla *> ;
  pp-fixed-prod(<ID> , <phyla *>)
<phyla *>        = <phyla +> ;
<phyla *>        = ;
  pp-prod-rhs ()
<phyla +>        = <phyla +> <phylum> ;
  pp-prod-rhs-post(<phyla +> , <phylum>)
<phyla +>        = <phylum> ;
  pp-prod-rhs(<phylum>)
```

```
<top box>        = <box> ;
  pp-top-box(<box>)
<top box>        = ;
  pp-empty-box ()
<box>                = <string box> ;
<box>                = <attrib occ> ;
  pp-attrib(<attrib occ>)
<box>                = <rec call> ;
<box>                = <conditional> ;
<box>                = <compound> ;
<box>                = <case match> ;
<box>                = <case tree> ;
<box>                = <pplist> ;
<string box>        = <STRING> ;
  pp-string-box(<STRING>)
<attrib occ>        = <attrib name> ;
  pp-attrib-occ-lhs(<attrib name>)
<attrib occ>        = <attrib name> "(" <symbol> ")" ;
  pp-attrib-occ(<attrib name> , <symbol>)
<symbol>        = <ID> ;
  pp-unqual-symb(<ID>)
<symbol>        = <ID> "." <qualifier> ;
  pp-qual-symb(<ID> , <qualifier>)
<qualifier>        = <INT> ;
  pp-int-qual(<INT>)
<qualifier>        = left ;
  pp-left-qual ()
<qualifier>        = right ;
  pp-right-qual ()
<qualifier>        = first ;
  pp-first-qual ()
<qualifier>        = last ;
  pp-last-qual ()
<rec call>         = <symbol> <context list> ;
  pp-rec-call(<symbol>, <context list>)
<context list>        = "(" <context *> ")"  ;
<context list>        = ;
  pp-contexts ()
<context *>        = <context +> ;
<context +>        = <context +> "," <context>;
  pp-contexts-post(<context +> , <context>)
<context +>        = <context> ;
  pp-contexts(<context>)
<context>        = <attrib name> ":=" <expr>;
  pp-context(<attrib name> , <expr>)
<attrib name>        = %ATTR ;
  pp-attr () with $pp-attr := %ATTR end with
<attrib name>        = $scx ;
  pp-attr () with $pp-attr := "$scx" end with
<attrib name>        = $arity ;
  pp-attr () with $pp-attr := "$arity" end with
<attrib name>        = $position ;
  pp-attr () with $pp-attr := "$position" end with
<conditional>        = if  <expr>  then
                        <box>
                  <elsif part>
                  else
                        <box>
                  end  if ;
  pp-conditional(<expr> , <box>.1 , <elsif part> , <box>.2)
<elsif part>        = <elsif part>
                  elsif  <expr>  then
                        <box> ;
  pp-elsif-part-post(<elsif part> , pp-elsif-clause(<expr> , <box>))
<elsif part>        = ;
```

```
  pp-elsif-part ()
<compound>        = "[" <SEP ID> <box +> "]" ;
  pp-compound-box(<SEP ID> , <box +>)
<box +>                = <box +> <local sep> <box> ;
  boxes-const(<box +> , <box> , <local sep>)
<box +>                = <box> ;
  pp-boxes-post(pp-boxes () , <box>)
<case tree>        = case  tree <symbol>  match
                      <match clauses>
                      <match default>
                  end  case ;
  pp-case-tree(<symbol>, <match clauses> , <match default>)
<case match>        = case <expr> is
                      <match clauses>
                      <match default>
                  end  case ;
  pp-case-match(<expr>, <match clauses> , <match default>)
<match clauses>        = <match clauses>
                  <match clause> ;
  pp-match-clauses-post(<match clauses> , <match clause>)
<match clauses>        = <match clause> ;
  pp-match-clauses(<match clause>)
<match clause>        = <case clauses>  ":"
                      <box>  ";" ;
  pp-match-clause(<case clauses> , <box>)
<case clauses>        = <case clauses> ","  <case clause> ;
  pp-patterns-post(<case clauses> , <case clause>)
<case clauses>        = <case clause> ;
  pp-patterns(<case clause>)
<match default>        = other  ":"
                      <box>  ";" ;
  pp-match-clause-other(<box>)
<case clause> = <root pattern> ;
<root pattern>        = <pattern cons> ;
<root pattern>        = <ID> ;
  pp-pattern-var-or-cons(<ID>)
<root pattern>        = <constant manifeste> ;
  pp-pattern-const-manif(<constant manifeste>)
<root pattern>        = const <expr> ;
        pp-single-case-const(<expr>)
<pattern cons>        = <ID>  "(" <patterns *> ")" ;
  pp-pattern-cons(<ID> , <patterns *>)
<patterns *>        = <patterns +> ;
<patterns *>        = ;
  pp-patterns ()
<patterns +>        = <patterns +> ","  <pattern> ;
  pp-patterns-post(<patterns +> , <pattern>)
<patterns +>        = <pattern> ;
  pp-patterns(<pattern>)
<constant manifeste> = <INT> ;
  pp-int-const (<INT>)
<constant manifeste> = %REAL ;
  pp-real-const (pp-real () with $pp-real := %REAL end with)
<constant manifeste> = %CHAR ;
  pp-char-const (pp-char () with $pp-char := %CHAR end with)
<constant manifeste> = %STRING ;
  pp-string-const (pp-string () with $pp-string := %STRING end with)
<pattern>        = <root pattern> ;
<pplist>        = pplist  <direction>
                         global <SEP ID>
                         <insert>
                         subbox  <box>
                  end  pplist ;
  pp-pplist-box(<direction> , <SEP ID> , <insert> , <box>)
<direction>        = left ;
  pp-left-dir ()
```

```
<direction>        = right ;
  pp-right-dir ()


<local sep>        = "<" <SEP ID> ">" ;
<local sep>        = ;
  no-sep-id ()
<insert>       = insert <STRING> ;
  pp-insert (<STRING>)
<insert>       = ;
  pp-no-insert ()

<expr>              = <INT> ;
  pp-int-const(<INT>)
<expr>              = <REAL> ;
  pp-real-const(<REAL>)
<expr>              = <CHAR> ;
  pp-char-const(<CHAR>)
<expr>              = <STRING> ;
  pp-string-const(<STRING>)
<expr>              = <ID> ;
  pp-applied-id(<ID>)
<expr>              = "(" <expr> ")" ;
<expr>              = <function call> ;
<expr>              = <attrib occ> ;
  pp-attrib-occ-as-expr(<attrib occ>)
<expr>              = <expr> "+" <expr> ;
  bin-func-call(pp-id() with $pp-id := "+" end with, <expr>.1 , <expr>.2)
<expr>              = <expr> "-" <expr> ;
  bin-func-call(pp-id() with $pp-id := "-" end with, <expr>.1 , <expr>.2)
<expr>              = <expr> "*" <expr> ;
  bin-func-call(pp-id() with $pp-id := "*" end with, <expr>.1 , <expr>.2)
<expr>              = <expr> "/" <expr> ;
  bin-func-call(pp-id() with $pp-id := "/" end with, <expr>.1 , <expr>.2)
<expr>              = <expr> "%" <expr> ;
  bin-func-call(pp-id() with $pp-id := "%" end with, <expr>.1 , <expr>.2)
<expr>              = <expr> "=" <expr> ;
  bin-func-call(pp-id() with $pp-id := "=" end with, <expr>.1 , <expr>.2)
<expr>              = <expr> "!=" <expr> ;
  bin-func-call(pp-id() with $pp-id := "!=" end with, <expr>.1 , <expr>.2)
<expr>              = <expr> "~" <expr> ;
  bin-func-call(pp-id() with $pp-id := "~" end with, <expr>.1 , <expr>.2)
<expr>              = <expr> "!~" <expr> ;
  bin-func-call(pp-id() with $pp-id := "!" end with, <expr>.1 , <expr>.2)
<expr>              = <expr> "<" <expr> ;
  bin-func-call(pp-id() with $pp-id := "<" end with, <expr>.1 , <expr>.2)
<expr>              = <expr> ">" <expr> ;
  bin-func-call(pp-id() with $pp-id := ">" end with, <expr>.1 , <expr>.2)
<expr>              = <expr> "<=" <expr> ;
  bin-func-call(pp-id() with $pp-id := "<=" end with, <expr>.1 , <expr>.2)
<expr>              = <expr> ">=" <expr> ;
  bin-func-call(pp-id() with $pp-id := ">=" end with, <expr>.1 , <expr>.2)
<expr>              = <expr> "&" <expr> ;
  bin-func-call(pp-id() with $pp-id := "&" end with, <expr>.1 , <expr>.2)
<expr>              = <expr> "|" <expr> ;
  bin-func-call(pp-id() with $pp-id := "|" end with, <expr>.1 , <expr>.2)
<expr>              = <expr> "and" <expr> ;
  bin-func-call(pp-id() with $pp-id := "and" end with, <expr>.1 , <expr>.2)
<expr>              = <expr> "or" <expr> ;
  bin-func-call(pp-id() with $pp-id := "or" end with, <expr>.1 , <expr>.2)
<expr>              = "-" <expr> ;
  unary-func-call(pp-id() with $pp-id := "-" end with, <expr>)
<expr>              = "+" <expr> ;
  unary-func-call(pp-id() with $pp-id := "+" end with, <expr>)
<expr>              = "!" <expr> ;
  unary-func-call(pp-id() with $pp-id := "!" end with, <expr>)
```

```
<function call>       = <ID> "(" <actual *> ")" ;
  pp-func-call(<ID> , <actual *>)
<actual *>       = <actual +> ;
<actual *>       = ;
  pp-actuals ()
<actual +>       = <actual +> "," <actual> ;
  pp-actuals-post(<actual +> , <actual>)
<actual +>       = <actual> ;
  pp-actuals(<actual>)
<actual>       = <expr> ;
  pp-actual(<expr>)
<INT>              = %INT ;
  pp-int () with $pp-int := %INT end with
<CHAR>              = %CHAR ;
  pp-char () with $pp-char := %CHAR end with
<STRING>       = %STRING ;
  pp-string () with $pp-string := %STRING end with
<ID>              = %ID ;
  pp-id () with $pp-id := %ID end with
<SEP ID>       = %ID ;
  pp-sep-id () with $pp-sep-id := %ID end with
<REAL>              = %REAL ;
  pp-real () with $pp-real := %REAL end with
<phylum>       = %ID ;
  pp-phylum () with $pp-phylum := %ID end with
```

A.A-5 olga-simpli.asx

```
{ -------------------------------------------------------
   -   grammaire abstraite du sous langage d'OLGA        -
   -   dans lequel sont generees les traductions         -
   -   des specifications PPAT par la GA ppat-olga       -
   ------------------------------------------------------- }

grammar olga-simpli is

root is COMPILATION-UNIT ;

  $id (ID-TER) : token ;
  $module (MODULE-ID) : token ;
  $phylum (PHYLUM) : token ;
  $int (INT) : token ;
  $attr (ATTR) : token ;
  $real (REAL) : token ;
  $char (CHAR) : token ;
  $string (STRING) : token ;

ID = unqual-id ;
unqual-id -> ID-TER;

ID-TER = id-ter ;
id-ter -> ;

TYPE-ID = type-id ;
type-id -> ID ;

COMPILATION-UNIT =  simple-AG ;
simple-AG -> AG-HEADER HOOK ;

        HOOK = hook-simple-AG ;
        hook-simple-AG -> DECLS BLOCK ;

AG-HEADER = AG-header ;
AG-header -> ID-TER AG-PROFILE ;
AG-PROFILE = functional-AG ;
functional-AG -> SYNTAX SYNTAXES ;
```

```
        SYNTAXES = syntaxes ;
        syntaxes -> SYNTAX * ;
        SYNTAX = syntax ;
        syntax -> ID-TER ;

SEMANTICS = productions ;
productions -> PRODUCTION-GROUP * ;

DECLS = decls ;
decls -> DECL * ;
DECL =  import-all simple-func-def rec-funcs-defs
        const-decl value-decl attribute-decl ;

MODULE-ID = module-id ;
module-id ->;

import-all -> MODULE-ID ;

PHYLA-NAMES = phyla-names ;
phyla-names -> PHYLUM * ;
PHYLUM = phylum ;
phylum -> ;

const-decl -> ID-TER BLOCK ;

value-decl -> ID-TER TYPE-ID BLOCK ;

simple-func-def -> FUNC-DEF ;
rec-funcs-defs -> FUNC-DEF * ;
        FUNC-DEF = func-def ;
        func-def -> FUNC-HEADER FUNC-BODY ;

attribute-decl -> ATTRIBUTE-KIND ATTR PHYLA-NAMES TYPE-ID ;

FUNC-HEADER = func-header ;
func-header -> CALL-MODE ID-TER FORMALS TYPE-ID EXCS;
        CALL-MODE = offline-mode ;
        offline-mode -> ;
        FORMALS = formals ;
        formals -> FORMAL * ;
                FORMAL = formal ;
                formal -> ID-TER TYPE-ID ;
        EXCS = excs ;
        excs  -> ID * ;

FUNC-BODY = simple-body decls-and-body ;
simple-body -> BLOCK ;
decls-and-body -> DECLS BLOCK ;

ATTRIBUTE-KIND = inh-kind syn-kind ;
inh-kind -> ;
syn-kind -> ;

PRODUCTION-GROUP = simple-prod-group prod-group-with-decls ;
simple-prod-group -> PRODUCTION SEM-RULES ;
prod-group-with-decls -> PRODUCTION DECLS SEM-RULES ;

PRODUCTION = ne-list-prod pe-list-prod fixed-prod ;
ne-list-prod -> ID-TER PHYLUM ;
pe-list-prod -> ID-TER PHYLUM ;
fixed-prod ->         ID-TER PROD-RHS ;
                PROD-RHS = prod-rhs ;
                prod-rhs -> PHYLUM * ;

SEM-RULES = sem-rules ;
sem-rules -> SEM-RULE * ;
```

```
SEM-RULE = simple-rule ;

simple-rule -> ATTRIB-OCC EXPR ;


ATTRIB-OCC = attrib-occ-lhs attrib-occ ;
attrib-occ-lhs -> ATTR ;
attrib-occ -> ATTR SYMBOL ;

        ATTR = attr ;
        attr -> ;
        SYMBOL = unqual-symb qual-symb ;
        unqual-symb -> ID-TER ;
        qual-symb -> ID-TER QUALIFIER ;
        QUALIFIER = int-qual up-qual left-qual right-qual first-qual last-qual ;
                int-qual -> INT ;
                up-qual -> ;
                left-qual -> ;
                right-qual -> ;
                first-qual -> ;
                last-qual -> ;

INT = int ;
int -> ;
CHAR = char ;
char -> ;
REAL = real ;
real -> ;
STRING = string ;
string -> ;

BLOCK = block-expr block-semantics ;
block-expr -> EXPR ;
block-semantics -> SEMANTICS ;

EXPR =  constant-expr int-const char-const string-const real-const applied-id
        attrib-occ-as-expr field-access func-call list-constr bool-and bool-or
        let-expr conditional case-match case-tree map-expr tree-with ;

constant-expr -> EXPR ;
int-const -> INT ;
real-const -> REAL ;
char-const -> CHAR ;
string-const -> STRING ;
applied-id -> ID ;
attrib-occ-as-expr -> ATTRIB-OCC ;

field-access -> EXPR ID-TER ;
func-call ->         ID-TER ACTUALS ;
        ACTUALS = actuals ;
        actuals -> ACTUAL * ;
        ACTUAL = actual ;
        actual -> EXPR ;

list-constr -> TYPE-ID LIST-ELTS ;
        LIST-ELTS = list-elts ;
        list-elts -> ACTUAL * ;

bool-and -> EXPR EXPR ;
bool-or -> EXPR EXPR ;

let-expr -> LET-CLAUSES BLOCK ;
        LET-CLAUSES = let-clauses ;
        let-clauses -> LET-CLAUSE + ;
        LET-CLAUSE = let-clause ;
```

```
        let-clause -> ID-TER TYPE-ID EXPR ;

conditional ->          EXPR EXPR ELSIF-PART EXPR ;
        ELSIF-PART = elsif-part ;
        elsif-part -> ELSIF-CLAUSE * ;
        ELSIF-CLAUSE = elsif-clause ;
        elsif-clause -> EXPR EXPR ;

case-match ->           EXPR MATCH-CLAUSES MATCH-CLAUSE ;
        MATCH-CLAUSES = match-clauses ;
        match-clauses -> MATCH-CLAUSE + ;
        MATCH-CLAUSE = match-clause match-clause-other;
        match-clause -> PATTERNS BLOCK ;
        match-clause-other -> EXPR ;
        PATTERN = pattern-cons pattern-var-or-const pattern-var-constraint
                  pattern-const-manif single-case-const ;
        pattern-cons -> ID PATTERNS ;
        pattern-var-or-const -> ID-TER ;
        pattern-var-constraint -> ID-TER TYPE-ID ;
        pattern-const-manif -> EXPR ;
        single-case-const -> EXPR ;
        PATTERNS = patterns ;
        patterns -> PATTERN * ;

case-tree -> SYMBOL MATCH-CLAUSES MATCH-CLAUSE ;

map-expr -> MAP-DIR ID EXPR ACTUALS ;
        MAP-DIR = left-dir right-dir ;
        left-dir -> ;
        right-dir -> ;

tree-with -> EXPR DECORATIONS ;
        DECORATIONS = decorations ;
        decorations -> DECORATION + ;
        DECORATION = decoration ;
        decoration -> ATTR EXPR ;
end grammar ;
```

A.B Grammaires attribuées

A.B-1 ag_ppat-check.olga

```
{ -----------------------------------------------------
   -  Grammaire attribue'e ppat-check :                -
   -    verification d'une specif ppat , construction: -
   -                * des tables olga,                 -
   -                * des separateurs,                 -
   -                * de la liste des types complexes  -
   -                  a afficher                       -
   ---------------------------------------------------- }

attribute grammar ppat-check (ppat-in , ppat-out) is
  from message_olga import all ;
  from ppat-types import all ;

attribute
{--- attributs de sortie ---}
  synthesized $S_TYPE(PP-PROG) : symbol_table_type ;
  synthesized $S_SEP(PP-PROG)  : st-sep ;
  synthesized $S_TTBP(PP-PROG) : st_type_olga ;
  synthesized $nom(PP-PROG)    : string ;

{--- attribut representant les tables globales ---}
```

```
  global $MODULE_TYPE(PP-HOOK) : module_type ;
  global $NAT(PP-HOOK)         : symbol_table_nat_olga ;
  global $PH(PP-HOOK)          : liste_ph ;
  global $OP(PP-HOOK)          : liste_op ;
  global $PH_OP(PP-HOOK)       : liste_ph_right_op ;

{--- nom interne du module ---}
  global $name_module(PP-HOOK) : info_name ;

{--- attributs representant les tables en cours de construction  ---}
  compound $h.MODULE_TYPE
      and $s.MODULE_TYPE(PP-HEADER,PP-HOOK,PP-DECLS,PP-DECL ) : module_type ;
  compound $h.NAT
      and $s.NAT(PP-HEADER,PP-HOOK )                           : symbol_table_nat_olga;
  global   $CVI(PP-BOX)                                        : symbol_table_cvi ;
  compound $h.CVI
      and $s.CVI(PP-MATCH-CLAUSES,PP-MATCH-CLAUSE,
                          PP-OTHER-CLAUSE,PP-PATTERNS,
                          PP-PATTERN )                         : symbol_table_cvi ;

{--- attributs pour les separateurs ---}
  global      $SEP(PP-HOOK)                     : st-sep ;
  compound    $h.SEP
      and     $s.SEP(PP-HOOK,PP-DECLS,PP-DECL ) : st-sep ;
  synthesized $sep-info(PP-SEPARATOR)           : sep-info ;
  synthesized $sep-param(PP-SEP-PARAM)          : sep-param ;

{--- attributs pour les types a afficher ---}
  compound $h.TTBP
      and $s.TTBP(PP-HOOK,PP-BLOCK,PP-RULES-T,PP-RULE-T) : st_type_olga ;
  global   $TTBP(PP-RULES-OP)                            : st_type_olga ;

{--- attributs divers ---}
  synthesized $ASX(PP-HEADER)                  : liste_asx_info ;
  synthesized $name(PP-HEADER,PP-ID)           : info_name ;
  synthesized $name_id(PP-ATTR,PP-SYMBOL)      : info_name_qua ;
  synthesized $ph(PP-PHYLUM,PP-SYMBOL)         : ph_info ;
  inherited   $h_ph(PP-CONTEXTS,PP-CONTEXT)    : ph_info ;
  synthesized $liste_ph(PP-PHYLA,PP-PROD-RHS) : liste_ph ;
  synthesized $op-prod(PP-PRODUCTION)          : op_info ;
  global      $op(PP-TOP-BOX)                  : op_info ;
  synthesized $nat(PP-ATTR)                    : nat_olga ;
  synthesized $pos(PP-SYMBOL,PP-QUALIFIER )    : int ;
  inherited   $h_pos(PP-CONTEXTS,PP-CONTEXT ) : int ;
  synthesized $kind_phylum(PP-SYMBOL,PP-QUALIFIER )   : kind_occ ;
  inherited   $h_kind_phylum(PP-CONTEXTS,PP-CONTEXT ) : kind_occ ;
  compound    $h.type
      and     $s.type(PP-TOP-BOX-T,PP-BOX-T,PP-BOXES-T )               : type_olga ;
  inherited   $type_pattern(PP-MATCH-CLAUSES,PP-MATCH-CLAUSE,
                          PP-OTHER-CLAUSE )                            : type_olga ;
  inherited   $case_match(PP-MATCH-CLAUSES,PP-MATCH-CLAUSE,
                        PP-OTHER-CLAUSE )                              : kind_expr ;
  synthesized $actual(PP-PATTERNS,PP-PATTERN,PP-ACTUALS,PP-ACTUAL )    : liste_type ;
  synthesized $liste_id(PP-PATTERNS,PP-PATTERN)                        : liste_id ;
  synthesized $type(PP-SYMBOL,PP-PATTERNS,PP-PATTERN,PP-ATTRIB-OCC,
                 PP-EXPR,PP-INT,PP-CHAR,PP-REAL,PP-STRING,PP-ACTUAL ) : type_olga ;
  synthesized $s_root(PP-PRODUCTION)        : bool ;
  inherited   $h_root(PP-TOP-BOX )          : bool ;
  inherited   $is-top(PP-BOX-T,PP-BOX )     : bool ;
  inherited   $var_loc(PP-BOX-T,PP-BOXES-T) : token ;
  synthesized $is_var_loc(PP-BOX-T)         : bool ;
  synthesized $correct(PP-PROG,PP-HEADER,PP-HOOK,PP-DECLS,PP-DECL,
                     PP-SEPARATOR,PP-SEP-PARAM,PP-PHYLA,PP-BLOCK,
                     PP-RULES-T,PP-RULE-T,PP-TOP-BOX-T,PP-BOX-T,
                     PP-BOXES-T,PP-RULES-OP,PP-RULE-OP,PP-PRODUCTION,
                     PP-PROD-RHS,PP-TOP-BOX,PP-BOX,PP-CONTEXTS,PP-CONTEXT,
```

```
                      PP-ELSIF-PART,PP-ELSIF-CLAUSE,PP-BOXES,PP-MATCH-CLAUSES,
                      PP-MATCH-CLAUSE,PP-PATTERNS,PP-PATTERN,PP-OTHER-CLAUSE,
                      PP-DIR,PP-INSERT,PP-EXPR,PP-ACTUALS,PP-ACTUAL,
                      PP-ATTRIB-OCC,PP-SYMBOL,PP-QUALIFIER,PP-ID,PP-SEP-ID,
                      PP-INT,PP-CHAR,PP-REAL,PP-STRING,PP-PHYLUM,PP-ATTR) : bool ;

{--- regles par defaut ---}
defaults

classes $CORRECT ($correct) : bool ;
        $Type ($type) : type_olga ;

default $CORRECT use
  where NULL use
        false
  end where ;

  where -> () use
        true
  end where ;

  where -> (X) *+ use
        map left &
          value true
          other $CORRECT (X)
        end map
  end where ;

  where -> (X) use
    $CORRECT (X)
  end where ;

  where -> (X , Y) use
    $CORRECT (X) & $CORRECT (Y)
  end where ;

  where -> (X , Y , Z) use
    $CORRECT (X) & $CORRECT (Y) & $CORRECT (Z)
  end where ;
end default ;

default $Type use
  where NULL use
        error-type-info
  end where ;
end default ;

{- ----------------------------------------------------------------- -}
{- PP-PROG                                                            -}
{- ----------------------------------------------------------------- -}
where null-PP-PROG ->  use
  $S_TTBP := st_type_olga () ;
  $S_SEP := st-sep () ;
  $S_TYPE := symbol_table_type () ;
  $nom := "" ;
end where ;

where pp-prog ->  PP-HEADER  PP-HOOK declare
value
  MODULE_TYPE : module_type := init_table_olga(
                                      $scx ,
                                      AG ,
                                      info_name(
                                        $name(PP-HEADER).name,
                                        id_no_module) ) ;
use
```

```
  $correct := $correct (PP-HEADER)
           & MODULE_TYPE.correct
           & $correct (PP-HOOK) ;
  $nom := string($name(PP-HEADER).name) ;
  $name_module (PP-HOOK) := info_name($name(PP-HEADER).name , id_no_module) ;
  $h.MODULE_TYPE (PP-HEADER) := MODULE_TYPE ;
  $h.MODULE_TYPE (PP-HOOK) := $s.MODULE_TYPE (PP-HEADER) ;
  $MODULE_TYPE (PP-HOOK) := $s.MODULE_TYPE (PP-HOOK) ;
  $h.NAT (PP-HEADER) := empty-nat-olga ;
  $h.NAT (PP-HOOK) := $s.NAT (PP-HEADER) ;
  $NAT (PP-HOOK) := $s.NAT (PP-HOOK) ;
  $PH (PP-HOOK) := rech_liste_ph ($ASX (PP-HEADER)) ;
  $OP (PP-HOOK) := rech_liste_op ($ASX (PP-HEADER)) ;
  $PH_OP (PP-HOOK) := table_ph_right_op(
                        rech_liste_ph($ASX (PP-HEADER)) ,
                        rech_liste_op($ASX (PP-HEADER)) ,
                        liste_ph_right_op () ) ;
  $S_TYPE := $s.MODULE_TYPE(PP-HOOK).TYPE ;
  $h.SEP (PP-HOOK) := st-sep () ;
  $S_SEP := $s.SEP (PP-HOOK) ;
  $SEP (PP-HOOK) := $s.SEP (PP-HOOK) ;
  $h.TTBP (PP-HOOK) := st_type_olga () ;
  $S_TTBP := $s.TTBP (PP-HOOK) ;
end where ;

{- ------------------------------------------------------------------------ -}
{- PP-HEADER                                                                -}
{- ------------------------------------------------------------------------ -}
where null-PP-HEADER ->  use
  $name := error-info-name ;
  $ASX := liste_asx_info () ;
end where ;

where pp-header ->  PP-ID  PP-ID  declare
value
  asx-module : asx_module := asx_read(
                                $scx ,
                                DF ,
                                $name (PP-ID.2) ,
                                asx_module(
                                  true ,
                                  empty-module ,
                                  init-type ,
                                  empty-op ,
                                  empty-ph ,
                                  init-attr ) ,
                                true , true , true ) ;
  TYPE : symbol_table_type := entree_type_asx(
                                asx-module.PHYLA ,
                                $h.MODULE_TYPE.TYPE ,
                                false ) ;
use
  $correct := $correct (PP-ID.1) & $correct (PP-ID.2) & asx-module.correct ;
  $name := $name (PP-ID) ;
  $s.MODULE_TYPE := module_type(
                      asx-module.correct ,
                      $h.MODULE_TYPE.CVI ,
                      TYPE ,
                      $h.MODULE_TYPE.FUNC ,
                      $h.MODULE_TYPE.MODULE ) ;
  $s.NAT := entree_nat_asx(
              IMPORTER,
              $scx ,
              asx-module.ATTRIBUT ,
              TYPE ,
              $h.NAT ,
```

```
              false ) ;
  $ASX := liste_asx_info(
            asx_info($name(PP-ID.2) , ASX_PROC , asx-module) ,
            liste_asx_info () ) ;
end where ;

{- ------------------------------------------------------------------ -}
{- PP-HOOK                                                            -}
{- ------------------------------------------------------------------ -}
where pp-hook ->  PP-DECLS  PP-BLOCK  use
  $h.MODULE_TYPE (PP-DECLS) := $h.MODULE_TYPE ;
  $s.MODULE_TYPE := $s.MODULE_TYPE (PP-DECLS) ;
  $h.NAT (PP-DECLS) := $h.NAT ;
  $s.NAT := $s.NAT (PP-DECLS) ;
  $h.SEP (PP-DECLS) := $h.SEP ;
  $s.SEP := $s.SEP (PP-DECLS) ;
  $h.TTBP (PP-BLOCK) := $h.TTBP ;
  $s.TTBP := $s.TTBP (PP-BLOCK) ;
end where ;

{- ------------------------------------------------------------------ -}
{- PP-DECLS                                                           -}
{- ------------------------------------------------------------------ -}
   { regles generees par defaut}
{- ------------------------------------------------------------------ -}
{- PP-DECL                                                            -}
{- ------------------------------------------------------------------ -}
where pp-import-all ->  PP-ID declare
value
  tables : module_type := module_read(
                                    $scx ,
                                    DC ,
                                    $name (PP-ID) ,
                                    $h.MODULE_TYPE , false , false ) ;
use
  $correct := $correct (PP-ID) & tables.correct ;
  $s.MODULE_TYPE := if tables.correct then
                 tables
              else
                 $h.MODULE_TYPE
              end if ;
end where ;

where pp-cxt-decl ->  PP-ATTR  PP-PHYLA PP-ID declare
value
  type_olga : type_olga := lookup_TYPE (
                                  info_name_qua(
                                    $name_module (PP-HOOK.up),
                                    $name(PP-ID) ) ,
                                  $h.MODULE_TYPE.TYPE) ;
  nat_decl : nat_liste_decl := placer_nat_decl(
                                false ,
                                $name_id (PP-ATTR) ,
                                error-info-name-qua ,
                                HERITE ,
                                $liste_ph (PP-PHYLA) ,
                                type_olga ,
                                $scx ,
                                $h.NAT ) ;
use
  $correct := if type_olga ~ error-type-info then
                message message_olga(4)
                position PP-ID
                severity 2
                value false
              else
```

