Le meta-decompilateur Pretty sous Mentor

Anne-Marie Vercoustre

CENTRE DE ROCQUENCOURT

Rapports Techniques

N°62

LE META-DÉCOMPILATEUR PRETTY SOUS MENTOR

Anne-Marie VERCOUSTRE

Décembre 1985

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.:(3) 954 90 20

LE META-DECOMPILATEUR PRETTY
sous Mentor

Anne-Marie Vercoustre
INRIA

Résumé

Mentor est un système permettant de manipuler des documents structurés, écrits en plusieurs langages: programmes, spécifications, rapports techniques. Si Mentor effectue ses manipulations sur une forme interne arborescente des objets, l'utilisateur peut voir en permanance sur l'écran la forme textuelle de ces objets. Le processeur permettant de passer de la forme arborescente à la forme textuelle est appelé décompilateur (ou décompilateur-paragrapheur). Cet article décrit Pretty, un langage permettant de spécifier des décompilateurs, ainsi que l'interpréteur correspondant: le méta décompilateur Pretty est guidé par des tables dépendantes du langage manipulé et générées automatiquement à partir de la spécification en Pretty .

Abstract

Mentor is a general multi-language system for the manipulation of structured documents such as programs, specifications or technical reports. The kernel of Mentor is a syntax directed editor in which objects are implemented by abstract syntax trees, but the user can everytime see a textual version of these objects. The process which computes the text from the abstract syntax, and displays it on the screen is called a decompiler (and pretty-printer). This paper describes Pretty, a language to specify decompilers, and its interpretor: to display the text, the Pretty interpretor is tables driven. The tables for a language are mechanically generated from the Pretty specification of that language.

LE META-DECOMPILATEUR PRETTY
sous Mentor

Anne-Marie Vercoustre
INRIA

1. Introduction

Pretty est un décompilateur d'arbre de syntaxe abstraite, c'est à dire un processeur permettant de produire un texte source, avec une mise en page adéquate, à partir de la représentation interne des programmes.
Pour chaque langage manipulé, cet outil est construit automatiquement à partir d'une spécification, écrite dans un langage appelé également Pretty, de la décompilation et du paragraphage souhaités.

Réalisé initialement pour l'éditeur syntaxique Minerve [Ver 83], le méta-décompilateur Pretty a été intégré à Mentor: grâce au mécanisme d' annotations du système Mentor [Don 84], le langage Pretty peut être vu comme une extension au langage Métal, qui est le langage permettant de spécifier le langage manipulé [Mel 82].

Pour spécifier son décompilateur, l'utilisateur dispose, sous Mentor, d'un environnement spécialisé lui permettant d'éditer de facon simplifiée, les annotations Pretty de son programme Métal. Il s'agit de spécifier, pour chaque opérateur de la syntaxe abstraite, un **schéma de décompilation** , ainsi qu'un modèle de mise en page permettant le **paragraphage** du texte du programme; ce modèle est précisé par un ensemble de directives de paragraphage associées au schéma de décompilation. De plus, l'utilisation de synonymes et de classes de synonymes, dans les schémas, permet de tenir compte du contexte dans lequel est décompilé l'opérateur (contexte hérité uniquement).
L'aspect multi-langages de Mentor permet une mise au point et une vérification interactive du décompilateur produit.

Le méta-décompilateur Pretty fonctionne avec les versions de Mentor multi-langages existant actuellement sous Unix (sur Vax, SM90 et Apollo) et Multics.

Nous présentons tout d'abord les concepts du langage Pretty, permettant la spécification d'un décompilateur Pretty.
La deuxième partie décrit l'environnement Pretty utilisable pour écrire cette spécification.
Puis, nous donnons le mode d'emploi, c'est à dire la suite des étapes à suivre, pour créer un décompilateur pour un langage, que nous appellerons 'Foo' dans la suite de ce papier.

