UNIVERSITÉ DE NICE
SOPHIA ANTIPOLIS

INRIA
PROJET CROAP

Visualisation graphique incrémentale par évaluation d'attributs

Roudet Christophe

Stage DEA Informatique de l'ESSI

à

INRIA Sophia-Antipolis
2004 Route des Lucioles
Sophia Antipolis
06902 Sophia Antipolis CEDEX

Encadreurs:
Isabelle Attali & Valérie Pascual

Rapporteur:
J. Farre

Novembre 1994 – Juin 1995

Remerciements

Je tiens à remercier tout particulièrement Isabelle Attali et Valérie Pascual pour les conseils et encouragements qu'elles n'ont pas manqués de me donner tout au long du stage.

J'adresse également des remerciements à Didier Parigot pour le temps qu'il a consacré à répondre à mes messages et pour ses visites qui m'ont permis de progresser plus rapidement.

Mes remerciements s'adressent aussi à toute l'équipe CROAP, en particulier à Francis Montagnac pour sa disponibilité et son aide souvent précieuse.

Enfin je remercie les autres stagiaires pour leur bonne humeur.

Résumé

Au sein du système CENTAUR, le langage Ppml permet de générer un affichage non graphique pour un arbre abstrait. Cet affichage est le plus souvent incrémental c'est à dire qu'une modification de l'arbre en entrée, entraine une mise à jour incrémentale de l'affichage. D'autre part, la machine graphique FIGUE permet d'afficher incrémentalement des arbres de boites graphiques. Le sujet consiste à connecter ces deux composants de manière incrémentale par évaluation d'une grammaire attribuée qui spécifie la traduction d'un arbre abstrait (ou une modification sur un arbre abstrait) en un arbre de boites que la machine FIGUE visualisera graphiquement et incrémentalement. On prendra comme application l'édition graphique incrémentale pour des formules de mathématiques.

Visualisation graphique incrémentale par évaluation d'attributs

Christophe Roudet

26 juin 1995

Table des matières

Table des figures

Chapitre 1

Introduction

Présentation du stage

La plupart des systèmes de développement tel que CENTAUR ou SYNTHESIZER GENERATOR ne permettent pas une édition graphique, il serait intéressant pour l'utilisateur d'avoir un environnement convivial, qui supporte à la fois une édition structurée graphique et textuelle. De tels environnements doivent en outre, offrir de bonnes performances pour être utilisables.

Le système CENTAUR [PBP87, ed.94] est un environnement générique et interactif qui, lorsqu'on lui fournit la description d'un langage de programmation particulier (syntaxe et sémantique entre autres), produit un environnement spécifique à ce langage. Cet environnement comprend un éditeur structuré, un interprèteur/debugger et d'autres outils, tous, possédant une interface graphique.

Au sein du système CENTAUR, le langage Ppml permet de générer un affichage (décompilation) non graphique incrémental pour un arbre abstrait. D'autre part, la machine FIGUE [Has92] permet d'afficher incrémentalement des arbres de boites graphiques (Fig. 1.1). Les spécifications Ppml sont compilées en un code Lisp et exécutées par la machine de formatage. Ces deux composants (CENTAUR et FIGUE) sont déja couplés, mais la décompilation graphique obtenue n'est pas incrémentale.

D'autre part, les grammaires attribuées sont devenues une méthode privilégiée pour décrire et implanter des 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. Elles ont été aussi utilisées dans le cadre de l'édition graphique incrémentale [FZ88]. L'incrémentalité permet d'obtenir de bonnes performances lors des actions d'éditions et de reformatage et, elle est obtenue “gratuitement” [FZJ91] (des algorithmes sont connus et ont été implantés).

Parmi les systèmes de traitement des grammaires attribuées, on trouve FNC-2 [JP94], développé par le projet ChLoE de l'INRIA. Cet outil sera utilisé pour le traitement des grammaires attribuées. Les travaux seront menés en collaboration avec Didier Parigot de l'équipe ChLoE de l'INRIA Rocquencourt.

Le sujet consiste à développer un nouveau compilateur Ppml (basé sur une grammaire attribuée) puis à connecter CENTAUR et FIGUE de manière incrémentale par évaluation d'une grammaire attribuée afin d'obtenir une décompilation graphique incrémentale.

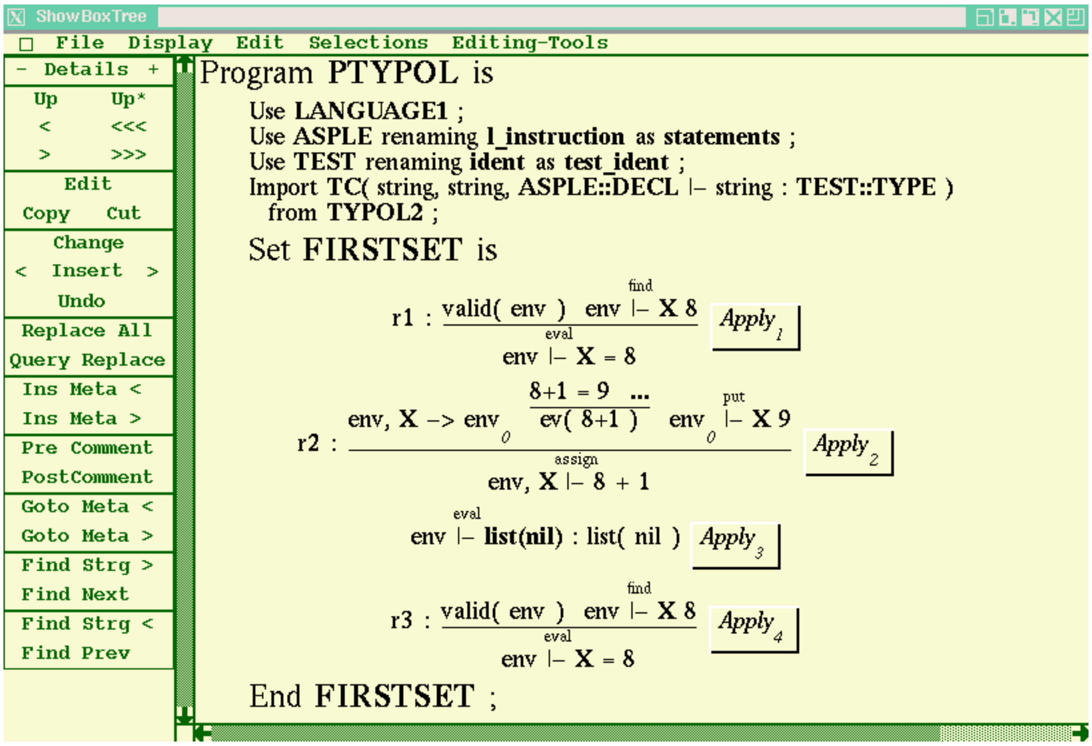

FIG. 1.1 - *Affichage par FIGUE de règles TYPOL*

Première partie

L'existant

Résumé

Dans un premier temps, nous présenterons le système CENTAUR et le formateur FIGUE et ensuite nous développerons le problème.

Chapitre 1

CENTAUR

Le système CENTAUR est un environnement de programmation générique et interactif qui permet de réaliser des environnements de programmation (éditeurs structurés, évaluateurs, analyseurs syntaxiques, vérificateurs de types,...) à partir de la syntaxe et de la sémantique d'un langage de programmation (Fig. 1.1). On peut Distinguer quatre composants principaux dans le système CENTAUR :

- le VTP (**V**irtual **T**ree **P**rocessor) : C'est le composant charnière de création, de mise-à-jour des arbres de syntaxe abstraite, de création d'annotations, de gestion d'occurences dans les arbres, ...
- la machinerie logique : Elle gère les outils sémantiques,
- l'environnement graphique,
- Sophtalk : Il gère la communication entre les composants.

1.1 Les environnements de développement

CENTAUR offre plusieurs environnements de développement ainsi que des outils génériques :

- l'environnement METAL : Il permet d'éditer, d'effectuer une vérification de types, et de compiler des programmes METAL,
- l'environnement Ppml : Il permet d'éditer et de compiler des programmes Ppml,
- l'environnement TYPOL : Il permet d'éditer et de debugger des spécifications TYPOL et d'effectuer la trace de la vérification de types et de l'exécution de programmes écrits en TYPOL,

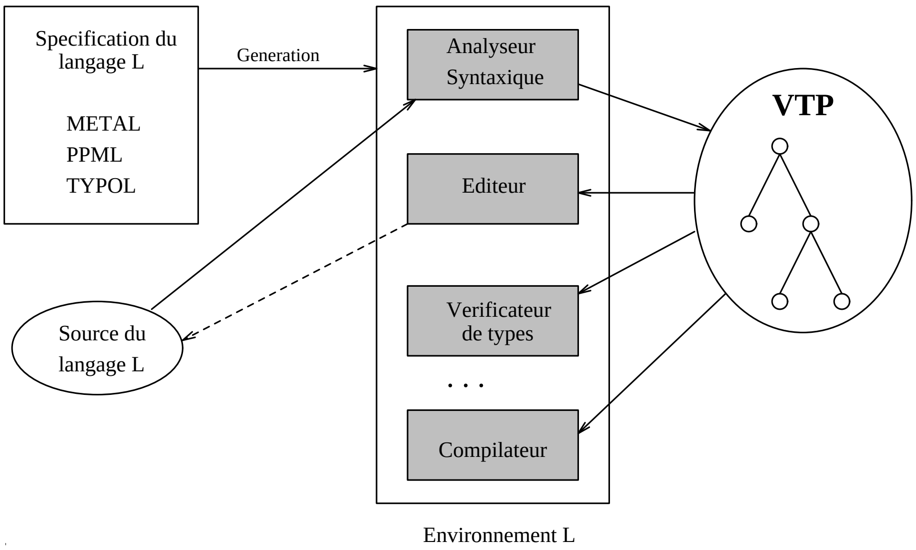

FIG. 1.1 - *exemple d'environnement de programmation*

- l'éditeur syntaxique qui permet une édition structurée pour un langage donné basé sur :
  - la spécification de la syntaxe abstraite d'un langage (définie en METAL),
  - une spécification Ppml de décompilation.

1.2 Les formalismes de CENTAUR

CENTAUR offre plusieurs formalismes de spécifications avec lesquels il est possible de définir divers aspects des langages de programmation, et des compilateurs qui transforment ces spécifications en différents composants de l'environnement de programmation. Les différents méta-langages sont :

1.2.1 METAL

METAL est un méta-langage utilisé pour spécifier la syntaxe abstraite et concrète. Le compilateur METAL est basé sur Lex et Yacc (donc les grammaires acceptables sont de la classe LALR(1)), et utilise le VTP pour construire les arbres abstraits.

Une spécification METAL peut être divisée en deux parties :

- La définition de la syntaxe abstraite, elle consiste à :
  - définir les différents opérateurs : opérateurs d'arité fixe, opérateurs

de listes et opérateur atomiques,

- définir des phyla (phylum).

Exemple :

```
definition of Exp is

abstract syntax
   exp_s -> Exp* ...;
   assign -> Var Exp;
   plus   -> Exp Exp;
   var   -> implemented as IDENTIFIER;
   litteralint   -> implemented as INTEGER;

   Exp_s ::= exp_s;
   Exp ::= assign plus var litteralint;
   Var ::= var;

end definition
```

- La définition de règles syntaxiques : elle consiste
  - à spécifier la syntaxe concrète,
  - à spécifier des actions : ce sont des fonctions de création d'arbres syntaxiques.

1.2.2 Ppml

Ppml [INR94] est un méta-langage utilisé pour spécifier la décompilation d'arbres abstraits de syntaxe en une représentation textuelle. Les utilisateurs peuvent ainsi définir la mise en page (le formatage), les couleurs, les fontes et la présentation de la sélection. Les règles de formatage sont de la forme :

$$pattern \rightarrow format$$

Où un *pattern* permet de reconnaître un nœud d'un arbre de syntaxe abstraite et un *format* décrit la sortie en termes de langage de boites. Les règles de formatages sont compilées en un code exécutable par la machine de formatage. Plusieurs pretty printers peuvent être spécifiés pour un même langage. Mais il faut en considérer un particulièrement important. C'est le pretty printer *standard* et il est nommé **std**. Ce pretty printer std affiche un arbre de syntaxe abstraite qui peut être reparsé directement. Il doit donc respecter et reproduire la syntaxe concrète d'un langage. S'il ne répond pas à cette exigence,il est possible que l'édition du texte affiché soit différente de celle souhaitée.

Exemple :

```
prettyprinter std of Exp is
constant
   tab=2;

default
   <hv 1,tab,0>

rules
   *x !0 -> [<h> ``...''];

   exp_s(**exps,exp)  ->  [<v> [<h> **exps ``;'']) *exp];
   plus(*exp1,*exp2)  ->  [<hv> *exp1 ``+'' *exp2];
   assign(*var,*exp2) ->  [<hv> *var ``:='' *exp2];
end prettyprinter
```

Les patterns

Ppml offre plusieurs techniques de pattern matching :

– les *pattern simple* : ce sont soit des variables, soit des noms spécifiques d'opérateurs ou une combinaison des deux. Nous trouvons dans cette catégorie de pattern :

  – les patterns pour opérateurs d'arité fixe :

  ```
  oper(*x, *y)
  ```

  Ppml accepte les patterns imbriqués :

  ```
  oper(oper2(*x), *y, oper3(oper4(*z)))
  ```

  – les patterns pour opérateurs de liste :

  ```
  oper_s()
  ```

  matche un nœud de liste n'ayant pas de descendant,

  ```
  oper_s(**x)
  ```

  matche un nœud de liste qui a un nombre quelconque de descendant,

  ```
  oper_s(*x, **y, *z)
  ```

  affecte à ***x** le premier élément de la liste, à ***z** le dernier et à ****y** le reste de la liste,

  - les patterns pour opérateurs atomique.

    ```
    ident *x
    ```

    matche un nœud atomique et affecte la valeur de **ident** à la variable *x.

- *les patterns contraints*, ils se comportent comme des patterns simples mais offre un mécanisme pour limiter leur champ d'action :

  ```
  *x where *x in {op1, op2, op3}
  ```

  matche seulement les opérateurs op1, op2 et op3,

- *les patterns de niveau de détail*, ce mécanisme permet de de contrôler le détail d'un programme affiché dans un éditeur CENTAUR selon la profondeur d'un nœud dans la syntaxe abstraite,

- les patterns d'annotation. Les annotations est le moyen sous CENTAUR, permettant de stocker de l'information sur les nœuds,

  ```
  *x ^myannot
  ```

- *les patterns de contexte.* Il arrive que les patterns de syntaxe abstraite soient insuffisant pour spécifier certaines règles. Des informations de contextes sont alors nécessaire pour indiquer quelles règles appliquer. Un contexte Ppml est un ensemble de règles identifiées par un nom de contexte.

Les boites et les séparateurs

Les boites sont des régions rectangulaires qui contiennent le caractère à afficher ou d'autres boites (Fig. 1.2) (même notion qu'en TEX).