```
              $correct (PP-PHYLA) & $correct (PP-ATTR) & nat_decl.correct
            end if ;
  $s.NAT := nat_decl.NAT ;
end where ;

where pp-sep-def ->  PP-SEP-ID  PP-SEPARATOR declare
value
  correct : bool := if lookup-SEP ($pp-sep-id (PP-SEP-ID) , $h.SEP) then
                        message message_olga (90)
                        position pp-sep-def
                        severity 2
                        value false
                    else
                        $correct (PP-SEPARATOR)
                    end if ;
use
  $correct := $correct (PP-SEP-ID) & correct ;
  $s.SEP := st-sep(
              $h.SEP ,
              sep(
                $pp-sep-id (PP-SEP-ID) ,
                $sep-info (PP-SEPARATOR) )) ;
end where ;

{- --------------------------------------------------------------------- -}
{- PP-SEPARATOR                                                          -}
{- --------------------------------------------------------------------- -}
where null-PP-SEPARATOR ->  use
  $sep-info := sep-info(error , sep-param ()) ;
end where ;

where pp-separator ->  PP-ID  PP-SEP-PARAM  declare
value
  info : sep-info := sep-info(
                        find-sep-kind($pp-id (PP-ID)) ,
                        $sep-param (PP-SEP-PARAM) ) ;
use
  $correct := verif-sep (info , $scx) & $correct (PP-SEP-PARAM) & $correct (PP-ID) ;
  $sep-info := info ;
end where ;

{- --------------------------------------------------------------------- -}
{- PP-SEP-PARAM                                                          -}
{- --------------------------------------------------------------------- -}
where null-PP-SEP-PARAM ->  use
  $sep-param := sep-param () ;
end where ;

where pp-sep-param ->  PP-INT * use
  $sep-param := map left sep-param
                value sep-param ()
                other int(string($pp-int (PP-INT)))
              end map ;
end where ;

{- --------------------------------------------------------------------- -}
{- PP-PHYLA                                                              -}
{- --------------------------------------------------------------------- -}
where null-PP-PHYLA ->  use
  $liste_ph := liste_ph () ;
end where ;

where pp-phyla ->  PP-PHYLUM * use
  $liste_ph := map left liste_ph
              value liste_ph ()
              other $ph (PP-PHYLUM)
```

```
              end map ;
end where ;

{- ------------------------------------------------------------------- -}
{- PP-BLOCK                                                            -}
{- ------------------------------------------------------------------- -}
where pp-block ->  PP-RULES-T  PP-RULES-OP  use
  $h.TTBP (PP-RULES-T) := $h.TTBP ;
  $s.TTBP := $s.TTBP (PP-RULES-T) ;
  $TTBP (PP-RULES-OP) := $s.TTBP (PP-RULES-T) ;
end where ;

{- ------------------------------------------------------------------- -}
{- PP-RULES-T                                                          -}
{- ------------------------------------------------------------------- -}
where pp-rules-t ->  PP-RULE-T * use
  { propagation de $TTBP par les regles par defaut }
end where ;

{- ------------------------------------------------------------------- -}
{- PP-RULE-T                                                           -}
{- ------------------------------------------------------------------- -}
where pp-rule-t ->  PP-ID  PP-TOP-BOX-T  declare
value
  name-qua : info_name_qua := info_name_qua(
                                $name_module (PP-HOOK.up),
                                $name (PP-ID) ) ;
  nouv-types : st-type-correct := rec-lookup-TTBP(
                                        name-qua ,
                                        $h.TTBP ,
                                        $MODULE_TYPE (PP-HOOK.up).TYPE ) ;
  T : type_olga := lookup_TYPE(name-qua , $MODULE_TYPE (PP-HOOK.up).TYPE) ;
use
  $correct := if (T !~ error-type-info) then
                extern_regle_type () & nouv-types.correct
                & $correct (PP-TOP-BOX-T) & $correct (PP-ID)
              else
                message message_olga (18)
                position PP-ID
                severity 2
                value false
              end if ;
  $s.TTBP := if nouv-types.correct then
                nouv-types.types
             else
                $h.TTBP
             end if ;
  $h.type (PP-TOP-BOX-T) := T ;
end where ;

{- ------------------------------------------------------------------- -}
{- PP-TOP-BOX-T                                                        -}
{- ------------------------------------------------------------------- -}
where pp-top-box-t ->  PP-BOX-T  use
  $h.type (PP-BOX-T) := $h.type ;
  $s.type := $h.type ;
  $is-top (PP-BOX-T) := true ;
  $var_loc (PP-BOX-T) := error-token ;
end where ;

where pp-empty-box-t ->  use
  $s.type := $h.type ;
end where ;

{- ------------------------------------------------------------------- -}
{- PP-BOX-T                                                            -}
```

```
{- ------------------------------------------------------------------- -}
where null-PP-BOX-T ->  use
  $is_var_loc := false ;
end where ;

where pp-pplist-box-t ->  PP-ID PP-DIR  PP-SEP-ID  PP-INSERT  PP-BOX-T  declare
value
  correct : bool := case $h.type.info is
                      info_type (liste : info_list) :
                        if $correct (PP-SEP-ID) then
                          if lookup-SEP(
                              $pp-sep-id (PP-SEP-ID) ,
                              $SEP (PP-HOOK.up))
                          then
                            true
                          else
                            message message_olga (92)
                            position PP-SEP-ID
                            severity 2
                            value false
                          end if
                        else
                          false
                        end if ;
                      other :
                        message message_olga (93)
                        position pp-pplist-box-t
                        severity 2
                        value false ;
                    end case ;
  type-elt-lst : type_olga := case $h.type.info is
                                info_type (liste : info_list) :
                                  liste.type_olga ;
                                other :
                                  error-type-info ;
                              end case ;
use
  $correct := if $is-top then
                $correct (PP-DIR)
                & $correct (PP-INSERT)
                & $correct (PP-BOX-T)
                & correct
              else
                message message_olga (96)
                position pp-pplist-box-t
                severity 2
                value false
              end if ;
  $h.type(PP-BOX-T) := type-elt-lst ;
  $var_loc(PP-BOX-T) := $pp-id(PP-ID) ;
  $s.type := $h.type ;
  $is-top (PP-BOX-T) := false ;
  $is_var_loc := false ;
end where ;

where pp-field-ref ->  PP-ID  declare
value
  correct : bool := case $h.type.info is
                      info_type (rec : info_record) :
                        if look-champ-record($name(PP-ID) , rec.liste_formal) then
                          true
                        else
                          if $var_loc = $pp-id(PP-ID) then
                                  true
                          else
                                message message_olga (14)
```

```
                                  position PP-ID
                                   severity 2
                                   value false
                             end if
                           end if ;
                         other :
                           if $var_loc = $pp-id(PP-ID) then
                                   true
                           else
                                   message message_olga (94)
                                   position pp-field-ref
                                   severity 2
                                   value false
                           end if ;
                       end case ;
  type-champ : type_olga := case $h.type.info is
                                 info_type (rec : info_record) :
                                   if $var_loc = $pp-id(PP-ID) then
                                     $h.type
                                   else
                                     look-CHAMP-RECORD($name(PP-ID) , rec.liste_formal)
                                   end if ;
                                 other :
                                   if $var_loc = $pp-id(PP-ID) then
                                     $h.type
                                   else
                                     error-type-info
                                   end if ;
                                 end case ;
use
  $correct := correct & $correct (PP-ID) ;
  $s.type := type-champ ;
  $is_var_loc := if $var_loc = $pp-id(PP-ID) then
                  true
                 else
                  false
                 end if ;
end where ;

where pp-compound-box-t ->  PP-SEP-ID  PP-BOXES-T  declare
value
  correct : bool := if $correct (PP-SEP-ID) then
                      if lookup-SEP ($pp-sep-id (PP-SEP-ID) , $SEP (PP-HOOK.up)) then
                        true
                      else
                        message message_olga (92)
                        position PP-SEP-ID
                        severity 2
                        value false
                      end if
                    else
                      false
                    end if ;
use
  $correct := $correct (PP-BOXES-T) & correct ;
  $h.type (PP-BOXES-T) := $h.type ;
  $s.type := $h.type ;
  $var_loc (PP-BOXES-T) := $var_loc ;
  $is_var_loc := false ;
end where ;

where pp-string-box-t ->  PP-STRING  use
  $s.type := $h.type ;
  $is_var_loc := false ;
end where ;
```

```
{- ----------------------------------------------------------------- -}
{- PP-BOXES-T                                                        -}
{- ----------------------------------------------------------------- -}
where pp-boxes-t ->  PP-BOX-T * use
  $h.type (PP-BOX-T) := $h.type ;
  $s.type := $h.type ;
  $is-top (PP-BOX-T) := false ;
  $var_loc (PP-BOX-T) := $var_loc ;
end where ;

{- ----------------------------------------------------------------- -}
{- PP-RULES-OP                                                       -}
{- ----------------------------------------------------------------- -}
  { regles generees par defaut }
{- ----------------------------------------------------------------- -}
{- PP-RULE-OP                                                        -}
{- ----------------------------------------------------------------- -}
where pp-rule-op ->  PP-PRODUCTION  PP-TOP-BOX  use
  $correct := $correct(PP-PRODUCTION) & $correct(PP-TOP-BOX) & extern_regle_op () ;
  $h_root (PP-TOP-BOX) := $s_root (PP-PRODUCTION) ;
  $op (PP-TOP-BOX) := $op-prod (PP-PRODUCTION) ;
end where ;

{- ----------------------------------------------------------------- -}
{- PP-PRODUCTION                                                     -}
{- ----------------------------------------------------------------- -}
where null-PP-PRODUCTION ->  use
  $op-prod := error-op ;
  $s_root := false ;
end where ;

where pp-fixed-prod ->  PP-ID  PP-PROD-RHS  declare
value
  op : op_info := lookup_OP(
                    info_name_qua(
                      $name_module (PP-HOOK.up) ,
                      $name (PP-ID) ) ,
                    $OP (PP-HOOK.up) ) ;
  ph_info : ph_info := head(op.info) ;
use
  $correct := $correct (PP-ID) & $correct (PP-PROD-RHS) and
              if op ~ error-op then
                message message_olga (38)
                position pp-fixed-prod
                severity 2
                value false
              elsif test_op_decl(op , $liste_ph(PP-PROD-RHS) , fixed_op) then
                true
              else
                message message_olga (79)
                position pp-fixed-prod
                severity 1
                value true
              end if ;
  $s_root := ph_info.root ;
  $op-prod := op ;
end where ;

where pp-pe-list-prod ->  PP-ID  PP-PHYLUM  declare
value
  op : op_info := lookup_OP(
                    info_name_qua(
                      $name_module (PP-HOOK.up) ,
                      $name (PP-ID) ) ,
                    $OP (PP-HOOK.up) ) ;
  ph_info : ph_info := head(op.info) ;
```

```
use
  $correct := $correct (PP-ID) & $correct (PP-PHYLUM) and
              if op ~ error-op then
                message message_olga (38)
                position pp-pe-list-prod
                severity 2
                value false
              elsif test_op_decl(op , liste_ph($ph (PP-PHYLUM)) , pe_list_op) then
                true
              else
                message message_olga (79)
                position pp-pe-list-prod
                severity 1
                value true
              end if ;
  $s_root := ph_info.root ;
  $op-prod := op ;
end where ;

where pp-ne-list-prod -> PP-ID  PP-PHYLUM  declare
value
  op : op_info := lookup_OP(
                    info_name_qua(
                      $name_module (PP-HOOK.up) ,
                      $name (PP-ID) ) ,
                    $OP (PP-HOOK.up) ) ;
  ph_info : ph_info := head(op.info) ;
use
  $correct := $correct (PP-ID) & $correct (PP-PHYLUM) and
              if op ~ error-op then
                message message_olga (38)
                position pp-ne-list-prod
                severity 2
                value false
              elsif test_op_decl(op , liste_ph($ph (PP-PHYLUM)) , ne_list_op) then
                true
              else
                message message_olga (79)
                position pp-ne-list-prod
                severity 1
                value true
              end if ;
  $s_root := ph_info.root ;
  $op-prod := op ;
end where ;

{- ------------------------------------------------------------------- -}
{- PP-PROD-RHS                                                         -}
{- ------------------------------------------------------------------- -}
where null-PP-PROD-RHS ->  use
  $liste_ph := liste_ph () ;
end where ;

where pp-prod-rhs ->  PP-PHYLUM * use
  $liste_ph := map left liste_ph
                value liste_ph ()
                other $ph (PP-PHYLUM)
               end map ;
end where ;

{- ------------------------------------------------------------------- -}
{- PP-TOP-BOX                                                          -}
{- ------------------------------------------------------------------- -}
where pp-top-box ->  PP-BOX declare
value
  dependance : bool := let bidon : bool := $h_root in true ;
```

```
use
  $correct := dependance & $correct(PP-BOX) ;
  $CVI (PP-BOX) := $MODULE_TYPE (PP-HOOK.up).CVI ;
  $is-top (PP-BOX) := true ;
end where ;

where pp-empty-box ->  use

end where ;


{- ------------------------------------------------------------------ -}
{- PP-BOX                                                              -}
{- ------------------------------------------------------------------ -}
where pp-pplist-box ->  PP-DIR  PP-SEP-ID  PP-INSERT  PP-BOX  declare
value
  correct : bool := (if (   ($op (PP-TOP-BOX.up).kind !~ pe_list_op)
                      &  ($op (PP-TOP-BOX.up).kind !~ ne_list_op)) then
                              message message_olga (91)
                              position pp-pplist-box
                              severity 2
                              value false
                     else
                        true
                     end if)
                    and
                    (if lookup-SEP($pp-sep-id (PP-SEP-ID) , $SEP (PP-HOOK.up)) then
                        true
                     else
                        message message_olga (92)
                        position PP-SEP-ID
                        severity 2
                        value false
                    end if ) ;
use
  $correct := correct
                & $correct (PP-BOX)
                & $correct (PP-INSERT)
                & $correct (PP-SEP-ID)
                & $correct (PP-DIR) ;
  $CVI (PP-BOX) := $CVI ;
  $is-top (PP-BOX) := false ;
end where ;

where pp-case-match -> PP-EXPR PP-MATCH-CLAUSES PP-OTHER-CLAUSE use
  $correct := if $is-top then
                $correct (PP-EXPR)
                & $correct (PP-MATCH-CLAUSES)
                & $correct (PP-OTHER-CLAUSE)
              else
                message message_olga (95)
                position pp-case-match
                severity 2
                value false
              end if ;
  $type_pattern (PP-MATCH-CLAUSES) := $type (PP-EXPR) ;
  $case_match (PP-MATCH-CLAUSES) := match_expr ;
  $type_pattern (PP-OTHER-CLAUSE) := $type (PP-EXPR) ;
  $case_match (PP-OTHER-CLAUSE) := match_expr ;
  $h.CVI (PP-MATCH-CLAUSES) := $CVI ;
  $h.CVI (PP-OTHER-CLAUSE) := $CVI ;
  $is-top (PP-MATCH-CLAUSES) := false ;
  $is-top (PP-OTHER-CLAUSE) := false ;
end where ;

where pp-case-tree ->  PP-SYMBOL  PP-MATCH-CLAUSES  PP-OTHER-CLAUSE  use
```

```
  { non encore implante , a remplacer par idem pp-case-match quand case-tree implante }
  $correct := message message_olga (0)
              position pp-case-tree
              severity 2
              value $correct (PP-MATCH-CLAUSES) & $correct (PP-OTHER-CLAUSE) & false ;
  $type_pattern (PP-MATCH-CLAUSES) := $type (PP-SYMBOL) ;
  $case_match (PP-MATCH-CLAUSES) := tree_expr ;
  $type_pattern (PP-OTHER-CLAUSE) := $type (PP-SYMBOL) ;
  $case_match (PP-OTHER-CLAUSE) := tree_expr ;
  $h.CVI (PP-MATCH-CLAUSES) := $CVI ;
  $h.CVI (PP-OTHER-CLAUSE) := $CVI ;
end where ;

where pp-compound-box ->  PP-SEP-ID  PP-BOXES  use
  $correct := if lookup-SEP ($pp-sep-id (PP-SEP-ID) , $SEP (PP-HOOK.up)) then
                $correct (PP-SEP-ID) & $correct (PP-BOXES)
              else
                message message_olga (92)
                position PP-SEP-ID
                severity 2
                value false
              end if ;
end where ;

where pp-conditional ->  PP-EXPR  PP-BOX  PP-ELSIF-PART  PP-BOX use
  $correct := if $correct (PP-EXPR.1)
                 & $correct(PP-BOX.1)
                 & $correct (PP-ELSIF-PART)
                 & $correct(PP-BOX.2) then
                        test_type_bool(
                          $scx,
                          $type (PP-EXPR.1),
                          $MODULE_TYPE (PP-HOOK.up).TYPE)
                else
                        false
                end if ;
  $CVI (PP-BOX.1) := $CVI ;
  $CVI (PP-BOX.2) := $CVI ;
  $is-top (PP-BOX.1) := false ;
  $is-top (PP-BOX.2) := false ;
end where ;

where pp-rec-call ->  PP-SYMBOL  PP-CONTEXTS  use
  $h_kind_phylum (PP-CONTEXTS) := $kind_phylum (PP-SYMBOL) ;
  $h_ph (PP-CONTEXTS) := $ph (PP-SYMBOL) ;
  $h_pos (PP-CONTEXTS) := $pos (PP-SYMBOL) ;
end where ;

where pp-attrib ->  PP-ATTRIB-OCC  use

end where ;

where pp-string-box ->  PP-STRING  use

end where ;

{- ------------------------------------------------------------------- -}
{- PP-CONTEXTS                                                          -}
{- ------------------------------------------------------------------- -}
  { regles generees par defaut }
{- ------------------------------------------------------------------- -}
{- PP-CONTEXT                                                           -}
{- ------------------------------------------------------------------- -}
where pp-context ->  PP-ATTR  PP-EXPR  declare
value
  nat_ph : rech_nat_ph_type :=
```

```
        if $op (PP-TOP-BOX.up) ~ error-op then
          rech_nat_ph_type (
            $nat (PP-ATTR) ,
            $NAT (PP-HOOK.up))
        else
          rech_nat_ph(
            $correct (PP-ATTR) ,
            $h_kind_phylum ,
            $name_id (PP-ATTR) ,
            $nat (PP-ATTR) ,
            $h_ph ,
            $op (PP-TOP-BOX.up) ,
            $h_pos ,
            false , 1 ,
            $scx ,
            $PH_OP (PP-HOOK.up) ,
            $NAT (PP-HOOK.up) )
        end if ;
  correct := if nat_ph.nat ~ error-nat-olga then
               false
            else
               true
            end if ;
use
  $correct := $correct (PP-ATTR) & $correct (PP-EXPR) & correct
             & (test_simple_rule (
                 $scx ,
                 rech_nat_type(
                   nat_ph.nat ,
                   $MODULE_TYPE (PP-HOOK.up).TYPE ,
                   $scx) ,
                 $type(PP-EXPR)) !~ error-type-info);
end where ;

{- ------------------------------------------------------------------- -}
{- PP-ELSIF-PART                                                        -}
{- ------------------------------------------------------------------- -}
   { regles generees par defaut }
{- ------------------------------------------------------------------- -}
{- PP-ELSIF-CLAUSE                                                      -}
{- ------------------------------------------------------------------- -}
where pp-elsif-clause ->  PP-EXPR  PP-BOX  use
  $correct := if $correct (PP-EXPR) & $correct (PP-BOX) then
               test_type_bool ($scx, $type (PP-EXPR), $MODULE_TYPE (PP-HOOK.up).TYPE)
             else
               false
             end if ;
  $CVI (PP-BOX) := $CVI (PP-BOX.up) ;
  $is-top (PP-BOX) := false ;
end where ;

{- ------------------------------------------------------------------- -}
{- PP-BOXES                                                             -}
{- ------------------------------------------------------------------- -}
where pp-boxes ->  PP-BOX * use
  $CVI (PP-BOX) := $CVI (PP-BOX.up) ;
  $is-top (PP-BOX) := false ;
end where ;

{- ------------------------------------------------------------------- -}
{- PP-MATCH-CLAUSES                                                     -}
{- ------------------------------------------------------------------- -}
where pp-match-clauses ->  PP-MATCH-CLAUSE + use
  $case_match (PP-MATCH-CLAUSE) := $case_match ;
  $h.CVI (PP-MATCH-CLAUSE) := $h.CVI ;
end where ;
```

```
{- ------------------------------------------------------------------- -}
{- PP-MATCH-CLAUSE                                                     -}
{- ------------------------------------------------------------------- -}
where pp-match-clause -> PP-PATTERNS PP-BOX use
  $CVI (PP-BOX) := $s.CVI (PP-PATTERNS) ;
  $is-top (PP-BOX) := false ;
end where ;

{- ------------------------------------------------------------------- -}
{- PP-PATTERNS                                                         -}
{- ------------------------------------------------------------------- -}
where null-PP-PATTERNS -> use
  $actual := liste_type () ;
  $liste_id := liste_id () ;
end where ;

where pp-patterns -> PP-PATTERN * use
  $actual := map left +
               value liste_type ()
               other $actual (PP-PATTERN)
            end map ;
  $liste_id := map left +
               value liste_id ()
               other $liste_id (PP-PATTERN)
              end map ;
  $type := map left test_type_egal_type
               value $type_pattern
               other $type (PP-PATTERN) , $scx , 9
          end map ;
end where ;

{- ------------------------------------------------------------------- -}
{- PP-PATTERN                                                          -}
{- ------------------------------------------------------------------- -}
where null-PP-PATTERN -> use
  $actual := liste_type () ;
  $liste_id := liste_id () ;
end where ;

where pp-pattern-var-or-cons -> PP-ID declare
value
  cvi_info : cvi_info := lookup_CVI (
                           info_name_qua($name_module(PP-HOOK.up) , $name(PP-ID)) ,
                           $h.CVI ) ;
use
  $correct := $correct (PP-ID)
              & if (cvi_info ~ error-cvi-info) then
                  message message_olga (11)
                  position pp-pattern-var-or-cons
                  severity 2
                  value false
                else
                  true
                end if ;
  $actual := liste_type (rech_cvi_type(cvi_info)) ;
  $liste_id := liste_id ($name(PP-ID)) ;
  $type := rech_cvi_type (cvi_info) ;
end where ;

where pp-pattern-cons -> PP-ID PP-PATTERNS declare
value
  func : func_info := lookup_FUNC (
                        info_name_qua(
                          $name_module (PP-HOOK.up) ,
                          $name (PP-ID) ) ,
```

```
                        $actual(PP-PATTERNS) ,
                        $MODULE_TYPE(PP-HOOK.up).FUNC) ;
  type_func : type_olga := rech_func_type(func) ;
use
  $correct := if (rech_info_func(info_info(func)) ~ error-func-info) then
                message message_olga(10)
                position PP-ID
                severity 2
                value false
              else
                true
              end if & $correct (PP-PATTERNS) & $correct (PP-ID) ;
  $type_pattern (PP-PATTERNS) := type-info-indefini ;
  $type := test_type_egal_type (type_func , $type_pattern , $scx , 8) ;
  $s.CVI := if (func ~ error-func-info) then
              $h.CVI
            else
              entree_ident_patterns (
                $scx ,
                $name_module(PP-HOOK.up) ,
                $liste_id (PP-PATTERNS) ,
                func.info.liste_formal ,
                $s.CVI(PP-PATTERNS) )
            end if ;
  $liste_id := liste_id (error-info-name) ;
  $actual := liste_type (type_func) ;
end where ;

where pp-pattern-const-manif -> PP-EXPR use
  $actual := liste_type($type(PP-EXPR)) ;
  $liste_id := liste_id (error-info-name) ;
  $type := $type(PP-EXPR) ;
end where ;

where pp-single-case-const -> PP-EXPR use
  $actual := liste_type($type(PP-EXPR)) ;
  $liste_id := liste_id (error-info-name) ;
  $type := $type(PP-EXPR) ;
end where ;

{- ------------------------------------------------------------------- -}
{- PP-OTHER-CLAUSE                                                     -}
{- ------------------------------------------------------------------- -}
where pp-match-clause-other ->  PP-BOX  use
  $CVI (PP-BOX) := $h.CVI ;
  $is-top (PP-BOX) := false ;
end where ;

{- ------------------------------------------------------------------- -}
{- PP-DIR                                                              -}
{- ------------------------------------------------------------------- -}
  { regles generees par defaut}
{- ------------------------------------------------------------------- -}
{- PP-INSERT                                                           -}
{- ------------------------------------------------------------------- -}
  { regles generees par defaut}
{- ------------------------------------------------------------------- -}
{- PP-EXPR                                                             -}
{- ------------------------------------------------------------------- -}
where pp-applied-id -> PP-ID declare
value
  cvi_info : cvi_info := lookup_CVI (
                          info_name_qua($name_module(PP-HOOK.up) , $name(PP-ID)) ,
                          $CVI (PP-BOX.up)) ;
use
  $correct := if cvi_info ~ error-cvi-info
```

```
              then
                message message_olga(6)
                position pp-applied-id
                severity 2
                value false
             else
                true
             end if ;
  $type := rech_cvi_type (cvi_info) ;
end where ;

where pp-func-call ->  PP-ID  PP-ACTUALS  declare
value
  func_info : func_info := if $correct (PP-ACTUALS) & $correct (PP-ID) then
                            lookup_FUNC (
                              info_name_qua (
                                $name_module (PP-HOOK.up) ,
                                $name (PP-ID) ) ,
                              $actual (PP-ACTUALS) ,
                              $MODULE_TYPE (PP-HOOK.up).FUNC)
                           else
                            error-func-info
                           end if ;
use
  $correct := if $correct (PP-ACTUALS) & $correct(PP-ID) then
                if rech_info_func(info_info(func_info)) ~ error-func-info  then
                  message message_olga(7)
                  position PP-ID
                  severity 2
                  value false
                else
                  true
                end if
              else
                false
              end if ;
  $type :=  rech_func_type (func_info) ;
end where ;

where pp-bool-or ->  PP-EXPR  PP-EXPR  use
  $type := $type (PP-EXPR.1) ;
  $correct := $correct (PP-EXPR.1)
              & $correct (PP-EXPR.2)
              & test_type_bool($scx, $type(PP-EXPR.1), $MODULE_TYPE(PP-HOOK.up).TYPE)
              & test_type_bool($scx, $type(PP-EXPR.2), $MODULE_TYPE(PP-HOOK.up).TYPE) ;
end where ;

where pp-bool-and ->  PP-EXPR  PP-EXPR  use
  $type := $type (PP-EXPR.1) ;
  $correct := $correct (PP-EXPR.1)
              & $correct (PP-EXPR.2)
              & test_type_bool($scx, $type(PP-EXPR.1), $MODULE_TYPE(PP-HOOK.up).TYPE)
              & test_type_bool($scx, $type(PP-EXPR.2), $MODULE_TYPE(PP-HOOK.up).TYPE) ;
end where ;

{- --------------------------------------------------------------------- -}
{- PP-ACTUALS                                                             -}
{- --------------------------------------------------------------------- -}
where null-PP-ACTUALS ->  use
  $actual := liste_type () ;
end where ;

where pp-actuals ->  PP-ACTUAL * use
  $actual := map left +
                value liste_type ()
                other $actual (PP-ACTUAL)
```

```
             end map ;
end where ;

{- ------------------------------------------------------------------ -}
{- PP-ACTUAL                                                           -}
{- ------------------------------------------------------------------ -}
where null-PP-ACTUAL ->  use
  $actual := liste_type () ;
end where ;

where pp-actual ->  PP-EXPR  use
  $type := $type (PP-EXPR) ;
  $actual := liste_type ($type (PP-EXPR)) ;
end where ;

{- ------------------------------------------------------------------ -}
{- PP-ATTRIB-OCC                                                       -}
{- ------------------------------------------------------------------ -}
where pp-attrib-occ ->  PP-ATTR  PP-SYMBOL  declare
value
  nat : nat_olga :=
        let
          nat_ph : rech_nat_ph_type :=
                if $op (PP-TOP-BOX.up) ~ error-op then
                  rech_nat_ph_type (
                    $nat (PP-ATTR) ,
                    $NAT (PP-HOOK.up))
                else
                  rech_nat_ph(
                    $correct (PP-ATTR) ,
                    $kind_phylum (PP-SYMBOL) ,
                    $name_id (PP-ATTR) ,
                    $nat (PP-ATTR) ,
                    $ph (PP-SYMBOL) ,
                    $op (PP-TOP-BOX.up) ,
                    $pos (PP-SYMBOL) ,
                    false , 1 ,
                    $scx ,
                    $PH_OP (PP-HOOK.up) ,
                    $NAT (PP-HOOK.up) )
                end if
        in
          nat_ph.nat ;
use
  $type := rech_nat_type(nat , $MODULE_TYPE (PP-HOOK.up) .TYPE , $scx ) ;
  $correct := if nat ~ error-nat-olga then
                false
              else
                $correct (PP-ATTR) & $correct (PP-SYMBOL)
              end if ;
end where ;

where pp-attrib-occ-lhs ->  PP-ATTR  declare
value
  nat : nat_olga :=
        let
          nat_ph : rech_nat_ph_type :=
                if $op (PP-TOP-BOX.up) ~ error-op then
                  rech_nat_ph_type(
                    $nat (PP-ATTR) ,
                    $NAT (PP-HOOK.up))
                else
                  rech_nat_ph(
                    $correct (PP-ATTR) ,
                    NORMAL ,
                    $name_id (PP-ATTR) ,
```

```
                  $nat (PP-ATTR) ,
                  head ($op (PP-TOP-BOX.up) .info) ,
                  $op (PP-TOP-BOX.up) ,
                  0 ,
                  false , 1 ,
                  $scx ,
                  $PH_OP (PP-HOOK.up) ,
                  $NAT (PP-HOOK.up) )
              end if
        in
          nat_ph.nat ;
use
  $type := rech_nat_type(nat , $MODULE_TYPE (PP-HOOK.up) .TYPE , $scx ) ;
  $correct := if nat ~ error-nat-olga then
              false
             else
              $correct (PP-ATTR)
             end if ;
end where ;

{- ------------------------------------------------------------------- -}
{- PP-SYMBOL                                                           -}
{- ------------------------------------------------------------------- -}
where null-PP-SYMBOL ->  use
  $pos := 0 ;
  $name_id := error-info-name-qua;
  $ph := error-ph;
  $kind_phylum := UNQUAL_SYMBOL;
end where ;

where pp-qual-symb ->  PP-ID  PP-QUALIFIER  declare
value
  name : info_name_qua := info_name_qua ($name_module(PP-HOOK.up) , $name(PP-ID) ) ;
  ph : ph_info := lookup_PH(name,$PH(PP-HOOK.up)) ;
use
  $type := rech_ph_type(ph , $MODULE_TYPE (PP-HOOK.up).TYPE) ;
  $name_id := name;
  $ph := if $kind_phylum(PP-QUALIFIER) = UP then
              ph
         else
              rech_ph_in_op(
                $op,ph,
                $pos(PP-QUALIFIER),
                $kind_phylum(PP-QUALIFIER),
                $scx)
        end if;
  $kind_phylum := $kind_phylum(PP-QUALIFIER);
  $pos := $pos(PP-QUALIFIER);
end where ;

where pp-unqual-symb ->  PP-ID  declare
value
  name : info_name_qua := info_name_qua ($name_module(PP-HOOK.up) , $name(PP-ID) ) ;
  symbol : type_unqual_symbol := rech_unqual_symbol(
                                   $op (PP-TOP-BOX.up) ,
                                   name ,
                                   $MODULE_TYPE (PP-HOOK.up) .TYPE ,
                                   $PH (PP-HOOK.up) ,
                                   $scx ) ;
use
  $correct := $correct (PP-ID) & symbol.correct ;
  $pos := symbol.pos ;
  $kind_phylum := symbol.kind ;
  $ph := symbol.ph ;
  $type := symbol.type_olga ;
  $name_id := name ;
```