2. Les spécifications Pretty

On suppose connu du lecteur la spécification de la syntaxe abstraite d'un nouveau langage à l'aide de Métal.
Pour construire un décompilateur pour le langage Foo, on utilise le langage Pretty; l'ensemble des spécifications n'est pas un programme Pretty monobloc, mais un ensemble d'unités de compilations , attachées en annotation de certains noeuds de l'arbre Métal décrivant le langage Foo.
L'annexe I donne la description complète de la spécification Pretty pour le langage Pascal. Celle-ci comprend les éléments suivants:

- un ensemble de *schémas de décompilation* attachés à chaque opérateur de la syntaxe abstraite et apparaissant comme des commentaires dans le programme Métal,
- des déclarations de synonymes et de classes, en annotation postfixée de la zone de syntaxe abstraite,
- des directives d'espacement pour certains symboles.

Nous allons voir comment utiliser ces éléments du langage Pretty pour spécifier les différentes fonctionnalités d'un décompilateur.

2.1. Schéma de décompilation

Un schéma de décompilation est une règle permettant d'associer à un opérateur une portion de texte dans le langage 'Foo'. Il existe deux types de schémas :
- les schémas fixes (**schemaf pretty**) associés aux opérateurs d'arité fixe.
- les schémas de liste (**schemal pretty**) associés aux opérateurs de liste.

Pour un opérateur d'arité fixe, dans la partie droite du schéma apparaissent des symboles ou des mots réservés du langage 'Foo', ainsi que des méta-variables correspondant à un appel récursif du décompilateur sur les fils de l'opérateur considéré.
On peut décompiler les fils dans un ordre différent de celui défini dans la syntaxe abstraite; dans ce cas, il faut préciser pour chaque méta-variable la position du fils correspondant; la syntaxe est la suivante:

```
decomp
  prog -> program $defid $1 $lident $2   %RC%
```

$1 précise que l'élément à décompiler récursivement, avec le schéma associé à l'opérateur **defid** , est le premier fils de l'opérateur **prog** ; de même, **$lident** correspond au deuxième fils, ce qui est explicité par $2. Si les positions sont omises, les fils sont décompilés dans l'ordre défini par la syntaxe abstraite. Dans cet exemple, les deux spécifications suivantes seraient équivalentes à la précédente:

```
decomp
  prog -> program $defid  $lident   %RC%
```

ou bien

```
decomp
   prog -> program  $1 $2   %RC%
```

L'indication de la position permet également de décompiler plusieurs fois le même fils, ou de ne pas décompiler certains fils.

Pour un opérateur de liste, le schéma de décompilation précise un délimiteur de début de liste (éventuellement vide), un délimiteur de fin de liste, ainsi que le séparateur des éléments de la liste.

```
abstract syntax
  lstat       -> $STAT1 * ... ;

*  decomp
*     lstat = begin $struct1 end ... ;      %LV,B,I,W%
*
```

La décompilation des éléments de la liste se fait par appel récursif du décompilateur sur chaque élément de la liste.
Il n'y a pas de possibilité de spécifier un traitement spécial pour un élément particulier, par exemple le premier ou le dernier, ni d'inverser deux éléments.

Entre les symboles % , on trouve les directives de paragraphage, que nous allons préciser.

2.2. Directives de paragraphage

Nous décrivons maintenant l'ensemble des directives de paragraphage utilisables. Ces directives constituent des modèles de paragraphage, interprétés au début ou à la fin de la décompilation de la structure considérée; l'effet d'une directive est local à l'opérateur pour lequel cette directive a été spécifiée (en particulier, l'indentation de la marge).
Au moment de la spécification, il n'y a pas de vérification effectuée sur les directives; les directives qui ne s'appliquent pas (par exemple, LV pour un opérateur d'arité fixe) sont ignorées.

2.2.1. Directives générales

Les directives qui s'appliquent à toutes les structures, quel que soit leur type (structure de liste, opérateur d'arité fixe, feuille terminale, synonyme) sont les suivantes:

- **LL** (Ligne) :signifie que le texte décompilé doit commencer sur une nouvelle ligne.

- **RC** (Retour Chariot) : signifie qu'un retour à la ligne sera effectué après la fin du texte associé à cette structure.

- **B** (Bloc) : est équivalente aux deux directives precédentes.

- **LB** (Ligne Blanche) : le texte sera précédé d'une ligne blanche.

- **J** : le texte associé à la structure sera décompilé avec une marge courante augmentée de la valeur d'indentation, dans la limite d' une valeur de marge maximum.