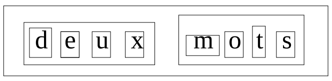

FIG. 1.2 - *Exemple de boites*

Les boites peuvent être soit :

- atomique, elle représente alors un élément lexical du langage,

- composée, c'est à dire qu'elle est constituée d'autres boites, positionnées les unes par rapport aux autres au moyen d'un *séparateur*.

Il y a cinq types de séparateurs :

1. <**h** d_x> sépare les éléments de d_x espaces,
2. <**v** d_i,d_y> sépare les éléments de d_y lignes et avec une indentation de d_i espaces par rapport à la boite englobante,
3. <**v1** d_i,d_y> se comporte comme <v d_i,d_y> sauf que l'indentation des éléments dépend de la marge gauche de la première boite englobante,
4. <**hv** d_x,d_i,d_y> sépare les éléments d'une boite de d_x espaces. Si les éléments de la boite englobante ne tiennent pas sur la même ligne, le séparateur <hv> se comporte alors comme le séparateur <v>,
5. <**hov** d_x,d_i,d_y> sépare les éléments d'une boite de d_x espaces. Si les éléments de la boite englobante ne tiennent pas sur la même ligne, alors tous les séparateurs <hov> sont transformés en séparateurs <v>, et les éléments suivant ces séparateurs, sont disposés avec un espacement vertical de d_y. Tous les éléments après le premier sont indentés par d_i espaces.

Il existe aussi deux structures de contrôle *if* et *case*. Ces deux constructions permettent de regrouper des règles similaires et ainsi de réduire la taille d'un programme Ppml.

Les ressources

Il est possible de définir pour chaque boite et chaque terminal des *classes de ressources*. Ces classes autorisent à définir la fonte et les couleurs associées aux tokens. Les ressources sont spécifiées dans un fichier séparé.

1.2.3 TYPOL

TYPOL est un langage de spécification basé sur des règles d'inférences utilisé pour définir la sémantique statique et dynamique des langages de programmations. Le compilateur TYPOL transforme les spécifications TYPOL en programmes Prolog. TYPOL permet d'obtenir des vérificateurs de types, des interprètes, des traducteurs ...

TYPOL est une implantation de la sémantique naturelle dans le système CENTAUR, un programme TYPOL est composé d'un ensemble de règles d'inférences regroupées en paquets (*sets*). Les règles d'inférences sont des règles de déduction de séquents à partir de séquents. Elles sont de la forme :

$$\frac{H_1 \text{ |- } T_1 : S_1 \ \ldots H_n \text{ |- } T_n : S_n}{H_0 \text{ |- } T_0 : S_0}$$

Le dénominateur s'appelle conclusion (séquent) et le numérateur est formé par les prémisses (liste de séquents). Une règle sans prémisse s'appelle un axiome.

1.2.4 Les paths de modification

Sous CENTAUR, le mécanisme de path de modification permet de gérer les modifications intervenant sur un arbre VTP. Un path désigne un ensemble de sous-arbres d'un arbre abstrait et est composé d'une série de déplacements, qui s'ils sont suivis depuis la racine de l'arbre, conduisent au sous-arbre désigné.

Les modifications sont de trois types :

1. insertions d'un sous-arbre,
2. modification d'un sous-arbre,
3. suppression d'un sous-arbre.

Lors d'une modification, des messages sont émis et le path de modification de l'arbre mis à jour. Il suffit alors d'intercepter ces messages et d'effectuer le traitement approprié. Dans le cadre de l'application, l'évaluateur doit être appelé après une édition, un copier/coller ou un chargement de fichier.

Chapitre 2

FIGUE

Dans ce chapitre nous allons présenter FIGUE qui est un formateur graphique incrémental.

2.1 Description de FIGUE

Le système FIGUE [Has92] (**F**ormatage **I**ncrémental **G**raphiq**UE**), est conçu pour décompiler des arbres d'objets graphiques imbriqués. Chaque objet a un *style*, qui détermine la construction de l'objet en fonction de ses sous-objets. Un objet peut-être atomique (c'est à dire qu'il n'a pas de sous-objet). Ces objets sont appelés *boxtree*s ou arbres de boites (c'est la même notion de boites que nous retrouvons en Ppml, cf. 1.2.2).

2.2 Décompilation d'un arbre syntaxe abstraite

La décompilation d'un arbre abstrait syntaxique est naturellement découpée en trois parties.

2.2.1 Décompilation

Cette phase consiste à transformer l'arbre abstrait syntaxique en un arbre de boites imbriquées. Les feuilles de cet arbre de boites sont la syntaxe concrète correspondant de l'arbre abstrait syntaxique d'entrée. Durant cette phase sont généralement résolues les questions de niveau de détail de la décompilation, ou de fontes. Chaque nœud non terminal possède un *style* (correspond à la notion de *séparateur* de Ppml) tel que *horizontal* ou *vertical*, qui détermine l'arrangement des sous-arbres de boites. En réalité les arbres de boites ont plusieurs *style*s (en fait, une liste de styles classée par ordre de préférence).

2.2.2 Formatage

Cette phase consiste à choisir, pour chaque arbre de boites, un style parmi la liste de styles. Le résultat est un arbre de boites formaté qui peut être affiché sur une *page*. Le choix final des styles doit être tel que le texte puisse être contenu dans une largeur de page donnée. Enfin, le résultat doit être "joli" (*pretty*), i.e. l'utilisateur doit définir des règles de préférences entre le choix de style et des options de reformatage.

2.2.3 Affichage

Cette phase consiste à afficher l'arbre de boites sur l'écran. L'affichage doit prendre en compte les couleurs et les fontes. Il est aussi nécessaire de traiter la *sélection*. Une *sélection* est une désignation d'une partie de l'arbre abstrait syntaxique, dont l'arbre syntaxique concret affiché correspondant est *grisé*.

2.3 Architecture de FIGUE

L'objet fondamental est appelé une *page*. Une page a une largeur et une profondeur qui peut être infinie. Lorsqu'un arbre de boites est formaté, il l'est de manière à ce qu'il rentre dans cette page. La page contient la représentation d'un arbre de boites après formatage, avec les indications de taille, de style et de position relative de chaque sous-arbre de boites. D'un autre coté, une page est conçue pour être indépendante des systèmes graphique utilisés. Par exemple, une page n'est pas nécessairement affichée sur l'écran. Lorsqu'une partie de la page est affichée, FIGUE appelle un ensemble de fonctions qui définissent l'interface d'un système graphique particulier. Pour utiliser un autre système graphique, il suffit de modifier ces fonctions. Une page sera affichée à l'écran à l'aide de vues (*views*). Certaines opérations sur les pages sont retardées, c'est à dire qu'elles seront effectuées quand une vue affichera le sous-arbre de boites correspondant. En mode normal (incrémental), les interactions entre une page et une vue sont :

- des modifications qui apparaissent dans le monde extérieurs et qui peuvent avoir des conséquences sur la page, sont des modifications de *path* sur l'arbre de boites, ou un changement de largeur de la page, ou un changement de la sélection de l'affichage de l'arbre de boites.

- si on demande à FIGUE de réagir à ces modifications, il faut lui fournir le path de modification, la nouvelle largeur de page et le nouveau path de sélection. La page est alors mise-à-jour, et les décisions de reformatage sont prises, i.e. changement de style pour certains sous-arbres de boites.

- selon les décisions de reformatage prises, FIGUE calcule les mouvements des arbres de boites inchangés, i.e. leur nouvelle position absolue dans la page. Les arbres de boites insérés, supprimés ou modifiés sont aussi localisés, ainsi que les changements de fontes, de couleurs ou de sélections.

La page envoie des messages aux vues concernées, afin que les vues stockent la liste de changement à effectuer en cas de mise-à-jour.

– lorsque une vue effectue une mise-à-jour, elle exécute la liste des changements précédemment stockée. Cela peut donner lieu à de nouvelles interactions avec la page, dans le cas ou des arbres de boites doivent être de nouveau (ré)affichés. Les arbres de boites à afficher sont parcourus, et des ordres atomiques d'affichage sont renvoyés à la vue.

2.4 Utilisation de FIGUE

La manière la plus convenable pour utiliser FIGUE, est de le connecter à un autre outil qui génère des arbres de boites non formatés. Ainsi , FIGUE a été connecté au décompilateur PPML. Une autre caractéristique nécessaire est une interaction entre l'arbre de boites de FIGUE et l'arbre original, pour qu'un path de sélection d'un arbre soit transformé en un path de sélection de l'autre.

Maintenant que ces deux outils ont été présentés, nous pouvons nous intéresser au problème.

Chapitre 3

Le compilateur Ppml actuel et ses limites

Un *pretty printer* ou *décompilateur* transforme un arbre abstrait en texte. En CENTAUR, les spécifications de décompilation sont écrites en Ppml, compilées en un code Lisp et exécutées par la machine de formatage. Étant donné un arbre abstrait et une spécification Ppml, le formateur Ppml génère une représentation textuelle de l'arbre.

Une spécification Ppml est une liste ordonnée de règles (cf. paragraphe 1.2.2). Le formateur Ppml génère une version textuelle d'un arbre en parcourant celui-ci, en cherchant un pattern (partie gauche des règles) qui matche chaque nœud de l'arbre. Lorsque ceci est vérifié, le formateur exécute la partie droite de la règle qui décrit l'organisation des tokens à l'aide d'un langage de boites. Ainsi, le compilateur Ppml existant, produit, à partir d'une spécification Ppml, un code Lisp traduisant un arbre abstrait d'un formalisme donné en un arbre de boites graphiques.

Le problème vient du fait que cette transformation d'arbre n'est pas réalisée de manière incrémentale par l'évaluateur (nous parlons ici du code Lisp pour un affichage graphique, car le code Lisp généré pour un affichage textuel effectue des traductions incrémentales) (Fig. 3.1). C'est à dire que dès qu'une partie de l'arbre abstrait est modifiée, la totalité de l'arbre est de nouveau évaluée, alors que dans la majorité des cas, seul une partie de l'arbre devrait être réévaluée. En effet dans la plupart des cas, l'évaluateur effectue des calculs inutiles puisque la disposition à l'écran de certaines boites, n'a pas à été modifiée.

On souhaite donc concevoir un nouveau compilateur Ppml, basé sur les grammaires attribuées, qui permettra de visualiser graphiquement et incrémentalement un arbre abstrait à partir d'une spécification Ppml quelconque.

La réalisation de cet outil met en œuvre les outils suivants : CENTAUR (pour la définition d'un formalisme pour les arbres de boites), FIGUE (pour la partie affichage graphique incrémental) et FNC-2 (pour le traitement des grammaires attribuées).

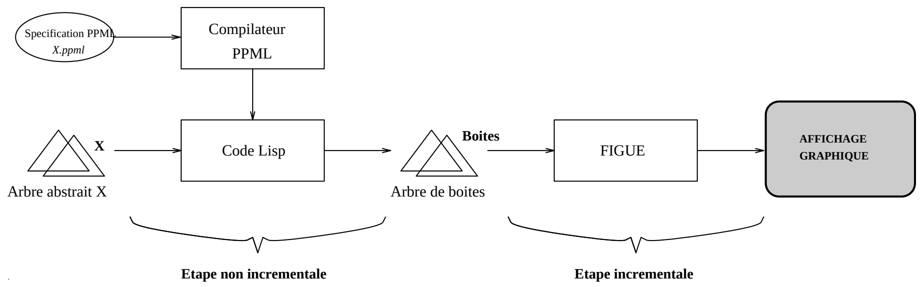

FIG. 3.1 - *La décompilation existante*

Deuxième partie

Solution proposée

Résumé

Dans un premier temps, nous donnerons le principe adopté pour résoudre le problème, puis dans un second, nous présenterons FNC-2, l'outil que nous utiliserons et le nouveau compilateur spécifié à l'aide des grammaires attribuées.

Chapitre 1

Principe général

Introduction

Pour résoudre le problème nous utilisons les grammaires attribuées, car elles sont devenues une méthode privilégiée pour décrire et implanter les programmes dirigés par la syntaxe et elles permettent d'obtenir l'incrémentalité sans grands efforts ce qui est intéressant dans notre cas.

Pour simplifier la résolution du problème, nous pouvons distinguer deux phases :

Une première phase consistera à générer, à partir d'une spécification PPML, une grammaire attribuée qui spécifie la traduction d'un arbre abstrait en un arbre de boites. Nous appellerons cette partie, *phase de traduction* (Fig. 1.1).

Une seconde phase concernera, quant à elle, l'affichage proprement dit. Nous appellerons cette phase, *phase de visualisation* (Fig. 1.2).

1.1 Le formalisme Boxtree

1.1.1 Définition d'un formalisme pour les arbres de boites

La machine FIGUE admet en entrée, des arbres de boites graphiques représentés sous forme de listes écrites en Lisp, c'est une forme *plate*, sans structure (pas de syntaxe abstraite) et donc difficilement manipulable. Cette représentation Lisp est ensuite transformée en un véritable arbres de boites graphiques par FIGUE. Quant aux grammaires attribuées, elles permettent d'effectuer des transformations sur des arbres, c'est pour cela qu'il a été nécessaire de définir un formalisme pour les arbres de boites, c'est le formalisme *Boxtree*.

Exemple :

Voici la représentation en Lisp :

```
(setq tree1
```

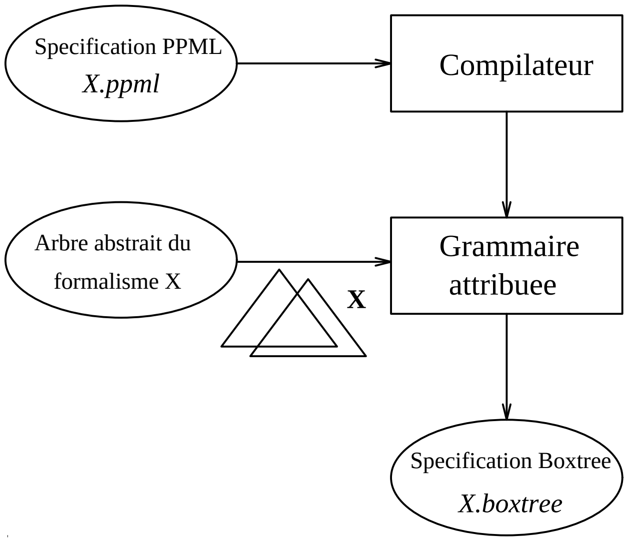

FIG. 1.1 - *phase de traduction*

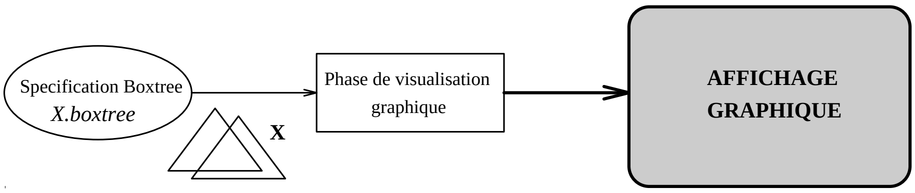

FIG. 1.2 - *phase de visualisation*

```
'((top)
  ((h-vi bold-norm-font)
   ``hello''
   ``world''
   ((v) ((box 0.4 2 0.5)) ((box 0.3 0.3))))))
```

et son équivalente en Boxtree :

```
boxtree tree1
begin
   top () is
      h-vi (font: bold-norm-font) is
         "hello"
         "world"
         v () is
            box () is
               0.4
               2
               0.5
            endbox
            box () is
               0.3
               0.3
            endbox
         endbox
      endbox
   endbox
end
```

1.1.2 L'environnement Boxtree

Nous avons donc spécifié, sous Centaur, un environnement pour le formalisme Boxtree. Cet environnement permet de parser des spécifications Boxtree, de les décompiler en Lisp et de visualiser le résultat de leur évaluation par FIGUE (Fig. 1.3).