```
end where ;

{- ------------------------------------------------------------------ -}
{- PP-QUALIFIER                                                        -}
{- ------------------------------------------------------------------ -}
where null-PP-QUALIFIER ->  use
  $kind_phylum := INT_QUAL ;
  $pos := 0;
end where ;

where pp-last-qual ->  use
  $pos := 1 ;
  $kind_phylum := LAST ;
end where ;

where pp-first-qual ->  use
  $pos := 1 ;
  $kind_phylum := FIRST ;
end where ;

where pp-right-qual ->  use
  $pos := 1 ;
  $kind_phylum := F2RIGHT ;
end where ;

where pp-left-qual ->  use
  $pos := 1 ;
  $kind_phylum := F2LEFT ;
end where ;

where pp-int-qual ->  PP-INT  use
  $pos := int (string ($pp-int (PP-INT))) ;
  $kind_phylum := INT_QUAL ;
end where ;

{- ------------------------------------------------------------------ -}
{- PP-ID                                                               -}
{- ------------------------------------------------------------------ -}
where null-PP-ID ->  use
  $name := info_name ( error-token , error_kind_id ) ;
end where ;

where pp-op-id ->  use
  $name := info_name($pp-id , id_operator) ;
end where ;

where pp-id ->  use
  $name := info_name($pp-id , id_ident ) ;
end where ;

{- ------------------------------------------------------------------ -}
{- PP-SEP-ID                                                           -}
{- ------------------------------------------------------------------ -}
  { regles generees par defaut }
{- ------------------------------------------------------------------ -}
{- PP-INT                                                              -}
{- ------------------------------------------------------------------ -}
where pp-int ->  use
  $type := type_olga_int ($MODULE_TYPE (PP-HOOK.up).TYPE) ;
end where ;

{- ------------------------------------------------------------------ -}
{- PP-CHAR                                                             -}
{- ------------------------------------------------------------------ -}
where pp-char ->  use
  $type := type_olga_char ($MODULE_TYPE (PP-HOOK.up).TYPE) ;
```

```
end where ;

{- ------------------------------------------------------------------ -}
{- PP-REAL                                                             -}
{- ------------------------------------------------------------------ -}
where pp-real ->  use
  $type := type_olga_real ($MODULE_TYPE (PP-HOOK.up).TYPE) ;
end where ;

{- ------------------------------------------------------------------ -}
{- PP-STRING                                                           -}
{- ------------------------------------------------------------------ -}
where pp-string ->  use
  $type := type_olga_string ($MODULE_TYPE (PP-HOOK.up).TYPE) ;
end where ;

{- ------------------------------------------------------------------ -}
{- PP-PHYLUM                                                           -}
{- ------------------------------------------------------------------ -}
where null-PP-PHYLUM ->  use
  $ph := error-ph ;
end where ;

where pp-phylum ->  declare
value
  correct : bool := name_correct ($scx , $pp-phylum) ;
  ph : ph_info := if correct then
                        lookup_PH(
                          info_name_qua(
                            $name_module(PP-HOOK.up) ,
                            info_name($pp-phylum , id_ident) ) ,
                          $PH (PP-HOOK.up) )
                  else
                        error-ph
                  end if ;
use
  $correct := correct and (if ph ~ error-ph then
                                message message_olga(36)
                                position pp-phylum
                                severity 2
                                value false
                           else
                                true
                           end if) ;
  $ph := ph ;

end where ;

{- ------------------------------------------------------------------ -}
{- PP-ATTR                                                             -}
{- ------------------------------------------------------------------ -}
where null-PP-ATTR ->  use
  $nat := error-nat-olga ;
  $name_id := error-info-name-qua ;
end where ;

where pp-attr ->  declare
value
  correct : bool := name_correct ($scx , $pp-attr) ;
  name : info_name_qua := info_name_qua(
                            $name_module(PP-HOOK.up),
                            info_name($pp-attr , id_attribut)) ;
  nat_info : nat_olga := lookup_NAT_OLGA (name , $NAT (PP-HOOK.up) ) ;
use
  $correct := correct ;
  $nat := if correct & (nat_info !~ error-nat-olga) then
```

```
          nat_info
        else
          error-nat-olga
        end if ;
  $name_id := name ;
end where ;
end grammar ;
```

A.B-2 ag_ppat-olga.olga

```
{  -----------------------------------------------------
   -  Grammaire attribuee ppat-olga                      -
   -  prenant en entree                                  -
   -        * une specif ppat                            -
   -  donnant en sortie:                                 -
   -        * une specif olga de transformation d'un arbre -
   -          en arbre de boites. (ATB)                  -
   ----------------------------------------------------- }

attribute grammar ppat-olga (ppat-out) : (olga-simpli)  is
  from ppat-types import all ;
{-----------------------------------------------}
{            declaration des types utilises pour}
{        representer les regles semantiques     }
{-----------------------------------------------}
type
  kind-qual = ( no-qual , dernier , premier , droit , gauche , entier ) ;
and
  t-symb = record
                symbol     : token ;
                kind-qual  : kind-qual ;
                qualifier  : token ;
           end record ;
and
  rep-occ = record
                attribut   : token ;
                symbol     : t-symb ;
               end record ;
and
  sem-rule = record
                rep-occ    : rep-occ ;
                occurence  : ATTRIB-OCC ;
                expression : EXPR ;
             end record ;
and
  l-sem-rule = list of sem-rule ;
and
  ei-part = record
                rep-occ     : rep-occ ;
                occurence   : ATTRIB-OCC ;
                part        : ELSIF-PART ;
             end record ;
and
  l-ei-part = list of ei-part ;
and
  ei-clause = record
                rep-occ     : rep-occ ;
                occurence   : ATTRIB-OCC ;
                clause      : ELSIF-CLAUSE ;
              end record ;
and
  l-ei-clause = list of ei-clause ;
and
  m-clause = record
                rep-occ     : rep-occ ;
```

```
                occurence   : ATTRIB-OCC ;
                clause      : MATCH-CLAUSE ;
             end record ;
and
  l-m-clause = list of m-clause ;
and
  m-clauses = record
                rep-occ      : rep-occ ;
                occurence    : ATTRIB-OCC ;
                clauses      : MATCH-CLAUSES ;
              end record ;
and
  l-m-clauses = list of m-clauses ;

const
  error-t-symb := t-symb (error-token , no-qual , error-token) ;
  error-rep-occ := rep-occ(error-token , error-t-symb) ;
  error-sem-rule := sem-rule(error-rep-occ , null-ATTRIB-OCC () , null-EXPR ()) ;

{-------------------------------------------------------------------------}
{ DEFINITION DES FONCTIONS                                                }
{-------------------------------------------------------------------------}

{-------------------------------------------------------------------------}
{ Fonction rec-sem-rules-creation                                         }
{-------------------------------------------------------------------------}
function sem-rules-creation(
                L-Sem-Rules : l-sem-rule ;
                Result : SEM-RULES ) : SEM-RULES is
{
  Fonction transformant une liste de representations de regles
  semantiques en une liste de regles semantiques OLGA .
}
  case L-Sem-Rules is
    l-sem-rule ( sem-rule ( Rep-Occ , ATO , E ) , LSR-T : l-sem-rule) :
    { Si plusieurs regles semantiques }
      sem-rules-creation(
        LSR-T , { L-Sem-Rules Tail }
        sem-rules-post(
          Result ,
          simple-rule(ATO , E))) ;
    l-sem-rule ( sem-rule ( Rep-Occ , ATO , E )) :
    { regle unique }
      sem-rules-post(Result , simple-rule(ATO , E)) ;
    other :
      Result ;
  end case
end function { sem-rules-creation } ;

{-------------------------------------------------------------------------}
{ Fonction lookup-ATO                                                     }
{-------------------------------------------------------------------------}
function lookup-ATO (
                rep-OCC : rep-occ ;
                OCC : ATTRIB-OCC ;
                L-Sem-Rules : l-sem-rule ;
                Mock-Exp : EXPR) : sem-rule is
{
  Fonction recherchant parmi L-Sem-Rules la regle correspondant a OCC
  et retournant une regle d'expression "bidon" Mock-Exp en cas d'echec .
}
  case L-Sem-Rules is
    l-sem-rule ( sem-rule ( rep-ATO , ATO , E ) , LSR-T : l-sem-rule) :
      if  rep-ATO = rep-OCC then
        sem-rule (rep-ATO , ATO , E)
        else
```

```
        lookup-ATO (rep-OCC , OCC ,LSR-T , Mock-Exp)
      end if  ;
    l-sem-rule ( sem-rule (rep-ATO , ATO , E)) :
      if rep-ATO = rep-OCC then
        sem-rule (rep-ATO , ATO , E)
      else
        sem-rule (rep-OCC , OCC , Mock-Exp )
      end if ;
    other :
      sem-rule (rep-OCC , OCC , Mock-Exp ) ;
  end case
end function { lookup-ATO } ;

{--------------------------------------------------------------------}
{ Fonction lookup-ato                                                }
{--------------------------------------------------------------------}
function lookup-ato (
                OCC : rep-occ ;
                LATO : l-sem-rule ) : bool is
{
  Fonction retournant true si OCC appartient a LATO et false autrement.
}
  case LATO is
    l-sem-rule (sem-rule (rep-ATO , ATO , Exp) , LATO-T : l-sem-rule) :
      if rep-ATO = OCC then
        true
      else
        lookup-ato (OCC , LATO-T)
      end if ;
     l-sem-rule (sem-rule (rep-ATO , ATO , Exp)) :
      if rep-ATO = OCC then
        true
      else
        false
      end if ;
    other :
      false ;
  end case
end function { lookup-ato } ;

{--------------------------------------------------------------------}
{ Fonction construct-ei-clause                                       }
{--------------------------------------------------------------------}
function construct-ei-clause (
                L-ATO : l-sem-rule ;
                Condition : EXPR ;
                LSR : l-sem-rule ;
                Result : l-ei-clause) : l-ei-clause is
{
  Fonction construisant une representation d'une elsif-clause
  pour toutes les occurences d'attribut de L-ATO .
}
  { parcourt de la liste des occurences d'attribut }
  case L-ATO is
    l-sem-rule (sem-rule (rep-ATO , ATO , Exp) , L-ATO-T : l-sem-rule) :
    { au moins deux occurences d'attribut }
      let
        rule : sem-rule := lookup-ATO (rep-ATO , ATO , LSR , Exp)
      in
        construct-ei-clause(
          L-ATO-T ,
          Condition ,
          LSR ,
          l-ei-clause(
            Result ,
            ei-clause (
```

```
              rep-ATO ,
              rule.occurence ,
              elsif-clause (Condition , rule.expression )))) ;
    l-sem-rule(sem-rule (rep-ATO , ATO , Exp)) :
    { une seule occurence d'attribut }
      let
        rule : sem-rule := lookup-ATO (rep-ATO , ATO , LSR , Exp)
      in
        l-ei-clause(
          Result ,
          ei-clause (
            rep-ATO ,
            rule.occurence ,
            elsif-clause (Condition , rule.expression ))) ;
    other :
    { pas d'occurence }
      Result ;
  end case
end function { construct-ei-clause } ;

{--------------------------------------------------------------------}
{ Fonction add-attr-occ                                              }
{--------------------------------------------------------------------}
function add-attr-occ (L-ATO , LSR : l-sem-rule ) :  l-sem-rule is
{
  Fonction rajoutant a L-ATO les occurences de LSR.
}
  case LSR is
    l-sem-rule (sem-rule (rep-ATO , ATO , Exp) , LSR-T : l-sem-rule) :
    { au moins deux occurences }
      if lookup-ato (rep-ATO , L-ATO) then
      { occurence deja representee }
        add-attr-occ (L-ATO , LSR-T)
      else
        add-attr-occ (l-sem-rule(L-ATO , sem-rule (rep-ATO , ATO , Exp )) , LSR-T)
      end if ;
    l-sem-rule (sem-rule (rep-ATO , ATO , Exp)) :
      if lookup-ato (rep-ATO , L-ATO) then
        L-ATO
      else
        l-sem-rule(L-ATO , sem-rule (rep-ATO , ATO , Exp ))
      end if ;
    other :
      L-ATO ;
  end case
end function { add-attr-occ } ;

{--------------------------------------------------------------------}
{ Fonction add-ei-clause                                             }
{--------------------------------------------------------------------}
function add-ei-clause (
               rep-OCC : rep-occ ;
               OCC : ATTRIB-OCC ;
               EIC : ELSIF-CLAUSE ;
               LEIP , Result : l-ei-part ) : l-ei-part is
  case LEIP is
    l-ei-part (ei-part (rep-ATO , ATO , EIP) , LEIP-T : l-ei-part) :
      if rep-ATO = rep-OCC then
        add-ei-clause(
          rep-OCC ,
          OCC ,
          EIC ,
          LEIP-T ,
          l-ei-part(Result , ei-part(rep-ATO , ATO , elsif-part-post (EIP , EIC))) )
      else
        add-ei-clause(
```

```
        rep-OCC ,
        OCC ,
        EIC ,
        LEIP-T ,
        l-ei-part(Result , ei-part(rep-ATO , ATO , EIP)) )
    end if ;
  l-ei-part (ei-part (rep-ATO , ATO , EIP)) :
    if rep-ATO = rep-OCC then
      l-ei-part(Result , ei-part(rep-ATO , ATO , elsif-part-post (EIP , EIC)))
    else
      add-ei-clause(
        rep-OCC ,
        OCC ,
        EIC ,
        l-ei-part () ,
        l-ei-part(Result , ei-part(rep-ATO , ATO , EIP)))
    end if ;
  other :
    l-ei-part(Result , ei-part(rep-OCC , OCC , elsif-part(EIC))) ;
 end case
end function { add-ei-clause } ;

{---------------------------------------------------------------------}
{ Fonction construct-ei-part                                          }
{---------------------------------------------------------------------}
function construct-ei-part (
                LEIP : l-ei-part ;
                LEIC : l-ei-clause ) : l-ei-part is
  case LEIC is
    l-ei-clause (ei-clause (rep-OCC , OCC , EIC) , LEIC-T : l-ei-clause) :
      construct-ei-part(
        add-ei-clause(rep-OCC , OCC , EIC , LEIP , l-ei-part ()) ,
        LEIC-T) ;
    l-ei-clause (ei-clause (rep-OCC , OCC , EIC)) :
      add-ei-clause(rep-OCC , OCC , EIC , LEIP , l-ei-part ()) ;
    other :
      LEIP ;
  end case
end function { construct-ei-part } ;

{---------------------------------------------------------------------}
{ Fonction lookup-eip                                                 }
{---------------------------------------------------------------------}
function lookup-eip (
                rep-ATO : rep-occ ;
                LEIP : l-ei-part ) : ELSIF-PART is
  case LEIP is
    l-ei-part (ei-part (rep-OCC , OCC , EIP) , LEIP-T : l-ei-part) :
      if rep-OCC = rep-ATO then
        EIP
      else
        lookup-eip (rep-ATO , LEIP-T)
      end if ;
    l-ei-part (ei-part (rep-OCC , OCC , EIP)) :
      if rep-OCC = rep-ATO then
        EIP
      else
        null-ELSIF-PART ()
      end if ;
    other :
      null-ELSIF-PART () ;
  end case
end function { lookup-eip } ;

{---------------------------------------------------------------------}
{ Fonction construct-conditional                                      }
```

```
{-------------------------------------------------------------------------}
function construct-conditional(
                LATO , LSR1 , LSR2 , Result : l-sem-rule ;
                LEIP : l-ei-part ;
                Condition : EXPR ) : l-sem-rule is
  case LATO is
    l-sem-rule (sem-rule (rep-ATO , ATO , Exp) , LATO-T : l-sem-rule) :
      let
        rule1 : sem-rule := lookup-ATO(rep-ATO , ATO , LSR1 , Exp) ,
        rule2 : sem-rule := lookup-ATO(rep-ATO , ATO , LSR2 , Exp) ,
        eip : ELSIF-PART := lookup-eip(rep-ATO , LEIP)
      in
        construct-conditional(
          LATO-T ,
          LSR1 ,
          LSR2 ,
          l-sem-rule(
            Result ,
            sem-rule(
              rep-ATO ,
              ATO ,
              conditional(
                Condition ,
                rule1.expression ,
                eip ,
                rule2.expression))) ,
          LEIP , Condition ) ;
    l-sem-rule (sem-rule (rep-ATO , ATO , Exp)) :
      let
        rule1 : sem-rule := lookup-ATO(rep-ATO , ATO , LSR1 , Exp) ,
        rule2 : sem-rule := lookup-ATO(rep-ATO , ATO , LSR2 , Exp) ,
        eip : ELSIF-PART := lookup-eip(rep-ATO , LEIP)
      in
        l-sem-rule(
          Result ,
          sem-rule(
            rep-ATO ,
            ATO ,
            conditional(
              Condition ,
              rule1.expression ,
              eip ,
              rule2.expression))) ;
    other :
      Result ;
  end case
end function { construct-conditional } ;

{-------------------------------------------------------------------------}
{ Fonction construct-m-clause                                             }
{-------------------------------------------------------------------------}
function construct-m-clause (
                L-ATO : l-sem-rule ;
                Patterns : PATTERNS ;
                LSR : l-sem-rule ;
                Result : l-m-clause) : l-m-clause is
{
  Fonction construisant une representation d'une match-clause
  pour toutes les occurences d'attribut de L-ATO .
}
  { parcourt de la liste des occurences d'attribut }
  case L-ATO is
    l-sem-rule (sem-rule (rep-ATO , ATO , Exp) , L-ATO-T : l-sem-rule) :
    { au moins deux occurences d'attribut }
      let
        rule : sem-rule := lookup-ATO (rep-ATO , ATO , LSR , Exp)
```

```
      in
        construct-m-clause(
          L-ATO-T ,
          Patterns ,
          LSR ,
          l-m-clause(
            Result ,
            m-clause (
              rep-ATO ,
              rule.occurence ,
              match-clause (Patterns, block-expr(rule.expression) )))) ;
    l-sem-rule(sem-rule (rep-ATO , ATO , Exp)) :
    { une seule occurence d'attribut }
      let
        rule : sem-rule := lookup-ATO (rep-ATO , ATO , LSR , Exp)
      in
        l-m-clause(
          Result ,
          m-clause (
            rep-ATO ,
            rule.occurence ,
            match-clause (Patterns , block-expr(rule.expression) ))) ;
    other :
    { pas d'occurence }
      Result ;
  end case
end function { construct-m-clause } ;

{-------------------------------------------------------------------------}
{ Fonction construct-other-clause                                         }
{-------------------------------------------------------------------------}
function construct-other-clause (
                L-ATO : l-sem-rule ;
                LSR : l-sem-rule ;
                Result : l-m-clause) : l-m-clause is
{
  Fonction construisant une representation d'une match-clause-other
  pour toutes les occurences d'attribut de L-ATO .
}
  { parcourt de la liste des occurences d'attribut }
  case L-ATO is
    l-sem-rule (sem-rule (rep-ATO , ATO , Exp) , L-ATO-T : l-sem-rule) :
    { au moins deux occurences d'attribut }
      let
        rule : sem-rule := lookup-ATO (rep-ATO , ATO , LSR , Exp)
      in
        construct-other-clause(
          L-ATO-T ,
          LSR ,
          l-m-clause(
            Result ,
            m-clause (
              rep-ATO ,
              rule.occurence ,
              match-clause-other (rule.expression)))) ;
    l-sem-rule(sem-rule (rep-ATO , ATO , Exp)) :
    { une seule occurence d'attribut }
      let
        rule : sem-rule := lookup-ATO (rep-ATO , ATO , LSR , Exp)
      in
        l-m-clause(
          Result ,
          m-clause (
            rep-ATO ,
            rule.occurence ,
            match-clause-other (rule.expression) )) ;
```

```
    other :
    { pas d'occurence }
      Result ;
  end case
end function { construct-other-clause } ;

{--------------------------------------------------------------------}
{ Fonction add-m-clause                                              }
{--------------------------------------------------------------------}
function add-m-clause (
                rep-OCC : rep-occ ;
                OCC : ATTRIB-OCC ;
                MC : MATCH-CLAUSE ;
                LMCS , Result : l-m-clauses ) : l-m-clauses is
  case LMCS is
    l-m-clauses (m-clauses (rep-ATO , ATO , MCS) , LMCS-T : l-m-clauses) :
      if rep-ATO = rep-OCC then
        add-m-clause(
          rep-OCC ,
          OCC ,
          MC ,
          LMCS-T ,
          l-m-clauses(
            Result ,
            m-clauses(rep-ATO , ATO , match-clauses-post (MCS , MC))) )
      else
        add-m-clause(
          rep-OCC ,
          OCC ,
          MC ,
          LMCS-T ,
          l-m-clauses(Result , m-clauses(rep-ATO , ATO , MCS)) )
      end if ;
    l-m-clauses (m-clauses (rep-ATO , ATO , MCS)) :
      if rep-ATO = rep-OCC then
        l-m-clauses(Result , m-clauses(rep-ATO , ATO , match-clauses-post (MCS , MC)))
      else
        add-m-clause(
          rep-OCC ,
          OCC ,
          MC ,
          l-m-clauses () ,
          l-m-clauses(Result , m-clauses(rep-ATO , ATO , MCS)))
      end if ;
    other :
      l-m-clauses(Result , m-clauses(rep-OCC , OCC , match-clauses(MC))) ;
  end case
end function { add-m-clause } ;

{--------------------------------------------------------------------}
{ Fonction construct-m-clauses                                       }
{--------------------------------------------------------------------}
function construct-m-clauses (
                LMCS : l-m-clauses ;
                LMC : l-m-clause ) : l-m-clauses is
  case LMC is
    l-m-clause (m-clause (rep-OCC , OCC , MC) , LMC-T : l-m-clause) :
      construct-m-clauses(
        add-m-clause(rep-OCC , OCC , MC , LMCS , l-m-clauses ()) ,
        LMC-T) ;
    l-m-clause (m-clause (rep-OCC , OCC , MC)) :
      add-m-clause(rep-OCC , OCC , MC , LMCS , l-m-clauses ()) ;
    other :
      LMCS ;
  end case
end function { construct-m-clauses } ;
```

```
{----------------------------------------------------------------------}
{ Fonction lookup-mcs                                                  }
{----------------------------------------------------------------------}
function lookup-mcs (
                rep-ATO : rep-occ ;
                LMCS : l-m-clauses ) : MATCH-CLAUSES is
  case LMCS is
    l-m-clauses (m-clauses (rep-OCC , OCC , MCS) , LMCS-T : l-m-clauses) :
      if rep-OCC = rep-ATO then
        MCS
      else
        lookup-mcs (rep-ATO , LMCS-T)
      end if ;
    l-m-clauses (m-clauses (rep-OCC , OCC , MCS)) :
      if rep-OCC = rep-ATO then
        MCS
      else
        null-MATCH-CLAUSES ()
      end if ;
    other :
      null-MATCH-CLAUSES () ;
  end case
end function { lookup-mcs } ;

{----------------------------------------------------------------------}
{ Fonction lookup-mc                                                   }
{----------------------------------------------------------------------}
function lookup-mc (
                rep-ATO : rep-occ ;
                LMC : l-m-clause ) : MATCH-CLAUSE is
  case LMC is
    l-m-clause (m-clause (rep-OCC , OCC , MC) , LMC-T : l-m-clause) :
      if rep-OCC = rep-ATO then
        MC
      else
        lookup-mc (rep-ATO , LMC-T)
      end if ;
    l-m-clause (m-clause (rep-OCC , OCC , MC)) :
      if rep-OCC = rep-ATO then
        MC
      else
        null-MATCH-CLAUSE ()
      end if ;
    other :
      null-MATCH-CLAUSE () ;
  end case
end function { lookup-mc } ;

{----------------------------------------------------------------------}
{ Fonction construct-case-match                                        }
{----------------------------------------------------------------------}
function construct-case-match (
                LATO : l-sem-rule ;
                LMCS : l-m-clauses ;
                LMC : l-m-clause ;
                Result : l-sem-rule ;
                Selector : EXPR ) : l-sem-rule is
  case LATO is
    l-sem-rule (sem-rule (rep-ATO , ATO , Exp) , LATO-T : l-sem-rule) :
      let
        mcs : MATCH-CLAUSES := lookup-mcs(rep-ATO , LMCS) ,
        mc : MATCH-CLAUSE := lookup-mc(rep-ATO , LMC)
      in
        construct-case-match(
          LATO-T ,
```

```
        LMCS ,
        LMC ,
        l-sem-rule(
          Result ,
          sem-rule(
            rep-ATO ,
            ATO ,
            case-match(
              Selector ,
              mcs ,
              mc))) ,
        Selector) ;
    l-sem-rule (sem-rule (rep-ATO , ATO , Exp)) :
      let
        mcs : MATCH-CLAUSES := lookup-mcs(rep-ATO , LMCS) ,
        mc : MATCH-CLAUSE := lookup-mc(rep-ATO , LMC)
      in
        l-sem-rule(
            Result ,
            sem-rule(
              rep-ATO ,
              ATO ,
              case-match(
                Selector ,
                mcs ,
                mc))) ;
    other :
      Result ;
  end case
end function { construct-case-match } ;

{-----------------------------------------------------------------------}
{ Fonction construct-case-tree                                          }
{-----------------------------------------------------------------------}
function construct-case-tree (
                LATO : l-sem-rule ;
                LMCS : l-m-clauses ;
                LMC : l-m-clause ;
                Result : l-sem-rule ;
                Symbol : SYMBOL ) : l-sem-rule is
  case LATO is
    l-sem-rule (sem-rule (rep-ATO , ATO , Exp) , LATO-T : l-sem-rule) :
      let
        mcs : MATCH-CLAUSES := lookup-mcs(rep-ATO , LMCS) ,
        mc : MATCH-CLAUSE := lookup-mc(rep-ATO , LMC)
      in
        construct-case-tree(
          LATO-T ,
          LMCS ,
          LMC ,
          l-sem-rule(
            Result ,
            sem-rule(
              rep-ATO ,
              ATO ,
              case-tree(
                Symbol ,
                mcs ,
                mc))) ,
          Symbol) ;
    l-sem-rule (sem-rule (rep-ATO , ATO , Exp)) :
      let
        mcs : MATCH-CLAUSES := lookup-mcs(rep-ATO , LMCS) ,
        mc : MATCH-CLAUSE := lookup-mc(rep-ATO , LMC)
      in
        l-sem-rule(
```

```
          Result ,
          sem-rule(
            rep-ATO ,
            ATO ,
            case-tree(
              Symbol ,
              mcs ,
              mc))) ;
    other :
      Result ;
  end case
end function { construct-case-tree } ;

{---------------------------------------------------------------------}
{ Fonction lst-expr-box_TO_expr-lst-box                               }
{---------------------------------------------------------------------}
function lst-expr-box_TO_expr-lst-box(
                boxes-post : ID-TER ;
                LOA : ACTUALS ;
                Result : EXPR) : EXPR is
{ fonction transformant une liste d'expression OLGA de type BOX
  en une expression OLGA de type BOXES }
  case LOA is
    actuals-pre(A : ACTUAL , LOA-T : ACTUALS) :
      lst-expr-box_TO_expr-lst-box(
        boxes-post ,
        LOA-T ,
        func-call(
          boxes-post ,
          actuals-pre(
            actual(Result) , actuals (A)))) ;
    actuals(A : ACTUAL) :
      func-call(
        boxes-post ,
        actuals-pre(
          actual(Result) , actuals (A))) ;
    other :
      Result ;
  end case
end function ;

{---------------------------------------------------------------------}
{ Fonction creer-def-fonc-predef-type                                 }
{---------------------------------------------------------------------}
function creer-def-fonc-predef-type (
                type-elt : string ;
                expression : EXPR) : FUNC-DEF is
{ fonction creant les fonctions de decompilation des types predefinis }
  func-def(
        func-header(
          offline-mode () ,
          id-ter () with $id := token("ppat_affichage_" + type-elt) end with ,
          formals-pre(
            formal(
              id-ter () with $id := token("object") end with ,
              type-id(
                unqual-id(
                  id-ter () with $id := token(type-elt) end with ))) ,
            formals(
              formal(
                id-ter () with $id := token("Result") end with ,
                type-id(
                  unqual-id(
                    id-ter () with $id := token("BOXES") end with ))))) ,
          type-id(
            unqual-id(
```

```
              id-ter () with $id := token("BOX") end with )) ,
            excs ()) ,
          simple-body(
            block-expr(
              tree-with(
                func-call(
                  id-ter () with $id := token("terminal") end with ,
                  actuals(
                    actual(
                      tree-with(
                        func-call(
                          id-ter () with $id := token("string-box") end with ,
                          actuals()) ,
                        decorations(
                          decoration(
                            attr () with $attr := token("$string-box") end with ,
                            expression)))))),
                decorations(
                  decoration(
                    attr () with $attr := token("$separator") end with ,
                    applied-id(
                      unqual-id(
                        id-ter () with $id := token("no-sep") end with))))))))
end function ;

{--------------------------------------------------------------------}
{ Fonction creer-def-fonc-predef                                     }
{--------------------------------------------------------------------}
function creer-def-fonc-predef () : DECL is
  let
    transformation : EXPR := func-call(
                                 id-ter() with $id := token("string") end with ,
                                 actuals(
                                   actual(
                                     applied-id(
                                       unqual-id(
                                         id-ter ()
                                           with
                                             $id := token("object")
                                           end with ))))) ,
    fonc-token : FUNC-DEF := creer-def-fonc-predef-type("token", transformation) ,
    fonc-int : FUNC-DEF := creer-def-fonc-predef-type("int", transformation) ,
    fonc-char : FUNC-DEF := creer-def-fonc-predef-type("char", transformation) ,
    fonc-real : FUNC-DEF := creer-def-fonc-predef-type("real", transformation) ,
    fonc-string : FUNC-DEF := creer-def-fonc-predef-type(
                               "string" ,
                               applied-id(
                                 unqual-id(
                                   id-ter () with $id := token("object") end with )))
  in
    rec-funcs-defs-pre(
      fonc-token ,
      rec-funcs-defs-pre(
        fonc-int ,
          rec-funcs-defs-pre(
            fonc-char ,
            rec-funcs-defs(fonc-string))))
{ a remplacer par quand string(real) definie }
{    rec-funcs-defs-pre(
      fonc-token ,
      rec-funcs-defs-pre(
        fonc-int ,
          rec-funcs-defs-pre(
            fonc-char ,
            rec-funcs-defs-pre(
              fonc-real ,
```