- **P** (Positionne) : la marge courante, pour le texte associé, est mise à la position courante dans la ligne, si cette position ne dépasse pas la taille de la marge maximum; sinon un retour à la ligne, avec indentation de la marge, est effectué.

exemple: (pour Pascal)

On doit spécifier la directive P pour l'opérateur **record** , si l'on veut la présentation suivante:

```
var  noeud : record
                op : integer;
                fils, frere : adress;
                lastnoeud : boolean
             end
```

La marge pour le texte du **record** est la position courante avant d'écrire le mot record; si cette position est trop à droite dans la ligne, on obtiendra :

```
var noeud, curnoeud, racine, newnoeud, locnoeud :
    record
       op : integer;
       fils, frere : adress;
       lastnoeud : boolean
    end
```

- **M** (Marque) : cette directive sert à marquer la position courante pour servir de marge, en cas de coupure d'une ligne plus longue que la taille effective de l'écran. Cette directive sera utilisée, par exemple, pour mémoriser la marge en début de liste horizontale.

2.2.2. Directives spécifiques des listes

Les directives suivantes ne s'appliquent qu'aux opérateurs de listes, ou aux synonymes de listes :

- **LV** (Liste Verticale) : cette directive signifie que la structure doit être décompilée avec ses éléments les uns en dessous des autres.

- **LH** (Liste Horizontale) : les éléments de la liste seront décompilés les uns à coté des autres, sur la même ligne.

- **I** (Indente) : les éléments de la liste verticale sont imprimés avec une indentation de la marge, par rapport aux délimiteurs de début et de fin de liste.

exemple:

```
begin
   aa: = aa + 3;
   insert(aa)
end
```

Les instructions de la liste sont décalées à droite par rapport aux délimiteurs de début (*begin*) et de fin (*end*).

- **A** (Aligne) : permet d'aligner, dans une liste verticale, un symbole apparaissant dans chaque élément de la liste.

exemple :

```
type noeud  = record
                 father  : adress;
                 son     : adress;
                 lastson : boolean;
                 code    : integer
              end;
     adress = ^noeud
```

Pour aligner les symboles ' = ' et ':' respectivement, la directive **A** doit être mise sur les opérateurs **type** et **var**.

Remarques:
- l'alignement se fait jusqu'à une longueur maximale (actuellement 15).
- l'alignement ne se fait que pour un texte gauche correspondant à une feuille ou une liste horizontale de feuilles.

- **H** : avec cette directive, les délimiteurs de début et de fin de liste sont respectivement sur la même ligne que le premier et le dernier élément de la liste. Pour une liste verticale indentée, le premier élément ne reste sur la ligne que si la valeur de la marge n'est pas dépassée; sinon la marge est positionnée à la position courante après le délimiteur de début de liste.

exemples: (avec une valeur d'indentation de 5)

```
type aa = integer;
     bb = boolean
...
case color of
   bleu:(aa:3;
         bb:false);
```

Si cette directive n'est pas présente, on obtient :

```
type
    aa = integer;
    bb = boolean
...
case color of
   bleu:(
        aa:3;
        bb:false
        )
```

- **D** : cette directive permet de décompiler les délimiteurs de début et de fin pour une liste vide.

2.2.3. Directives spécifiques des feuilles

Les directives suivantes permettent de préciser si les identificateurs doivent être imprimés en minuscules ou en majuscules:

- **K** : les feuilles correspondantes sont en majuscules;
- **C** : les feuilles correspondantes sont en minuscules;
- **Y** : les feuilles sont imprimées en minuscules ou majuscules selon la valeur du flag IDLC de Mentor;
- **T** (Telquel) : les feuilles correspondantes sont imprimées telles qu'elles sont en mémoire, c'est à dire telles que l' utilisateur les a entrées, si l'analyseur lexical n'effectue pas de conversion.