1.2 Traduction d'une spécification PPML quelconque en un arbre Boxtree

Maintenant que nous disposons d'un formalisme pour les arbres de boites graphiques, il s'agit de générer une spécification de ce formalisme, à partir d'une spécification PPML (Fig. 1.1). Nous pouvons distinguer deux étapes dans cette phase :

– **Première étape : compilation d'une spécification PPML en une grammaire attribuée**

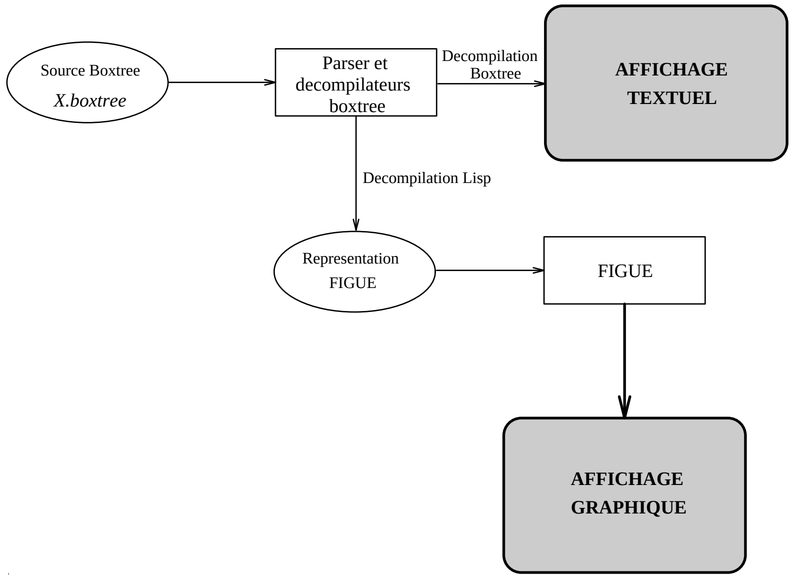

FIG. 1.3 - *Environnement Boxtree*

Il s'agit dans cette étape de générer une spécification de grammaire attribuée (écrite en OLGA). C'est une étape de transformation sur des arbres abstraits, prenant en entrée une spécification de décompilation et fournissant en sortie une spécification de grammaire attribuée, spécifique à la décompilation du langage d'entrée.

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

Il s'agit de mettre en correspondance les deux syntaxes abstraites, celle de Ppml et celle d'OLGA. Il faut donc écrire une grammaire OLGA qui génère une autre grammaire OLGA.

– **Deuxième étape : évaluation d'un arbre abstrait**

  La grammaire ainsi obtenue spécifiera quant à elle, la transformation d'un arbre abstrait du formalisme X, en un arbre de boites Boxtree. On générera un évaluateur incrémental à partir de la grammaire obtenue à l'étape précédente. Il faudra, à ce niveau, s'occuper des *paths de modification* afin qu'une modification de l'arbre abstrait en entrée, apparaisse dans l'arbre de boites et que cela se fasse de façon incrémentale.

Ces deux étapes peuvent être représentées par la figure 1.4.

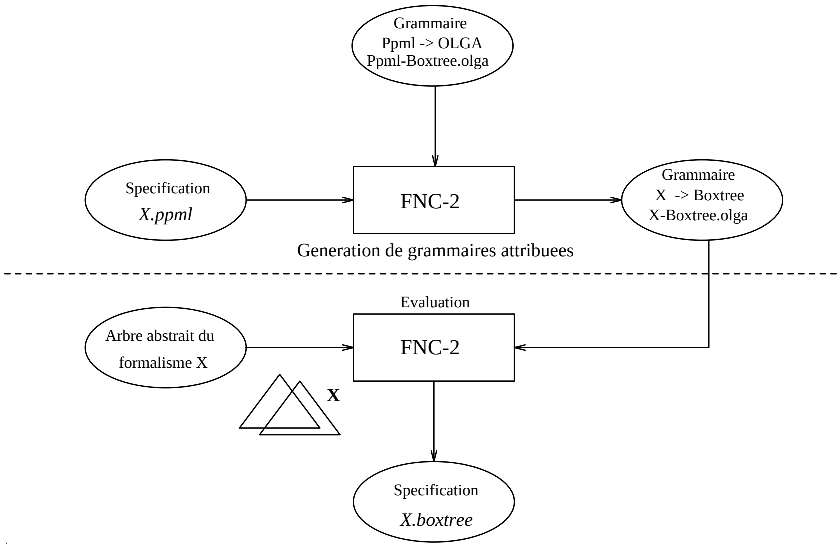

FIG. 1.4 - *Les deux étapes de la phase de traduction*

1.3 Phase de visualisation

Cette phase concerne la partie visualisation incrémentale et graphique par Figue de l'arbre de boites, il faudra se préoccuper des paths de modification et du reformatage incrémental de la page Figue, pour qu'une modification de l'arbre de boites n'entraine que le reformatage de la partie concernée de la page.

Chapitre 2

FNC-2

Nous allons maintenant présenter succinctement le système FNC-2 [JDP] qui va nous permettre d'écrire le nouveau compilateur Ppml.

2.1 Généralités

FNC-2 [1] est un système de traitement des grammaires attribuées.

2.1.1 Description d'une application pour FNC-2

Une grammaire attribuée spécifie la transformation d'un arbre abstrait attribué en zéro, un ou plusieurs autres arbres attribués. Ce concept est le fondement de la modularité du système FNC-2. Une *grosse* application peut-être découpée en un nombre de passes quelconque, chaque passe prenant en entrée la représentation intermédiaire générée par la précédente, la transformant en une autre représentation intermédiaire pouvant servir d'entrée à la passe suivante. Chaque passe peut être décrite par une grammaire attribuée (ou par d'autres outils) et ainsi chacune de ces grammaires est beaucoup plus petite et par conséquent plus facilement manipulable qu'une grammaire attribuée unique représentant l'application globale.

2.2 Les syntaxes abstraites attribuées

Les arbres abstraits attribués qui sont en entrée et en sortie d'un évaluateur, sont décrits par des *syntaxes abstraites attribuées.* Ces dernière 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.

1. FNC-2 signifie **F**ortement **N**on **C**irculaire

En résumé, 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, ce qui offre 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), et exprime de façon plus claire les imbrications des structures sémantiques.

- les arbres abstraits sont en général plus petits que les arbres concrets puisqu'ils sont débarrassés des mots clefs et des autres informations encombrantes.

Ces arbres abstraits sont obtenus à partir d'un texte ou comme résultat d'une grammaire attribuée.

Le formalisme utilisé pour la spécification des syntaxes abstraites est proche de METAL, celui utilisé sous le système CENTAUR. Il repose sur la définition de *phyla* et d'*opérateurs* comme en METAL. Le sous langage d'OLGA utilisé pour définir les syntaxes abstraites attribuées s'appelle *asx*.

2.3 OLGA

Le langage OLGA [2] [MJP90] 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 des règles sémantiques,...

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

2.3.1 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, ce qui donne un certain niveau de sûreté. 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). Le nom des fonctions peut-être surchargé, ce qui augmente la lisibilité des spécifications.

Tout comme Ada, OLGA possède la notion de modules, permettant de définir un ensemble d'objets liés entre eux. Cette particularité participe à la 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

2. OLGA signifie **O**uf!, un **L**angage pour les **G**rammaires **A**ttribuées

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.

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.

2.3.2 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 sorties 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 d'entrée.

– les grammaires attribuées *procédurales* : ces grammaires prennent en entrée un arbre décoré et fournissent en sortie le même arbre (au niveau de la structure), mais décoré de façon différente. 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.

Les attributs

On distingue trois classes d'attributs :

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

– les attributs *locaux* : ils servent au calcul mais ne sont pas accroché à l'arbre de sortie et sont donc par conséquent invisibles à l'extérieur de la grammaire.

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

Ces attributs ne sont pas déclarés dans la même spécification :

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

Les attributs *exportés* sont définis quant-à-eux dans la spécification de la syntaxe abstraite attribuée de la grammaire de sortie, mais il faut définir leur mode de calcul, c'est-à-dire leur catégories.

Il existe quatre catégories pour les attributs *exportés* et les attributs de *travail*:

- *synthétisé*: la valeur d'un tel attribut est calculé au cours de la visite du noeud correspondant au phylum sur lequel il est accroché. L'information circule 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. L'information se propage de la racine vers les feuilles.
- *mixte*: un tel attribut peut être considéré comme l'association de deux attributs, un synthétisé et un hérité.
- *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 l'information.

Règles de copie par défaut

Le système FNC-2 fournit, afin de clarifier les spécifications de grammaires attribuées, un mécanisme de génération de règle 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 spécifiée par l'utilisateur.

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 occurrences d'attributs, et sont donc de la forme d'une affectation d'une expression à une occurrence 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 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.

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 à la partie intéressante au niveau de la sémantique.

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 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 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 ne contiennent que des définitions d'attributs.

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 de langage *atc* des appels aux fonctions associées aux opérateurs. De façon pratique, puisque 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.

2.4 La version incrémentale de FNC-2

FNC-2 génère soit du code C, soit du code Lisp. Dans le cadre de ce stage, seule la version Lisp sera utilisée. L'évaluateur peut être ainsi aisément couplé avec CENTAUR, ce qui a déjà été réalisé dans le cadre d'un petit exemple *bird* écrit par Didier Parigot.

2.4.1 Principe de base

Un évaluateur en version exhaustive est composé d'un ensemble de fonctions mutuellement récursives (principe de séquence de visite).
Le générateur d'évaluateurs (FNC-2) pour décider que sur un non-terminal *Nt* il faudra n visites, calcule pour chaque attribut **synthétisé**, l'ensemble des attributs **hérités** dont il dépend quels que soient les sous-arbres (dérivables à partir de *Nt*). C'est la notion de **sélecteur d'argument**.

Pour faire une version incrémentale, il faut étendre cette notion de sélecteur d'argument aux attributs hérités et il faut ajouter a la valeur de l'attribut un *tag*, signifiant que la valeur de l'attribut est consistant ou non (modifiée ou pas). Ainsi un évaluateur incrémental est composé de deux familles de visites :

- visite pour les synthétisés (V_S),
- visite pour les hérités (V_H).

2.4.2 Le contrôle

En fait, un évaluateur incrémental est un évaluateur classique exhaustif, mais qui peut commencer l'évaluation n'importe où dans l'arbre. Pour rendre cet évaluateur incrémental on rajoute un *contrôle*: "est-il nécessaire de faire cette visite ?".

Le contrôle se décompose en deux types :

– contrôle sur l'appel des visites,

– contrôle "autour" de l'évaluation des attributs.

Si sur un nœud, le sous-arbre ne contient aucune modification et l'ensemble des attributs hérités est consistant alors la visite du nœud n'est pas nécessaire. Ce raisonnement est appliqué pour les deux visites, V_S et V_H.

Chapitre 3

Un nouveau compilateur pour Ppml

3.1 Ppml* : un sous ensemble de Ppml

Il est souvent difficile d'implanter d'une manière globale toutes les possibilités d'un langage dès la première version d'un compilateur. Nous avons donc choisi de prendre un sous ensemble de Ppml que nous appellerons Ppml*. Nous allons donner les principales restrictions du langage et les problèmes que pourront poser leur implantation.

3.1.1 Déclarations

Seule les déclarations de constantes sont prises en compte actuellement, car il existe aussi des déclarations de constantes en OLGA, alors que pour les valeurs par défaut, il n'y a pas de notion équivalente en OLGA.

3.1.2 Règles

Plusieurs règles de décompilation peuvent être définies pour un même opérateur en Ppml. Par contre OLGA n'accepte qu'une seule production par opérateur et Ppml* aussi par conséquent. De même, les chapitres (*chapter*) qui permettent de regrouper un ensemble de règles ne sont pas traités.

3.1.3 Patterns

Ppml fournit plusieurs techniques de patterns matching, par contre Ppml* n'autorise pas :

1. les patterns imbriqués (par exemple op(op1(*x),*y)) Il existe en OLGA une construction *case is* qui permet de faire du pattern matching sur

les attributs, et en principe ce type de pattern ne devrait pas être trop compliqué à implanter.

2. les patterns pour opérateurs de liste où l'on peut préciser la tête ou la queue de la liste.
   Exemples :

   – oper_s(*head,**rest),
   – oper_s(**rest,*tail),
   – oper_s(*head,**rest,*tail).

   Il existe en OLGA une construction qui permet de définir les attributs selon la position des descendants, il s'agit de *case position*. Ce qui est plus difficile, c'est d'effectuer les bonnes constructions d'arbres selon la position dans la liste.

3. les patterns contraints,

4. les patterns de niveau de détail (print level patterns),

5. les patterns d'annotations,

6. les patterns de contexte, ici il s'agira typiquement d'ajouter un attribut hérité qui descendra l'information de contexte vers les descendants du nœud.

3.1.4 Formatage

Dans cette partie de règle sont spécifiées les représentations textuelles des nœuds des arbres abstraits.
Tout ce qui concerne les appels de fonctions externes, les construction *if* et *case* et les classes de ressources n'existent pas en Ppml*. En ce qui concerne la prise en compte des ressources, le problème vient du fait que les ressources sont associées à une spécification de décompilation Ppml et que dans le cadre de l'application, nous n'utilisons pas de décompilateur Ppml, il semble difficile dans ce cas de prendre en considération les ressources. Si une solution est trouvée, l'implantation se fera à l'aide d'un attribut hérité puisque les ressources sont partagées par les sous boites, il faudra alors acheminer l'information vers ces sous boites.

3.1.5 Autres restrictions

– Il existe en Ppml des variables anonymes, et il n'y a pas d'équivalent en OLGA,

– la surcharge d'opérateur n'est pas autorisée en Ppml*: en OLGA, il faut préciser le phylum auquel appartient l'opérateur et pour l'instant nous n'avons pas un accès immédiat au phylum d'un opérateur donné,

– en Ppml il n'est pas nécessaire d'écrire une règle pour les opérateurs atomiques, alors qu'il faut obligatoirement en écrire une en Ppml*. Sinon il faudrait avoir accès à la syntaxe abstraite du formalisme traité afin de générer automatiquement ces règles.