```
            rec-funcs-defs(fonc-string))))) }
end function ;

{------------------------------------------------------------}
{                   DECLARATION DES ATTRIBUTS                }
{------------------------------------------------------------}
attribute
{------------------------------------------------------------}
{                  attributs represantant des                }
{            sous-arbres de l'arbre olga de sortie           }
{------------------------------------------------------------}
 synthesized $Rhs-Symbol-s(PP-PRODUCTION) : SYMBOL ;
 synthesized $Ag-Header(PP-HEADER) : AG-HEADER ;
 synthesized $Block(PP-BLOCK) : BLOCK ;
 synthesized $Compilation-Unit(PP-PROG) : COMPILATION-UNIT ;
 synthesized $Decl(PP-DECL,PP-BLOCK,PP-RULES-T ) : DECL ;
 synthesized $Decls(PP-DECLS,PP-TOP-BOX,PP-BOX,PP-BOXES,PP-ELSIF-PART,
                  PP-ELSIF-CLAUSE,PP-MATCH-CLAUSES,PP-MATCH-CLAUSE,
                  PP-OTHER-CLAUSE ) : DECLS ;
 synthesized $Elsif-Clause(PP-ELSIF-CLAUSE) : ELSIF-CLAUSE ;
 synthesized $Elsif-Part(PP-ELSIF-PART) : ELSIF-PART ;
 synthesized $Expr(PP-ATTRIB-OCC,PP-BOX,PP-BOXES,PP-BOX-T,PP-BOXES-T,
                 PP-EXPR,PP-INSERT,PP-SEP-PARAM,PP-SEPARATOR,PP-BOX-T,
                 PP-STRING ) : EXPR ;
 synthesized $Actuals(PP-ACTUALS) : ACTUALS ;
 synthesized $Actual(PP-ACTUAL) : ACTUAL ;
 synthesized $Hook(PP-HOOK) : HOOK ;
 synthesized $Map-Dir(PP-DIR) : MAP-DIR ;
 synthesized $Match-Clause(PP-MATCH-CLAUSE,PP-OTHER-CLAUSE) : MATCH-CLAUSE ;
 synthesized $Match-Clauses(PP-MATCH-CLAUSES) : MATCH-CLAUSES ;
 synthesized $Pattern(PP-PATTERN) : PATTERN ;
 synthesized $Patterns(PP-PATTERNS) : PATTERNS ;
 synthesized $Phyla-Names(PP-PHYLA) : PHYLA-NAMES ;
 synthesized $Phylum(PP-PHYLUM) : PHYLUM ;
 synthesized $Prod-Rhs(PP-PROD-RHS) : PROD-RHS ;
 synthesized $Production(PP-PRODUCTION) : PRODUCTION ;
 synthesized $Production-Group(PP-RULE-OP) : PRODUCTION-GROUP ;
 synthesized $Sem-Rules(PP-TOP-BOX) : SEM-RULES ;
 synthesized $Semantics(PP-RULES-OP) : SEMANTICS ;
 synthesized $Symbol(PP-SYMBOL) : SYMBOL ;
 synthesized $Qualifier(PP-QUALIFIER) : QUALIFIER ;
 synthesized $Func-Def(PP-RULE-T) : FUNC-DEF ;
 synthesized $Func-Body(PP-TOP-BOX-T) : FUNC-BODY ;

{------------------------------------------------------------}
{ attributs de calculs intermediaires                        }
{------------------------------------------------------------}
 inherited   $Rhs-Symbol-h(PP-TOP-BOX,PP-BOX,PP-BOXES,PP-ELSIF-CLAUSE,
                        PP-ELSIF-PART,PP-MATCH-CLAUSE,PP-MATCH-CLAUSES,
                        PP-OTHER-CLAUSE) : SYMBOL ;
 inherited   $Symbol-h(PP-CONTEXTS,PP-CONTEXT) : SYMBOL ;
 {--- representations internes d'arbres OLGA ---}
 inherited   $symb-h(PP-CONTEXTS,PP-CONTEXT ) : t-symb ;
 synthesized $symb(PP-SYMBOL) : t-symb ;
 synthesized $kind-qual(PP-QUALIFIER) : kind-qual ;
 synthesized $val-qual(PP-QUALIFIER) : token ;
 synthesized $L-Ei-Clause(PP-ELSIF-CLAUSE) : l-ei-clause ;
 synthesized $L-Ei-Part(PP-ELSIF-PART) : l-ei-part ;
 compound    $h.L-Attr-Occ
      and    $s.L-Attr-Occ(PP-ELSIF-PART,PP-ELSIF-CLAUSE,PP-MATCH-CLAUSES,
                       PP-MATCH-CLAUSE,PP-OTHER-CLAUSE) : l-sem-rule ;
 inherited   $L-Attr-Occ(PP-ELSIF-PART,PP-ELSIF-CLAUSE,PP-MATCH-CLAUSES,
                      PP-MATCH-CLAUSE,PP-OTHER-CLAUSE ) : l-sem-rule ;
 synthesized $L-M-Clause(PP-MATCH-CLAUSE,PP-OTHER-CLAUSE) : l-m-clause ;
 synthesized $L-M-Clauses(PP-MATCH-CLAUSES) : l-m-clauses ;
 synthesized $L-Sem-Rule(PP-BOX,PP-BOXES,PP-CONTEXTS) : l-sem-rule ;
```

```
 synthesized $Sem-Rule(PP-CONTEXT) : sem-rule ;
 {--- attributs booleens de decision ---}
 synthesized $is-no-insert(PP-INSERT) : bool ;
 synthesized $is-left-dir(PP-DIR) : bool ;
 {--- compteur de formats liste au sein d'une meme production ---}
 compound    $h.num-pplist
      and $s.num-pplist(PP-BOX,PP-BOXES,PP-ELSIF-PART,PP-ELSIF-CLAUSE,
                        PP-MATCH-CLAUSES,PP-MATCH-CLAUSE,PP-OTHER-CLAUSE ) : int ;
{--- regles par defaut ---}
defaults
  classes $EXPRESSION   ($Expr)  : EXPR ;
          $DECLARATIONS ($Decls) : DECLS ;

  default $EXPRESSION use
    where NULL use
      null-EXPR ()
    end where ;
  end default ;

  default $DECLARATIONS use
    where NULL use
        null-DECLS ()
    end where ;
  end default ;

{- ------------------------------------------------------------------- -}
{- PP-PROG                                                              -}
{- ------------------------------------------------------------------- -}
where null-PP-PROG ->  use
  $Compilation-Unit := null-COMPILATION-UNIT () ;
end where ;

where pp-prog -> PP-HEADER PP-HOOK use
  $Compilation-Unit := simple-AG(
                         $Ag-Header (PP-HEADER) ,
                         $Hook (PP-HOOK) ) ;
end where ;

{- ------------------------------------------------------------------- -}
{- PP-HEADER                                                            -}
{- ------------------------------------------------------------------- -}
where null-PP-HEADER ->  use
  $Ag-Header := null-AG-HEADER () ;
end where ;

where pp-header -> PP-ID PP-ID use
  $Ag-Header := AG-header(
                  id-ter () with $id := $pp-id (PP-ID.1) end with ,
                  functional-AG(
                    syntax(
                      id-ter () with $id := $pp-id (PP-ID.2) end with ) ,
                    syntaxes(
                      syntax(
                        id-ter () with $id := token("boxes-in") end with )))) ;
end where ;

{- ------------------------------------------------------------------- -}
{- PP-HOOK                                                              -}
{- ------------------------------------------------------------------- -}
where null-PP-HOOK ->  use
  $Hook := null-HOOK () ;
end where ;

where pp-hook -> PP-DECLS PP-BLOCK declare
value
  declaration-att : DECLS := decls-pre(
```

```
                              attribute-decl(
                                syn-kind () ,
                                attr () with $attr := token("$Ppat") end with ,
                                phyla-names (),
                                type-id(
                                  unqual-id(
                                    id-ter () with $id := token("PPAT") end with))) ,
                              decls(
                                attribute-decl(
                                  syn-kind () ,
                                  attr () with $attr := token("$Box") end with ,
                                  phyla-names () ,
                                  type-id(
                                    unqual-id(
                                      id-ter ()
                                        with $id := token("BOX") end with ))))) ;
use
  $Hook := hook-simple-AG(
             decls-merge(
                declaration-att ,
                decls-post(
                  $Decls (PP-DECLS) ,    { contexts and separators declarations}
                  $Decl (PP-BLOCK) )) ,  { functions (rules for types) declarations }
             $Block (PP-BLOCK) );
end where ;

{- -------------------------------------------------------------------- -}
{- PP-DECLS                                                              -}
{- -------------------------------------------------------------------- -}
where pp-decls -> PP-DECL * declare
value
 predef : DECL := import-all(
                    module-id () with $module := token("ppat-predef") end with ) ;
use
  $Decls := map left decls-post
              value  decls (predef)
              other  $Decl (PP-DECL)
            end map ;
end where ;

{- -------------------------------------------------------------------- -}
{- PP-DECL                                                               -}
{- -------------------------------------------------------------------- -}
where null-PP-DECL ->  use
  $Decl := null-DECL () ;
end where ;

where pp-sep-def -> PP-SEP-ID PP-SEPARATOR use
  $Decl := const-decl(
            id-ter () with $id := $pp-sep-id (PP-SEP-ID) end with ,
            block-expr(
              constant-expr($Expr (PP-SEPARATOR) ))) ;
end where ;

where pp-cxt-decl -> PP-ATTR PP-PHYLA PP-ID use
  $Decl := attribute-decl(
             inh-kind () , { tous les contextes sont herites }
             attr () with $attr := $pp-attr (PP-ATTR) end with ,
             $Phyla-Names (PP-PHYLA) ,
             type-id(
               unqual-id( { tous les contextes sont entiers }
                 id-ter () with $id := $pp-id(PP-ID) end with ))) ;
end where ;

where pp-import-all -> PP-ID use
  $Decl := import-all(
```

```
            module-id () with $module := $pp-id (PP-ID) end with ) ;
end where ;

{- ------------------------------------------------------------------ -}
{- PP-SEPARATOR                                                        -}
{- ------------------------------------------------------------------ -}
where pp-separator -> PP-ID PP-SEP-PARAM use
  $Expr := func-call(
            id-ter () with $id := token("sep-info") end with ,
            actuals-pre(
              actual(
                applied-id(
                  unqual-id(
                    id-ter () with $id := $pp-id (PP-ID) end with ))) ,
              actuals(
                actual($Expr (PP-SEP-PARAM) )))) ;
end where ;

{- ------------------------------------------------------------------ -}
{- PP-SEP-PARAM                                                        -}
{- ------------------------------------------------------------------ -}
where pp-sep-param -> PP-INT * declare
value
  liste : LIST-ELTS := map left list-elts-post
                        value list-elts ()
                        other actual(
                                int-const(
                                  int () with $int := $pp-int (PP-INT) end with ))
            end map ;
use
  $Expr := list-constr(
            type-id(
              unqual-id(
                id-ter () with $id := token("sep-param") end with )) ,
            liste ) ;
end where ;

{- ------------------------------------------------------------------ -}
{- PP-PHYLA                                                            -}
{- ------------------------------------------------------------------ -}
where null-PP-PHYLA ->  use
  $Phyla-Names := null-PHYLA-NAMES () ;
end where ;

where pp-phyla -> PP-PHYLUM * use
  $Phyla-Names := map left phyla-names-post
                    value phyla-names ()
                    other $Phylum (PP-PHYLUM)
                  end map ;
end where ;

{- ------------------------------------------------------------------ -}
{- PP-BLOCK                                                            -}
{- ------------------------------------------------------------------ -}
where null-PP-BLOCK ->  use
  $Decl := null-DECL () ;
  $Block := null-BLOCK () ;
end where ;

where pp-block -> PP-RULES-T PP-RULES-OP use
  $Decl := $Decl (PP-RULES-T) ;
  $Block := block-semantics($Semantics (PP-RULES-OP) ) ;
end where ;

{- ------------------------------------------------------------------ -}
{- PP-RULES-T                                                          -}
```

```
{- ------------------------------------------------------------------ -}
where null-PP-RULES-T ->  use
  $Decl := null-DECL () ;
end where ;

where pp-rules-t ->  PP-RULE-T * use
  $Decl := map left rec-funcs-defs-post
            value creer-def-fonc-predef ()
            other $Func-Def (PP-RULE-T)
           end map ;
end where ;

{- ------------------------------------------------------------------ -}
{- PP-RULE-T                                                           -}
{- ------------------------------------------------------------------ -}
where null-PP-RULE-T ->  use
  $Func-Def := null-FUNC-DEF () ;
end where ;

where pp-rule-t ->  PP-ID  PP-TOP-BOX-T  declare
value
  {--- construction de la fonction associee a un type complexe ---}
  fonc-assoc-type : FUNC-DEF :=
    let
      nom-fonction : string := "ppat_affichage_" + string ($pp-id (PP-ID)) ,
      form1 : FORMAL := formal(
                          id-ter () with $id := token ("object") end with ,
                          type-id(
                            unqual-id(id-ter () with $id := $pp-id (PP-ID) end with ))) ,
      form2 : FORMAL := formal(
                          id-ter () with $id := token ("Result") end with ,
                          type-id(
                            unqual-id(id-ter () with $id := token("BOXES") end with ))) ,
      typ-fonc : TYPE-ID := type-id(
                                unqual-id(id-ter () with $id := token("BOX") end with ))
    in
      func-def(
        func-header(
          offline-mode () ,
          id-ter () with $id := token(nom-fonction) end with ,
          formals-pre(form1, formals(form2)) ,
          typ-fonc ,
          excs () ) ,
        $Func-Body (PP-TOP-BOX-T) ) ;
use
  $Func-Def := fonc-assoc-type ;
end where ;

{- ------------------------------------------------------------------ -}
{- PP-TOP-BOX-T                                                        -}
{- ------------------------------------------------------------------ -}
where null-PP-TOP-BOX-T ->  use
  $Func-Body := null-FUNC-BODY () ;
end where ;

where pp-top-box-t ->  PP-BOX-T  use
  $Func-Body := simple-body(
                  block-expr($Expr (PP-BOX-T) )) ;
end where ;

where pp-empty-box-t ->  use
  $Func-Body := simple-body(
                  block-expr(
                    tree-with(
                      func-call(
                        id-ter () with $id := token("empty-box") end with ,
```

```
                     actuals () ),
                   decorations(
                     decoration(
                       attr () with $attr := token("$separator") end with ,
                       applied-id(
                         unqual-id(
                           id-ter () with $id := token("no-sep") end with ))))))) ;
end where ;

{- ------------------------------------------------------------------- -}
{- PP-BOX-T                                                            -}
{- ------------------------------------------------------------------- -}
where pp-pplist-box-t ->  PP-ID PP-DIR  PP-SEP-ID  PP-INSERT  PP-BOX-T  declare
value
  selector : EXPR := applied-id(
                     unqual-id(id-ter () with $id := token("object") end with )) ;
  LST-T : ID-TER := id-ter () with $id := token("LST-T") end with ;
  ELT : ID-TER := id-ter () with $id := token("ELT") end with ;
  Result : ID-TER := id-ter () with $id := token("Result") end with ;
  box : ID-TER := id-ter () with $id := token("box") end with ;
  boxes : ID-TER := id-ter () with $id := token("boxes") end with ;
  expr-empty-boxes : EXPR := func-call(boxes , actuals ()) ;
  boxes-pre : ID-TER := id-ter () with $id := token("boxes-pre") end with ;
  boxes-post : ID-TER := id-ter () with $id := token("boxes-post") end with ;
  nm-type-liste : token := $s.type.name.name.name ;
  nm-type-elt : token := $s.type(PP-BOX-T).name.name.name ;
  id-type-liste : ID := unqual-id(id-ter () with $id := nm-type-liste end with ) ;
  id-type-elt : ID := unqual-id(id-ter () with $id := nm-type-elt end with ) ;
  id-fonc-liste : ID-TER := id-ter ()
                         with $id := token(
                                       "ppat_affichage_"
                                       + string(nm-type-liste))
                         end with ;
  id-fonc-elt : ID := unqual-id(
                       id-ter ()
                         with $id := token(
                                       "ppat_affichage_"
                                       + string(nm-type-elt))
                         end with ) ;
  pattern-elt : PATTERN := pattern-var-constraint(ELT , type-id(id-type-elt)) ;
  pattern-liste : PATTERN := pattern-var-constraint(LST-T , type-id(id-type-liste)) ;
  pattern-liste-left : PATTERN := pattern-cons(
                                 id-type-liste ,
                                 patterns-pre(
                                   pattern-elt ,
                                   patterns(pattern-liste))) ;
  pattern-liste-right : PATTERN := pattern-cons(
                                 id-type-liste ,
                                 patterns-pre(
                                   pattern-liste ,
                                   patterns(pattern-elt))) ;
  patterns-liste : PATTERNS := if $is-left-dir (PP-DIR) then
                             patterns(pattern-liste-left)
                            else
                             patterns(pattern-liste-right)
                            end if ;
  patterns-elt : PATTERNS := patterns(
                            pattern-cons(
                              id-type-liste ,
                              patterns(pattern-elt))) ;
  para1-lst : ACTUAL := actual(applied-id(unqual-id (LST-T))) ;
  para1-elt : ACTUAL := actual(
                       func-call(
                         id-ter () with $id := nm-type-liste end with  ,
                         actuals ())) ;
  para2-lst : ACTUAL :=
```

```
  let
    ancienne-liste : ACTUAL := actual(applied-id(unqual-id(Result))) ,
    comp-box : ACTUAL := actual(
                          func-call(
                            boxes-pre ,
                              actuals-pre(
                                actual($Expr(PP-BOX-T)) ,
                                actuals(
                                  actual(
                                    func-call(
                                      boxes ,
                                      actuals(actual($Expr (PP-INSERT))))))))) ,
    nvel-elt : ACTUAL := actual(
                          tree-with(
                            func-call(
                              box ,
                              actuals(comp-box)) ,
                            decorations(
                              decoration(
                                attr () with $attr := token("$separator") end with ,
                                applied-id(
                                  unqual-id(
                                    id-ter ()
                                      with $id := token("sep-null") end with ))))))
      in
        actual(
          func-call(boxes-post, actuals-pre(ancienne-liste, actuals(nvel-elt)))) ;
para2-elt : ACTUAL :=
  let
    ancienne-liste : ACTUAL := actual(applied-id(unqual-id(Result))) ,
    comp-box : ACTUAL := actual(
                          func-call(
                            boxes ,
                              actuals(
                                actual($Expr(PP-BOX-T))))) ,
    nvel-elt : ACTUAL := actual(
                          tree-with(
                            func-call(
                              box ,
                              actuals(comp-box)) ,
                            decorations(
                              decoration(
                                attr () with $attr := token("$separator") end with ,
                                applied-id(
                                  unqual-id(
                                    id-ter ()
                                      with $id := token("sep-null") end with ))))))
      in
        actual(
          func-call(boxes-post, actuals-pre(ancienne-liste, actuals(nvel-elt)))) ;
block-liste : BLOCK := block-expr(
                         func-call(
                           id-fonc-liste ,
                           actuals-pre(para1-lst , actuals(para2-lst)))) ;
block-elt : BLOCK := block-expr(
                       func-call(
                         id-fonc-liste ,
                         actuals-pre(para1-elt , actuals(para2-elt)))) ;
clause-liste : MATCH-CLAUSE := match-clause(patterns-liste , block-liste) ;
clause-elt : MATCH-CLAUSE := match-clause(patterns-elt , block-elt) ;
clauses : MATCH-CLAUSES := match-clauses-pre(
                             clause-liste ,
                             match-clauses(clause-elt)) ;
clause-autre : MATCH-CLAUSE := match-clause-other(
                                 tree-with(
                                   func-call(
```

```
                                     box ,
                                     actuals(actual(applied-id(unqual-id(Result))))) ,
                                   decorations(
                                     decoration(
                                       attr ()
                                         with $attr := token("$separator") end with ,
                                       applied-id(
                                         unqual-id(
                                           id-ter ()
                                             with
                                               $id := $pp-sep-id (PP-SEP-ID)
                                             end with )))))) ;
use
  $Expr := case-match(selector , clauses , clause-autre ) ;
end where ;

where pp-field-ref ->  PP-ID  declare
value
  nom-fonction : string := "ppat_affichage_" + string ($s.type.name.name.name) ;
  expression-record : EXPR :=func-call(
              id-ter () with $id := token (nom-fonction) end with ,
              actuals-pre(
                actual(
                  field-access(
                    applied-id(
                      unqual-id(id-ter () with $id := token("object") end with )) ,
                    id-ter () with $id := $pp-id (PP-ID) end with )) ,
                actuals(
                  actual(
                    func-call(
                      id-ter () with $id := token("boxes") end with ,
                      actuals ())))))) ;
  expr-var-loc : EXPR := func-call(
              id-ter () with $id := token (nom-fonction) end with ,
              actuals-pre(
                actual(
                  applied-id(
                      unqual-id(id-ter () with $id := token("ELT") end with ))) ,
                actuals(
                  actual(
                    func-call(
                      id-ter () with $id := token("boxes") end with ,
                      actuals ())))))) ;
use
  $Expr := if $is_var_loc then
                expr-var-loc
           else
                expression-record
           end if ;
end where ;

where pp-compound-box-t ->  PP-SEP-ID  PP-BOXES-T  use
  $Expr := tree-with(
            func-call(
              id-ter () with $id := token("box") end with ,
              actuals(actual($Expr (PP-BOXES-T)))) ,
            decorations(
              decoration(
                attr () with $attr := token("$separator") end with ,
                applied-id(
                  unqual-id(
                    id-ter () with $id := $pp-sep-id (PP-SEP-ID) end with ))))) ;
end where ;

where pp-string-box-t ->  PP-STRING  use
  $Expr := tree-with(
```

```
            func-call(
               id-ter () with $id := token("terminal") end with ,
               actuals(actual($Expr(PP-STRING)))) ,
            decorations(
              decoration(
                attr () with $attr := token("$separator") end with ,
                applied-id(
                  unqual-id(
                    id-ter () with $id := token("no-sep") end with ))))) ;
end where ;

{- ------------------------------------------------------------------ -}
{- PP-BOXES-T                                                          -}
{- ------------------------------------------------------------------ -}
where pp-boxes-t ->  PP-BOX-T * declare
value
  lst-expr-box : ACTUALS := map left actuals-post
                        value actuals ()
                        other actual($Expr (PP-BOX-T))
                      end map ;
  expr-lst-box-empty : EXPR := func-call(
                                 id-ter () with $id := token("boxes") end with ,
                                 actuals ()) ;
use
  $Expr := lst-expr-box_TO_expr-lst-box(
            id-ter () with $id := token("boxes-post") end with ,
            lst-expr-box ,
            expr-lst-box-empty) ;
end where ;

{- ------------------------------------------------------------------ -}
{- PP-RULES-OP                                                         -}
{- ------------------------------------------------------------------ -}
where null-PP-RULES-OP ->  use
  $Semantics := null-SEMANTICS () ;
end where ;

where pp-rules-op -> PP-RULE-OP * use
  $Semantics := map left productions-post
                  value productions ()
                  other $Production-Group (PP-RULE-OP)
                end map ;
end where ;

{- ------------------------------------------------------------------ -}
{- PP-RULE-OP                                                          -}
{- ------------------------------------------------------------------ -}
where null-PP-RULE-OP ->  use
  $Production-Group := null-PRODUCTION-GROUP () ;
end where ;

where pp-rule-op -> PP-PRODUCTION PP-TOP-BOX use
  $Production-Group := case $Decls (PP-TOP-BOX) is
                        decls () :
                          simple-prod-group(
                            $Production (PP-PRODUCTION) ,
                            $Sem-Rules (PP-TOP-BOX) );
                        other :
                          prod-group-with-decls(
                            $Production (PP-PRODUCTION) ,
                            $Decls (PP-TOP-BOX) ,
                            $Sem-Rules (PP-TOP-BOX) );
                       end case ;
  $Rhs-Symbol-h (PP-TOP-BOX) := $Rhs-Symbol-s (PP-PRODUCTION) ;
end where ;
```

```
{- ------------------------------------------------------------------ -}
{- PP-PRODUCTION                                                      -}
{- ------------------------------------------------------------------ -}
where null-PP-PRODUCTION ->  use
  $Production := null-PRODUCTION () ;
  $Rhs-Symbol-s := null-SYMBOL () ;
end where ;

where pp-fixed-prod -> PP-ID PP-PROD-RHS use
  $Production := fixed-prod(
                   id-ter () with $id := $pp-id (PP-ID) end with ,
                   $Prod-Rhs (PP-PROD-RHS) ) ;
  $Rhs-Symbol-s := null-SYMBOL () ;
end where ;

where pp-pe-list-prod -> PP-ID PP-PHYLUM use
  $Rhs-Symbol-s := unqual-symb(
                     id-ter () with $id := $pp-phylum (PP-PHYLUM) end with ) ;
  $Production := pe-list-prod(
                   id-ter () with $id := $pp-id (PP-ID) end with ,
                   phylum () with $phylum := $pp-phylum (PP-PHYLUM) end with ) ;
end where ;

where pp-ne-list-prod -> PP-ID PP-PHYLUM use
  $Rhs-Symbol-s := unqual-symb(
                     id-ter () with $id := $pp-phylum (PP-PHYLUM) end with ) ;
  $Production := ne-list-prod(
                   id-ter () with $id := $pp-id (PP-ID) end with ,
                   phylum () with $phylum := $pp-phylum (PP-PHYLUM) end with ) ;
end where ;

{- ------------------------------------------------------------------ -}
{- PP-PROD-RHS                                                        -}
{- ------------------------------------------------------------------ -}
where null-PP-PROD-RHS ->  use
  $Prod-Rhs := null-PROD-RHS () ;
end where ;

where pp-prod-rhs -> PP-PHYLUM * use
  $Prod-Rhs := map left prod-rhs-post
                 value prod-rhs ()
                 other $Phylum (PP-PHYLUM)
               end map ;
end where ;

{- ------------------------------------------------------------------ -}
{- PP-TOP-BOX                                                         -}
{- ------------------------------------------------------------------ -}
where null-PP-TOP-BOX ->  use
  $Sem-Rules := null-SEM-RULES () ;
end where ;

where pp-top-box -> PP-BOX use
  $Rhs-Symbol-h (PP-BOX) := $Rhs-Symbol-h ;
  $Decls := if $h_root then
                decls-pre(
                  value-decl(
                    id-ter () with $id := token("ppat-top-expression") end with ,
                    type-id(unqual-id(id-ter () with $id := token("BOX") end with)) ,
                    block-expr($Expr(PP-BOX))) ,
                  $Decls (PP-BOX))
            else
                $Decls (PP-BOX)
            end if ;
  $Sem-Rules := if $h_root then
                  sem-rules-pre(
```

```
                    simple-rule(
                      attrib-occ-lhs(
                        attr () with $attr := token("$Ppat") end with ) ,
                      func-call(
                        id-ter () with $id := token("ppat") end with ,
                        actuals(
                          actual(
                            applied-id(
                              unqual-id(
                                id-ter ()
                                  with
                                    $id := token("ppat-top-expression")
                                  end with)))))) ,
                    sem-rules-creation($L-Sem-Rule (PP-BOX) , sem-rules ()))
                else
                  sem-rules-pre(
                    simple-rule(
                      attrib-occ-lhs(
                        attr () with $attr := token("$Box") end with ) ,
                      $Expr (PP-BOX)) ,
                    sem-rules-creation($L-Sem-Rule (PP-BOX) , sem-rules ()))
                end if ;
  $h.num-pplist(PP-BOX) := 0 ;
end where ;

where pp-empty-box -> declare
value
  expr-empty-box : EXPR := tree-with(
                            func-call(
                              id-ter () with $id := token("empty-box") end with ,
                              actuals ()) ,
                            decorations(
                              decoration(
                                attr () with $attr := token("$separator") end with ,
                                applied-id(
                                  unqual-id(
                                    id-ter ()
                                      with $id := token("no-sep") end with))))) ;
use
  $Decls := decls () ;
  $Sem-Rules := if $h_root then
                  sem-rules(
                    simple-rule(
                      attrib-occ-lhs(
                        attr () with $attr := token("$Ppat") end with ) ,
                      func-call(
                        id-ter () with $id := token("ppat") end with ,
                        actuals(actual(expr-empty-box)))))
                else
                  sem-rules(
                    simple-rule(
                      attrib-occ-lhs(
                        attr () with $attr := token("$Box") end with ) ,
                      expr-empty-box))
                end if ;
end where ;

{- --------------------------------------------------------------------- -}
{- PP-BOX                                                                -}
{- --------------------------------------------------------------------- -}
where null-PP-BOX ->  use
  $L-Sem-Rule := l-sem-rule () ;
end where ;

where pp-pplist-box ->  PP-DIR  PP-SEP-ID  PP-INSERT  PP-BOX  declare
value
```

```
  separateur : ACTUAL := actual(
                            applied-id(
                                unqual-id(
                             id-ter ()
                               with $id := $pp-sep-id (PP-SEP-ID) end with ))) ;
  boxes-post : ID := unqual-id(id-ter () with $id := token("boxes-post") end with ) ;
  initial-value : EXPR := func-call(
                           id-ter () with $id := token("boxes") end with ,
                           actuals () ) ;
  dernier : EXPR := if $is-left-dir (PP-DIR) then
                       attrib-occ-as-expr(
                         attrib-occ-lhs(
                           attr() with $attr := token("$arity") end with ))
                    else
                       int-const(int() with $int := token("1") end with)
                    end if ;
  condition : EXPR := func-call(
                       id-ter () with $id := token("=") end with ,
                       actuals-pre(
                         actual(attrib-occ-as-expr(
                           attrib-occ(
                             attr () with $attr := token("$position") end with ,
                             $Rhs-Symbol-h ))) ,
                         actuals(actual(dernier)))) ;
  box-compound : EXPR := tree-with(
                     func-call(
                       id-ter () with $id := token("box") end with ,
                       actuals(
                         actual(
                           func-call(
                             id-ter () with $id := token("boxes-pre") end with ,
                             actuals-pre(
                               actual($Expr (PP-BOX)) ,
                               actuals(
                                 actual(
                                   func-call(
                                     id-ter() with $id := token("boxes") end with ,
                                     actuals(actual($Expr (PP-INSERT)))))))))) ,
                     decorations(
                       decoration(
                         attr () with $attr := token("$separator") end with ,
                         applied-id(
                           unqual-id(
                             id-ter () with $id := token("sep-null") end with ))))) ;
  other-value : ACTUALS := actuals(
                           actual(
                             conditional(
                               condition ,
                               $Expr (PP-BOX) ,
                               elsif-part () ,
                               box-compound ))) ;
  name-value : string := "ppat-list-box-" + string($h.num-pplist) ;
use
  $Rhs-Symbol-h (PP-BOX) := $Rhs-Symbol-h ;
  $L-Sem-Rule := $L-Sem-Rule (PP-BOX) ;
  $Decls := decls(
            value-decl(
              id-ter () with $id := token(name-value) end with ,
              type-id(
                unqual-id(
                  id-ter () with $id := token("BOXES") end with)) ,
              block-expr(
                map-expr(
                  $Map-Dir(PP-DIR), boxes-post, initial-value, other-value)))) ;
  $Expr := tree-with(
          func-call(
```