2.3. Autres directives

2.3.1. Espacements

Dans le texte décompilé, les mots et les symboles peuvent être, ou non, séparés par un espacement; par défaut, les mots réservés du langage sont précédés et suivis d'un blanc, alors que les symboles et autres terminaux sont collés les uns aux autres.
Pour obtenir un espacement avant ou après un symbole donné, on peut utiliser les commandes **blanc0** et **blanc1** , qui demandent interactivement l'entrée d'un symbole. Ces commandes peuvent être utilisées dans une partie **abs_syn** de Métal, ou sur un schéma de décompilation; elles créent une annotation Pretty (de nom ^espace), attachée à l'opérateur **abs_syn** courant, qui apparait sous la forme suivante :

```
devnoncolle
  : =
  =

dernoncolle
  :
  : =
  =
  ,
```

Ceci signifie que les symboles ': =' et ' =' seront précédés et suivis d'un espacement, les symboles ':' et ',' seront suivis d'un espacement, mais collés au texte qui les précède.

On notera toutefois que l'effet de la commande est global, c'est à dire que ce symbole sera alors précédé ou suivi d'un blanc dans l'ensemble du programme, quel que soit le contexte.
Pour obtenir un espacement dans un contexte particulier, il faut utiliser la fonction prédéfinie F(11), dans le schéma de décompilation correspondant à ce contexte(cf 2.7); cette fonction pourra également être utilisée pour séparer par un blanc des terminaux qui ne sont pas séparés par un symbole (par exemple, dans une liste d'identificateurs ou de nombres sans séparateur de liste).

2.3.2. Holophraste

On appelle holophraste, la possibilité d'obtenir un affichage partiel des arbres. Le mécanisme standard dans Mentor est basé sur la profondeur d'emboitement des sous arbres par rapport à la position courante: les sous arbres dont la profondeur est supérieure à un paramètre, appelé *niveau de détails* , sont remplacés, à l'affichage, par un symbole d'abréviation (le symbole # pour un opérateur d'arité fixe, et ... pour un opérateur de liste).

Il apparait qu'il n'est pas satisfaisant d'appliquer ce mécanisme de facon uniforme sur les sous-arbres de même niveau d'emboitement.

exemple:

```
AA,BB,CC : integer;
```

Si l'on ne prend pas de précaution, ce texte peut être holophrasté en :

```
... : # ;
```

ou

```
#,#,# : # ;
```

selon les niveaux de visibilité atteints.

La directive **W** (Weigth) permet de spécifier explicitement l'effet d' **holophraste** sur le texte; si cette directive n'est pas précisée pour un opérateur, celui-ci sera considéré comme de même *niveau de détails* que ses fils; il faudra donc mettre cette directive sur les structures dont tous les fils peuvent effectivement être holophrastés; en général, on ne mettra pas cette directive sur les listes dont les éléments sont des feuilles, comme par exemple les listes d' identificateurs.

autre exemple:
pour obtenir l'effet suivant (en Pascal):

```
var
  AAA : ... ;
```

BBB : # ;

On a utilisé une spécification utilisant un synonyme, qui est donnée en 2.5.3, dans l'exemple 2 .
Une autre facon d'obtenir cette présentation, est d'utiliser, dans le schéma de l'opérateur dcl , un appel de procedure spéciale (cf 2.7) qui mettra explicitement du *poids* sur la partie droite de la déclaration.

2.4. Priorité des opérateurs

2.4.1. Présentation

Lors de la décompilation d'un arbre de syntaxe abstraite, il s'agit de restituer les parenthèses dans les expressions, en tenant compte de la priorité des opérateurs.
exemple:

(AA + (3 in BB)) * CC

ne correspond pas au même arbre que le texte:

AA + 3 in BB * CC

Le mécanisme de priorité introduit dans Pretty, est adapté au cas où les opérateurs binaires correspondent à des noeuds binaires, (comme dans la syntaxe abstraite de Pascal sous Mentor), et beaucoup moins au cas où les opérateurs sont des noeuds ternaires.

2.4.2. Noeuds binaires

Dans ce cas, il suffit de donner la priorité de l'opérateur, dans la liste des directives, sous la forme **R(n)** où **n** est un entier positif: un opérateur sera décompilé entre parenthèses :
- s'il n'est pas le noeud racine de l'arbre décompilé
- si sa priorité est inférieure ou égale à celle de l'opérateur de son père.