3.1.6 Exemple de spécification Ppml*

```
prettyprinter fig of minimath is

rules

main(*program) -> [<h> *program];

program(*inst, *axiom) -> [<v> *inst *axiom];

program_vide *x -> [<h>];

integral_inst(*b1, *b2, *expr, *dx) ->
    [<h> [<exp> [<ind> [<integral>] [<h> *b1]] *b2] [<h> *expr *dx]];

sqrt_inst(*deg, *expr) -> [<sqrt> *expr *deg];

integral_expr(*b1, *b2, *expr, *dx) ->
    [<h> [<exp> [<ind> [<integral>] [<h> *b1]] *b2] [<h> *expr *dx]];

sqrt_expr(*deg, *expr) -> [<sqrt> *expr *deg];

chaine_expr(*string) -> [<h> *string];

chaine_string *x -> [<h> *x];

end prettyprinter
```

Nous pouvons remarquer qu'il y a des séparateurs qui n'existent pas en Ppml comme <*sqrt*> ou <*integral*>. Justement, ce qui est intéressant, c'est de pouvoir ajouter de nouveaux séparateurs. c'est sur cet exemple que les tests et la mise au point ont été effectués.

3.2 Le formalisme Boxtree

Boxtree est un formalisme destiné à spécifier des arbres de boites graphiques visualisables à l'aide de FIGUE. Nous allons surtout regarder la syntaxe abstraite du formalisme Boxtree et la décompilation Lisp en laissant de coté la syntaxe concrète.

3.2.1 La syntaxe abstraite de Boxtree

La syntaxe abstraite permet de représenter la structure d'arbre des boites, ce que la représentation sous forme de listes Lisp ne permet pas.. Pour cela, on défini des *opérateurs* et des *phyla*. Sur la figure 3.1, les phyla sont en gras et en majuscule. La structure choisie est proche de la structure des listes Lisp de boites, ce qui facilitera la décompilation.

Les principaux phyla sont :

- **TRNAME** représentant le nom de l'arbre,
- **BOXTREE** représentant l'arbre proprement dit,
- **SONS** représentant la liste des sous-arbres,
- **STYLE** représentant le *style* de la boite graphique,
- **CBNAME** étant le nom du *style*,
- **PARAM** représentant les paramètres (fonte et indentations).

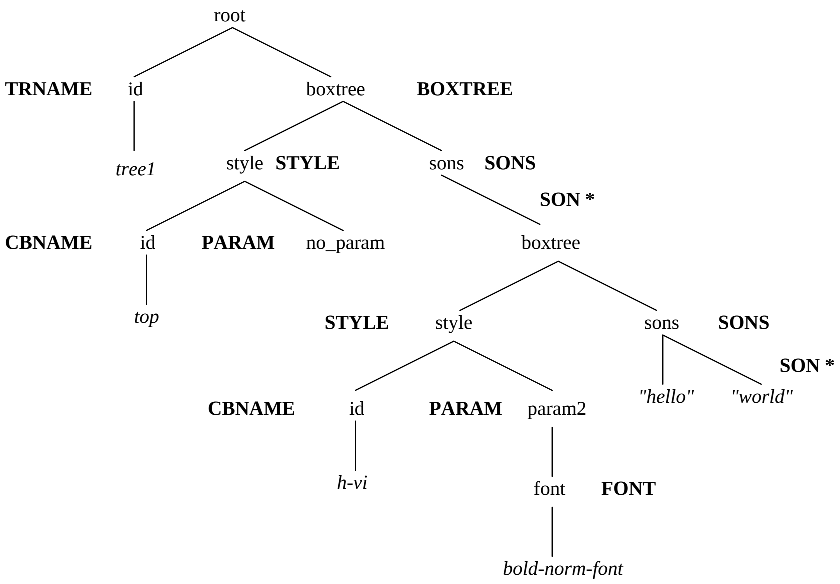

FIG. 3.1 - *Exemple d'arbre de boites Boxtree*

3.2.2 La décompilation des arbres de boites en liste Lisp

Comme nous l'avons dit au chapitre 1.1.1 (page 23) , FIGUE prend en entrée la représentation Lisp des arbres de boites, nous effectuons donc une décompilation des arbres Boxtree en Lisp. Dans ce cas on perd la structure d'arbre

Boxtree, mais cela n'est pas important dans la mesure ou FIGUE ne se sert pas des arbres Boxtree.

Les principaux schémas de traductions sont les suivants :

- *root(***TRNAME, BOXTREE***)* est décompilé en :

  (setq **TRNAME** *'* **BOXTREE***)*

- *boxtree(***STYLE, SONS***)* est décompilé en :

  *(***STYLE SONS***)*

  sauf dans le cas oú le style est soit *box* soit *atom*, la décompilation est alors la suivante :

  *boxtree(style(id 'box'| 'atom',***PARAM***),* **SONS***)* est traduit en :

  ((box **PARAM SONS***))*

- *style(***CBNAME, PARAM***)* est décompilé en :

  *(***CBNAME PARAM***)*

Par exemple, la décompilation de l'arbre de la figure 3.1 donne :

```
(setq tree1
    '((top)
      ((h-vi bold-norm-font)
        ''hello''
        ''world''
      )
     )
)
```

3.2.3 L'environnement Boxtree

Le schéma correspondant à la génération de l'environnement Boxtree est décrit par la figure 3.2. Les spécifications représentées sont :

- **boxtree.metal**, la syntaxe abstraite et concrète du formalisme Boxtree,
- **boxtree-std.ppml**, la spécification de décompilation standard,
- **boxtree-lisp.ppml**, la spécification de décompilation Lisp.

Après compilation de ces différentes spécifications, on obtient un parseur et deux décompilateurs. De plus, quelques ressources ont été définies pour l'éditeur Boxtree, entre autres, un bouton permettant l'affichage par FIGUE de la décompilation Lisp.

Le parseur prend en entrée une spécification Boxtree (Fig. 3.3) et fait une décompilation Lisp (Fig. 3.4) qui est ensuite traitée par FIGUE (Fig. 3.5).

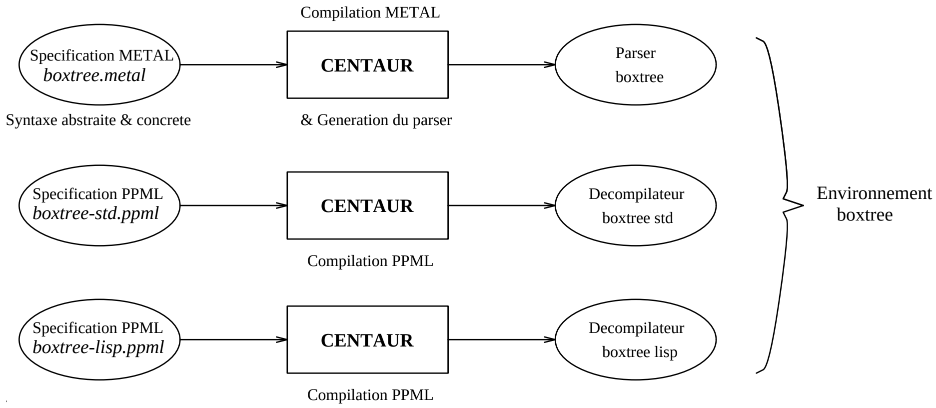

FIG. 3.2 - *Génération de l'environnement Boxtree*

3.3 Phase de traduction

Comme nous l'avons vu au paragraphe 1.2, on souhaite obtenir un arbre de boites à partir d'une spécification Ppml*. Pour cela, une première étape consiste à écrire une grammaire qui génère une grammaire qui spécifie une telle traduction. Dans la suite de ce paragraphe nous allons montrer les principaux schémas de traduction.

3.3.1 Les principaux schémas de traduction

Rappelons qu'il faut mettre en correspondance les structures des deux syntaxes abstraites Ppml* et OLGA.

Le principe général de traduction est le suivant :

> *Les déclarations de constantes Ppml* sont transformées en déclarations de constantes OLGA.*
> *Chaque règle de formatage Ppml est transformée en une production, des déclarations et une règle sémantique OLGA.*

Si le rapprochement est facile pour les grandes structures, il devient plus complexe à représenter dès qu'un phylum a plusieurs opérateurs ou qu'une même structure Ppml* doit être rapprochée de plusieurs structures OLGA.

traduction des grandes structures

Ainsi au niveau le plus élevé, il est naturel de faire correspondre la racine des deux arbres (Fig. 3.7) et les déclarations de constantes (Fig. 3.8) puisque que de telles déclarations sont possibles en OLGA.

Ce type de traduction est simple et ne pose pas de problème.

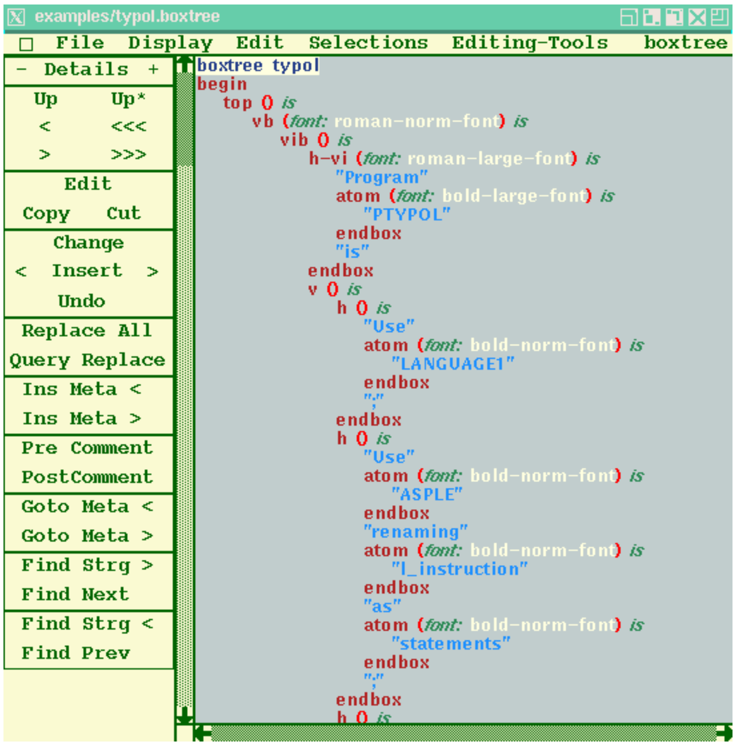

FIG. 3.3 - *Exemple de spécification Boxtree*

traduction des règles

Une règle Ppml* est composée de deux parties (cf. 1.2.2 page 8) :

$$pattern \longrightarrow format$$

- **le pattern** est traduit en une production OLGA, mais il existe deux types de productions OLGA :
  - les productions simples,
  - les productions avec déclarations.

  Et chacun d'eux peut être soit une production pour un opérateur de liste, soit une production pour un opérateur d'arité fixe. Il faut donc déterminer quel type de production nous devons générer.

  En OLGA, les listes se traitent a l'aide de l'opérateur *map* et dans notre cas nous avons besoin d'une telle construction lorsque le pattern est sur un opérateur de liste.

  Ainsi si nous avons un pattern de liste, la production est une production pour opérateur de liste avec déclaration d'un map (Fig. 3.10), sinon si nous

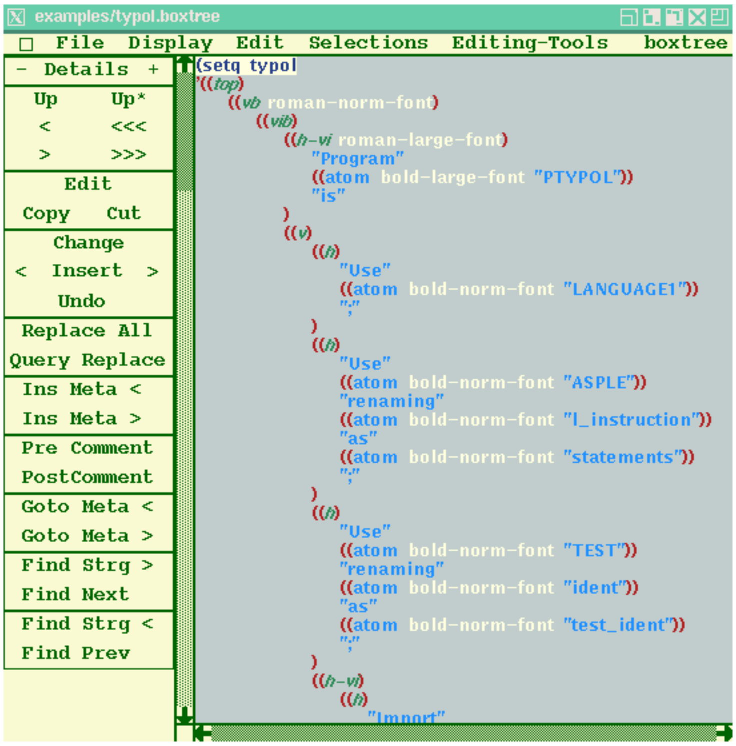

FIG. 3.4 - *Décompilation Lisp d'un arbre Boxtree*

avons un pattern d'arité fixe, il suffit de construire une production simple (Fig. 3.9). La détermination du type de pattern se fait avec un attribut synthétisé de type booléen *$Sub-list-op* qui permet de "remonter" cette indication le long de l'arbre.

– **le format** est traduit en une règle sémantique qui doit spécifier la construction d'un arbre de boites. Pour cela on applique le principe suivant :

> *Pour chaque boite Ppml* apparaissant dans le format, nous construisons une boite Boxtree.*
>
> *Une boite Boxtree ayant deux fils, une liste de sous-boites (***sons***) et un style (***style***).*

Exemple

au format Ppml :

```
[<h> A [<v> B]]
```

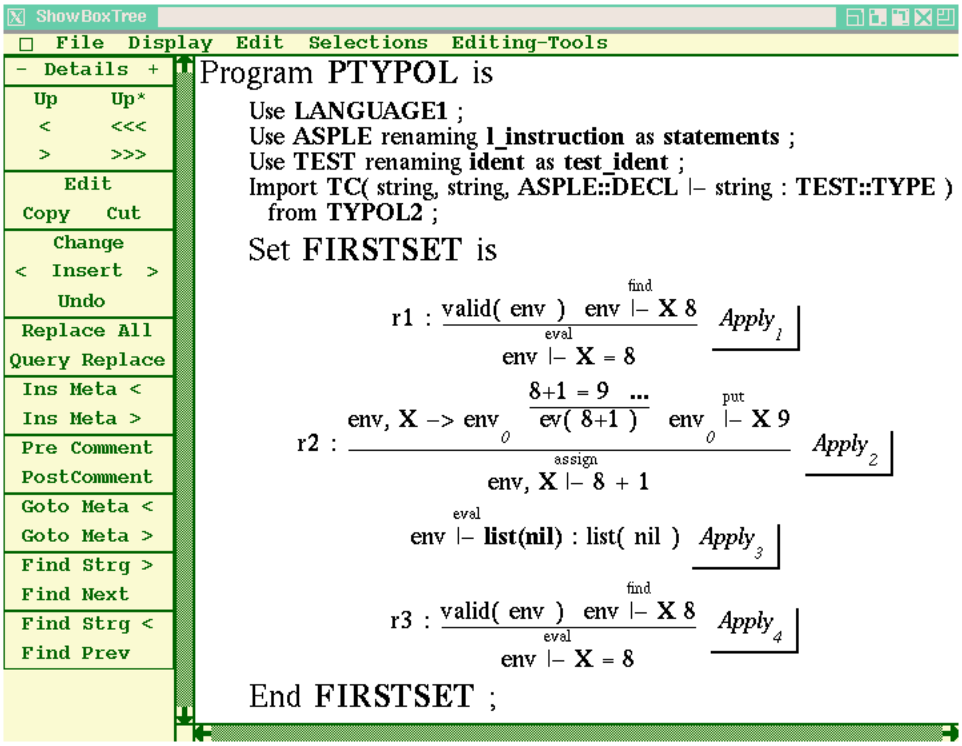

Fig. 3.5 - *Affichage par FIGUE*

correspond l'arbre de boites (simplifié):

```
boxtree(sons(A, boxtree(sons(B),
                style(v))),
        style(h))
```

Ici le problème intervient au niveau de la construction de la liste de fils (opérateur de liste *sons*). En OLGA, il faut utiliser les fonctions sons-post ou sons-pre pour construire une liste. L'exemple ci-dessus devient :

```
boxtree(sons-post(sons(A), boxtree(sons(B),
                           style(v))),
        style(h))
```

Nous avons donc écrit une fonction qui permet d'effectuer une telle transformation pour pouvoir construire correctement l'arbre de boites.