```
            id-ter () with $id := token("box") end with,
            actuals(
              actual(
                applied-id(
                  unqual-id(
                    id-ter () with $id := token(name-value) end with ))))) ,
          decorations(
            decoration(
              attr () with $attr := token("$separator") end with ,
              applied-id(
                unqual-id(
                  id-ter () with $id := $pp-sep-id (PP-SEP-ID) end with ))))) ;
  $h.num-pplist(PP-BOX) := $h.num-pplist + 1 ;
  $s.num-pplist := $s.num-pplist(PP-BOX) ;
end where ;

where pp-case-tree -> PP-SYMBOL PP-MATCH-CLAUSES PP-OTHER-CLAUSE use
  $Decls := decls-merge ($Decls(PP-MATCH-CLAUSES) , $Decls(PP-OTHER-CLAUSE)) ;
  $Expr := case-tree(
          $Symbol (PP-SYMBOL) ,
          $Match-Clauses (PP-MATCH-CLAUSES) ,
          $Match-Clause (PP-OTHER-CLAUSE) ) ;
  $h.L-Attr-Occ (PP-MATCH-CLAUSES) := l-sem-rule () ;
  $h.L-Attr-Occ (PP-OTHER-CLAUSE) := $s.L-Attr-Occ (PP-MATCH-CLAUSES) ;
  $L-Attr-Occ (PP-MATCH-CLAUSES) := $s.L-Attr-Occ (PP-OTHER-CLAUSE) ;
  $L-Attr-Occ (PP-OTHER-CLAUSE) := $s.L-Attr-Occ (PP-OTHER-CLAUSE) ;
  $L-Sem-Rule := construct-case-tree(
                $s.L-Attr-Occ (PP-OTHER-CLAUSE) ,
                $L-M-Clauses (PP-MATCH-CLAUSES) ,
                $L-M-Clause (PP-OTHER-CLAUSE) ,
                l-sem-rule () ,
                $Symbol (PP-SYMBOL)) ;
  $h.num-pplist(PP-MATCH-CLAUSES) := $h.num-pplist ;
  $h.num-pplist(PP-OTHER-CLAUSE) := $s.num-pplist(PP-MATCH-CLAUSES) ;
  $s.num-pplist := $s.num-pplist(PP-OTHER-CLAUSE) ;
end where ;

where pp-case-match -> PP-EXPR PP-MATCH-CLAUSES PP-OTHER-CLAUSE use
  $Decls := decls-merge ($Decls(PP-MATCH-CLAUSES) , $Decls(PP-OTHER-CLAUSE)) ;
  $Expr := case-match(
          $Expr (PP-EXPR) ,
          $Match-Clauses (PP-MATCH-CLAUSES) ,
          $Match-Clause (PP-OTHER-CLAUSE) ) ;
  $h.L-Attr-Occ (PP-MATCH-CLAUSES) := l-sem-rule () ;
  $h.L-Attr-Occ (PP-OTHER-CLAUSE) := $s.L-Attr-Occ (PP-MATCH-CLAUSES) ;
  $L-Attr-Occ (PP-MATCH-CLAUSES) := $s.L-Attr-Occ (PP-OTHER-CLAUSE) ;
  $L-Attr-Occ (PP-OTHER-CLAUSE) := $s.L-Attr-Occ (PP-OTHER-CLAUSE) ;
  $L-Sem-Rule := construct-case-match(
                $s.L-Attr-Occ (PP-OTHER-CLAUSE) ,
                $L-M-Clauses (PP-MATCH-CLAUSES) ,
                $L-M-Clause (PP-OTHER-CLAUSE) ,
                l-sem-rule () ,
                $Expr (PP-EXPR)) ;
  $h.num-pplist(PP-MATCH-CLAUSES) := $h.num-pplist ;
  $h.num-pplist(PP-OTHER-CLAUSE) := $s.num-pplist(PP-MATCH-CLAUSES) ;
  $s.num-pplist := $s.num-pplist(PP-OTHER-CLAUSE) ;
end where ;

where pp-compound-box -> PP-SEP-ID PP-BOXES use
  $Decls := $Decls(PP-BOXES) ;
  $Rhs-Symbol-h (PP-BOXES) := $Rhs-Symbol-h ;
  $L-Sem-Rule := $L-Sem-Rule (PP-BOXES) ;
  $Expr := tree-with(
          func-call(
            id-ter () with $id := token("box") end with ,
            actuals(
```

```
              actual($Expr (PP-BOXES)))) ,
          decorations(
            decoration(
              attr () with $attr := token("$separator") end with ,
              applied-id(
                unqual-id(
                  id-ter () with $id := $pp-sep-id (PP-SEP-ID) end with ))))) ;
  $h.num-pplist(PP-BOXES) := $h.num-pplist ;
  $s.num-pplist := $s.num-pplist(PP-BOXES) ;
end where ;

where pp-conditional ->  PP-EXPR  PP-BOX  PP-ELSIF-PART  PP-BOX use
  $Decls := decls-merge(decls-merge($Decls(PP-BOX.1) , $Decls(PP-ELSIF-PART)) ,
                        $Decls(PP-BOX.2)) ;
  $Expr := conditional(
            $Expr (PP-EXPR) ,
            $Expr (PP-BOX.1) ,
            $Elsif-Part (PP-ELSIF-PART) ,
            $Expr (PP-BOX.2) ) ;
  $L-Sem-Rule := construct-conditional(
                   $s.L-Attr-Occ (PP-ELSIF-PART) ,
                   $L-Sem-Rule (PP-BOX.1) ,
                   $L-Sem-Rule (PP-BOX.2) ,
                   l-sem-rule () ,
                   $L-Ei-Part (PP-ELSIF-PART) ,
                   $Expr (PP-EXPR)) ;
  $h.L-Attr-Occ (PP-ELSIF-PART) := add-attr-occ (
                                    add-attr-occ(
                                      l-sem-rule() ,
                                      $L-Sem-Rule(PP-BOX.1)) ,
                                    $L-Sem-Rule (PP-BOX.2) ) ;
  $L-Attr-Occ (PP-ELSIF-PART) := $s.L-Attr-Occ (PP-ELSIF-PART) ;
  $h.num-pplist(PP-BOX.1) := $h.num-pplist ;
  $h.num-pplist(PP-ELSIF-PART) := $s.num-pplist(PP-BOX.1) ;
  $h.num-pplist(PP-BOX.2) := $s.num-pplist(PP-ELSIF-PART) ;
  $s.num-pplist := $s.num-pplist(PP-BOX.2) ;
end where ;

where pp-rec-call -> PP-SYMBOL PP-CONTEXTS use
  $Decls := decls () ;
  $Symbol-h (PP-CONTEXTS) := $Symbol (PP-SYMBOL) ;
  $symb-h (PP-CONTEXTS) := $symb (PP-SYMBOL) ;
  $L-Sem-Rule := $L-Sem-Rule (PP-CONTEXTS) ;
  $Expr := attrib-occ-as-expr(
            attrib-occ(
              attr () with $attr := token("$Box") end with ,
              $Symbol(PP-SYMBOL))) ;
end where ;

where pp-attrib ->  PP-ATTRIB-OCC  declare
value
 nm-fonc : string := "ppat_affichage_" +
                     string($type(PP-ATTRIB-OCC).name.name.name) ;
use
  $Decls := decls () ;
  $Expr := func-call(
              id-ter () with $id := token(nm-fonc) end with ,
              actuals-pre(
                actual($Expr (PP-ATTRIB-OCC)) ,
                actuals(
                  actual(
                    func-call(
                      id-ter () with $id := token("boxes") end with,
                      actuals ())))))) ;
  $L-Sem-Rule := l-sem-rule () ;
end where ;
```

```
where pp-string-box -> PP-STRING use
  $Decls := decls () ;
  $L-Sem-Rule := l-sem-rule () ;
  $Expr := tree-with(
            func-call(
              id-ter () with $id := token("terminal") end with ,
              actuals(actual($Expr(PP-STRING)))) ,
            decorations(
              decoration(
                attr () with $attr := token("$separator") end with ,
                applied-id(
                  unqual-id(
                    id-ter () with $id := token("no-sep") end with ))))) ;
end where ;

{- ------------------------------------------------------------------- -}
{- PP-CONTEXTS                                                          -}
{- ------------------------------------------------------------------- -}
where null-PP-CONTEXTS ->  use
  $L-Sem-Rule := l-sem-rule () ;
end where ;

where pp-contexts -> PP-CONTEXT * use
  $L-Sem-Rule := map left l-sem-rule
                   value l-sem-rule ()
                   other $Sem-Rule (PP-CONTEXT)
                 end map ;
end where ;

{- ------------------------------------------------------------------- -}
{- PP-CONTEXT                                                           -}
{- ------------------------------------------------------------------- -}
where null-PP-CONTEXT ->  use
  $Sem-Rule :=         error-sem-rule ;
end where ;

where pp-context -> PP-ATTR PP-EXPR use
  $Sem-Rule := sem-rule(
                rep-occ(
                  $pp-attr (PP-ATTR) ,
                  $symb-h ) ,
                attrib-occ (
                  attr () with $attr := $pp-attr (PP-ATTR) end with ,
                  $Symbol-h) ,
                $Expr(PP-EXPR)) ;
end where ;

{- ------------------------------------------------------------------- -}
{- PP-ELSIF-PART                                                        -}
{- ------------------------------------------------------------------- -}
where null-PP-ELSIF-PART ->  use
  $L-Ei-Part := l-ei-part () ;
  $Elsif-Part := null-ELSIF-PART () ;
end where ;

where pp-elsif-part ->  PP-ELSIF-CLAUSE * use
  $Elsif-Part := map left elsif-part-post
                        value elsif-part ()
                        other $Elsif-Clause (PP-ELSIF-CLAUSE)
                 end map ;
  $L-Ei-Part := map left construct-ei-part
                        value l-ei-part ()
                        other $L-Ei-Clause (PP-ELSIF-CLAUSE)
                end map ;
  $Decls := map left decls-merge
```

```
                value decls()
                other $Decls(PP-ELSIF-CLAUSE)
            end map ;
end where ;

{- ------------------------------------------------------------------- -}
{- PP-ELSIF-CLAUSE                                                     -}
{- ------------------------------------------------------------------- -}
where null-PP-ELSIF-CLAUSE ->  use
  $L-Ei-Clause := l-ei-clause () ;
  $Elsif-Clause := null-ELSIF-CLAUSE () ;
end where ;

where pp-elsif-clause ->  PP-EXPR  PP-BOX  use
  $Elsif-Clause := elsif-clause($Expr (PP-EXPR) , $Expr (PP-BOX)) ;
  $s.L-Attr-Occ := add-attr-occ ($h.L-Attr-Occ , $L-Sem-Rule (PP-BOX)) ;
  $L-Ei-Clause := construct-ei-clause (
                    $L-Attr-Occ ,
                    $Expr (PP-EXPR) ,
                    $L-Sem-Rule (PP-BOX) ,
                    l-ei-clause () ) ;
  $h.num-pplist(PP-BOX) := $h.num-pplist ;
  $s.num-pplist := $s.num-pplist(PP-BOX) ;
  $Decls := $Decls(PP-BOX) ;
end where ;

{- ------------------------------------------------------------------- -}
{- PP-BOXES                                                            -}
{- ------------------------------------------------------------------- -}
where null-PP-BOXES ->  use
  $L-Sem-Rule := l-sem-rule () ;
end where ;

where pp-boxes -> PP-BOX * declare
value
  lst-expr-box : ACTUALS := map left actuals-post
                      value actuals ()
                      other actual($Expr (PP-BOX))
                    end map ;
  expr-lst-box-empty : EXPR := func-call(
                              id-ter () with $id := token("boxes") end with ,
                              actuals ()) ;
use
  $Rhs-Symbol-h (PP-BOX) := $Rhs-Symbol-h ;
  $L-Sem-Rule := map left +
                 value l-sem-rule ()
                 other $L-Sem-Rule (PP-BOX)
               end map ;
  $Expr := lst-expr-box_TO_expr-lst-box(
            id-ter () with $id := token("boxes-post") end with ,
            lst-expr-box ,
            expr-lst-box-empty) ;
  $Decls := map left decls-merge
                value decls ()
                other $Decls(PP-BOX)
            end map ;
end where ;

{- ------------------------------------------------------------------- -}
{- PP-MATCH-CLAUSES                                                    -}
{- ------------------------------------------------------------------- -}
where null-PP-MATCH-CLAUSES ->  use
  $Match-Clauses := null-MATCH-CLAUSES () ;
  $L-M-Clauses := l-m-clauses () ;
end where ;
```

```
where pp-match-clauses ->  PP-MATCH-CLAUSE + use
  $Match-Clauses := map left match-clauses-post
                        value match-clauses ()
                        other $Match-Clause (PP-MATCH-CLAUSE)
                    end map ;
  $L-M-Clauses := map left construct-m-clauses
                        value l-m-clauses ()
                        other $L-M-Clause (PP-MATCH-CLAUSE)
                  end map ;
  $Decls := map left decls-merge
                value decls ()
                other $Decls(PP-MATCH-CLAUSE)
            end map ;
end where ;

{- ------------------------------------------------------------------- -}
{- PP-MATCH-CLAUSE                                                      -}
{- ------------------------------------------------------------------- -}
where null-PP-MATCH-CLAUSE ->  use
  $Match-Clause := null-MATCH-CLAUSE () ;
  $L-M-Clause := l-m-clause () ;
end where ;

where pp-match-clause ->  PP-PATTERNS  PP-BOX  use
  $Match-Clause := match-clause($Patterns (PP-PATTERNS), block-expr($Expr (PP-BOX))) ;
  $s.L-Attr-Occ := add-attr-occ ($h.L-Attr-Occ , $L-Sem-Rule (PP-BOX)) ;
  $L-M-Clause := construct-m-clause (
                        $L-Attr-Occ ,
                        $Patterns (PP-PATTERNS) ,
                        $L-Sem-Rule (PP-BOX) ,
                        l-m-clause ()) ;
  $h.num-pplist(PP-BOX) := $h.num-pplist ;
  $s.num-pplist := $s.num-pplist(PP-BOX) ;
  $Decls := $Decls(PP-BOX) ;
end where ;

{- ------------------------------------------------------------------- -}
{- PP-PATTERNS                                                          -}
{- ------------------------------------------------------------------- -}
where null-PP-PATTERNS ->  use
  $Patterns := null-PATTERNS () ;
end where ;

where pp-patterns -> PP-PATTERN * use
  $Patterns := map left patterns-post
                 value patterns ()
                 other $Pattern (PP-PATTERN)
               end map ;
end where ;

{- ------------------------------------------------------------------- -}
{- PP-PATTERN                                                           -}
{- ------------------------------------------------------------------- -}
where null-PP-PATTERN ->  use
  $Pattern := null-PATTERN () ;
end where ;

where pp-pattern-var-or-cons -> PP-ID use
  $Pattern := pattern-var-or-const(id-ter () with $id := $pp-id (PP-ID) end with ) ;
end where ;

where pp-pattern-cons -> PP-ID PP-PATTERNS use
  $Pattern := pattern-cons(
                unqual-id(id-ter () with $id := $pp-id (PP-ID) end with ) ,
                $Patterns (PP-PATTERNS) ) ;
end where ;
```

```
where pp-pattern-const-manif -> PP-EXPR use
  $Pattern := pattern-const-manif($Expr(PP-EXPR)) ;
end where ;

where pp-single-case-const -> PP-EXPR use
  $Pattern := single-case-const($Expr(PP-EXPR)) ;
end where ;

{- ------------------------------------------------------------------ -}
{- PP-OTHER-CLAUSE                                                     -}
{- ------------------------------------------------------------------ -}
where null-PP-OTHER-CLAUSE ->  use
  $Match-Clause := null-MATCH-CLAUSE () ;
  $L-M-Clause := l-m-clause () ;
end where ;

where pp-match-clause-other ->  PP-BOX  use
  $Match-Clause := match-clause-other ($Expr (PP-BOX)) ;
  $s.L-Attr-Occ := add-attr-occ ($h.L-Attr-Occ , $L-Sem-Rule (PP-BOX)) ;
  $L-M-Clause := construct-other-clause (
                       $L-Attr-Occ ,
                       $L-Sem-Rule (PP-BOX) ,
                       l-m-clause ()) ;
  $h.num-pplist(PP-BOX) := $h.num-pplist ;
  $s.num-pplist := $s.num-pplist(PP-BOX) ;
  $Decls := $Decls (PP-BOX) ;
end where ;

{- ------------------------------------------------------------------ -}
{- PP-DIR                                                              -}
{- ------------------------------------------------------------------ -}
where null-PP-DIR ->  use
  $is-left-dir := true ;
  $Map-Dir := null-MAP-DIR () ;
end where ;

where pp-right-dir -> use
  $Map-Dir := right-dir () ;
  $is-left-dir := false ;
end where ;

where pp-left-dir -> use
  $Map-Dir := left-dir () ;
  $is-left-dir := true ;
end where ;

{- ------------------------------------------------------------------ -}
{- PP-INSERT                                                           -}
{- ------------------------------------------------------------------ -}
where null-PP-INSERT ->  use
  $is-no-insert := true ;
end where ;

where pp-insert -> PP-STRING use
  $Expr := tree-with(
            func-call(
              id-ter () with $id := token("terminal") end with ,
              actuals(actual($Expr(PP-STRING)))) ,
            decorations(
              decoration(
                attr () with $attr := token("$separator") end with ,
                applied-id(
                  unqual-id(
                    id-ter () with $id := token("no-sep") end with ))))) ;
  $is-no-insert := false ;
```

```
end where ;

where pp-no-insert ->  use
  $is-no-insert := true ;
  $Expr := tree-with(
            func-call(
              id-ter () with $id := token("terminal") end with ,
              actuals(
                actual(
                  tree-with(
                    func-call(
                      id-ter () with $id := token("string-box") end with ,
                      actuals ()),
                    decorations(
                      decoration(
                        attr () with $attr := token("$string-box") end with ,
                        string-const(string () with $string := token("") end with ))))))),
            decorations(
              decoration(
                attr () with $attr := token("$separator") end with ,
                applied-id(
                  unqual-id(
                    id-ter () with $id := token("no-sep") end with ))))) ;
end where ;

{- ------------------------------------------------------------------ -}
{- PP-EXPR                                                            -}
{- ------------------------------------------------------------------ -}
where pp-applied-id -> PP-ID use
  $Expr := applied-id(unqual-id(id-ter () with $id := $pp-id(PP-ID) end with)) ;
end where ;

where pp-real-const -> PP-REAL use
  $Expr := real-const(real () with $real := $pp-real (PP-REAL) end with ) ;
end where ;

where pp-string-const -> PP-STRING use
  $Expr := string-const(string () with $string := $pp-string(PP-STRING) end with ) ;
end where ;

where pp-char-const -> PP-CHAR use
  $Expr := char-const (char () with $char := $pp-char (PP-CHAR) end with ) ;
end where ;

where pp-int-const -> PP-INT use
  $Expr := int-const (int () with $int := $pp-int (PP-INT) end with ) ;
end where ;

where pp-attrib-occ-as-expr -> PP-ATTRIB-OCC use
  $Expr := $Expr (PP-ATTRIB-OCC) ;
end where ;

where pp-func-call ->  PP-ID  PP-ACTUALS  use
  $Expr := func-call(
            id-ter () with $id := $pp-id (PP-ID) end with ,
            $Actuals (PP-ACTUALS)) ;
end where ;

where pp-bool-or ->  PP-EXPR  PP-EXPR  use
  $Expr := bool-or($Expr(PP-EXPR.1) , $Expr(PP-EXPR.2)) ;
end where ;

where pp-bool-and ->  PP-EXPR  PP-EXPR  use
  $Expr := bool-and($Expr(PP-EXPR.1) , $Expr(PP-EXPR.2)) ;
end where ;
```

```
{- ------------------------------------------------------------------- -}
{- PP-ACTUALS                                                          -}
{- ------------------------------------------------------------------- -}
where null-PP-ACTUALS ->  use
  $Actuals := null-ACTUALS () ;
end where ;

where pp-actuals ->  PP-ACTUAL * use
  $Actuals := map left actuals-post
               value actuals ()
               other $Actual (PP-ACTUAL)
             end map ;
end where ;

{- ------------------------------------------------------------------- -}
{- PP-ACTUAL                                                           -}
{- ------------------------------------------------------------------- -}
where null-PP-ACTUAL ->  use
  $Actual := null-ACTUAL () ;
end where ;

where pp-actual ->  PP-EXPR  use
  $Actual := actual ($Expr (PP-EXPR)) ;
end where ;

{- ------------------------------------------------------------------- -}
{- PP-ATTRIB-OCC                                                       -}
{- ------------------------------------------------------------------- -}
where null-PP-ATTRIB-OCC ->  use

end where ;

where pp-attrib-occ -> PP-ATTR PP-SYMBOL use
  $Expr := attrib-occ-as-expr(
             attrib-occ(
               attr () with $attr := $pp-attr (PP-ATTR) end with ,
               $Symbol (PP-SYMBOL) )) ;
end where ;

where pp-attrib-occ-lhs -> PP-ATTR use
  $Expr := attrib-occ-as-expr(
             attrib-occ-lhs(attr () with $attr := $pp-attr (PP-ATTR) end with )) ;
end where ;

{- ------------------------------------------------------------------- -}
{- PP-SYMBOL                                                           -}
{- ------------------------------------------------------------------- -}
where null-PP-SYMBOL ->  use
  $Symbol := null-SYMBOL () ;
  $symb := error-t-symb ;
end where ;

where pp-qual-symb -> PP-ID PP-QUALIFIER use
  $Symbol := qual-symb(
               id-ter () with $id := $pp-id (PP-ID) end with ,
               $Qualifier(PP-QUALIFIER) ) ;
  $symb := t-symb (
                $pp-id (PP-ID) ,
                $kind-qual (PP-QUALIFIER) ,
                $val-qual (PP-QUALIFIER) ) ;
end where ;

where pp-unqual-symb -> PP-ID use
  $Symbol := unqual-symb(id-ter () with $id := $pp-id (PP-ID) end with ) ;
  $symb := t-symb (
                $pp-id (PP-ID) ,
```

```
                no-qual ,
                token ("0") ) ;
end where ;

{- ------------------------------------------------------------------- -}
{- PP-QUALIFIER                                                         -}
{- ------------------------------------------------------------------- -}
where null-PP-QUALIFIER ->  use
  $Qualifier := null-QUALIFIER () ;
  $val-qual := token ("0") ;
  $kind-qual := no-qual ;
end where ;

where pp-last-qual ->  use
  $Qualifier := last-qual () ;
  $kind-qual := dernier ;
  $val-qual := token ("0") ;
end where ;

where pp-first-qual ->  use
  $Qualifier := first-qual () ;
  $kind-qual := premier ;
  $val-qual := token ("0") ;
end where ;

where pp-right-qual ->  use
  $Qualifier := right-qual () ;
  $kind-qual := droit ;
  $val-qual := token ("0") ;
end where ;

where pp-left-qual ->  use
  $Qualifier := left-qual () ;
  $kind-qual := gauche ;
  $val-qual := token ("0") ;
end where ;

where pp-int-qual ->  PP-INT  use
  $Qualifier := int-qual(int () with $int := $pp-int(PP-INT) end with );
  $kind-qual := entier ;
  $val-qual := $pp-int (PP-INT) ;
end where ;

{- ------------------------------------------------------------------- -}
{- PP-STRING                                                            -}
{- ------------------------------------------------------------------- -}
where pp-string ->  use
  $Expr := tree-with(
            func-call(
              id-ter () with $id := token("string-box") end with ,
              actuals ()),
            decorations(
              decoration(
                attr () with $attr := token("$string-box") end with ,
                string-const(string () with $string := $pp-string end with )))) ;
end where ;

{- ------------------------------------------------------------------- -}
{- PP-PHYLUM                                                            -}
{- ------------------------------------------------------------------- -}
where null-PP-PHYLUM ->  use
  $Phylum := null-PHYLUM () with $phylum := token("erreur-token") end with ;
end where ;

where pp-phylum ->  use
  $Phylum := phylum () with $phylum := $pp-phylum end with ;
```

```
end where ;

{- ------------------------------------------------------------------- -}
{- il n'y a pas de regles pour les phylums suivants:                    -}
{- PP-ID PP-SEP-ID PP-INT PP-CHAR PP-REAL PP-ATTR                       -}
{- ------------------------------------------------------------------- -}

end grammar ;
```

A.C Modules de déclarations et de définitions

A.C-1 dec_ppat-predef.olga

```
{  ------------------------------------
   -    module de definition des types    -
   -        et des constantes predefinies -
   -        de ppat                       -
   ----------------------------------    }

declaration module ppat-predef is
type
  sep-param = list of int ;

  sep-kind = ( h , v , hv , hov , error ) ;

  sep-info = record
               kind :               sep-kind ;
               parametres :       sep-param ;
            end record ;
const
  no-sep := sep-info (error , sep-param ()) ;
  sep-null := sep-info (h , sep-param(0)) ;
end module ;
```

A.C-2 dec_ppat-types.olga

```
{  ------------------------------------
   -    module de declaration pour le    -
   -         traitement des separateurs  -
   -        et des types devant etre     -
   -        affiches                     -
   ----------------------------------   }

declaration module ppat-types is
  from ppat-predef import all ;
  from attr_typage import all ;
type
  sep = record
          name :                token ;
          caracteristiques :        sep-info ;
        end record ;

  st-sep = list of sep ;

{--- definition des types permettant la manipulation des Types To Be Printed ---}
  st-type-correct = record
                        correct : bool ;
                        types :   st_type_olga ;
                    end record ;

{--- declaration des fonctions de manipulation des separateurs ---}
function lookup-SEP (
                key : token ; { nom du separateur }
```

```
                st : st-sep   { liste des separateurs deja definis }
        ) : bool ;            { retourne true si present et false sinon }

function lookup-sep (
                key : token ; { nom du separateur }
                st : st-sep   { liste des separateurs deja definis }
        ) : sep-info ;        { retourne les informations relatives au sep }

function verif-sep (
                elt : sep-info ;
                scx : source-index { position dans le texte }
        ) : bool ;                 { verifie coherence type<->parametres }

function find-sep-kind (key : token) : sep-kind ; { retourne le type d'un separateur }

{--- declaration des fonctions de manipulation des Types To Be Printed ---}
function rec-lookup-TTBP (
                key : info_name_qua ;  { type recherche }
                ttbp : st_type_olga ;  { liste des TTBP }
                st : symbol_table_type { table des types }
        ) : st-type-correct ; { retourne la liste des nouveaux ttbp}
                              { et true si pas d'erreurs }

function lookup-ttbp(key : info_name_qua ; ttbp : st_type_olga) : bool ;

function rec-lookup-ttbp-record (
                info : liste_formal ;
                result : st-type-correct ;
                st : symbol_table_type ) : st-type-correct ;

function rec-lookup-ttbp-union (
                info : liste_type ;
                result : st-type-correct ;
                st : symbol_table_type ) : st-type-correct ;

function rec-lookup-ttbp-list-set (
                info : type_olga ;
                result : st-type-correct ;
                st : symbol_table_type ) : st-type-correct ;

{--- declaration de fonctions auxilliaires ---}
function look-champ-record (
                key : info_name ;
                formals : liste_formal ) : bool ;

function look-CHAMP-RECORD (key : info_name ; formals : liste_formal) : type_olga ;

{--- fonctions externes de bon deroulement ---}
function extern_regle_type () : bool   ; {affichage d'un "t" pour chaque regle de type}
function extern_regle_op () : bool ; {affichage d'un "o" pour chaque regle d'operateur}
end module ;
```

A.C-3 def_ppat-types.olga

```
{  -------------------------------------------------------
   - Module de definition pour ppat-check                -
   ------------------------------------------------------- }

definition module ppat-types is
  from message_olga import all ;
function lookup-SEP (key : token ; st : st-sep) : bool is
  case st is
        st-sep (sep (n : token , c : sep-info)) :
          if (n = key ) then
            true
```

```
          else
            false
          end if ;
        st-sep (sep (n : token , c : sep-info) , ST : st-sep) :
          if (n = key) then
            true
          else
            lookup-SEP (key , ST)
          end if ;
        other : false ;
  end case
end function ;

function lookup-sep (key : token ; st : st-sep ) : sep-info is
  case st is
        st-sep (sep (n , c)) :
          if (n = key) then
            c
          else
           sep-info (error , sep-param ())
          end if ;
        st-sep (sep (n , c) , ST : st-sep) :
          if (n = key) then
            c
          else
            lookup-sep (key , ST)
          end if ;
        other : sep-info (error , sep-param ()) ;
  end case
end function ;

function verif-sep (elt : sep-info ;
                    scx : source-index ) : bool is
  let
    k : sep-kind := elt.kind ,
    P : sep-param := elt.parametres
  in
    case k is
      h :
        case P is
          sep-param (dx : int) : true ;
          other :
                message message_olga(97)
                position scx
                severity 2
                value false ;
        end case ;
      v :
        case P is
          sep-param(di : int , sep-param (dh : int)) : true ;
          other :
                message message_olga(97)
                position scx
                severity 2
                value  false ;
        end case ;
      hv :
        case P is
          sep-param ( dx : int , sep-param ( di : int , sep-param (dh : int))) : true ;
          other :
                message message_olga(97)
                position scx
                severity 2
                value false ;
        end case ;
     hov :
```