Dans l'exemple précédent, il faut spécifier les priorités de la facon suivante:
in sans directive (équivalent à **R(0)**)
plus avec la directive **R(1)**
mult avec la directive **R(2)**

2.4.3. Noeuds ternaires

Dans la représentation ternaire des opérateurs, le code opération est le deuxième fils, ce qui permet d'effectuer, dans la syntaxe abstraite, des regroupements d'opérateurs. exemple:(Asple)

```
abstract syntax
  binary     -> EXP OP EXP;
  plus       -> implemented as SINGLETON;
  times      -> implemented as SINGLETON;
  equal      -> implemented as SINGLETON;
  different  -> implemented as SINGLETON;

  OP         :: = plus times equal different;
```

Il faudrait écrire:

```
binary__op(plus,times,plus) -> ( $exp) * ( $exp) %%
binary__op(plus,times,exp)  -> ( $exp) *   $exp  %%
binary__op(exp,times,plus)  ->   $exp  * ( $exp) %%
```

et de même pour les autres opérateurs, ce que l'on ne peut faire en Pretty.

Remarque:
Dans la syntaxe abstraite choisie pour Ada, la situation est encore pire, car les codes opérations sont regroupés en un seul opérateur terminal, et ne sont différenciables que par la *valeur* de ce terminal.

La spécification en utilisant les priorités peut encore être utilisée pour des noeuds d'arité trois, si l'on regroupe, dans la syntaxe abstraite, les opérateurs ayant même priorité. Sinon, il faut écrire un programme spécifique et utiliser le mécanisme d'appel de procedure décrit en 2.7.

2.5. Synonymes

2.5.1. Présentation

Pour décompiler un opérateur en tenant compte du contexte dans lequel il se trouve, nous avons introduit un mécanisme de synonymes, qui permet de conserver tout *le contexte hérité.*

A un synonyme peut être associé un schéma de décompilation, qui sera utilisé à la place de celui de l'opérateur courant, lorsqu'on se trouve dans un contexte de décompilation où apparait ce synonyme.

exemple: (Pascal)

```
abstract syntax
  repeat      -> LSTAT EXP;

*decomp
*   repeat    -> repeat $lstat_syn until $exp      %B,W%
*   lstat_syn = $stat ... ;    %LV,B,J,W%
*   lstat     = begin $stat end ... ;    %LV,B,J,W%
*
*synonymes
*   lstat_syn :: lstat         %%
*
```

Dans le contexte d'une instruction *repeat* , l'opérateur **lstat** doit être décompilé sans les mots-clefs *begin* et *end*

Ceci se fait, en introduisant le synonyme **lstat_syn** dans le schéma de l'opérateur **repeat** ; dans ce contexte, le schéma de **lstat_syn** sera utilisé à la place de celui de **lstat** , qui, lui, contient les mots *begin* et *end.*

2.5.2. Déclaration de synonymes

Les synonymes doivent être déclarés avant leur utilisation dans un schéma de décompilation, soit dans la partie de syntaxe abstraite courante, soit dans une précédente, au sens du parcours en préordre de l'arbre étal On peut utiliser la commande dclsyn qui demande :

- le nom du synonyme,
- la structure pour laquelle on crée ce synonyme;

On peut déclarer des synonymes d'opérateur, de phylum ou de classe. (Les classes sont définies en 2.6).

2.5.3. Utilisation

Nous avons vu, dans l'exemple ci-dessus, l'utilisation d'un synonyme pour un opérateur de liste; ceci n'est possible que si l'opérateur apparait seul dans le phylum; dans le cas général, il faut utiliser une classe, ce que nous verrons en 2.6.

Dans le cas d'un synonyme de phylum, le schéma du synonyme n'est pas utilisé *à la place* du schéma de l'opérateur courant (qui appartient au phylum considéré), mais *avant* la décompilation de cet opérateur.
exemple 1:

```
abstract syntax
  initvar_dcl -> VARBL TYPE INITVAR;

  INITVAR     :: = EXP void;

*decomp
*   initvar_dcl -> $varbl : $type $initvar_syn     %%
*   initvar_syn -> : = $exp     %%
*
*synonymes
*   initvar_syn :: initvar     %%
*
```

L'utilisation du synonyme permet ici de mettre en facteur le symbole : = pour tous les opérateurs apparaissant dans l'expression d'initialisation, et de ne pas écrire ce symbole lorsque la partie initialisation est vide.