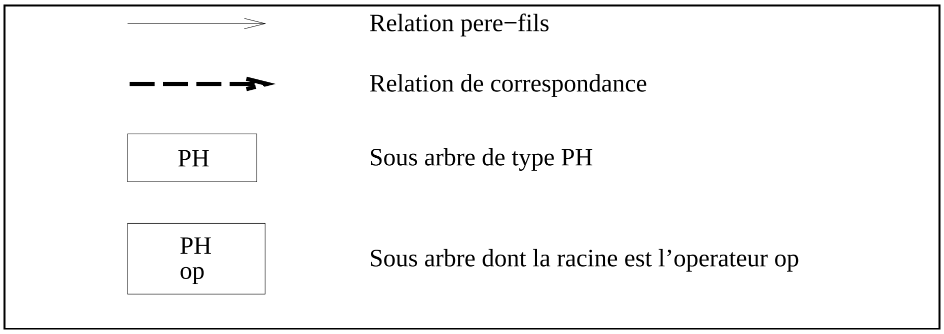

FIG. 3.6 - *Légende des schémas de rapprochement*

3.3.2 Traducteur incrémental

A la fin de l'étape précédente, nous obtenons un arbre représentant une grammaire OLGA, pour la compiler avec FNC-2, il est nécessaire de le décompiler pour obtenir un fichier "texte" de cette grammaire (Fig. 3.11).

A partir de ce fichier, un évaluateur incrémental peut être généré. Cet évaluateur prend en entrée un arbre du formalisme pour lequel la spécification Ppml* a été définie, les éventuelles modifications sur cet arbre et fournit un arbre de boites Boxtree. Les modifications sont obtenues grâce aux *paths de modification.*

3.4 Phase de visualisation

A ce stade, nous disposons d'un arbre de boites Boxtree, qu'il faut visualiser avec FIGUE. Dans cette étape, il faut traiter un nouveau problème de paths de modification :

3.4.1 Le problème des paths de modification

Au paragraphe précédent, nous avons vu que l'arbre Boxtree est recalculé après chaque modification, nous devons donc aussi reformater la page FIGUE, pour cela, nous devons fournir a FIGUE des paths de modification.

Les paths de modification de FIGUE ressemblent beaucoup aux paths de modification du CENTAUR, néanmoins ceci n'est pas suffisant pour passer facilement d'un path CENTAUR à un path FIGUE : il faudrait que les arbres Boxtree et que les arbres FIGUE soient de structure proche. C'est le cas; les arbres FIGUE ont guidés la définition du formalisme Boxtree dans l'optique de simplifier la résolution de ce problème.

Une fonction permet de passer d'un path de modification Boxtree à un path de modification FIGUE.

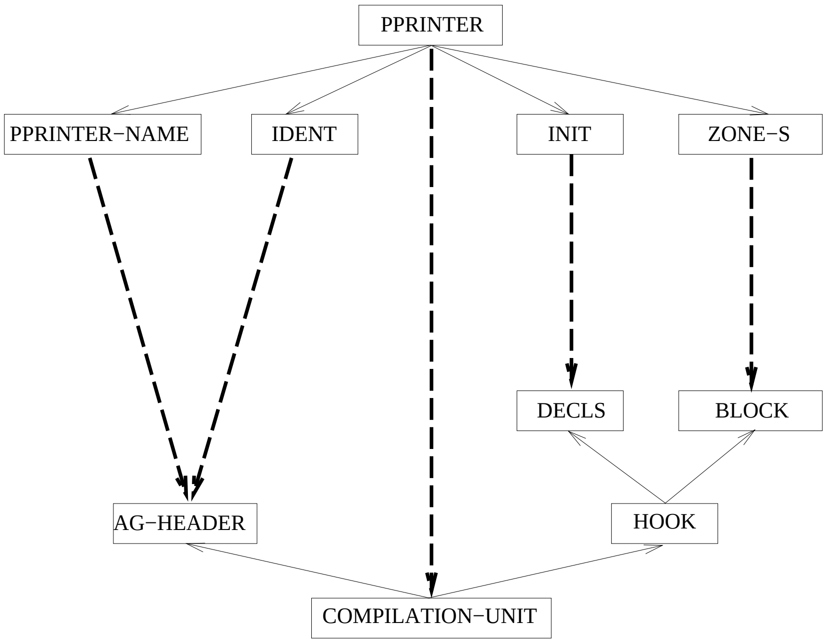

FIG. 3.7 - *Rapprochement des structures Ppml* (en haut) et OLGA (en bas)*

3.5 Architecture du compilateur

Nous travaillons sur deux systèmes CENTAUR et FNC-2 et chacun d'eux, utilise son propre formalisme pour décrire la syntaxe abstraite. En CENTAUR le formalisme utilisé est *Metal* et en FNC-2 il s'agit de *asx*. C'est pour cela que nous avons parfois besoin de l'une et/ou de l'autre spécification selon que nous travaillons sous CENTAUR, sous FNC-2 ou sous les deux systèmes. Il existe une fonction qui génère une spécification asx à partir d'un formalisme CENTAUR. Le système global peut être représenté par deux schémas :

– étape de compilation (Fig. 3.12).

  Les spécifications représentées sur ce schéma sont :

  – **ppml.asx**, la syntaxe abstraite (ou un sous-ensemble) du langage PPML,
  – **olga.asx**, la syntaxe abstraite (ou un sous-ensemble) du langage OLGA,
  – **ppml-olga.olga**, une grammaire attribuée effectuant la traduction d'un arbre PPML en un arbre OLGA,
  – **olga.ppml**, la spécification de décompilation d'un arbre OLGA,

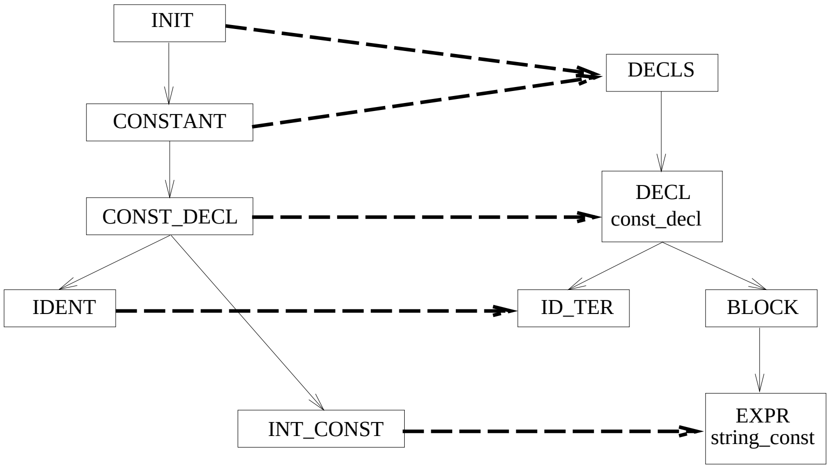

FIG. 3.8 - *Rapprochement des déclarations de constantes Ppml* (à gauche) et OLGA (à droite)*

  - **olga.metal**, la syntaxe abstraite d'un arbre OLGA,
  - **X-boxtree.olga**, la grammaire attribuée résultat de cette étape,
  - **X.ppml**, la spécification de décompilation PPML, qui est l'entrée de cette étape.

- étape d'évaluation (Fig. 3.13).

  Les spécifications représentées sur ce schéma sont :

  - **X.asx**, la syntaxe abstraite du langage X,
  - **Boxtree.asx**, la syntaxe abstraite de Boxtree,
  - **boxtree.ppml**, la spécification de décompilation d'un arbre Boxtree,
  - **boxtree.metal**, la spécification du formalisme Boxtree,
  - **X-boxtree.olga**, la grammaire attribuée résultat de l'étape précédente.

3.6 L'exemple minimath

La mise au point et les tests du compilateur ont été fait sur un petit formalisme : *minimath* qui permet d'éditer des expressions contenant des racines et des intégrales.

3.6.1 Syntaxe abstraite de minimath

```
abstract syntax
```

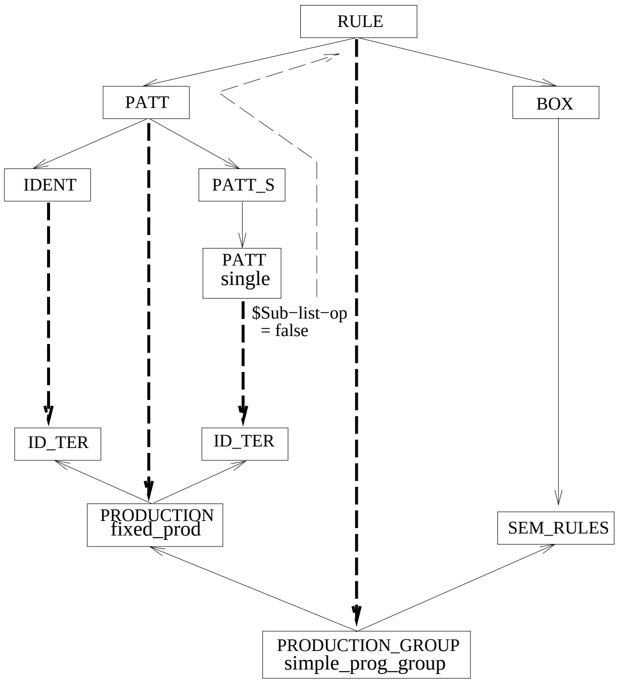

FIG. 3.9 - *Traduction d'un pattern d'arité fixe*

```
MAIN  ::= main ;
main  -> PROGRAM ;
PROGRAM  ::= program program_vide ;
program  -> INST PROGRAM ;
program_vide  -> ;
INST  ::= sqrt_inst integral_inst ;
sqrt_inst  -> STRING EXPR ;
sqrt_expr  -> STRING EXPR ;
EXPR  ::= sqrt_expr integral_expr chaine_expr ;
integral_inst  -> STRING STRING EXPR STRING ;
integral_expr  -> STRING STRING EXPR STRING ;
STRING  ::= chaine_string ;
chaine_expr  -> STRING ;
chaine_string  -> implemented as IDENTIFIER ;
```

3.6.2 L'environnement minimath

Nous avons mis en place dans cet environnement l'appel automatique à l'évaluateur incrémental lors d'une modification et l'affichage incrémental par FIGUE de l'arbre de boites.

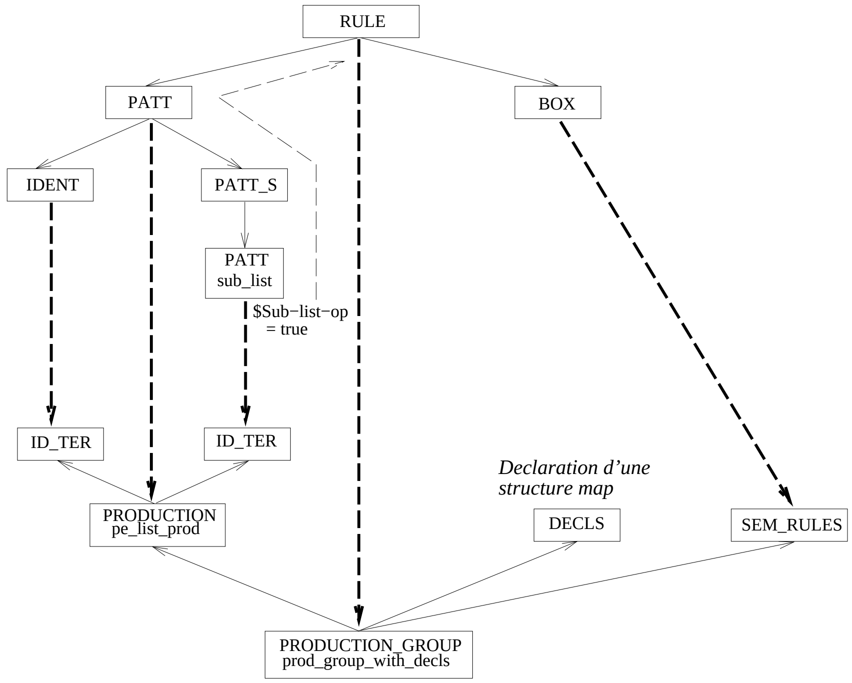

FIG. 3.10 - *Traduction d'un pattern de liste*

Nous pouvons observer les différentes étapes de la décompilation graphique d'un arbre minimath :

- la décompilation initiale de l'arbre (Fig. 3.14),
- l'arbre Boxtree intermédiaire (Fig. 3.15),
- et enfin, la décompilation graphique par FIGUE (Fig. 3.16).