```
        case P is
          sep-param ( dx : int , sep-param ( di : int , sep-param (dh : int))) : true ;
          other :
                message message_olga(97)
                position scx
                severity 2
                value false ;
        end case ;
    other :
        message message_olga(98)
        position scx
        severity 2
        value false ;
  end case
end function ;

function find-sep-kind (key : token) : sep-kind is
  if key = token("h") then h
  elsif key = token("v") then v
  elsif key = token("hv") then hv
  elsif key = token("hov") then hov
  else error
  end if
end function ;

function ttbp-predef (
                t : type_olga ;
                def : bool ;
                ttbp : st_type_olga ) : st-type-correct is
  if def then
  { si le type est deja dans ttbp }
    st-type-correct (true , ttbp)
  else
  { si le type n'est pas dans ttbp }
    st-type-correct (true , st_type_olga (ttbp , t))
  end if
end function ;

function rec-lookup-TTBP (
                key : info_name_qua ;
                ttbp : st_type_olga ;
                st : symbol_table_type) : st-type-correct is

  if lookup_in_block_TYPE (key , st) then
  { si le type existe dans la table des types }
    let
      t : type_olga := lookup_TYPE (key , st) ,
      def : bool := lookup-ttbp (key , ttbp)
    in
      case t.info is
        info_type (X : info_enumered) :
          ttbp-predef (t , def , ttbp) ;
        info_type (X : info_subrange) :
          ttbp-predef (t , def , ttbp) ;
        info_type (X : info_asx_type) :
          ttbp-predef (t , def , ttbp) ;
        info_type (X : info_opaque) :
          ttbp-predef (t , def , ttbp) ;
        info_type (X : info_record) :
          rec-lookup-ttbp-record(
            X.liste_formal ,
            if def then
              st-type-correct (true , ttbp)
            else
              st-type-correct (true , st_type_olga (ttbp , t))
            end if ,
```

```
            st ) ;
        info_type (X : info_union) :
          rec-lookup-ttbp-union(
            X.liste_type ,
            if def then
              st-type-correct (true , ttbp)
            else
              st-type-correct (true , st_type_olga (ttbp , t))
            end if ,
            st ) ;
        info_type (X : info_list) :
          rec-lookup-ttbp-list-set(
            X.type_olga ,
            if def then
              st-type-correct (true , ttbp)
            else
              st-type-correct (true , st_type_olga (ttbp , t))
            end if ,
            st ) ;
        info_type (X : info_set) :
          rec-lookup-ttbp-list-set(
            X.type_olga ,
            if def then
              st-type-correct (true , ttbp)
            else
              st-type-correct (true , st_type_olga (ttbp , t))
            end if ,
            st ) ;
        other :
          st-type-correct (false ,ttbp) ;
      end case
  else
  { le type n'existe pas dans la table des types }
    st-type-correct ( false , ttbp)
  end if
end function ;

and

function lookup-ttbp (
                key : info_name_qua ;
                ttbp : st_type_olga ) : bool is
  case ttbp is
    st_type_olga ( T : type_olga , ST : st_type_olga) :
      if (T.name = key) then
        true
      else
        lookup-ttbp (key , ST)
      end if ;
    st_type_olga (T : type_olga) :
      if (T.name = key) then
        true
      else
        false
      end if ;
    other : false ;
  end case
end function ;

and

function rec-lookup-ttbp-record (
                info : liste_formal ;
                result : st-type-correct ;
                st : symbol_table_type ) : st-type-correct is
  case info is
```

```
    liste_formal (F : formal , SF : liste_formal) :
      rec-lookup-ttbp-record (
        SF ,
        rec-lookup-TTBP (
          (F.type_olga).name ,
          result.types ,
          st ) ,
        st ) ;
    liste_formal (F : formal) :
      rec-lookup-TTBP (
        (F.type_olga).name ,
        result.types ,
        st ) ;
    other :
      result ;
  end case
end function ;

and

function rec-lookup-ttbp-union (
                info : liste_type ;
                result : st-type-correct ;
                st : symbol_table_type ) : st-type-correct is
  case info is
    liste_type (T : type_olga , ST : liste_type) :
      rec-lookup-ttbp-union (
        ST ,
        rec-lookup-TTBP (
          T.name ,
          result.types ,
          st ) ,
        st ) ;
    liste_type (T : type_olga) :
      rec-lookup-TTBP (
        T.name ,
        result.types ,
        st ) ;
    other :
      result ;
  end case
end function ;

and

function rec-lookup-ttbp-list-set (
                info : type_olga ;
                result : st-type-correct ;
                st : symbol_table_type ) : st-type-correct is
  rec-lookup-TTBP (info.name , result.types , st )
end function ;

function look-champ-record (key : info_name ; formals : liste_formal) : bool is
  case formals is
    liste_formal (F : formal , SF : liste_formal) :
      if (F.name = key) then
        true
      else
        look-champ-record(key , SF)
      end if ;
    liste_formal (F : formal) :
      if (F.name = key) then
        true
      else
        false
      end if ;
```

```
    other : false ;
  end case
end function ;

function look-CHAMP-RECORD (key : info_name ; formals : liste_formal) : type_olga is
  case formals is
    liste_formal (F : formal , SF : liste_formal) :
      if (F.name = key) then
        F.type_olga
      else
        look-CHAMP-RECORD(key , SF)
      end if ;
    liste_formal (F : formal) :
      if (F.name = key) then
        F.type_olga
      else
        error-type-info
      end if ;
    other : error-type-info ;
  end case
end function ;

function extern_regle_type () : bool is external ;
function extern_regle_op () : bool is external ;
end module ;
```

Annexe B

Spécifications de la phase d'exécution

B.A Grammaires abstraites et concrètes

B.A-1 boxes-in.asx

```
{  ------------------------------------------------------
   -   syntaxe abstraite d'un arbre de boites decore par:  -
   -     * les separateurs                                 -
   -     * le contenu des chaines                          -
   ------------------------------------------------------- }

grammar boxes-in is
from ppat-predef import all ;

root is PPAT ;

  $separator        (BOX)              : sep-info ;
  $string-box       (STRING-BOX)       : string ;

PPAT = ppat ;
ppat -> BOX ;

BOX = box terminal empty-box ;
box -> BOXES ;
terminal -> STRING-BOX ;
empty-box -> ;

BOXES = boxes ;
boxes -> BOX * ;

STRING-BOX = string-box ;
string-box -> ;

end grammar ;
```

B.A-2 boxes-out.asx

```
{  ------------------------------------------------------
   -   syntaxe abstraite d'un arbre de boites decore par:  -
   -     * les separateurs                                 -
   -     * le contenu des chaines                          -
   -     * les coordonnees et les dimensions des boites    -
   ------------------------------------------------------- }
```

```
grammar boxes-out is
import grammar boxes-in ;
from ppat-predef import all ;
from calcul-coord-box import all ;

root is PPAT ;
  $Coord        (PPAT , BOX)        : coord-box;
end grammar ;
```

B.B Grammaires attribuées

B.B-1 ag_boxes-coord.olga

```
{  -------------------------------------------------------
   -   premiere ga du run time de ppat                    -
   -   calcul des coordonnees des boites                  -
   ------------------------------------------------------- }

attribute grammar boxes-coord (boxes-in,boxes-out) is
from ppat-predef import all ;
from calcul-coord-box import all ;

attribute
 synthesized $DIM-nat(BOX) : dim-box ;
 synthesized $L-DIM-nat(BOXES) : l-dim-box ;
 inherited $XY(BOX,BOXES) : xy-box ;
 synthesized $Coord(PPAT,BOX) : coord-box ;
 synthesized $L-Coord(BOXES) : l-coord-box ;
 inherited $sep(BOXES) : sep-info ;
 inherited $is-vert(BOXES) : bool ;

{- --------------------------------------------------------------------- -}
{- PPAT                                                                  -}
{- --------------------------------------------------------------------- -}
where null-PPAT ->  use
  $Coord := coord-nul ;
end where ;

where ppat ->  BOX  use
  $XY(BOX) := xy-nul ;
  $Coord := $Coord (BOX) ;
end where ;

{- --------------------------------------------------------------------- -}
{- BOX                                                                   -}
{- --------------------------------------------------------------------- -}
where null-BOX ->  use
  $DIM-nat := dim-nul ;
  $Coord := coord-box($XY , dim-nul) ;
end where ;

where empty-box ->  use
  $DIM-nat := dim-nul ;
  $Coord := coord-box($XY , dim-nul) ;
end where ;

where terminal ->  STRING-BOX  use
  $DIM-nat := dim-box(length($string-box(STRING-BOX)) ,
                      1 ,
                      length($string-box(STRING-BOX))) ;
  $Coord := coord-box($XY ,
                      dim-box(length($string-box(STRING-BOX)) ,
                              1 ,
```

```
                              length($string-box(STRING-BOX)))) ;
end where ;

where box ->  BOXES  declare
value
  dimensions-naturelles : dim-box := DIM-NAT($separator , $L-DIM-nat(BOXES)) ;
  is-vert : bool := if $XY.X + dimensions-naturelles.W < largeur-page then
                      false
                   else
                      true
                   end if ;
use
  $DIM-nat := dimensions-naturelles ;
  $Coord := coord-box($XY ,
                     DIM($L-Coord (BOXES) , $XY , dim-nul , 0)) ;
  $sep(BOXES) := $separator ;
  $XY(BOXES) := $XY ;
  $is-vert(BOXES) := is-vert ;
end where ;

{- ------------------------------------------------------------------- -}
{- BOXES                                                               -}
{- ------------------------------------------------------------------- -}


where null-BOXES ->  use
  $L-DIM-nat := l-dim-box () ;
  $L-Coord := l-coord-box () ;
end where ;

where boxes ->  BOX * use
  $L-DIM-nat := map left l-dim-box
                value l-dim-box ()
                other $DIM-nat (BOX)
               end map ;
  $XY (BOX) := case position is
                first : XY($sep ,
                           $DIM-nat(BOX) ,
                           $is-vert ,
                           true ,
                           coord-box($XY, dim-nul) ,
                           $XY) ;
                other : XY($sep ,
                           $DIM-nat(BOX) ,
                           $is-vert ,
                           false ,
                           $Coord(BOX.left) ,
                           $XY) ;
               end case ;
  $L-Coord := map left l-coord-box
                value l-coord-box ()
                other $Coord(BOX)
              end map ;
end where ;

{- ------------------------------------------------------------------- -}
{- pas de regles pour STRING-BOX                                       -}
{- ------------------------------------------------------------------- -}
end grammar ;
```

B.C Modules de déclarations et de définitions

B.C-1 dec_calcul-coord-box.olga

```
{ -----------------------------------------------------
   -  Module de declaration pour le calcul            -
   -  des coordonnees des boites                      -
   ----------------------------------------------------- }
declaration module calcul-coord-box is
  from ppat-predef import all ;

type
  dim-box = record
               W        : int ;
               H        : int ;
               Wlast    : int ;
  end record ;

  l-dim-box = list of dim-box ;

  xy-box = record
               X        : int ;
               Y        : int ;
  end record ;

  coord-box = record
               COORD      : xy-box ;
               DIM        : dim-box ;
  end record ;

  l-coord-box = list of coord-box ;

function extern-largeur-page () : int ;
function extern-ppat-name () : string ;

function DIM-NAT (separator : sep-info ; L-DIM : l-dim-box) : dim-box ;
function DIM (L-Coord : l-coord-box ;
             XY-Top : xy-box ;
             Result : dim-box ;
             dernier : int) : dim-box ;
function XY (separator : sep-info ;
             DIM-nat : dim-box ;
             is-vert , is-first : bool ;
             Coord-left : coord-box ;
             XY-top : xy-box ) : xy-box ;

const
  largeur-page := extern-largeur-page () ;
  dim-nul := dim-box(0,0,0) ;
  xy-nul := xy-box(0,0) ;
  coord-nul := coord-box(xy-nul , dim-nul) ;
end module ;
```

B.C-2 def_calcul-coord-box.olga

```
{ -----------------------------------------------------
   -  Module de definition pour le calcul              -
   -  des coordonnees des boites                      -
   ----------------------------------------------------- }
definition module calcul-coord-box is

function extern-largeur-page () : int is external ;
function longueur (chaine : string) : int is external ;
function extern-ppat-name () : string is external ;
```

```
function para1(parametres : sep-param) : int is
  head(parametres)
end function ;

function para2(parametres : sep-param) : int is
  head(tail(parametres))
end function ;

function para3(parametres : sep-param) : int is
  head(tail(tail(parametres)))
end function ;

function max(l1 , l2 : int) : int is
  if (l1 > l2) then
        l1
  else
        l2
  end if
end function ;

function DIM-NAT-h(dx : int ;
                   L-DIM : l-dim-box ;
                   is-first : bool ;
                   Result : dim-box) : dim-box is
  case L-DIM is
    l-dim-box(DIM : dim-box , L-DIM-T : l-dim-box) :
      if is-first then
        DIM-NAT-h(dx , L-DIM-T , false , DIM)
      else
        DIM-NAT-h(dx , L-DIM-T , false ,
                  dim-box(max(Result.W , Result.Wlast + dx + DIM.W) ,
                          Result.H
                            + DIM.H
                            - if (Result.H < 1) then 0 else 1 end if ,
                          Result.Wlast + dx + DIM.Wlast))
      end if ;
    l-dim-box(DIM : dim-box) :
      if is-first then
        DIM
      else
        dim-box(max(Result.W , Result.Wlast + dx + DIM.W) ,
                Result.H
                  + DIM.H
                  - if (Result.H < 1) then 0 else 1 end if ,
                Result.Wlast + dx + DIM.Wlast)
      end if ;
    other :
      Result ;
  end case
end function ;

function DIM-NAT-v(di , dh : int ; L-DIM : l-dim-box ; Result : dim-box) : dim-box is
  case L-DIM is
    l-dim-box(DIM : dim-box , L-DIM-T : l-dim-box) :
      DIM-NAT-v(di , dh , L-DIM-T ,
               dim-box(max(Result.W , DIM.W) , Result.H + DIM.H + dh , 0)) ;
    l-dim-box(DIM : dim-box) :
      dim-box(max(Result.W , DIM.W) + di , Result.H + DIM.H , DIM.Wlast + di );
    other :
      dim-box(Result.W + di , Result.H , Result.Wlast + di) ;
  end case
end function ;

function DIM-NAT(separator : sep-info ; L-DIM : l-dim-box) : dim-box is
  if (separator.kind = h) | (separator.kind = hv) | (separator.kind = hov) then
    DIM-NAT-h(para1(separator.parametres) , L-DIM , true , dim-nul)
```

```
  elsif separator.kind = v then
    DIM-NAT-v(para1(separator.parametres) ,
             para2(separator.parametres) ,
             L-DIM , dim-nul)
  else
    dim-nul
  end if
end function ;

function DIM(L-Coord : l-coord-box ;
             XY-Top : xy-box ;
             Result : dim-box ;
             dernier : int) : dim-box is
  case L-Coord is
    l-coord-box(C : coord-box , L-Coord-T : l-coord-box) :
        DIM(L-Coord-T , XY-Top ,
            dim-box(max(Result.W , C.COORD.X - XY-Top.X + C.DIM.W) ,
                    max(Result.H , C.COORD.Y - XY-Top.Y + C.DIM.H) ,
                    if (C.COORD.Y + C.DIM.H) = dernier then
                        max(Result.Wlast , C.COORD.X - XY-Top.X + C.DIM.Wlast)
                    elsif (C.COORD.Y + C.DIM.H) > dernier then
                        C.COORD.X - XY-Top.X + C.DIM.Wlast
                    else
                        Result.Wlast
                    end if) ,
            max(dernier , C.COORD.Y + C.DIM.H)) ;
    l-coord-box(C : coord-box) :
        dim-box(max(Result.W , C.COORD.X - XY-Top.X + C.DIM.W) ,
                max(Result.H , C.COORD.Y - XY-Top.Y + C.DIM.H) ,
                if (C.COORD.Y + C.DIM.H) = dernier then
                    max(Result.Wlast , C.COORD.X - XY-Top.X + C.DIM.Wlast)
                elsif (C.COORD.Y + C.DIM.H) > dernier then
                    C.COORD.X - XY-Top.X + C.DIM.Wlast
                else
                    Result.Wlast
                end if) ;
    other :
        Result ;
  end case
end function ;

function XY (separator : sep-info ;
             DIM-nat : dim-box ;
             is-vert , is-first : bool ;
             Coord-left : coord-box ;
             XY-Top : xy-box) : xy-box is
  if separator.kind = h then
    let
      dx : int := para1(separator.parametres)
    in
      if is-first then
        XY-Top
      else
        xy-box(Coord-left.COORD.X + Coord-left.DIM.Wlast + dx ,
               Coord-left.COORD.Y
                + Coord-left.DIM.H
                - if (Coord-left.DIM.H != 0) then 1 else 0 end if)
      end if
  elsif separator.kind = v then
    let
      di : int := para1(separator.parametres) ,
      dh : int := para2(separator.parametres)
    in
      if is-first then
        xy-box(XY-Top.X + di , XY-Top.Y)
      else
```

```
        xy-box(XY-Top.X + di ,
               Coord-left.COORD.Y + Coord-left.DIM.H + dh)
      end if
  elsif separator.kind = hv then
    let
      dx : int := para1(separator.parametres) ,
      di : int := para2(separator.parametres) ,
      dh : int := para3(separator.parametres)
    in
      if is-first then
        XY-Top
      elsif (Coord-left.COORD.X + Coord-left.DIM.Wlast + dx + DIM-nat.W)
             < largeur-page
      then
        xy-box(Coord-left.COORD.X + Coord-left.DIM.Wlast + dx ,
               Coord-left.COORD.Y
                + Coord-left.DIM.H
                - if (Coord-left.DIM.H != 0) then 1 else 0 end if)
      else
        xy-box(XY-Top.X + di ,
               Coord-left.COORD.Y + Coord-left.DIM.H + dh)
      end if
  elsif separator.kind = hov then
    let
      dx : int := para1(separator.parametres) ,
      di : int := para2(separator.parametres) ,
      dh : int := para3(separator.parametres)
    in
      if is-first then
        if is-vert then
          xy-box(XY-Top.X + di , XY-Top.Y)
        else
          XY-Top
        end if
      else
        if is-vert then
          xy-box(XY-Top.X + di ,
                 Coord-left.COORD.Y + Coord-left.DIM.H + dh)
        else
          xy-box(Coord-left.COORD.X + Coord-left.DIM.Wlast + dx ,
                 Coord-left.COORD.Y
                  + Coord-left.DIM.H
                  - if (Coord-left.DIM.H != 0) then 1 else 0 end if)
        end if
      end if
  else
    xy-nul
  end if
end function ;
end module ;
```

Annexe C

Exemple issu de l'auto-amorçage

C.A Spécification PPAT de décompilation d'un arbre `olga-simpli`

```
ppat ppat_olga-simpli (olga-simpli) ;
context
        $sep(ACTUALS , ACTUAL)        : string ;
        $is-bin(ACTUALS , ACTUAL )    : int ;
        $is-last(ACTUAL)              : bool ;
separator
  <h0> = <h 0> ;
  <v00> = <v 0 , 0> ;
  <v01> = <v 0 , 1> ;
  <v04> = <v 0 , 4> ;
  <v20> = <v 2 , 0> ;
  <hv020> = <hv 0 , 2 , 0> ;
  <hv100> = <hv 1 , 0 , 0> ;
  <hv120> = <hv 1 , 2 , 0> ;
  <hov100> = <hov 1 , 0 , 0> ;
  <hov120> = <hov 1 , 2 , 0> ;

{- ------------------------------------------------------------------ -}
{- ID                                                                 -}
{- ------------------------------------------------------------------ -}
where unqual-id ->  ID-TER  pprint
        ID-TER
end where ;

{- ------------------------------------------------------------------ -}
{- ID-TER                                                             -}
{- ------------------------------------------------------------------ -}
where id-ter ->  pprint
        $id
end where ;

{- ------------------------------------------------------------------ -}
{- TYPE-ID                                                            -}
{- ------------------------------------------------------------------ -}
where type-id ->  ID  pprint
        ID
end where ;

{- ------------------------------------------------------------------ -}
{- COMPILATION-UNIT                                                   -}
```

```
{- ------------------------------------------------------------------ -}
where simple-AG ->  AG-HEADER  HOOK  pprint
        [v01 AG-HEADER HOOK "end grammar ;" ""]
end where ;

{- ------------------------------------------------------------------ -}
{- HOOK                                                               -}
{- ------------------------------------------------------------------ -}
where hook-simple-AG ->  DECLS  BLOCK  pprint
        [v04 DECLS BLOCK]
end where ;

{- ------------------------------------------------------------------ -}
{- AG-HEADER                                                          -}
{- ------------------------------------------------------------------ -}
where AG-header ->  ID-TER  AG-PROFILE  pprint
        [hv120 "attribute grammar" ID-TER AG-PROFILE "is"]
end where ;

{- ------------------------------------------------------------------ -}
{- AG-PROFILE                                                         -}
{- ------------------------------------------------------------------ -}
where functional-AG ->  SYNTAX  SYNTAXES  pprint
        [hv120 "(" SYNTAX ")" ":" "(" SYNTAXES ")"]
end where ;

{- ------------------------------------------------------------------ -}
{- SYNTAXES                                                           -}
{- ------------------------------------------------------------------ -}
where syntaxes ->  SYNTAX * pprint
        pplist left
                global hov100
                insert ","
                subbox SYNTAX
        end pplist
end where ;

{- ------------------------------------------------------------------ -}
{- SYNTAX                                                             -}
{- ------------------------------------------------------------------ -}
where syntax ->  ID-TER  pprint
        ID-TER
end where ;

{- ------------------------------------------------------------------ -}
{- SEMANTICS                                                          -}
{- ------------------------------------------------------------------ -}
where productions ->  PRODUCTION-GROUP * pprint
        pplist left
                global v01
                subbox PRODUCTION-GROUP
        end pplist
end where ;

{- ------------------------------------------------------------------ -}
{- DECLS                                                              -}
{- ------------------------------------------------------------------ -}
where decls ->  DECL * pprint
        pplist left
                global v00
                subbox DECL
        end pplist
end where ;

{- ------------------------------------------------------------------ -}
{- DECL                                                               -}
```

```
{- ------------------------------------------------------------------ -}
where attribute-decl ->  ATTRIBUTE-KIND  ATTR  PHYLA-NAMES  TYPE-ID  pprint
        [v00 "attribute"
             [v20 [hv120 ATTRIBUTE-KIND ATTR "(" PHYLA-NAMES ")" ":" TYPE-ID ";"]]]
end where ;

where value-decl ->  ID-TER  TYPE-ID  BLOCK  pprint
        [v00 "value"
             [v20 [hv120 ID-TER ":" TYPE-ID ":=" BLOCK ";"]]]
end where ;

where const-decl ->  ID-TER  BLOCK  pprint
        [v00 "const"
            [v20 [hv120 ID-TER ":=" BLOCK ";"]]]
end where ;

where rec-funcs-defs ->  FUNC-DEF * pprint
        pplist left
                global v01
                insert " and"
                subbox FUNC-DEF
        end pplist
end where ;

where simple-func-def ->  FUNC-DEF  pprint
        FUNC-DEF
end where ;

where import-all ->  MODULE-ID  pprint
        [hv120 "from" MODULE-ID "import all ;"]
end where ;

{- ------------------------------------------------------------------ -}
{- MODULE-ID                                                          -}
{- ------------------------------------------------------------------ -}
where module-id ->  pprint
        $module
end where ;

{- ------------------------------------------------------------------ -}
{- PHYLA-NAMES                                                        -}
{- ------------------------------------------------------------------ -}
where phyla-names ->  PHYLUM * pprint
        pplist left
                global hov100
                insert ","
                subbox PHYLUM
        end pplist
end where ;

{- ------------------------------------------------------------------ -}
{- PHYLUM                                                             -}
{- ------------------------------------------------------------------ -}
where phylum ->  pprint
        $phylum
end where ;

{- ------------------------------------------------------------------ -}
{- FUNC-DEF                                                           -}
{- ------------------------------------------------------------------ -}
where func-def ->  FUNC-HEADER  FUNC-BODY  pprint
        [v00 FUNC-HEADER [hv120 FUNC-BODY ";"]]
end where ;

{- ------------------------------------------------------------------ -}
{- FUNC-HEADER                                                        -}
```

```
{- ---------------------------------------------------------------- -}
where func-header -> CALL-MODE ID-TER FORMALS TYPE-ID EXCS pprint
        [hv120 "function" ID-TER "(" FORMALS ")" " :" TYPE-ID "is"]
end where ;

{- ---------------------------------------------------------------- -}
{- FORMALS                                                          -}
{- ---------------------------------------------------------------- -}
where formals -> FORMAL * pprint
        pplist left
                global hov100
                insert ";"
                subbox FORMAL
        end pplist
end where ;

{- ---------------------------------------------------------------- -}
{- FORMAL                                                           -}
{- ---------------------------------------------------------------- -}
where formal -> ID-TER TYPE-ID pprint
        [hv100 ID-TER ":" TYPE-ID]
end where ;

{- ---------------------------------------------------------------- -}
{- FUNC-BODY                                                        -}
{- ---------------------------------------------------------------- -}
where decls-and-body -> DECLS BLOCK pprint
        [v01 DECLS [v00 BLOCK "end function"]]
end where ;

where simple-body -> BLOCK pprint
        [v00 BLOCK "end function"]
end where ;

{- ---------------------------------------------------------------- -}
{- ATTRIBUTE-KIND                                                   -}
{- ---------------------------------------------------------------- -}
where syn-kind -> pprint
        "synthesized"
end where ;

where inh-kind -> pprint
        "inherited"
end where ;

{- ---------------------------------------------------------------- -}
{- PRODUCTION-GROUP                                                 -}
{- ---------------------------------------------------------------- -}
where prod-group-with-decls -> PRODUCTION DECLS SEM-RULES pprint
        [v00 [hv120 "where" PRODUCTION "declare"]
             [v20 DECLS]
             "use"
             [v20 SEM-RULES]
             "end where ;"]
end where ;

where simple-prod-group -> PRODUCTION SEM-RULES pprint
        [v00 [hv120 "where" PRODUCTION "use"]
             [v20 SEM-RULES]
             "end where ;"]
end where ;

{- ---------------------------------------------------------------- -}
{- PRODUCTION                                                       -}
{- ---------------------------------------------------------------- -}
where fixed-prod -> ID-TER PROD-RHS pprint
```

```
        [hv120 ID-TER "->" PROD-RHS]
end where ;

where pe-list-prod ->  ID-TER  PHYLUM  pprint
        [hv120 ID-TER "->" PHYLUM "*"]
end where ;

where ne-list-prod ->  ID-TER  PHYLUM  pprint
        [hv120 ID-TER "->" PHYLUM "+"]
end where ;

{- ------------------------------------------------------------------ -}
{- PROD-RHS                                                            -}
{- ------------------------------------------------------------------ -}
where prod-rhs ->  PHYLUM * pprint
        pplist left
                global hov100
                subbox PHYLUM
        end pplist
end where ;

{- ------------------------------------------------------------------ -}
{- SEM-RULES                                                           -}
{- ------------------------------------------------------------------ -}
where sem-rules ->  SEM-RULE * pprint
        pplist left
                global v00
                subbox SEM-RULE
        end pplist
end where ;

{- ------------------------------------------------------------------ -}
{- SEM-RULE                                                            -}
{- ------------------------------------------------------------------ -}
where simple-rule ->  ATTRIB-OCC  EXPR  pprint
        [hv120 ATTRIB-OCC ":=" EXPR ";"]
end where ;

{- ------------------------------------------------------------------ -}
{- ATTRIB-OCC                                                          -}
{- ------------------------------------------------------------------ -}
where attrib-occ ->  ATTR  SYMBOL  pprint
        [hv020 ATTR "(" SYMBOL ")"]
end where ;

where attrib-occ-lhs ->  ATTR  pprint
        ATTR
end where ;

{- ------------------------------------------------------------------ -}
{- ATTR                                                                -}
{- ------------------------------------------------------------------ -}
where attr ->  pprint
        $attr
end where ;

{- ------------------------------------------------------------------ -}
{- SYMBOL                                                              -}
{- ------------------------------------------------------------------ -}
where qual-symb ->  ID-TER  QUALIFIER  pprint
        [hv020 ID-TER "." QUALIFIER]
end where ;

where unqual-symb ->  ID-TER  pprint
        ID-TER
end where ;
```

```
{- ------------------------------------------------------------------ -}
{- QUALIFIER                                                           -}
{- ------------------------------------------------------------------ -}
where last-qual ->  pprint
        "last"
end where ;

where first-qual ->  pprint
        "first"
end where ;

where right-qual ->  pprint
        "right"
end where ;

where left-qual ->  pprint
        "left"
end where ;

where up-qual ->  pprint
        "up"
end where ;

where int-qual ->  INT  pprint
        INT
end where ;

{- ------------------------------------------------------------------ -}
{- INT                                                                 -}
{- ------------------------------------------------------------------ -}
where int ->  pprint
        $int
end where ;

{- ------------------------------------------------------------------ -}
{- CHAR                                                                -}
{- ------------------------------------------------------------------ -}
where char ->  pprint
        $char
end where ;

{- ------------------------------------------------------------------ -}
{- REAL                                                                -}
{- ------------------------------------------------------------------ -}
where real ->  pprint
        $real
end where ;

{- ------------------------------------------------------------------ -}
{- STRING                                                              -}
{- ------------------------------------------------------------------ -}
where string ->  pprint
        [h0 "\\\"" $string "\\\""]
end where ;

{- ------------------------------------------------------------------ -}
{- BLOCK                                                               -}
{- ------------------------------------------------------------------ -}
where block-semantics ->  SEMANTICS  pprint
        SEMANTICS
end where ;

where block-expr ->  EXPR  pprint
        EXPR
end where ;
```

```
{- ------------------------------------------------------------------ -}
{- EXPR                                                                -}
{- ------------------------------------------------------------------ -}
where tree-with ->  EXPR  DECORATIONS  pprint
        [hov100 EXPR
                "with" DECORATIONS "end with"]
end where ;

where map-expr ->  MAP-DIR  ID  EXPR  ACTUALS  pprint
        [v00 [hv120 "map" MAP-DIR ID]
             [v20 "value" <hv120> EXPR
                  "other" <hv120> ACTUALS($sep := "," , $is-bin := 0)]
            "end map"]
end where ;

where case-tree ->  SYMBOL  MATCH-CLAUSES  MATCH-CLAUSE  pprint
        [v00 [hv120 "case tree" SYMBOL "is"]
             [v20 MATCH-CLAUSES
                  MATCH-CLAUSE]
            "end case"]
end where ;

where case-match ->  EXPR  MATCH-CLAUSES  MATCH-CLAUSE  pprint
        [v00 [hv120 "case" EXPR "is"]
             [v20 MATCH-CLAUSES
                MATCH-CLAUSE]
            "end case"]
end where ;

where conditional ->  EXPR  EXPR  ELSIF-PART  EXPR  pprint
        [v00 [hv120 "if" EXPR.1 "then"]
             [v20 EXPR.2]
             ELSIF-PART
             "else"
             [v20 EXPR.3]
             "end if"]
end where ;

where let-expr ->  LET-CLAUSES  BLOCK  pprint
        [v00 "let"
             LET-CLAUSES
             "in"
             [v20 BLOCK]]
end where ;

where bool-or ->  EXPR  EXPR  pprint
        [hv120 EXPR.1 "or" EXPR.2]
end where ;

where bool-and ->  EXPR  EXPR  pprint
        [hv120 EXPR.1 "and" EXPR.2]
end where ;

where list-constr ->  TYPE-ID  LIST-ELTS  pprint
        [hv120 TYPE-ID "[|" LIST-ELTS "|]"]
end where ;

where func-call ->  ID-TER  ACTUALS  pprint
        if ($id(ID-TER) = token("*"))
          |($id(ID-TER) = token("/"))
          |($id(ID-TER) = token("%"))
          |($id(ID-TER) = token("="))
          |($id(ID-TER) = token("!="))
          |($id(ID-TER) = token("~"))
          |($id(ID-TER) = token("<"))
```