On pourra ainsi décompiler une déclaration de variable avec ou sans initialisation :

```
A : integer := 3;
B : boolean;
```

exemple 2:

```
abstract syntax
  var__dcl    -> ID__L TYPE;

*decomp
*   var__dcl -> $id__l : $type__syn          %%
*
*synonymes
*   type__syn :: type %W%
*
```

Le synonyme **type__syn** du phylum **type** n'a pas de schéma de décompilation associé, mais seulement la directive **W** , permettant l'holophraste suivant:

exemple:

```
AA,BB : #;
CC    : ...;
```

Le schéma d'un synonyme d'opérateur de liste doit être un schéma de liste (opérateur **schemaf pretty**).

Le schéma d'un synonyme d'opérateur d'arité fixe, de phylum ou de classe, doit être de type schéma fixe (opérateur **schemaf pretty**).

2.6. Classes

2.6.1. Présentation

Dans le cas où un opérateur doit être décompilé en tenant compte du contexte du phylum auquel il appartient, il faut généraliser le mécanisme de synonymes et utiliser une *classe* : une *classe* est un ensemble d'opérateurs ou de synonymes d'opérateurs appartenant à un phylum; plus précisement, chaque classe correspond à un phylum *p* , au sens suivant:

pour tout élément *c* appartenant à la classe, il existe un opérateur *o* appartenant au phylum *p* , tel que: *c = o* ou *c synonyme de o*

2.6.2. Déclaration

Les classes doivent être déclarées avant leur utilisation; on peut utiliser la commande **dclclasse** qui demande le nom de la classe et l'ensemble de ses éléments; en fait, il n'est pas nécessaire de donner tous les éléments de la classe, mais seulement ceux qui sont des synonymes (cela optimise d'ailleurs la décompilation).
On notera que l'on peut avoir également des synonymes de classes, et donc qu'une partie de déclaration Pretty peut comporter plusieurs zones de déclarations de synonymes ou de déclaration de classes, puisqu'il ne peut y avoir de référence en avant.

2.6.3. Utilisation

Reprenons l'exemple de synonyme de phylum donné en 2.5.3

```
abstract syntax
  initvar_dcl -> VARBL TYPE INITVAR;

  INITVAR     :: = EXP void;

*synonymes
*   initvar_syn :: initvar     %%
*
*decomp
*   initvar_dcl -> $varbl : $type $initvar_syn     %%
*   initvar_syn -> : = $exp     %%
*
```

L'utilisation d'un synonyme de phylum, comme ci-dessus, convient lorsque effectivement aucun noeud n'est créé pour une expression d'initialisation vide; si un noeud non vide, mais se décompilant en un texte vide, est créé, il faut utiliser une classe, comme suit:

```
abstract syntax
  initvar_dcl -> VARBL TYPE INITVAR;

  INITVAR     :: = EXP void;

*synonymes
*   initvar_syn :: exp         %%
*classes
*   initvar_class :: = initvar_syn void %%
```

```
*
*decomp
*   initvar_dcl -> $varbl : $type $initvar_class   %%
*   initvar_syn -> := $exp     %%
*
```

Autre exemple:

On peut utiliser les classes pour restituer les parenthèses dans la décompilation des opérateurs, si l'on ne veut pas utiliser le mécanisme de priorité (par exemple, si les priorités à gauche et à droite sont différentes).
On écrira alors:

```
abstract syntax
  plus       -> EXP EXP;
  mult       -> EXP EXP;
  dans       -> EXP EXP;

  EXP        :: = PLUS MULT DANS IDENT CST;

*synonymes
*   plus_syn :: plus %%
*   dans_syn :: dans %%
*classes
*   exp1 :: = dans_syn plus mult ident cst %%
*   exp2 :: = plus_syn dans_syn mult ident cst %%
*
*decomp
*   plus    -> $exp1 + $exp1  %%
*   mult    -> $exp2 * $exp2  %%
*   dans    -> $exp in $exp   %%
*   plus_syn -> ( $exp1 + $exp1 )      %%
*   dans_syn -> ( $exp in $exp )       %%
*
```

Dans le contexte de l'opérateur **mult** , les opérateurs **plus** et **dans** seront décompilés avec des parenthèses, car on utilisera les schémas trouvés dans la classe **exp2** ,soit plus_syn et dans_syn; de même, dans le contexte de **plus** , on utilisera les schémas trouvés par la classe **exp1** , et donc seul **dans** sera décompilé avec des parenthèses, en utilisant le schéma de dans_syn.