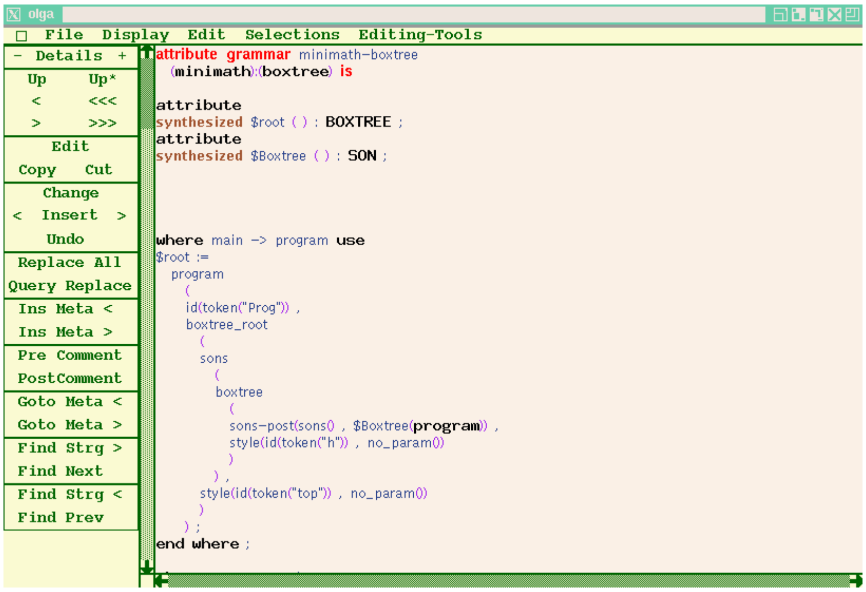

FIG. 3.11 - *Exemple de décompilation d'un arbre OLGA*

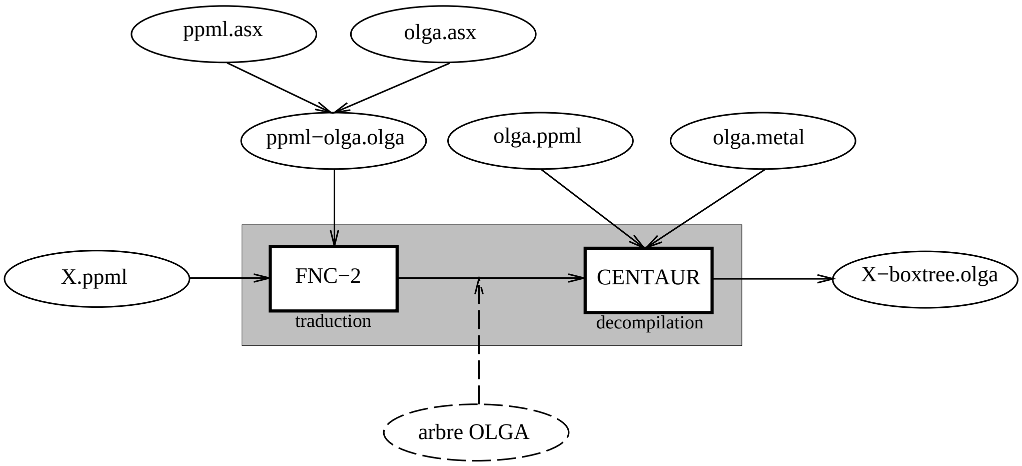

FIG. 3.12 - *Schéma de l'étape de compilation*

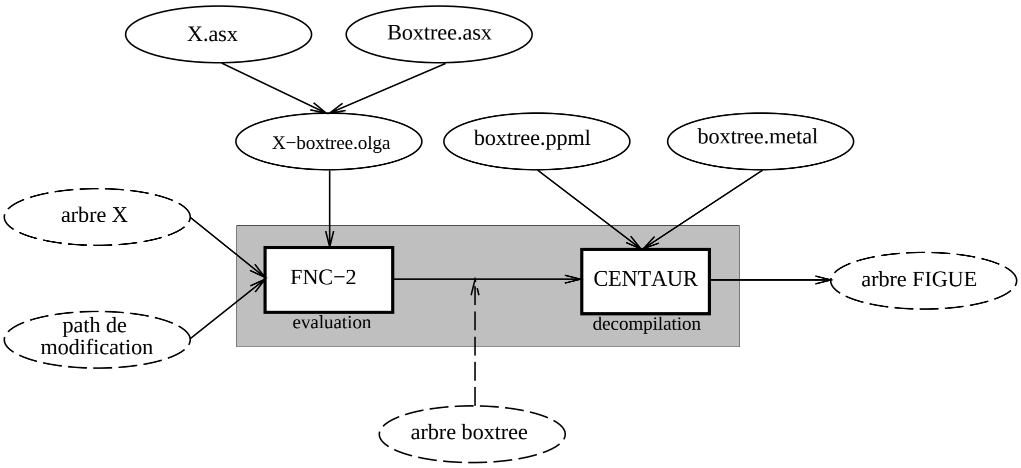

FIG. 3.13 - *Schéma de l'étape d'évaluation*

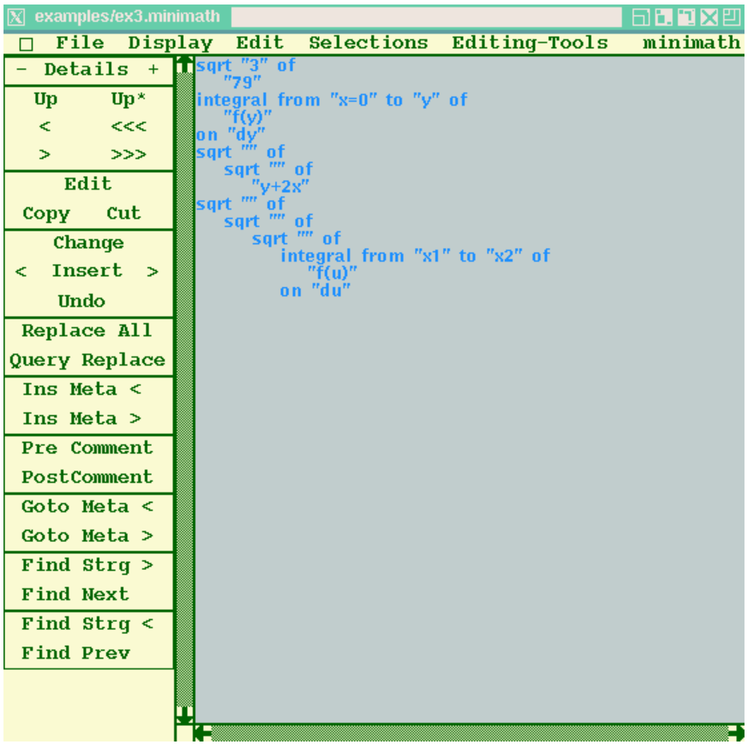

FIG. 3.14 - *Exemple d'arbre minimath*

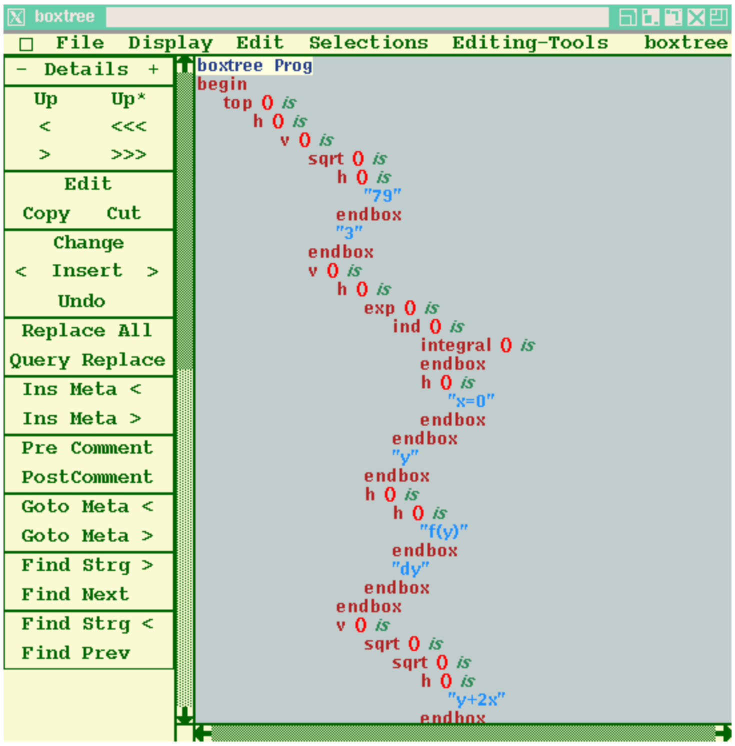

FIG. 3.15 - *Arbre Boxtree intermédiaire*

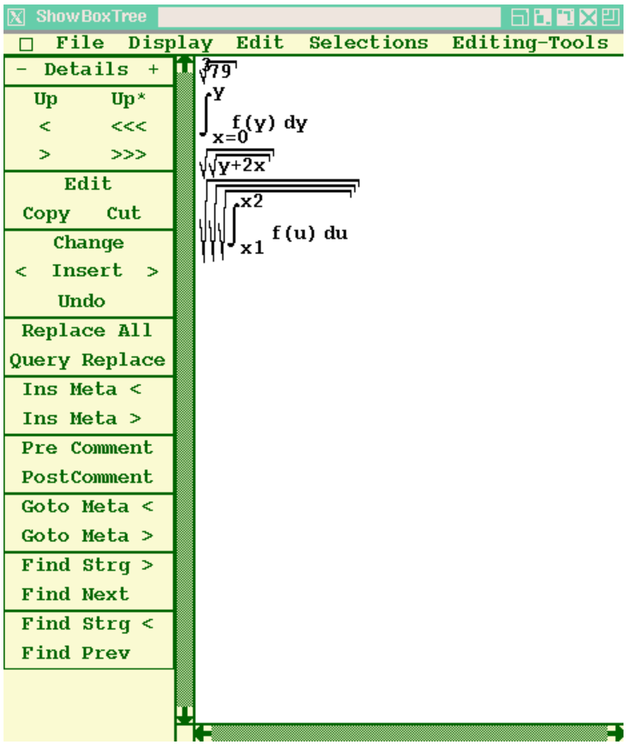

FIG. 3.16 - *Décompilation graphique par FIGUE*

Conclusion

Dans un premier temps nous avons spécifié un formalisme pour les arbres de boites, c'est le formalisme *Boxtree.* il devait nous permettre :

- de manipuler facilement les arbres de boites,
- de passer à FIGUE par décompilation Lisp,
- de traiter aisément le problème des paths de modification.

Ces objectifs ont été tenus par le fait, entre autres, que le formalisme Boxtree est proche du formalisme reconnu par FIGUE (les deux dernier points sont ainsi plus aisés à résoudre).

Dans un second temps, nous avons écrit un nouveau compilateur pour Ppml*, basé sur une grammaire attribuée et générant une autre grammaire attribuée, puis nous avons résolu les problèmes liés aux path de modifications. Ces problèmes interviennent à deux niveaux :

1. entre un arbre VTP et un arbre Boxtree,
2. entre un arbre Boxtree et un arbre FIGUE.

Nous pouvons remarquer que l'incrémetalité intervient aussi à deux niveaux, en effet l'évaluateur obtenu après compilation de la grammaire générée, traduit de façon incrémentale, un arbre VTP en un arbre Boxtree et le réaffichage d'une page par FIGUE, est aussi réalisé incrémentalement.

De nombreuses spécifications ont été nécessaires pour le développement du compilateur, certaines ont été récupérées (**R**), d'autres générées (**G**) ou écrites (**E**).

1. Les spécifications CENTAUR sont :
   - pour le formalisme Boxtree :
     - Boxtree.metal (E),
     - Boxtree-std.ppml (E),
     - Boxtree-lisp.ppml (E),
     - Boxtree.ll (E).

- pour le formlisme Ppml* :
  - ppml*.metal récupérée puis modifiée,
  - ppml*-std.ppml récupérée puis modifiée,
  - ppml*.ll (E).
- pour le formalisme olga :
  - olga_simpli.metal (E),
  - olga_simpli-std.ppml (E).

2. Les spécifications FNC-2 sont :
   - les syntaxes abstraites :
     - boxtree.asx (G),
     - ppml*.asx (G),
     - olga_simpli.asx (R).
   - la grammaire attribuée :
     - ppml*-olga.olga (E).

Perspectives

Le compilateur obtenu semble fonctionner correctement, néanmoins, le sous-ensemble Ppml*, n'est pas suffisant pour pouvoir compiler la majorité des spécifications Ppml.

Il faudrait l'étendre (même au delà du Ppml actuel), et s'occuper des problèmes de sélection et d'édition dans la fenêtre graphique FIGUE. En effet il serait souhaitable de pouvoir éditer directement dans la fenêtre graphique, mais pour cela il faut s'occuper de la mise en correspondance des différents arbres (Fig. 3.17), il est nécessaire de savoir que tel sous-arbre (ou nœud) de l'arbre VTP correspond à tel sous-arbre (ou nœud) de l'arbre Boxtree et de même entre l'arbre Boxtree et FIGUE afin de gérer correctement la sélection et l'édition.

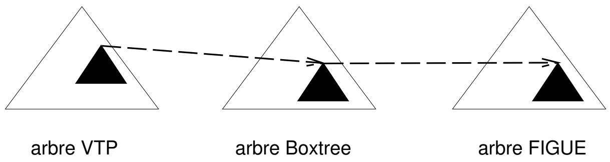

FIG. 3.17 - *Correspondance entre les arbres VTP, Boxtree et FIGUE*

De plus, la réalisation de ce compilateur a montrer l'efficacité des grammaires attribuées à traiter ce type de problème et l'évaluation incrémentale donne de bonnes performances dans le domaine de l'édition et de l'affichage graphique.

Enfin, outre, l'intérêt de réaliser un outil, la conception de ce nouveau compilateur, a permis de tester, et parfois d'améliorer (enfin, je l'espère), le système

FNC-2 dans sa version Lisp. En effet, la mise au point de ce nouveau compilateur est l'une des premières application écrite en utilisant FNC-2 (les deux premières applications majeures sont FNC-2[1] et PPAT [Pla89]), et sa réalisation a soulevé des problèmes qui n'avaient pas été traités (surcharge des opérateurs, opérateurs de liste dans la version incrémentale,...). Ainsi lorsque le système FNC-2 sera au point, il sera possible d'avoir un compilateur Ppml incrémental.