```
          |($id(ID-TER) = token(">"))
          |($id(ID-TER) = token("<="))
          |($id(ID-TER) = token(">="))
          |($id(ID-TER) = token("&"))
          |($id(ID-TER) = token("|")) then
                [hv120 "("
                        ACTUALS($sep := string($id(ID-TER)) , $is-bin := 2)
                        ")"]
        elsif ($id(ID-TER) = token("+"))
             |($id(ID-TER) = token("-"))
             |($id(ID-TER) = token("!")) then
                if ($arity(ACTUALS) = 2) then
                        [hv120 "("
                                ACTUALS($sep := string($id(ID-TER)) , $is-bin := 2)
                                ")" ]
                else
                        [hv120 "("
                            ID-TER
                            ACTUALS($sep := string($id(ID-TER)) , $is-bin := 1)
                            ")" ]
                end if
        else
                [hv120 ID-TER "(" ACTUALS($sep := ",", $is-bin := 0) ")"]
        end if
end where ;

where field-access ->  EXPR  ID-TER  pprint
        [hv020 EXPR "." ID-TER]
end where ;

where attrib-occ-as-expr ->  ATTRIB-OCC  pprint
        ATTRIB-OCC
end where ;

where applied-id ->  ID  pprint
        ID
end where ;

where real-const ->  REAL  pprint
        REAL
end where ;

where string-const ->  STRING  pprint
        STRING
end where ;

where char-const ->  CHAR  pprint
        CHAR
end where ;

where int-const ->  INT  pprint
        INT
end where ;

where constant-expr ->  EXPR  pprint
        EXPR
end where ;

{- ------------------------------------------------------------------- -}
{- ACTUALS                                                              -}
{- ------------------------------------------------------------------- -}
where actuals ->  ACTUAL * pprint
        pplist left
                global hov100
                subbox ACTUAL($sep := $sep ,
                              $is-bin := $is-bin ,
```

```
                              $is-last := ($position(ACTUAL) = $arity))
        end pplist
end where ;

{- ------------------------------------------------------------------- -}
{- ACTUAL                                                               -}
{- ------------------------------------------------------------------- -}
where actual ->  EXPR  pprint
        if ($is-bin = 1) then
                [hv120 $sep  EXPR]
        elsif $is-last then
                EXPR
        else
                [hv120 EXPR $sep]
        end if
end where ;

{- ------------------------------------------------------------------- -}
{- LIST-ELTS                                                            -}
{- ------------------------------------------------------------------- -}
where list-elts ->  ACTUAL * pprint
        pplist left
                global hov100
                subbox ACTUAL($sep := "," ,
                              $is-bin := 0 ,
                              $is-last := ($position(ACTUAL) = $arity))
        end pplist
end where ;

{- ------------------------------------------------------------------- -}
{- LET-CLAUSES                                                          -}
{- ------------------------------------------------------------------- -}
where let-clauses ->  LET-CLAUSE + pprint
        pplist left
                global v20
                insert ","
                subbox LET-CLAUSE
        end pplist
end where ;

{- ------------------------------------------------------------------- -}
{- LET-CLAUSE                                                           -}
{- ------------------------------------------------------------------- -}
where let-clause ->  ID-TER  TYPE-ID  EXPR  pprint
        [hv120 ID-TER ":" TYPE-ID ":=" EXPR]
end where ;

{- ------------------------------------------------------------------- -}
{- ELSIF-PART                                                           -}
{- ------------------------------------------------------------------- -}
where elsif-part ->  ELSIF-CLAUSE * pprint
        pplist left
                global v00
                subbox ELSIF-CLAUSE
        end pplist
end where ;

{- ------------------------------------------------------------------- -}
{- ELSIF-CLAUSE                                                         -}
{- ------------------------------------------------------------------- -}
where elsif-clause ->  EXPR  EXPR  pprint
        [v00 [hov120 "elsif" EXPR.1 "then"]
             [v20 EXPR.2]]
end where ;

{- ------------------------------------------------------------------- -}
```

```
{- MATCH-CLAUSES                                                      -}
{- ------------------------------------------------------------------- -}
where match-clauses ->  MATCH-CLAUSE + pprint
        pplist left
                global v20
                subbox MATCH-CLAUSE
        end pplist
end where ;

{- ------------------------------------------------------------------- -}
{- MATCH-CLAUSE                                                       -}
{- ------------------------------------------------------------------- -}
where match-clause-other ->  EXPR  pprint
        [hv120 "other:" EXPR ";"]
end where ;

where match-clause ->  PATTERNS  BLOCK  pprint
        [hv120 PATTERNS ":" BLOCK ";"]
end where ;

{- ------------------------------------------------------------------- -}
{- PATTERN                                                            -}
{- ------------------------------------------------------------------- -}
where single-case-const ->  EXPR  pprint
        EXPR
end where ;

where pattern-const-manif ->  EXPR  pprint
        EXPR
end where ;

where pattern-var-constraint ->  ID-TER  TYPE-ID  pprint
        [hv120 ID-TER ":" TYPE-ID]
end where ;

where pattern-var-or-const ->  ID-TER  pprint
        ID-TER
end where ;

where pattern-cons ->  ID  PATTERNS  pprint
        [hv120 ID "(" PATTERNS ")"]
end where ;

{- ------------------------------------------------------------------- -}
{- PATTERNS                                                           -}
{- ------------------------------------------------------------------- -}
where patterns ->  PATTERN * pprint
        pplist left
                global hov100
                insert ","
                subbox PATTERN
        end pplist
end where ;

{- ------------------------------------------------------------------- -}
{- MAP-DIR                                                            -}
{- ------------------------------------------------------------------- -}
where right-dir ->  pprint
        "right"
end where ;

where left-dir ->  pprint
        "left"
end where ;

{- ------------------------------------------------------------------- -}
```

```
{- DECORATIONS                                                          -}
{- -------------------------------------------------------------------- -}
where decorations ->  DECORATION + pprint
        pplist left
                global hov100
                insert ","
                subbox DECORATION
        end pplist
end where ;

{- -------------------------------------------------------------------- -}
{- DECORATION                                                           -}
{- -------------------------------------------------------------------- -}
where decoration ->  ATTR  EXPR  pprint
        [hv120 ATTR ":=" EXPR]
end where ;

end ppat ;
```

C.B Grammaire produite par PPAT

Cette grammaire est produite par PPAT à partir de la spécification précédente, utilisée pour décompiler l'arbre correspondant à la traduction en OLGA de la spécification précédente (auto-amorcage). La largeur du support est de 80 caractères.

```
attribute grammar ppat_olga-simpli ( olga-simpli ) : ( boxes-in ) is

attribute
  synthesized $Ppat (  ) : PPAT ;
attribute
  synthesized $Box (  ) : BOX ;
from ppat-predef import all ;
attribute
  inherited $sep ( ACTUALS, ACTUAL ) : string ;
attribute
  inherited $is-bin ( ACTUALS, ACTUAL ) : int ;
attribute
  inherited $is-last ( ACTUAL ) : bool ;
const
  h0 := sep-info ( h , sep-param [| 0 |] ) ;
const
  v00 := sep-info ( v , sep-param [| 0 , 0 |] ) ;
const
  v01 := sep-info ( v , sep-param [| 0 , 1 |] ) ;
const
  v04 := sep-info ( v , sep-param [| 0 , 4 |] ) ;
const
  v20 := sep-info ( v , sep-param [| 2 , 0 |] ) ;
const
  hv020 := sep-info ( hv , sep-param [| 0 , 2 , 0 |] ) ;
const
  hv100 := sep-info ( hv , sep-param [| 1 , 0 , 0 |] ) ;
const
  hv120 := sep-info ( hv , sep-param [| 1 , 2 , 0 |] ) ;
const
  hov100 := sep-info ( hov , sep-param [| 1 , 0 , 0 |] ) ;
const
  hov120 := sep-info ( hov , sep-param [| 1 , 2 , 0 |] ) ;
function ppat_affichage_token ( object : token; Result : BOXES )  : BOX is
terminal ( string-box (  ) with $string-box := string ( object ) end with )
with
$separator := no-sep
end with
```

```
end function ; and

function ppat_affichage_int ( object : int; Result : BOXES )  : BOX is
terminal ( string-box (  ) with $string-box := string ( object ) end with )
with
$separator := no-sep
end with
end function ; and

function ppat_affichage_char ( object : char; Result : BOXES )  : BOX is
terminal ( string-box (  ) with $string-box := string ( object ) end with )
with
$separator := no-sep
end with
end function ; and

function ppat_affichage_string ( object : string; Result : BOXES )  : BOX is
terminal ( string-box (  ) with $string-box := object end with )
with
$separator := no-sep
end with
end function ;




where null-ID -> use
  $Box := empty-box (  ) with $separator := no-sep end with ;
end where ;

where unqual-id -> ID-TER use
  $Box := $Box(ID-TER) ;
end where ;

where null-ID-TER -> use
  $Box := empty-box (  ) with $separator := no-sep end with ;
end where ;

where id-ter -> use
  $Box := ppat_affichage_token ( $id , boxes (  ) ) ;
end where ;

where null-TYPE-ID -> use
  $Box := empty-box (  ) with $separator := no-sep end with ;
end where ;

where type-id -> ID use
  $Box := $Box(ID) ;
end where ;

where null-COMPILATION-UNIT -> use
  $Ppat := ppat ( empty-box (  ) with $separator := no-sep end with ) ;
end where ;

where simple-AG -> AG-HEADER HOOK declare
  value
    ppat-top-expression : BOX :=
      box (
        boxes-post (
          boxes-post (
            boxes-post (
              boxes-post ( boxes (  ) , $Box(AG-HEADER) ) , $Box(HOOK) ) ,
            terminal (
              string-box (  ) with $string-box := "end grammar ;" end with )
            with
            $separator := no-sep
```

```
          end with ) ,
        terminal ( string-box (  ) with $string-box := "" end with )
        with
        $separator := no-sep
        end with ) )
    with
    $separator := v01
    end with ;
use
  $Ppat := ppat ( ppat-top-expression ) ;
end where ;

where null-HOOK -> use
  $Box := empty-box (  ) with $separator := no-sep end with ;
end where ;

where hook-simple-AG -> DECLS BLOCK use
  $Box :=
    box ( boxes-post ( boxes-post ( boxes (  ) , $Box(DECLS) ) , $Box(BLOCK) )
      )
    with
    $separator := v04
    end with ;
end where ;

where null-AG-HEADER -> use
  $Box := empty-box (  ) with $separator := no-sep end with ;
end where ;

where AG-header -> ID-TER AG-PROFILE use
  $Box :=
    box (
      boxes-post (
        boxes-post (
          boxes-post (
            boxes-post (
              boxes (  ) ,
              terminal (
                string-box (  )
                with
                $string-box := "attribute grammar"
                end with )
              with
              $separator := no-sep
              end with ) ,
            $Box(ID-TER) ) ,
          $Box(AG-PROFILE) ) ,
        terminal ( string-box (  ) with $string-box := "is" end with )
        with
        $separator := no-sep
        end with ) )
    with
    $separator := hv120
    end with ;
end where ;

where null-AG-PROFILE -> use
  $Box := empty-box (  ) with $separator := no-sep end with ;
end where ;

where functional-AG -> SYNTAX SYNTAXES use
  $Box :=
    box (
      boxes-post (
        boxes-post (
          boxes-post (
```

```
            boxes-post (
              boxes-post (
                boxes-post (
                  boxes-post (
                    boxes (  ) ,
                    terminal ( string-box (  ) with $string-box := "(" end with
                      )
                    with
                    $separator := no-sep
                    end with ) ,
                  $Box(SYNTAX) ) ,
                terminal ( string-box (  ) with $string-box := ")" end with )
                with
                $separator := no-sep
                end with ) ,
              terminal ( string-box (  ) with $string-box := ":" end with )
              with
              $separator := no-sep
              end with ) ,
            terminal ( string-box (  ) with $string-box := "(" end with )
            with
            $separator := no-sep
            end with ) ,
          $Box(SYNTAXES) ) ,
        terminal ( string-box (  ) with $string-box := ")" end with )
        with
        $separator := no-sep
        end with ) )
    with
    $separator := hv120
    end with ;
end where ;

where null-SYNTAXES -> use
  $Box := empty-box (  ) with $separator := no-sep end with ;
end where ;

where syntaxes -> SYNTAX * declare
  value
    ppat-list-box-0 : BOXES :=
      map left boxes-post
        value boxes (  )
        other
          if ( $position(SYNTAX) = $arity ) then
            $Box(SYNTAX)
          else
            box (
              boxes-pre (
                $Box(SYNTAX) ,
                boxes (
                  terminal ( string-box (  ) with $string-box := "," end with )
                  with
                  $separator := no-sep
                  end with ) ) )
            with
            $separator := sep-null
            end with
          end if
      end map ;
use
  $Box := box ( ppat-list-box-0 ) with $separator := hov100 end with ;
end where ;

where null-SYNTAX -> use
  $Box := empty-box (  ) with $separator := no-sep end with ;
end where ;
```

```
where syntax -> ID-TER use
  $Box := $Box(ID-TER) ;
end where ;

where null-SEMANTICS -> use
  $Box := empty-box (  ) with $separator := no-sep end with ;
end where ;

where productions -> PRODUCTION-GROUP * declare
  value
    ppat-list-box-0 : BOXES :=
      map left boxes-post
        value boxes (  )
        other
          if ( $position(PRODUCTION-GROUP) = $arity ) then
            $Box(PRODUCTION-GROUP)
          else
            box (
              boxes-pre (
                $Box(PRODUCTION-GROUP) ,
                boxes (
                  terminal ( string-box (  ) with $string-box := "" end with )
                  with
                  $separator := no-sep
                  end with ) ) )
            with
            $separator := sep-null
            end with
          end if
      end map ;
use
  $Box := box ( ppat-list-box-0 ) with $separator := v01 end with ;
end where ;

where null-DECLS -> use
  $Box := empty-box (  ) with $separator := no-sep end with ;
end where ;

where decls -> DECL * declare
  value
    ppat-list-box-0 : BOXES :=
      map left boxes-post
        value boxes (  )
        other
          if ( $position(DECL) = $arity ) then
            $Box(DECL)
          else
            box (
              boxes-pre (
                $Box(DECL) ,
                boxes (
                  terminal ( string-box (  ) with $string-box := "" end with )
                  with
                  $separator := no-sep
                  end with ) ) )
            with
            $separator := sep-null
            end with
          end if
      end map ;
use
  $Box := box ( ppat-list-box-0 ) with $separator := v00 end with ;
end where ;

where null-DECL -> use
```

```
  $Box := empty-box (  ) with $separator := no-sep end with ;
end where ;

where attribute-decl -> ATTRIBUTE-KIND ATTR PHYLA-NAMES TYPE-ID use
  $Box :=
    box (
      boxes-post (
        boxes-post (
          boxes (  ) ,
          terminal ( string-box (  ) with $string-box := "attribute" end with )
          with
          $separator := no-sep
          end with ) ,
        box (
          boxes-post (
            boxes (  ) ,
            box (
              boxes-post (
                boxes-post (
                  boxes-post (
                    boxes-post (
                      boxes-post (
                        boxes-post (
                          boxes-post (
                            boxes-post ( boxes (  ) , $Box(ATTRIBUTE-KIND) ) ,
                            $Box(ATTR) ) ,
                          terminal (
                            string-box (  ) with $string-box := "(" end with )
                          with
                          $separator := no-sep
                          end with ) ,
                        $Box(PHYLA-NAMES) ) ,
                      terminal (
                        string-box (  ) with $string-box := ")" end with )
                      with
                      $separator := no-sep
                      end with ) ,
                    terminal ( string-box (  ) with $string-box := ":" end with
                      )
                    with
                    $separator := no-sep
                    end with ) ,
                  $Box(TYPE-ID) ) ,
                terminal ( string-box (  ) with $string-box := ";" end with )
                with
                $separator := no-sep
                end with ) )
            with
            $separator := hv120
            end with ) )
        with
        $separator := v20
        end with ) )
    with
    $separator := v00
    end with ;
end where ;

where value-decl -> ID-TER TYPE-ID BLOCK use
  $Box :=
    box (
      boxes-post (
        boxes-post (
          boxes (  ) ,
          terminal ( string-box (  ) with $string-box := "value" end with )
          with
```

```
          $separator := no-sep
          end with ) ,
        box (
          boxes-post (
            boxes ( ) ,
            box (
              boxes-post (
                boxes-post (
                  boxes-post (
                    boxes-post (
                      boxes-post (
                        boxes-post ( boxes ( ) , $Box(ID-TER) ) ,
                        terminal (
                          string-box ( ) with $string-box := ":" end with )
                        with
                        $separator := no-sep
                        end with ) ,
                      $Box(TYPE-ID) ) ,
                    terminal (
                      string-box ( ) with $string-box := ":=" end with )
                    with
                    $separator := no-sep
                    end with ) ,
                  $Box(BLOCK) ) ,
                terminal ( string-box ( ) with $string-box := ";" end with )
                with
                $separator := no-sep
                end with ) )
            with
            $separator := hv120
            end with ) )
        with
        $separator := v20
        end with ) )
    with
    $separator := v00
    end with ;
end where ;

where const-decl -> ID-TER BLOCK use
  $Box :=
    box (
      boxes-post (
        boxes-post (
          boxes ( ) ,
          terminal ( string-box ( ) with $string-box := "const" end with )
          with
          $separator := no-sep
          end with ) ,
        box (
          boxes-post (
            boxes ( ) ,
            box (
              boxes-post (
                boxes-post (
                  boxes-post (
                    boxes-post ( boxes ( ) , $Box(ID-TER) ) ,
                    terminal (
                      string-box ( ) with $string-box := ":=" end with )
                    with
                    $separator := no-sep
                    end with ) ,
                  $Box(BLOCK) ) ,
                terminal ( string-box ( ) with $string-box := ";" end with )
                with
                $separator := no-sep
```

```
                end with ) )
            with
            $separator := hv120
            end with ) )
        with
        $separator := v20
        end with ) )
    with
    $separator := v00
    end with ;
end where ;

where rec-funcs-defs -> FUNC-DEF * declare
  value
    ppat-list-box-0 : BOXES :=
      map left boxes-post
        value boxes (  )
        other
          if ( $position(FUNC-DEF) = $arity ) then
            $Box(FUNC-DEF)
          else
            box (
              boxes-pre (
                $Box(FUNC-DEF) ,
                boxes (
                  terminal (
                    string-box (  ) with $string-box := " and" end with )
                  with
                  $separator := no-sep
                  end with ) ) )
            with
            $separator := sep-null
            end with
          end if
      end map ;
use
  $Box := box ( ppat-list-box-0 ) with $separator := v01 end with ;
end where ;

where simple-func-def -> FUNC-DEF use
  $Box := $Box(FUNC-DEF) ;
end where ;

where import-all -> MODULE-ID use
  $Box :=
    box (
      boxes-post (
        boxes-post (
          boxes-post (
            boxes (  ) ,
            terminal ( string-box (  ) with $string-box := "from" end with )
            with
            $separator := no-sep
            end with ) ,
          $Box(MODULE-ID) ) ,
        terminal ( string-box (  ) with $string-box := "import all ;" end with
          )
        with
        $separator := no-sep
        end with ) )
    with
    $separator := hv120
    end with ;
end where ;

where null-MODULE-ID -> use
```

```
  $Box := empty-box (  ) with $separator := no-sep end with ;
end where ;

where module-id -> use
  $Box := ppat_affichage_token ( $module , boxes (  ) ) ;
end where ;

where null-PHYLA-NAMES -> use
  $Box := empty-box (  ) with $separator := no-sep end with ;
end where ;

where phyla-names -> PHYLUM * declare
  value
    ppat-list-box-0 : BOXES :=
      map left boxes-post
        value boxes (  )
        other
          if ( $position(PHYLUM) = $arity ) then
            $Box(PHYLUM)
          else
            box (
              boxes-pre (
                $Box(PHYLUM) ,
                boxes (
                  terminal ( string-box (  ) with $string-box := "," end with )
                  with
                  $separator := no-sep
                  end with ) ) )
            with
            $separator := sep-null
            end with
          end if
      end map ;
use
  $Box := box ( ppat-list-box-0 ) with $separator := hov100 end with ;
end where ;

where null-PHYLUM -> use
  $Box := empty-box (  ) with $separator := no-sep end with ;
end where ;

where phylum -> use
  $Box := ppat_affichage_token ( $phylum , boxes (  ) ) ;
end where ;

where null-FUNC-DEF -> use
  $Box := empty-box (  ) with $separator := no-sep end with ;
end where ;

where func-def -> FUNC-HEADER FUNC-BODY use
  $Box :=
    box (
      boxes-post (
        boxes-post ( boxes (  ) , $Box(FUNC-HEADER) ) ,
        box (
          boxes-post (
            boxes-post ( boxes (  ) , $Box(FUNC-BODY) ) ,
            terminal ( string-box (  ) with $string-box := ";" end with )
            with
            $separator := no-sep
            end with ) )
        with
        $separator := hv120
        end with ) )
    with
    $separator := v00
```

```
    end with ;
end where ;

where null-FUNC-HEADER -> use
  $Box := empty-box (  ) with $separator := no-sep end with ;
end where ;

where func-header -> CALL-MODE ID-TER FORMALS TYPE-ID EXCS use
  $Box :=
    box (
      boxes-post (
        boxes-post (
          boxes-post (
            boxes-post (
              boxes-post (
                boxes-post (
                  boxes-post (
                    boxes-post (
                      boxes (  ) ,
                      terminal (
                        string-box (  ) with $string-box := "function" end with
                        )
                      with
                      $separator := no-sep
                      end with ) ,
                    $Box(ID-TER) ) ,
                  terminal ( string-box (  ) with $string-box := "(" end with )
                  with
                  $separator := no-sep
                  end with ) ,
                $Box(FORMALS) ) ,
              terminal ( string-box (  ) with $string-box := ")" end with )
              with
              $separator := no-sep
              end with ) ,
            terminal ( string-box (  ) with $string-box := " :" end with )
            with
            $separator := no-sep
            end with ) ,
          $Box(TYPE-ID) ) ,
        terminal ( string-box (  ) with $string-box := "is" end with )
        with
        $separator := no-sep
        end with ) )
    with
    $separator := hv120
    end with ;
end where ;

where null-CALL-MODE -> use
  $Box := empty-box (  ) with $separator := no-sep end with ;
end where ;

where offline-mode -> use
  $Box := empty-box (  ) with $separator := no-sep end with ;
end where ;

where null-FORMALS -> use
  $Box := empty-box (  ) with $separator := no-sep end with ;
end where ;

where formals -> FORMAL * declare
  value
    ppat-list-box-0 : BOXES :=
      map left boxes-post
        value boxes (  )
```

```
        other
          if ( $position(FORMAL) = $arity ) then
            $Box(FORMAL)
          else
            box (
              boxes-pre (
                $Box(FORMAL) ,
                boxes (
                  terminal ( string-box (  ) with $string-box := ";" end with )
                  with
                  $separator := no-sep
                  end with ) ) )
            with
            $separator := sep-null
            end with
          end if
      end map ;
use
  $Box := box ( ppat-list-box-0 ) with $separator := hov100 end with ;
end where ;

where null-FORMAL -> use
  $Box := empty-box (  ) with $separator := no-sep end with ;
end where ;

where formal -> ID-TER TYPE-ID use
  $Box :=
    box (
      boxes-post (
        boxes-post (
          boxes-post ( boxes (  ) , $Box(ID-TER) ) ,
          terminal ( string-box (  ) with $string-box := ":" end with )
          with
          $separator := no-sep
          end with ) ,
        $Box(TYPE-ID) ) )
    with
    $separator := hv100
    end with ;
end where ;

where null-EXCS -> use
  $Box := empty-box (  ) with $separator := no-sep end with ;
end where ;

where excs -> ID * use
  $Box := empty-box (  ) with $separator := no-sep end with ;
end where ;

where null-FUNC-BODY -> use
  $Box := empty-box (  ) with $separator := no-sep end with ;
end where ;

where decls-and-body -> DECLS BLOCK use
  $Box :=
    box (
      boxes-post (
        boxes-post ( boxes (  ) , $Box(DECLS) ) ,
        box (
          boxes-post (
            boxes-post ( boxes (  ) , $Box(BLOCK) ) ,
            terminal (
              string-box (  ) with $string-box := "end function" end with )
            with
            $separator := no-sep
            end with ) )
```

```
        with
        $separator := v00
        end with ) )
    with
    $separator := v01
    end with ;
end where ;

where simple-body -> BLOCK use
  $Box :=
    box (
      boxes-post (
        boxes-post ( boxes (  ) , $Box(BLOCK) ) ,
        terminal ( string-box (  ) with $string-box := "end function" end with
          )
        with
        $separator := no-sep
        end with ) )
    with
    $separator := v00
    end with ;
end where ;

where null-ATTRIBUTE-KIND -> use
  $Box := empty-box (  ) with $separator := no-sep end with ;
end where ;

where syn-kind -> use
  $Box :=
    terminal ( string-box (  ) with $string-box := "synthesized" end with )
    with
    $separator := no-sep
    end with ;
end where ;

where inh-kind -> use
  $Box :=
    terminal ( string-box (  ) with $string-box := "inherited" end with )
    with
    $separator := no-sep
    end with ;
end where ;

where null-PRODUCTION-GROUP -> use
  $Box := empty-box (  ) with $separator := no-sep end with ;
end where ;

where prod-group-with-decls -> PRODUCTION DECLS SEM-RULES use
  $Box :=
    box (
      boxes-post (
        boxes-post (
          boxes-post (
            boxes-post (
              boxes-post (
                boxes (  ) ,
                box (
                  boxes-post (
                    boxes-post (
                      boxes-post (
                        boxes (  ) ,
                        terminal (
                          string-box (  ) with $string-box := "where" end with
                          )
                        with
                        $separator := no-sep
```

```
                    end with ) ,
                  $Box(PRODUCTION) ) ,
                terminal (
                  string-box (  ) with $string-box := "declare" end with )
                with
                $separator := no-sep
                end with ) )
            with
            $separator := hv120
            end with ) ,
          box ( boxes-post ( boxes (  ) , $Box(DECLS) ) )
          with
          $separator := v20
          end with ) ,
        terminal ( string-box (  ) with $string-box := "use" end with )
        with
        $separator := no-sep
        end with ) ,
      box ( boxes-post ( boxes (  ) , $Box(SEM-RULES) ) )
      with
      $separator := v20
      end with ) ,
    terminal ( string-box (  ) with $string-box := "end where ;" end with )
    with
    $separator := no-sep
    end with ) )
  with
  $separator := v00
  end with ;
end where ;

where simple-prod-group -> PRODUCTION SEM-RULES use
  $Box :=
    box (
      boxes-post (
        boxes-post (
          boxes-post (
            boxes (  ) ,
            box (
              boxes-post (
                boxes-post (
                  boxes-post (
                    boxes (  ) ,
                    terminal (
                      string-box (  ) with $string-box := "where" end with )
                    with
                    $separator := no-sep
                    end with ) ,
                  $Box(PRODUCTION) ) ,
                terminal ( string-box (  ) with $string-box := "use" end with )
                with
                $separator := no-sep
                end with ) )
            with
            $separator := hv120
            end with ) ,
          box ( boxes-post ( boxes (  ) , $Box(SEM-RULES) ) )
          with
          $separator := v20
          end with ) ,
        terminal ( string-box (  ) with $string-box := "end where ;" end with )
        with
        $separator := no-sep
        end with ) )
    with
    $separator := v00
```