1. FNC-2 étant presque totalement auto-amorcé, c'est la première application du système FNC-2

Troisième partie

Annexes

Annexe A

Le formalisme Boxtree

A.1 Syntaxe abstraite et concrète : Boxtree.metal

```
-- $Id: metalmodel,v 1.2 94/04/14 15:56:17 croap1 Exp $
definition of boxtree is
   chapter 'axiom'
      rules
         <axiom> ::= <root> ;
            <root>
   end chapter ;

   chapter 'root'
      rules
         <root> ::= "boxtree" <id> "begin" <boxtree_root> "end" ;
            program (<id>, <boxtree_root>)

      abstract syntax
         PROGRAM  ::= program ;
         program  -> ID BOXT ;
         BOXT  ::= boxtree_root  ;
   end chapter ;

   chapter 'boxtree'
      rules
         <boxtree_root> ::= <style> "is" <sons> "endbox" ;
            boxtree_root (<sons>, <style>)
         <boxtree> ::= <style> "is" <sons> "endbox" ;
            boxtree (<sons>, <style>)

      abstract syntax
         boxtree_root  -> SONS STYLE ;
         boxtree  -> SONS STYLE ;
         SONS  ::= sons ;
         STYLE  ::= style ;
```

```
end chapter ;

chapter 'sons'
   rules
      <sons> ::= ;
         sons-list (())
      <sons> ::= <sons> <son> ;
         sons-post (<sons>, <son>)

   abstract syntax
      sons  -> SON * ... ;
      SON  ::= boxtree str nombre ;
end chapter ;

chapter 'son'
   rules
      <son> ::= <boxtree> ;
         <boxtree>
      <son> ::= <str> ;
         <str>
      <son> ::= <nombre> ;
         <nombre>
end chapter ;

chapter 'style'
   rules
      <style> ::= <id> "(" <param> ")" ;
         style (<id>, <param>)

   abstract syntax
      style  -> ID PARAM ;
      PARAM  ::= no_param param param1 param2 ;
end chapter ;

chapter 'param'
   rules
      <param> ::= ;
         no_param ()
      <param> ::= "font" ":" <id> ";" "indent" ":" <lst_ind> ;
         param (<id>, <lst_ind>)
      <param> ::= "indent" ":" <lst_ind> ;
         param1 (<lst_ind>)
      <param> ::= "font" ":" <id> ;
         param2 (<id>)

   abstract syntax
      no_param  -> implemented as SINGLETON ;
      param  -> ID LST_IND ;
      param1  -> LST_IND ;
      param2  -> ID ;
```

```
      LST_IND  ::= lst_ind ;
end chapter ;

chapter 'lst-ind'
   rules
      <lst_ind> ::= ;
         lst_ind-list (())
      <lst_ind> ::= <lst_ind> <number> ;
         lst_ind-post (<lst_ind>, <number>)

   abstract syntax
      lst_ind  -> NUMBER * ... ;
      NUMBER  ::= number ;
end chapter ;

chapter 'terminals'
   rules
      <id> ::= %ID ;
         id-atom(%ID)
      <texte> ::= %STRING ;
         texte-atom(%STRING)
      <number> ::= %NUMBER ;
         number-atom(%NUMBER)
      <nombre> ::= <number> ;
         nombre (<number>)
      <str> ::= <texte> ;
         str (<texte>)

   abstract syntax
      number  -> implemented as IDENTIFIER ;
      id  -> implemented as IDENTIFIER ;
      texte  -> implemented as STRING ;
      nombre  -> NUMBER ;
      str  -> STR ;
      STR  ::= texte ;
      ID  ::= id ;
end chapter ;

chapter 'entry points'
   rules
      <axiom> ::= "[PROGRAM]" <root> ;
         <root>
      <axiom> ::= "[BOXT]" <boxtree_root> ;
         <boxtree_root>
      <axiom> ::= "[SON]" <boxtree> ;
         <boxtree>
      <axiom> ::= "[SONS]" <sons> ;
         <sons>
      <axiom> ::= "[SON]" <son> ;
         <son>
```

```
         <axiom> ::= "[SON]" <str> ;
            <str>
         <axiom> ::= "[SON]" <nombre> ;
            <nombre>
         <axiom> ::= "[STYLE]" <style> ;
            <style>
         <axiom> ::= "[PARAM]" <param> ;
            <param>
         <axiom> ::= "[LST_IND]" <lst_ind> ;
            <lst_ind>
         <axiom> ::= "[STR]" <texte> ;
            <texte>
         <axiom> ::= "[NUMBER]" <number> ;
            <number>
         <axiom> ::= "[ID]" <id> ;
            <id>
   end chapter ;
end definition
```

A.2 Spécifications de décompilation

A.2.1 Boxtree-std.ppml

```
prettyprinter std of boxtree is

rules

*x !0 -> [<h> "..."];

program(*trname, *boxt) ->
    [<v>
     in class = Debut : [<h 1> "boxtree" *trname]
     [<v 3,0> in class = mot_cle : "begin" *boxt] in class = mot_cle : "end"];

boxtree_root(*sons, *style) ->
    [<v>
     [<v 3,0> in class = Style : [<h 1> *style "is"] *sons]
     in class = mot_cle : "endbox"];

boxtree(*sons, *style) ->
    [<v>
     [<v 3,0> in class = Style : [<h 1> *style "is"] *sons]
     in class = mot_cle : "endbox"];

style(*cbname, *param) ->
    [<h 1>
     in class = mot_cle : [<h> *cbname]
     [<h> in class = Par : "(" *param in class = Par : ")"]];
```

```
sons(**son) -> [<h> in class = Son : [<v> (**son)]];

no_param -> [<h>];

param(*font, *lst_ind) ->
    [<h 1>
     "font:" in class = Param : [<h> *font] "indent:"
     in class = Param : [<h> *lst_ind]];

param1(*lst_ind) -> [<h 1> "indent:" in class = Param : [<h> *lst_ind]];

param2(*font) -> [<h 1> "font:" in class = Param : [<h> *font]];

lst_ind(**dx, *dx1) -> [<h 1> (**dx) *dx1];

nombre(*number) -> [<h> *number];

str(*texte) -> [<h> *texte];

end prettyprinter
```

A.2.2 Boxtree-lisp.ppml

```
prettyprinter lisp of boxtree is

rules

*x !0 -> [<h> "..."];

program(*trname, *boxt) ->
    [<v>
     in class = Debut : [<h 1> "(setq" *trname]
     [<h> in class = Parenth : "'" *boxt] in class = Parenth : ")"];

boxtree_root(*sons, style(id 'atom', *param)) ->
    [<h>
     in class = Parenth : "((" [<h 1> "atom" *param *sons]
     in class = Parenth : "))"];

boxtree_root(*sons, style(id 'box', *param)) ->
    [<h>
     in class = Parenth : "((" [<h 1> "box" *param *sons]
     in class = Parenth : "))"];

boxtree_root(*sons, *style) ->
    [<v>
     [<v 3,0> [<h> in class = Parenth : "(" *style] *sons]
     in class = Parenth : ")"];
```

```
boxtree(*sons, style(id 'atom', *param)) ->
    [<h>
     in class = Parenth : "((" [<h 1> "atom" *param *sons]
     in class = Parenth : "))"];

boxtree(*sons, style(id 'box', *param)) ->
    [<h>
     in class = Parenth : "((" [<h 1> "box" *param *sons]
     in class = Parenth : "))"];

boxtree(*sons, *style) ->
    [<v>
     [<v 3,0> [<h> in class = Parenth : "(" *style] *sons]
     in class = Parenth : ")"];

style(*cbname, *param) ->
    [<h>
     in class = Parenth : "(" in class = Style : [<h 1> *cbname *param]
     in class = Parenth : ")"];

sons(**son) -> [<v> (**son)];

no_param -> [<h>];

param(*font, *lst_ind) -> [<h> in class = Param : [<h 1> *font *lst_ind]];

param1(*lst_ind) -> [<h> in class = Parm : [<h 1> "0" *lst_ind]];

param2(*font) -> [<h> in class = Param : [<h> *font]];

lst_ind(**dx, *dx1) -> [<h 1> (**dx) *dx1];

nombre(*number) -> [<h> *number];

str(*texte) -> [<h> *texte];

end prettyprinter
```

Annexe B

Grammaire new_ppml-olga.olga

```
{--------------------------------------------------
 - Grammaire attribuee new_ppml-olga              -
 - prenant en entree                              -
 -   * une specification new_ppml                 -
 - donnant en sortie                              -
 -   * une specification olga de transformation   -
 -     d'un arbre en un arbre de boites graphiques. -
 --------------------------------------------------}

attribute grammar new_ppml-olga  ( new_ppml-base ) : ( olga_simpli ) is

{----------------------------}
{- Declaration des fonctions -}
{----------------------------}

{----------------------------------------------------}
{- Fonction transformant une liste d'expression OLGA -}
{- de type SONS en une expression OLGA de type SONS  -}
{----------------------------------------------------}

{- la traduction en Lisp est incorrecte pour l'instant -}

function lst-expr-box_TO_expr-lst-box(
                sons-post : ID_TER ;
                LOA : ACTUALS;
                Result : EXPR) : EXPR is
 case LOA is
   actuals-pre(A : ACTUAL, LOA-T : ACTUALS) :
     lst-expr-box_TO_expr-lst-box(
         sons-post,
         LOA-T,
```

```
         func_call(
           sons-post,
           actuals-pre(
             actual(Result), actuals(A))));
   actuals (A :ACTUAL) :
      func_call(
          sons-post,
          actuals-pre(
              actual(Result), actuals(A))) ;
   other :
       Result ;
 end case
end function ;


{-----------------------------}
{- Declaration des attributs -}
{-----------------------------}
attribute

synthesized $Compilation-Unit (PPRINTER) : COMPILATION_UNIT;
synthesized $Ag-name (IDENT) : ID_TER;
synthesized $Decls (INIT,CONSTANT) : DECLS;
synthesized $Decl (CONST_DECL) : DECL;
synthesized $Block (ZONE_S) : BLOCK;
synthesized $Const-ident (IDENT) : ID_TER ;
synthesized $String-ident (IDENT, INT_CONST) : STRING ;
synthesized $Production (ZONE) : SEMANTICS ;
synthesized $Rule (RULE) : PRODUCTION_GROUP;
synthesized $Patt (PATT) : PRODUCTION ;
synthesized $Box (BOX) : SEM_RULES ;
synthesized $Patt-s (PATT_S) : PROD_RHS ;
synthesized $Patt1 (PATT, IDENT) : ID_TER ;
synthesized $Single-s-patt (PATT_S) : bool;
synthesized $Sub-list (PATT,ELEM) : bool;
synthesized $Sub-list-op (PATT,PATT_S) : bool;
synthesized $Patt-s-single (PATT_S) : ID_TER;
synthesized $Map-decl (PATT) : DECLS;
synthesized $Id-int (ID_INTEGER) : EXPR;
synthesized $Param (S_PARAM_S) : ACTUALS;
synthesized $Elem(SEP, ELEM) : EXPR;
synthesized $Elem-s (ELEM_S) : EXPR;
synthesized $Actuals (ELEM_S) : ACTUALS ;
synthesized $Atom-arg (ATOM_ARG) : ID_TER ;
synthesized $Atom-bool (PATT,ATOM_ARG) : bool ;
inherited $Atom-bool-h (BOX) : bool;
inherited $First-rule-h (RULE, BOX) : bool;
compound $Bug (PATT) : bool ;
```

```
{----------------------}
{- Regles semantiques -}
{----------------------}

where revlist::ELEM -> ELEM_S  use
   $Elem :=  $Elem-s(ELEM_S);
   $Sub-list := false;
end where ;

where seplist::ELEM -> ELEM_S  use
   $Elem := func_call(id_ter(token("")),$Actuals(ELEM_S));
   $Sub-list := true;
end where ;

where s_param_s -> ID_INTEGER  * declare
value
 expr-lst-box-empty : EXPR := func_call(id_ter(token("lst_ind")),
                                         actuals());
 temp := map left actuals-post
              value actuals()
              other actual(
                     func_call(
                      id_ter(token("id")),
                      actuals(
                       actual(
                        func_call(
                          id_ter(token("token")),
                          actuals(
                            actual($Id-int(ID_INTEGER))))))))
           end map;
 use $Param := case arity is
               0 : actuals(
                            actual(
                             func_call(
                              id_ter(token("no_param")),
                              actuals())));
               other : actuals(actual(
                        func_call(
                         id_ter(token("param1")),
                         actuals(actual(
                         lst-expr-box_TO_expr-lst-box(
                         id_ter(token("lst_ind-post")),
                                temp,
                                expr-lst-box-empty))))));
              end case;

end where ;

where sep::DEFAULT_DECL -> IDENT S_PARAM_S  use
```

```
end where ;

where sep::SEP -> IDENT S_PARAM_S  use
   $Elem :=  func_call(
                id_ter(token("style")),
                actuals-pre(
                   actual(
                     func_call(
                       id_ter(token("id")),
                       actuals(
                         actual(func_call(
                                id_ter(token("token")),
                                actuals(
                                 actual(
                                  string_const($String-ident(IDENT))))))))),
                   $Param(S_PARAM_S)));
end where ;

where sep::ELEM -> IDENT S_PARAM_S  use
   $Elem := func_call(
                 id_ter(token("style")),
                 actuals-pre(
                   actual(string_const($String-ident(IDENT))),
                   $Param(S_PARAM_S)));
end where ;

where box::ELEM -> SEP ELEM_S  use
   $Elem :=  func_call(
                  id_ter(token("boxtree")),
                  actuals-pre(
                    actual($Elem-s(ELEM_S)),
                    actuals(actual($Elem(SEP)))));
   $Sub-list := false;
end where ;

where box::BOX -> SEP ELEM_S  use
   $Box := if $First-rule-h
   then
     if $Atom-bool-h
     then
        sem_rules(
             simple_rule(
               attrib_occ_lhs(attr(token("$root"))),
               func_call(
                  id_ter(token("program")),
                  actuals-pre(
                    actual(
                      func_call(
                         id_ter(token("id")),
                         actuals(
```

```
         actual(
           func_call(
             id_ter(token("token")),
             actuals(
               actual(
                 string_const(
                    string(token("Prog")))))))))),
actuals(
 actual(func_call(
   id_ter(token("boxtree_root")),
   actuals-pre(
    actual(
     func_call(
       id_ter(token("sons")),
       actuals(
        actual(
         func_call(
          id_ter(token("str")),
          actuals(
            actual(func_call(
                    id_ter(token("texte")),
                    actuals(actual(
                      func_call(
                       id_ter(token("token")),
                       actuals(actual(
                         func_call(
                          id_ter(token("string")),
                          actuals(actual(
                           applied_id(unqual_id(
                            id_ter(token("M"))))))))))))))))))),
 actuals(
  actual(
   func_call(
     id_ter(token("style")),
     actuals-pre(
       actual(
         func_call(
            id_ter(token("id")),
            actuals(
               actual(
                 func_call(
                    id_ter(token("token")),
                    actuals(
                      actual(
                        string_const(
                         string(token("top")))))))))),
       actuals(
         actual(
           func_call(
             id_ter(token("no_param")),
```

```
                           actuals()))))))))))))))
else   { fin if Atom-bool}
   sem_rules(
        simple_rule(
          attrib_occ_lhs(attr(token("$root"))),
          func_call(
             id_ter(token("program")),
             actuals-pre(
               actual(
                 func_call(
                    id_ter(token("id")),
                    actuals(
                       actual(
                         func_call(
                           id_ter(token("token")),
                           actuals(
                             actual(
                               string_const(
                                  string(token("Prog")))))))))),
              actuals(
               actual(func_call(
                 id_ter(token("boxtree_root")),
                 actuals-pre(
                  actual(
                   func_call(
                     id_ter(token("sons")),
                     actuals(
                      actual(
                       func_call(
                        id_ter(token("boxtree")),
                        actuals-pre(
                          actual($Elem-s(ELEM_S)),
                          actuals(actual($Elem(SEP)))))))))),
               actuals(
                actual(
                 func_call(
                   id_ter(token("style")),
                   actuals-pre(
                     actual(
                       func_call(
                          id_ter(token("id")),
                          actuals(
                             actual(
                               func_call(
                                  id_ter(token("token")),
                                  actuals(
                                    actual(
                                      string_const(
                                       string(token("top")))))))))),
                     actuals(
```

```
                            actual(
                              func_call(
                                id_ter(token("no_param")),
                                actuals()))))))))))))))



end if  {fin else Atom-bool-h }
   else
     if $Atom-bool-h
     then
       sem_rules(
             simple_rule(
               attrib_occ_lhs(attr(token("$Boxtree"))),
               func_call(
                 id_ter(token("str")),
                 actuals(
                   actual(func_call(
                      id_ter(token("texte")),
                      actuals(actual(
                        func_call(
                          id_ter(token("token")),
                          actuals(actual(
                            func_call(
                              id_ter(token("string")),
                              actuals(actual(
                                 applied_id(unqual_id(
                                        id_ter(token("M")))))))))))))))))

     else
       sem_rules(
             simple_rule(
               attrib_occ_lhs(attr(token("$Boxtree"))),
               func_call(
                  id_ter(token("boxtree")) ,
                  actuals-pre(
                    actual($Elem-s(ELEM_S)),
                    actuals(actual($Elem(SEP)))))))
     end if
   end if;
end where ;

where elem_s -> ELEM  * declare
value
   lst-expr-box : ACTUALS :=
              map left actuals-post
                value actuals()
                other actual($Elem(ELEM))
              end map;
   expr-lst-box-empty : EXPR := func_call(
```

```
                                  id_ter(token("sons")),
                                  actuals());
 use
   $Elem-s := case arity is
               0: expr-lst-box-empty;
               other : if $Sub-list(ELEM.last)
                       then
                         func_call(
                           id_ter(token("")),
                           lst-expr-box)
                       else
                         lst-expr-box_TO_expr-lst-box(
                         id_ter(token("sons-post")),
                         lst-expr-box ,
                         expr-lst-box-empty)
                       end if;
              end case;
$Actuals := lst-expr-box ;
end where ;

where atom_patt -> IDENT ATOM_ARG  use
   $Patt := fixed_prod($Const-ident(IDENT),prod_rhs($Atom-arg(ATOM_ARG)));
   $Sub-list-op := false ;
   $Atom-bool := true ;
end where ;

where patt_s -> PATT  * use
   $h.Bug(PATT) := case position is
             first : true;
             other : $s.Bug(PATT.left);
           end case ;
   $Patt-s :=   map left prod_rhs-post
                          value prod_rhs()
                          other $Patt1(PATT)
                         end map;
   $Atom-bool := false;
   $Sub-list-op := case arity is
                       0 : false ;
                       other : $Sub-list(PATT.last) and $s.Bug(PATT.last);
                   end case;
   $Single-s-patt := case arity is
                       0 : false;
                       other :
                         if $Sub-list(PATT.last) then
                           true
                         else false
                         end if;
                     end case;
   $Patt-s-single := case arity is
                       0 :  id_ter(token("")) ;
```

```
                      other :
                        if $Sub-list(PATT.last) then
                          $Patt1(PATT.last)
                        else id_ter(token(""))
                        end if;
                    end case;
   $Map-decl := case arity is
                  0 : decls() ;
                  other :$Map-decl(PATT.last);
                end case;
end where ;

where node_patt -> IDENT PATT_S  use
   $Patt := if $Single-s-patt(PATT_S) then
               pe_list_prod(
               $Const-ident(IDENT),
               $Patt-s-single(PATT_S))
            else
               fixed_prod(
               $Const-ident(IDENT),
               $Patt-s(PATT_S))
            end if;
   $Sub-list-op := $Sub-list-op(PATT_S) ;
   $Atom-bool := false ;
end where ;

where sublist_metavar::ELEM -> M  use
   $Elem := applied_id(unqual_id(id_ter(token("temp")))) ;
   $Sub-list := true;
end where ;

where sublist_metavar::ATOM_ARG -> M  use
   $Atom-arg := id_ter(token("M")) ;
   $Atom-bool := true ;
end where ;

where sublist_metavar::PATT -> M  use
   $Patt1 := id_ter(M);
   $Sub-list := true ;
   $Sub-list-op := true ;
   $Atom-bool := false ;
   $Patt := fixed_prod(id_ter(token("")),prod_rhs());
   $Map-decl := decls(
                 value_decl(
                   id_ter(token("temp")),
                   type_id(
                     unqual_id(
                       id_ter(token("SONS")))),
                   block_expr(
                     map_expr(
```

```
                       left_dir(),
                       unqual_id(id_ter(token("sons-post"))),
                       func_call(id_ter(token("sons")),
                           actuals()),
                       actuals(
                           actual(
                             attrib_occ_as_expr(
                               attrib_occ(
                                 attr(token("$Boxtree")) ,
                                 applied_id(unqual_id(id_ter(M) ))))))))));
end where ;

where single_metavar::ELEM -> M  use
   $Elem := func_call(
             id_ter(token("$Boxtree")),
             actuals(
              actual(
               applied_id(unqual_id(id_ter(M) )))));
   $Sub-list := false;
end where ;

where single_metavar::ATOM_ARG -> M  use
   $Atom-arg := id_ter(token("M"));
   $Atom-bool := true;
end where ;

where single_metavar::PATT -> M  use
   $Patt1 := id_ter (M);
   $Sub-list := false;
   $Sub-list-op := false ;
   $Atom-bool := false ;
   $Patt := fixed_prod(id_ter(token("")) ,prod_rhs());
   $Map-decl := decls ();
end where ;

where anonymous_metavar::ATOM_ARG -> void  use
   $Atom-bool:=true;
   $Atom-arg := id_ter(token("M"));
end where ;

where anonymous_metavar::PATT -> void  use
   $Patt1 := id_ter(token("")) ;
   $Patt := fixed_prod(id_ter(token("")) ,prod_rhs());
   $Sub-list := false;
   $Sub-list-op := false ;
   $Map-decl := decls();
   $Atom-bool := false;
end where ;

where terminal -> M  use
```

```
   $Elem := func_call(
             id_ter(token("str")),
             actuals(
               actual(
                func_call(
                 id_ter(token("texte")),
                 actuals(
                  actual(
                   func_call(
                     id_ter(token("token")),
                     actuals(
                      actual(
                        string_const(string(token("\\\"\\\""+M+"\\\"\\\""))))))))))));
   $Sub-list := false;
end where ;

where string::CHAPTER_NAME -> M  use

end where ;

where string::ATOM_ARG -> M  use
   $Atom-bool := true ;
   $Atom-arg := id_ter(token("M"));
end where ;

where int_const::PPRINTER_NAME -> M  use
end where ;

where int_const::INT_CONST -> M  use
   $String-ident := string(token(M)) ;
end where ;

where int_const::ID_INTEGER -> M  use
   $Id-int :=string_const(string(token(M))) ;
end where ;

where ident::PPRINTER_NAME -> M  use
end where ;

where ident::PATT -> M  use
   $Patt1 := id_ter(M) ;
   $Patt := fixed_prod(id_ter(token("")) ,prod_rhs());
   $Sub-list := false;
   $Sub-list-op := false ;
   $Map-decl := decls();
   $Atom-bool := false;
end where ;

where ident::IDENT -> M  use
   $Const-ident := id_ter(token(M)) ;
```

```
   $String-ident := string(token(M)) ;
   $Ag-name := id_ter(token(M+"-boxtree"));
   $Patt1 := id_ter(token(M));
   $Sub-list := false;
   $Map-decl := decls();
end where ;

where ident::ID_INTEGER -> M  use
   $Id-int := string_const(string(M));
end where ;

where ident::CHAPTER_NAME -> token  use

end where ;

where chapt -> CHAPTER_NAME INIT ZONE_S  use
   $Production := productions();
end where ;

where rule -> PATT BOX  use
   $First-rule-h (BOX) := $First-rule-h(rule) ;
   $h.Bug(PATT) := true ;
   $Atom-bool-h (BOX) := $Atom-bool(PATT);
   $Rule := if $Sub-list-op(PATT)  then
                 prod_group_with_decls(
                    $Patt(PATT),
                    $Map-decl(PATT),
                    $Box(BOX))
            else
                 simple_prod_group(
                    $Patt(PATT),
                    $Box(BOX))

            end if;
end where ;

where rule_s -> RULE  + use
   $First-rule-h (RULE):= case position is
                      first : true;
                      other : false ;
                    end case;
   $Production := map left productions-post
                    value productions()
                    other $Rule(RULE)
                  end map;
end where ;

where zone_s -> ZONE  *
declare
value
```

```
   temp : SEMANTICS := map left productions-merge
                value productions()
                other $Production(ZONE)
              end map;
   use
      $Block := block_semantics(temp);
end where ;

where _default -> DEFAULT_DECL  * use

end where ;
where const_decl -> IDENT INT_CONST  use
   $Decl := const_decl(
              $Const-ident(IDENT),
              block_expr(
                string_const($String-ident(INT_CONST))));
   $Decls := decls();
end where ;

where constant -> CONST_DECL  * use
   $Decls := map left decls-post
                value decls()
                other $Decl(CONST_DECL)
              end map;
end where ;

where init -> CONSTANT DEFAULT  use
   $Decls := $Decls(CONSTANT) ;
end where ;

where pprinter -> PPRINTER_NAME IDENT INIT ZONE_S  declare
value
   declaration-att : DECLS := decls-pre(
                                 attribute_decl(
                                  syn_kind(),
                                  attr(token("$root")) ,
                                  phyla_names(),
                                  type_id(
                                    unqual_id(
                                     id_ter(token("BOXTREE")) ))),
                                 decls(
                                  attribute_decl(
                                   syn_kind(),
                                   attr(token("$Boxtree")) ,
                                   phyla_names(),
                                   type_id(
                                     unqual_id(
                                      id_ter(token("SON")) ))))) ;
use
   $Compilation-Unit := simple_AG(
```

```
                          ag_header($Ag-name(IDENT),
                             functional_AG(
                                syntax(
                                   $Const-ident(IDENT)),
                                   syntaxes(
                                       syntax(
                                            id_ter(token("boxtree")) )))),
                           hook_simple_AG(
                              decls-merge(
                                declaration-att ,
                                $Decls(INIT)),
                              $Block(ZONE_S)));
end where ;

end grammar ;
```

Annexe C

Exemple de grammaire générée

La grammaire suivante a été obtenue après compilation de la grammaire présentée au paragraphe 3.1.6 page 36.

```
attribute grammar minimath-boxtree (minimath):(boxtree) is

attribute
synthesized $root ( ) : BOXTREE ;
attribute
synthesized $Boxtree ( ) : SON ;




where main -> program use
$root :=
  program
    (
    id(token("Prog")) ,
    boxtree_root
      (
      sons
        (
        boxtree
          (
          sons-post(sons() , $Boxtree(program)) ,
          style(id(token("h")) , no_param())
          )
        ) ,
      style(id(token("top")) , no_param())
      )
    ) ;
```

```
end where ;

where program -> inst
                 axiom use
$Boxtree :=
  boxtree
    (
    sons-post(sons-post(sons() , $Boxtree(inst)) , $Boxtree(axiom)) ,
    style(id(token("v")) , no_param())
    ) ;
end where ;

where program_vide -> M use
$Boxtree := str(texte(token(string(M)))) ;
end where ;

where integral_inst -> b1
                       b2
                       expr
                       dx use
$Boxtree :=
  boxtree
    (
    sons-post
      (
      sons-post
        (
        sons() ,
        boxtree
          (
          sons-post
            (
            sons-post
              (
              sons() ,
              boxtree
                (
                sons-post
                  (
                  sons-post
                    (
                    sons() ,
                    boxtree(sons() , style(id(token("integral")) , no_param()))
                    ) ,
                  boxtree
                    (
                    sons-post(sons() , $Boxtree(b1)) ,
                    style(id(token("h")) , no_param())
                    )
                  ) ,
```

```
                style(id(token("ind")) , no_param())
                )
              ) ,
            $Boxtree(b2)
            ) ,
          style(id(token("exp")) , no_param())
          )
        ) ,
      boxtree
        (
        sons-post(sons-post(sons() , $Boxtree(expr)) , $Boxtree(dx)) ,
        style(id(token("h")) , no_param())
        )
      ) ,
    style(id(token("h")) , no_param())
    ) ;
end where ;

where sqrt_inst -> deg
                   expr use
$Boxtree :=
  boxtree
    (
    sons-post(sons-post(sons() , $Boxtree(expr)) , $Boxtree(deg)) ,
    style(id(token("sqrt")) , no_param())
    ) ;
end where ;

where integral_expr -> b1
                       b2
                       expr
                       dx use
$Boxtree :=
  boxtree
    (
    sons-post
      (
      sons-post
        (
        sons() ,
        boxtree
          (
          sons-post
            (
            sons-post
              (
              sons() ,
              boxtree
                (
                sons-post
```

```
                  (
                  sons-post
                    (
                    sons() ,
                    boxtree(sons() , style(id(token("integral")) , no_param()))
                    ) ,
                  boxtree
                    (
                    sons-post(sons() , $Boxtree(b1)) ,
                    style(id(token("h")) , no_param())
                    )
                  ) ,
                style(id(token("ind")) , no_param())
                )
              ) ,
            $Boxtree(b2)
            ) ,
          style(id(token("exp")) , no_param())
          )
        ) ,
      boxtree
        (
        sons-post(sons-post(sons() , $Boxtree(expr)) , $Boxtree(dx)) ,
        style(id(token("h")) , no_param())
        )
      ) ,
    style(id(token("h")) , no_param())
    ) ;
end where ;

where sqrt_expr -> deg
                   expr use
$Boxtree :=
  boxtree
    (
    sons-post(sons-post(sons() , $Boxtree(expr)) , $Boxtree(deg)) ,
    style(id(token("sqrt")) , no_param())
    ) ;
end where ;

where chaine_expr -> string use
$Boxtree :=
  boxtree
    (
    sons-post(sons() , $Boxtree(string)) , style(id(token("h")) , no_param())
    ) ;
end where ;

where chaine_string -> M use
$Boxtree := str(texte(token(string(M)))) ;
```

```
end where ;

end grammar ;
```

Bibliographie

[ed.94] Ian Jacobs ed. *Centaur 2.0 Manual.* INRIA-Sophia Antipolis, 1994.

[FZ88] P. Franchi-Zannettacci. Attribute specifications for graphical interface generation. Technical Report 937, INRIA, Sophia-Antipolis, Dec 1988.

[FZJ91] P. Franchi-Zannettacci and M. Jourdan. Conception des environnements de programmation première partie : Attribute grammars. In *Ecole Jeunes Chercheurs du GRECO de programmation.* ESSI - Sophia Antipolis, april 1991.

[Has92] L. Hascoët. *FIGUE An Incremental Graphic Formatter User's manual for Version 1.* INRIA, Sophia–Antipolis, Sept 1992.

[INR94] INRIA, Sophia–Antipolis. *The Ppml Manual*, Jun 1994.

[JDP] M. Jourdan & D. Parigot. Intenals ans Externals of the FNC-2 Attribute Grammar System.

[JP94] M. Jourdan and D. Parigot. *The FNC-2 User's Guide and References Manual release 1.18.* INRIA, Rocquencourt, Nov 1994. This manual is periodically updated.

[MJP90] C. Le Bellec M. Jourdan and D. Parigot. The olga attribute grammar description language : Designation, implementation and evaluation. In P. Deransart and M. Jourdan, editors, *Attribute Grammars and their applications (WAGA), Paris*, pages 222–237. Lect. Notes in Comp. Sci., Springer Verlag, New York–Heideilberg–Berlin, 1990.

[PBP87] Th. Despeyroux J. Incerpi G. Kahn B. Lang P. Borras, D. Clément and V. Pascual. Centaur : the system. Technical Report 777, INRIA, Sophia–Antipolis - Rocquencourt, 1987.

[Pla89] É. Planes. *PPAT : un décompilateur d'arbres attribués pour le système FNC-2.* Rapport de DEA, Dépt d'informatique, Univ. d'Orléans, Sept 1989.