```
    end with ;
end where ;

where null-PRODUCTION -> use
  $Box := empty-box (  ) with $separator := no-sep end with ;
end where ;

where fixed-prod -> ID-TER PROD-RHS use
  $Box :=
    box (
      boxes-post (
        boxes-post (
          boxes-post ( boxes (  ) , $Box(ID-TER) ) ,
          terminal ( string-box (  ) with $string-box := "->" end with )
          with
          $separator := no-sep
          end with ) ,
        $Box(PROD-RHS) ) )
    with
    $separator := hv120
    end with ;
end where ;

where pe-list-prod -> ID-TER PHYLUM use
  $Box :=
    box (
      boxes-post (
        boxes-post (
          boxes-post (
            boxes-post ( boxes (  ) , $Box(ID-TER) ) ,
            terminal ( string-box (  ) with $string-box := "->" end with )
            with
            $separator := no-sep
            end with ) ,
          $Box(PHYLUM) ) ,
        terminal ( string-box (  ) with $string-box := "*" end with )
        with
        $separator := no-sep
        end with ) )
    with
    $separator := hv120
    end with ;
end where ;

where ne-list-prod -> ID-TER PHYLUM use
  $Box :=
    box (
      boxes-post (
        boxes-post (
          boxes-post (
            boxes-post ( boxes (  ) , $Box(ID-TER) ) ,
            terminal ( string-box (  ) with $string-box := "->" end with )
            with
            $separator := no-sep
            end with ) ,
          $Box(PHYLUM) ) ,
        terminal ( string-box (  ) with $string-box := "+" end with )
        with
        $separator := no-sep
        end with ) )
    with
    $separator := hv120
    end with ;
end where ;

where null-PROD-RHS -> use
```

```
  $Box := empty-box (  ) with $separator := no-sep end with ;
end where ;

where prod-rhs -> PHYLUM * declare
  value
    ppat-list-box-0 : BOXES :=
      map left boxes-post
        value boxes (  )
        other
          if ( $position(PHYLUM) = $arity ) then
            $Box(PHYLUM)
          else
            box (
              boxes-pre (
                $Box(PHYLUM) ,
                boxes (
                  terminal ( string-box (  ) with $string-box := "" end with )
                  with
                  $separator := no-sep
                  end with ) ) )
            with
            $separator := sep-null
            end with
          end if
      end map ;
use
  $Box := box ( ppat-list-box-0 ) with $separator := hov100 end with ;
end where ;

where null-SEM-RULES -> use
  $Box := empty-box (  ) with $separator := no-sep end with ;
end where ;

where sem-rules -> SEM-RULE * declare
  value
    ppat-list-box-0 : BOXES :=
      map left boxes-post
        value boxes (  )
        other
          if ( $position(SEM-RULE) = $arity ) then
            $Box(SEM-RULE)
          else
            box (
              boxes-pre (
                $Box(SEM-RULE) ,
                boxes (
                  terminal ( string-box (  ) with $string-box := "" end with )
                  with
                  $separator := no-sep
                  end with ) ) )
            with
            $separator := sep-null
            end with
          end if
      end map ;
use
  $Box := box ( ppat-list-box-0 ) with $separator := v00 end with ;
end where ;

where null-SEM-RULE -> use
  $Box := empty-box (  ) with $separator := no-sep end with ;
end where ;

where simple-rule -> ATTRIB-OCC EXPR use
  $Box :=
    box (
```

```
      boxes-post (
        boxes-post (
          boxes-post (
            boxes-post ( boxes (  ) , $Box(ATTRIB-OCC) ) ,
            terminal ( string-box (  ) with $string-box := ":=" end with )
            with
            $separator := no-sep
            end with ) ,
          $Box(EXPR) ) ,
        terminal ( string-box (  ) with $string-box := ";" end with )
        with
        $separator := no-sep
        end with ) )
    with
    $separator := hv120
    end with ;
end where ;

where null-ATTRIB-OCC -> use
  $Box := empty-box (  ) with $separator := no-sep end with ;
end where ;

where attrib-occ -> ATTR SYMBOL use
  $Box :=
    box (
      boxes-post (
        boxes-post (
          boxes-post (
            boxes-post ( boxes (  ) , $Box(ATTR) ) ,
            terminal ( string-box (  ) with $string-box := "(" end with )
            with
            $separator := no-sep
            end with ) ,
          $Box(SYMBOL) ) ,
        terminal ( string-box (  ) with $string-box := ")" end with )
        with
        $separator := no-sep
        end with ) )
    with
    $separator := hv020
    end with ;
end where ;

where attrib-occ-lhs -> ATTR use
  $Box := $Box(ATTR) ;
end where ;

where null-ATTR -> use
  $Box := empty-box (  ) with $separator := no-sep end with ;
end where ;

where attr -> use
  $Box := ppat_affichage_token ( $attr , boxes (  ) ) ;
end where ;

where null-SYMBOL -> use
  $Box := empty-box (  ) with $separator := no-sep end with ;
end where ;

where qual-symb -> ID-TER QUALIFIER use
  $Box :=
    box (
      boxes-post (
        boxes-post (
          boxes-post ( boxes (  ) , $Box(ID-TER) ) ,
          terminal ( string-box (  ) with $string-box := "." end with )
```

```
          with
          $separator := no-sep
          end with ) ,
        $Box(QUALIFIER) ) )
    with
    $separator := hv020
    end with ;
end where ;

where unqual-symb -> ID-TER use
  $Box := $Box(ID-TER) ;
end where ;

where null-QUALIFIER -> use
  $Box := empty-box (  ) with $separator := no-sep end with ;
end where ;

where last-qual -> use
  $Box :=
    terminal ( string-box (  ) with $string-box := "last" end with )
    with
    $separator := no-sep
    end with ;
end where ;

where first-qual -> use
  $Box :=
    terminal ( string-box (  ) with $string-box := "first" end with )
    with
    $separator := no-sep
    end with ;
end where ;

where right-qual -> use
  $Box :=
    terminal ( string-box (  ) with $string-box := "right" end with )
    with
    $separator := no-sep
    end with ;
end where ;

where left-qual -> use
  $Box :=
    terminal ( string-box (  ) with $string-box := "left" end with )
    with
    $separator := no-sep
    end with ;
end where ;

where up-qual -> use
  $Box :=
    terminal ( string-box (  ) with $string-box := "up" end with )
    with
    $separator := no-sep
    end with ;
end where ;

where int-qual -> INT use
  $Box := $Box(INT) ;
end where ;

where null-INT -> use
  $Box := empty-box (  ) with $separator := no-sep end with ;
end where ;

where int -> use
```

```
  $Box := ppat_affichage_token ( $int , boxes (  ) ) ;
end where ;

where null-CHAR -> use
  $Box := empty-box (  ) with $separator := no-sep end with ;
end where ;

where char -> use
  $Box := ppat_affichage_token ( $char , boxes (  ) ) ;
end where ;

where null-REAL -> use
  $Box := empty-box (  ) with $separator := no-sep end with ;
end where ;

where real -> use
  $Box := ppat_affichage_token ( $real , boxes (  ) ) ;
end where ;

where null-STRING -> use
  $Box := empty-box (  ) with $separator := no-sep end with ;
end where ;

where string -> use
  $Box :=
    box (
      boxes-post (
        boxes-post (
          boxes-post (
            boxes (  ) ,
            terminal ( string-box (  ) with $string-box := "\"" end with )
            with
            $separator := no-sep
            end with ) ,
          ppat_affichage_token ( $string , boxes (  ) ) ) ,
        terminal ( string-box (  ) with $string-box := "\"" end with )
        with
        $separator := no-sep
        end with ) )
    with
    $separator := h0
    end with ;
end where ;

where null-BLOCK -> use
  $Box := empty-box (  ) with $separator := no-sep end with ;
end where ;

where block-semantics -> SEMANTICS use
  $Box := $Box(SEMANTICS) ;
end where ;

where block-expr -> EXPR use
  $Box := $Box(EXPR) ;
end where ;

where null-EXPR -> use
  $Box := empty-box (  ) with $separator := no-sep end with ;
end where ;

where tree-with -> EXPR DECORATIONS use
  $Box :=
    box (
      boxes-post (
        boxes-post (
          boxes-post (
```

```
          boxes-post ( boxes (  ) , $Box(EXPR) ) ,
          terminal ( string-box (  ) with $string-box := "with" end with )
          with
          $separator := no-sep
          end with ) ,
        $Box(DECORATIONS) ) ,
      terminal ( string-box (  ) with $string-box := "end with" end with )
      with
      $separator := no-sep
      end with ) )
    with
    $separator := hov100
    end with ;
end where ;

where map-expr -> MAP-DIR ID EXPR ACTUALS use
  $Box :=
    box (
      boxes-post (
        boxes-post (
          boxes-post (
            boxes (  ) ,
            box (
              boxes-post (
                boxes-post (
                  boxes-post (
                    boxes (  ) ,
                    terminal (
                      string-box (  ) with $string-box := "map" end with )
                    with
                    $separator := no-sep
                    end with ) ,
                  $Box(MAP-DIR) ) ,
                $Box(ID) ) )
            with
            $separator := hv120
            end with ) ,
          box (
            boxes-post (
              boxes-post (
                boxes (  ) ,
                box (
                  boxes-post (
                    boxes-post (
                      boxes (  ) ,
                      terminal (
                        string-box (  ) with $string-box := "value" end with )
                      with
                      $separator := no-sep
                      end with ) ,
                    $Box(EXPR) ) )
                with
                $separator := hv120
                end with ) ,
              box (
                boxes-post (
                  boxes-post (
                    boxes (  ) ,
                    terminal (
                      string-box (  ) with $string-box := "other" end with )
                    with
                    $separator := no-sep
                    end with ) ,
                  $Box(ACTUALS) ) )
              with
              $separator := hv120
```

```
            end with ) )
          with
          $separator := v20
          end with ) ,
        terminal ( string-box (  ) with $string-box := "end map" end with )
        with
        $separator := no-sep
        end with ) )
    with
    $separator := v00
    end with ;
  $sep(ACTUALS) := "," ;
  $is-bin(ACTUALS) := 0 ;
end where ;

where case-tree -> SYMBOL MATCH-CLAUSES MATCH-CLAUSE use
  $Box :=
    box (
      boxes-post (
        boxes-post (
          boxes-post (
            boxes (  ) ,
            box (
              boxes-post (
                boxes-post (
                  boxes-post (
                    boxes (  ) ,
                    terminal (
                      string-box (  ) with $string-box := "case tree" end with
                      )
                    with
                    $separator := no-sep
                    end with ) ,
                  $Box(SYMBOL) ) ,
                terminal ( string-box (  ) with $string-box := "is" end with )
                with
                $separator := no-sep
                end with ) )
            with
            $separator := hv120
            end with ) ,
          box (
            boxes-post (
              boxes-post ( boxes (  ) , $Box(MATCH-CLAUSES) ) ,
              $Box(MATCH-CLAUSE) ) )
          with
          $separator := v20
          end with ) ,
        terminal ( string-box (  ) with $string-box := "end case" end with )
        with
        $separator := no-sep
        end with ) )
    with
    $separator := v00
    end with ;
end where ;

where case-match -> EXPR MATCH-CLAUSES MATCH-CLAUSE use
  $Box :=
    box (
      boxes-post (
        boxes-post (
          boxes-post (
            boxes (  ) ,
            box (
              boxes-post (
```

```
              boxes-post (
                boxes-post (
                  boxes ( ) ,
                  terminal (
                    string-box ( ) with $string-box := "case" end with )
                  with
                  $separator := no-sep
                  end with ) ,
                $Box(EXPR) ) ,
              terminal ( string-box ( ) with $string-box := "is" end with )
              with
              $separator := no-sep
              end with ) )
          with
          $separator := hv120
          end with ) ,
        box (
          boxes-post (
            boxes-post ( boxes ( ) , $Box(MATCH-CLAUSES) ) ,
            $Box(MATCH-CLAUSE) ) )
        with
        $separator := v20
        end with ) ,
      terminal ( string-box ( ) with $string-box := "end case" end with )
      with
      $separator := no-sep
      end with ) )
  with
  $separator := v00
  end with ;
end where ;

where conditional -> EXPR EXPR ELSIF-PART EXPR use
 $Box :=
  box (
   boxes-post (
    boxes-post (
     boxes-post (
      boxes-post (
       boxes-post (
        boxes-post (
         boxes ( ) ,
         box (
          boxes-post (
           boxes-post (
            boxes-post (
             boxes ( ) ,
             terminal (
              string-box ( ) with $string-box := "if" end with )
             with
             $separator := no-sep
             end with ) ,
            $Box(EXPR.1) ) ,
           terminal (
            string-box ( ) with $string-box := "then" end with )
           with
           $separator := no-sep
           end with ) )
         with
         $separator := hv120
         end with ) ,
        box ( boxes-post ( boxes ( ) , $Box(EXPR.2) ) )
        with
        $separator := v20
        end with ) ,
       $Box(ELSIF-PART) ) ,
```

```
            terminal ( string-box (  ) with $string-box := "else" end with )
            with
            $separator := no-sep
            end with ) ,
          box ( boxes-post ( boxes (  ) , $Box(EXPR.3) ) )
          with
          $separator := v20
          end with ) ,
        terminal ( string-box (  ) with $string-box := "end if" end with )
        with
        $separator := no-sep
        end with ) )
    with
    $separator := v00
    end with ;
end where ;

where let-expr -> LET-CLAUSES BLOCK use
  $Box :=
    box (
      boxes-post (
        boxes-post (
          boxes-post (
            boxes-post (
              boxes (  ) ,
              terminal ( string-box (  ) with $string-box := "let" end with )
              with
              $separator := no-sep
              end with ) ,
            $Box(LET-CLAUSES) ) ,
          terminal ( string-box (  ) with $string-box := "in" end with )
          with
          $separator := no-sep
          end with ) ,
        box ( boxes-post ( boxes (  ) , $Box(BLOCK) ) )
        with
        $separator := v20
        end with ) )
    with
    $separator := v00
    end with ;
end where ;

where bool-or -> EXPR EXPR use
  $Box :=
    box (
      boxes-post (
        boxes-post (
          boxes-post ( boxes (  ) , $Box(EXPR.1) ) ,
          terminal ( string-box (  ) with $string-box := "or" end with )
          with
          $separator := no-sep
          end with ) ,
        $Box(EXPR.2) ) )
    with
    $separator := hv120
    end with ;
end where ;

where bool-and -> EXPR EXPR use
  $Box :=
    box (
      boxes-post (
        boxes-post (
          boxes-post ( boxes (  ) , $Box(EXPR.1) ) ,
          terminal ( string-box (  ) with $string-box := "and" end with )
```

```
          with
          $separator := no-sep
          end with ) ,
        $Box(EXPR.2) ) )
    with
    $separator := hv120
    end with ;
end where ;

where list-constr -> TYPE-ID LIST-ELTS use
  $Box :=
    box (
      boxes-post (
        boxes-post (
          boxes-post (
            boxes-post ( boxes (  ) , $Box(TYPE-ID) ) ,
            terminal ( string-box (  ) with $string-box := "[|" end with )
            with
            $separator := no-sep
            end with ) ,
          $Box(LIST-ELTS) ) ,
        terminal ( string-box (  ) with $string-box := "|]" end with )
        with
        $separator := no-sep
        end with ) )
    with
    $separator := hv120
    end with ;
end where ;

where func-call -> ID-TER ACTUALS use
  $Box :=
    if
      (
        (
          (
            (
              (
                (
                  (
                    (
                      (
                        (
                          (
                            ( $id(ID-TER) = token ( "*" ) ) |
                            ( $id(ID-TER) = token ( "/" ) ) ) |
                          ( $id(ID-TER) = token ( "%" ) ) ) |
                        ( $id(ID-TER) = token ( "=" ) ) ) |
                      ( $id(ID-TER) = token ( "!=" ) ) ) |
                    ( $id(ID-TER) = token ( "~" ) ) ) |
                  ( $id(ID-TER) = token ( "<" ) ) ) |
                ( $id(ID-TER) = token ( ">" ) ) ) |
              ( $id(ID-TER) = token ( "<=" ) ) ) |
            ( $id(ID-TER) = token ( ">=" ) ) ) |
          ( $id(ID-TER) = token ( "&" ) ) ) |
        ( $id(ID-TER) = token ( "|" ) ) ) then
      box (
        boxes-post (
          boxes-post (
            boxes-post (
              boxes (  ) ,
              terminal ( string-box (  ) with $string-box := "(" end with )
              with
              $separator := no-sep
              end with ) ,
            $Box(ACTUALS) ) ,
```

```
      terminal ( string-box (  ) with $string-box := ")" end with )
      with
      $separator := no-sep
      end with ) )
  with
  $separator := hv120
  end with
  elsif
  (
    ( ( $id(ID-TER) = token ( "+" ) ) | ( $id(ID-TER) = token ( "-" ) ) ) |
    ( $id(ID-TER) = token ( "!" ) ) )
  then
  if ( $arity(ACTUALS) = 2 ) then
    box (
      boxes-post (
        boxes-post (
          boxes-post (
            boxes (  ) ,
            terminal ( string-box (  ) with $string-box := "(" end with )
            with
            $separator := no-sep
            end with ) ,
          $Box(ACTUALS) ) ,
        terminal ( string-box (  ) with $string-box := ")" end with )
        with
        $separator := no-sep
        end with ) )
    with
    $separator := hv120
    end with
  else
    box (
      boxes-post (
        boxes-post (
          boxes-post (
            boxes-post (
              boxes (  ) ,
              terminal ( string-box (  ) with $string-box := "(" end with )
              with
              $separator := no-sep
              end with ) ,
            $Box(ID-TER) ) ,
          $Box(ACTUALS) ) ,
        terminal ( string-box (  ) with $string-box := ")" end with )
        with
        $separator := no-sep
        end with ) )
    with
    $separator := hv120
    end with
  end if
else
  box (
    boxes-post (
      boxes-post (
        boxes-post (
          boxes-post ( boxes (  ) , $Box(ID-TER) ) ,
          terminal ( string-box (  ) with $string-box := "(" end with )
          with
          $separator := no-sep
          end with ) ,
        $Box(ACTUALS) ) ,
      terminal ( string-box (  ) with $string-box := ")" end with )
      with
      $separator := no-sep
      end with ) )
```

```
  with
  $separator := hv120
  end with
 end if ;
$sep(ACTUALS) :=
 if
  (
   (
    (
     (
      (
       (
        (
         (
          (
           (
            (
             ( $id(ID-TER) = token ( "*" ) ) |
             ( $id(ID-TER) = token ( "/" ) ) ) |
            ( $id(ID-TER) = token ( "%" ) ) ) |
           ( $id(ID-TER) = token ( "=" ) ) ) |
          ( $id(ID-TER) = token ( "!=" ) ) ) |
         ( $id(ID-TER) = token ( "~" ) ) ) |
        ( $id(ID-TER) = token ( "<" ) ) ) |
       ( $id(ID-TER) = token ( ">" ) ) ) |
      ( $id(ID-TER) = token ( "<=" ) ) ) |
     ( $id(ID-TER) = token ( ">=" ) ) ) |
    ( $id(ID-TER) = token ( "&" ) ) ) |
   ( $id(ID-TER) = token ( "|" ) ) ) then
  string ( $id(ID-TER) )
  elsif
  (
   ( ( $id(ID-TER) = token ( "+" ) ) | ( $id(ID-TER) = token ( "-" ) ) ) |
   ( $id(ID-TER) = token ( "!" ) ) )
  then
  if ( $arity(ACTUALS) = 2 ) then
   string ( $id(ID-TER) )
  else
   string ( $id(ID-TER) )
  end if
 else
  ","
 end if ;
$is-bin(ACTUALS) :=
 if
  (
   (
    (
     (
      (
       (
        (
         (
          (
           (
            (
             ( $id(ID-TER) = token ( "*" ) ) |
             ( $id(ID-TER) = token ( "/" ) ) ) |
            ( $id(ID-TER) = token ( "%" ) ) ) |
           ( $id(ID-TER) = token ( "=" ) ) ) |
          ( $id(ID-TER) = token ( "!=" ) ) ) |
         ( $id(ID-TER) = token ( "~" ) ) ) |
        ( $id(ID-TER) = token ( "<" ) ) ) |
       ( $id(ID-TER) = token ( ">" ) ) ) |
      ( $id(ID-TER) = token ( "<=" ) ) ) |
     ( $id(ID-TER) = token ( ">=" ) ) ) |
```

```
          ( $id(ID-TER) = token ( "&" ) ) ) |
        ( $id(ID-TER) = token ( "|" ) ) ) then
      2
      elsif
      (
        ( ( $id(ID-TER) = token ( "+" ) ) | ( $id(ID-TER) = token ( "-" ) ) ) |
        ( $id(ID-TER) = token ( "!" ) ) )
      then
      if ( $arity(ACTUALS) = 2 ) then
        2
      else
        1
      end if
    else
      0
    end if ;
end where ;

where field-access -> EXPR ID-TER use
  $Box :=
    box (
      boxes-post (
        boxes-post (
          boxes-post ( boxes (  ) , $Box(EXPR) ) ,
          terminal ( string-box (  ) with $string-box := "." end with )
          with
          $separator := no-sep
          end with ) ,
        $Box(ID-TER) ) )
    with
    $separator := hv020
    end with ;
end where ;

where attrib-occ-as-expr -> ATTRIB-OCC use
  $Box := $Box(ATTRIB-OCC) ;
end where ;

where applied-id -> ID use
  $Box := $Box(ID) ;
end where ;

where real-const -> REAL use
  $Box := $Box(REAL) ;
end where ;

where string-const -> STRING use
  $Box := $Box(STRING) ;
end where ;

where char-const -> CHAR use
  $Box := $Box(CHAR) ;
end where ;

where int-const -> INT use
  $Box := $Box(INT) ;
end where ;

where constant-expr -> EXPR use
  $Box := $Box(EXPR) ;
end where ;

where null-ACTUALS -> use
  $Box := empty-box (  ) with $separator := no-sep end with ;
end where ;
```

```
where actuals -> ACTUAL * declare
  value
    ppat-list-box-0 : BOXES :=
      map left boxes-post
        value boxes (  )
        other
          if ( $position(ACTUAL) = $arity ) then
            $Box(ACTUAL)
          else
            box (
              boxes-pre (
                $Box(ACTUAL) ,
                boxes (
                  terminal ( string-box (  ) with $string-box := "" end with )
                  with
                  $separator := no-sep
                  end with ) ) )
            with
            $separator := sep-null
            end with
          end if
      end map ;
use
  $Box := box ( ppat-list-box-0 ) with $separator := hov100 end with ;
  $sep(ACTUAL) := $sep ;
  $is-bin(ACTUAL) := $is-bin ;
  $is-last(ACTUAL) := ( $position(ACTUAL) = $arity ) ;
end where ;

where null-ACTUAL -> use
  $Box := empty-box (  ) with $separator := no-sep end with ;
end where ;

where actual -> EXPR use
  $Box :=
    if ( $is-bin = 1 ) then
      box (
        boxes-post (
          boxes-post ( boxes (  ) , ppat_affichage_string ( $sep , boxes (  ) )
            ) ,
          $Box(EXPR) ) )
      with
      $separator := hv120
      end with
    elsif $is-last then
      $Box(EXPR)
    else
      box (
        boxes-post (
          boxes-post ( boxes (  ) , $Box(EXPR) ) ,
          ppat_affichage_string ( $sep , boxes (  ) ) ) )
      with
      $separator := hv120
      end with
    end if ;
end where ;

where null-LIST-ELTS -> use
  $Box := empty-box (  ) with $separator := no-sep end with ;
end where ;

where list-elts -> ACTUAL * declare
  value
    ppat-list-box-0 : BOXES :=
      map left boxes-post
        value boxes (  )
```

```
        other
          if ( $position(ACTUAL) = $arity ) then
            $Box(ACTUAL)
          else
            box (
              boxes-pre (
                $Box(ACTUAL) ,
                boxes (
                  terminal ( string-box (  ) with $string-box := "" end with )
                  with
                  $separator := no-sep
                  end with ) ) )
            with
            $separator := sep-null
            end with
          end if
      end map ;
use
  $Box := box ( ppat-list-box-0 ) with $separator := hov100 end with ;
  $sep(ACTUAL) := "," ;
  $is-bin(ACTUAL) := 0 ;
  $is-last(ACTUAL) := ( $position(ACTUAL) = $arity ) ;
end where ;

where null-LET-CLAUSES -> use
  $Box := empty-box (  ) with $separator := no-sep end with ;
end where ;

where let-clauses -> LET-CLAUSE + declare
  value
    ppat-list-box-0 : BOXES :=
      map left boxes-post
        value boxes (  )
        other
          if ( $position(LET-CLAUSE) = $arity ) then
            $Box(LET-CLAUSE)
          else
            box (
              boxes-pre (
                $Box(LET-CLAUSE) ,
                boxes (
                  terminal ( string-box (  ) with $string-box := "," end with )
                  with
                  $separator := no-sep
                  end with ) ) )
            with
            $separator := sep-null
            end with
          end if
      end map ;
use
  $Box := box ( ppat-list-box-0 ) with $separator := v20 end with ;
end where ;

where null-LET-CLAUSE -> use
  $Box := empty-box (  ) with $separator := no-sep end with ;
end where ;

where let-clause -> ID-TER TYPE-ID EXPR use
  $Box :=
    box (
      boxes-post (
        boxes-post (
          boxes-post (
            boxes-post (
              boxes-post ( boxes (  ) , $Box(ID-TER) ) ,
```

```
              terminal ( string-box (  ) with $string-box := ":" end with )
              with
              $separator := no-sep
              end with ) ,
            $Box(TYPE-ID) ) ,
          terminal ( string-box (  ) with $string-box := ":=" end with )
          with
          $separator := no-sep
          end with ) ,
        $Box(EXPR) ) )
    with
    $separator := hv120
    end with ;
end where ;

where null-ELSIF-PART -> use
  $Box := empty-box (  ) with $separator := no-sep end with ;
end where ;

where elsif-part -> ELSIF-CLAUSE * declare
  value
    ppat-list-box-0 : BOXES :=
      map left boxes-post
        value boxes (  )
        other
          if ( $position(ELSIF-CLAUSE) = $arity ) then
            $Box(ELSIF-CLAUSE)
          else
            box (
              boxes-pre (
                $Box(ELSIF-CLAUSE) ,
                boxes (
                  terminal ( string-box (  ) with $string-box := "" end with )
                  with
                  $separator := no-sep
                  end with ) ) )
            with
            $separator := sep-null
            end with
          end if
      end map ;
use
  $Box := box ( ppat-list-box-0 ) with $separator := v00 end with ;
end where ;

where null-ELSIF-CLAUSE -> use
  $Box := empty-box (  ) with $separator := no-sep end with ;
end where ;

where elsif-clause -> EXPR EXPR use
  $Box :=
    box (
      boxes-post (
        boxes-post (
          boxes (  ) ,
          box (
            boxes-post (
              boxes-post (
                boxes-post (
                  boxes (  ) ,
                  terminal (
                    string-box (  ) with $string-box := "elsif" end with )
                  with
                  $separator := no-sep
                  end with ) ,
                $Box(EXPR.1) ) ,
```

```
              terminal ( string-box (  ) with $string-box := "then" end with )
              with
              $separator := no-sep
              end with ) )
          with
          $separator := hov120
          end with ) ,
        box ( boxes-post ( boxes (  ) , $Box(EXPR.2) ) )
        with
        $separator := v20
        end with ) )
    with
    $separator := v00
    end with ;
end where ;

where null-MATCH-CLAUSES -> use
  $Box := empty-box (  ) with $separator := no-sep end with ;
end where ;

where match-clauses -> MATCH-CLAUSE + declare
  value
    ppat-list-box-0 : BOXES :=
      map left boxes-post
        value boxes (  )
        other
          if ( $position(MATCH-CLAUSE) = $arity ) then
            $Box(MATCH-CLAUSE)
          else
            box (
              boxes-pre (
                $Box(MATCH-CLAUSE) ,
                boxes (
                  terminal ( string-box (  ) with $string-box := "" end with )
                  with
                  $separator := no-sep
                  end with ) ) )
            with
            $separator := sep-null
            end with
          end if
      end map ;
use
  $Box := box ( ppat-list-box-0 ) with $separator := v20 end with ;
end where ;

where null-MATCH-CLAUSE -> use
  $Box := empty-box (  ) with $separator := no-sep end with ;
end where ;

where match-clause-other -> EXPR use
  $Box :=
    box (
      boxes-post (
        boxes-post (
          boxes-post (
            boxes (  ) ,
            terminal ( string-box (  ) with $string-box := "other:" end with )
            with
            $separator := no-sep
            end with ) ,
          $Box(EXPR) ) ,
        terminal ( string-box (  ) with $string-box := ";" end with )
        with
        $separator := no-sep
        end with ) )
```

```
    with
    $separator := hv120
    end with ;
end where ;

where match-clause -> PATTERNS BLOCK use
  $Box :=
    box (
      boxes-post (
        boxes-post (
          boxes-post (
            boxes-post ( boxes (  ) , $Box(PATTERNS) ) ,
            terminal ( string-box (  ) with $string-box := ":" end with )
            with
            $separator := no-sep
            end with ) ,
          $Box(BLOCK) ) ,
        terminal ( string-box (  ) with $string-box := ";" end with )
        with
        $separator := no-sep
        end with ) )
    with
    $separator := hv120
    end with ;
end where ;

where null-PATTERN -> use
  $Box := empty-box (  ) with $separator := no-sep end with ;
end where ;

where single-case-const -> EXPR use
  $Box := $Box(EXPR) ;
end where ;

where pattern-const-manif -> EXPR use
  $Box := $Box(EXPR) ;
end where ;

where pattern-var-constraint -> ID-TER TYPE-ID use
  $Box :=
    box (
      boxes-post (
        boxes-post (
          boxes-post ( boxes (  ) , $Box(ID-TER) ) ,
          terminal ( string-box (  ) with $string-box := ":" end with )
          with
          $separator := no-sep
          end with ) ,
        $Box(TYPE-ID) ) )
    with
    $separator := hv120
    end with ;
end where ;

where pattern-var-or-const -> ID-TER use
  $Box := $Box(ID-TER) ;
end where ;

where pattern-cons -> ID PATTERNS use
  $Box :=
    box (
      boxes-post (
        boxes-post (
          boxes-post (
            boxes-post ( boxes (  ) , $Box(ID) ) ,
            terminal ( string-box (  ) with $string-box := "(" end with )
```

```
            with
            $separator := no-sep
            end with ) ,
          $Box(PATTERNS) ) ,
        terminal ( string-box (  ) with $string-box := ")" end with )
        with
        $separator := no-sep
        end with ) )
    with
    $separator := hv120
    end with ;
end where ;

where null-PATTERNS -> use
  $Box := empty-box (  ) with $separator := no-sep end with ;
end where ;

where patterns -> PATTERN * declare
  value
    ppat-list-box-0 : BOXES :=
      map left boxes-post
        value boxes (  )
        other
          if ( $position(PATTERN) = $arity ) then
            $Box(PATTERN)
          else
            box (
              boxes-pre (
                $Box(PATTERN) ,
                boxes (
                  terminal ( string-box (  ) with $string-box := "," end with )
                  with
                  $separator := no-sep
                  end with ) ) )
            with
            $separator := sep-null
            end with
          end if
      end map ;
use
  $Box := box ( ppat-list-box-0 ) with $separator := hov100 end with ;
end where ;

where null-MAP-DIR -> use
  $Box := empty-box (  ) with $separator := no-sep end with ;
end where ;

where right-dir -> use
  $Box :=
    terminal ( string-box (  ) with $string-box := "right" end with )
    with
    $separator := no-sep
    end with ;
end where ;

where left-dir -> use
  $Box :=
    terminal ( string-box (  ) with $string-box := "left" end with )
    with
    $separator := no-sep
    end with ;
end where ;

where null-DECORATIONS -> use
  $Box := empty-box (  ) with $separator := no-sep end with ;
end where ;
```

```
where decorations -> DECORATION + declare
  value
    ppat-list-box-0 : BOXES :=
      map left boxes-post
        value boxes (  )
        other
          if ( $position(DECORATION) = $arity ) then
            $Box(DECORATION)
          else
            box (
              boxes-pre (
                $Box(DECORATION) ,
                boxes (
                  terminal ( string-box (  ) with $string-box := "," end with )
                  with
                  $separator := no-sep
                  end with ) ) )
            with
            $separator := sep-null
            end with
          end if
      end map ;
use
  $Box := box ( ppat-list-box-0 ) with $separator := hov100 end with ;
end where ;

where null-DECORATION -> use
  $Box := empty-box (  ) with $separator := no-sep end with ;
end where ;

where decoration -> ATTR EXPR use
  $Box :=
    box (
      boxes-post (
        boxes-post (
          boxes-post ( boxes (  ) , $Box(ATTR) ) ,
          terminal ( string-box (  ) with $string-box := ":=" end with )
          with
          $separator := no-sep
          end with ) ,
        $Box(EXPR) ) )
    with
    $separator := hv120
    end with ;
end where ;

end grammar ;
```