Remarque:
Les classes **exp1** et **exp2** seront décrites de facon incomplète et optimale par:

```
classes
   exp1 :: dans_syn          %%
   exp2 :: dans_syn plus_syn %%
```

2.7. Appel de procédures

La spécification d'appels de procédure dans les schémas de décompilation correspond à un mécanisme d'échappatoire, pour effectuer des traitements spécifiques (programmés à la main) ou des actions de paragraphage directement exécutables.

2.7.1. Procedures prédéfinies

Les procédures prédéfinies correspondent à des actions de paragraphage (qui gagneraient à avoir une syntaxe plus explicite !); ce sont:
F(1) : positionne sur une nouvelle ligne, avec la marge courante;
F(2) : inhibe le retour à la ligne suivant;
F(3) : augmente le *poids* (pour l'holophraste) de 1;
F(4) : diminue le *poids* de 1;
F(5) : positionne la marge à la position courante;
F(6) : effectue une tabulation;
F(7) : augmente la valeur de la marge de la valeur d'indentation;
F(11) : met un espacement entre le symbole (ou mot) précédent et le symbole (ou mot) suivant;
F(12) : supprime l'espacement entre le symbole précédent et le symbole suivant;
F(18) : effectue un positionnement à la marge suivante, si cette postion n'est pas dépassée; sinon un retour à la ligne est effectué, avec indentation de la marge;
F(19) : met à "vraie" une assertion permettant de doubler les quotes dans la chaine qui suit;

exemple 1:

```
decomp
   meta -> F(11) $meta F(11) %%
```

L'appel de procédure prédéfinie F(11) permet de séparer les méta variables, qui, en l'absence de séparation par des symboles du langage, apparaitraient collées les unes aux autres.

exemple 2:

```
decomp
   for2 -> F(18) for $ident1 $step1 do $stat2 %W%
```

Si une instruction **for** est décompilée dans le contexte d'un **if** , la présentation, avec une valeur d'indentation de 6, sera la suivante:

```
if AAAAA then bb: = 3
else  for i: = 1 to 10 do
            begin
            CC[i]: = 0
            end
```

Si la valeur d'indentation est inférieure à 5, la présentation sera :

```
if AAAAA then bb: = 3
else
    for i: = 1 to 10 do
        begin
        CC[i]: = 0
        end
```

2.7.2. Procédures dépendantes du langage

Il est possible d'utiliser les appels de procédures pour effectuer des traitements spécifiques non réalisables avec les mécanismes existants: par exemple le traitement du "*dangling else*" Pascal ou le traitement des priorités en Asple ou Ada.

Malheureusement, ce mécanisme prévu pour Minerve est mal adapté à un environnement multi-langages comme celui de Mentor; pour être réellement utilisable, les appels à ces programmes devraient passer par l'interface SYSUSER utilisée pour MENTORKIT [Mel 85], ce qui n'est pas le cas actuellement.

3. Environnement PRETTY

Pour utiliser l'environnnement Pretty, il faut charger le fichier Mentol **PRETTYUSER** ; une documentation sur cet environnement peut être consultée intéractivement sous Mentor.

3.1. Création de squelettes de schémas

Ces procédures servent à initialiser ou à déplacer les schémas de décompilation.

- **prompt**
  Crée les squelettes de schémas Pretty, en annotation du programme ; peut être appelé en haut du programme , sur un chapitre, une partie de syntaxe abstraite ou un opérateur; un schéma est créé pour chaque opérateur du sous arbre, *en détruisant les anciens schémas, s'il en existait* ; à n'utiliser, donc, que pour initialiser les schémas, ou très localement !

- **grouper**
  Cette commande regroupe tous les schémas de décompilation d'une partie de syntaxe abstraite à la fin de celle-ci; on peut appeler la commande **grouper** , à n'importe quel niveau dans l'arbre; les schémas de décompilation contenus dans le sous-arbre sont alors regroupés à la fin de la partie de syntaxe abstraite.

  exemple: (pour Asple)
  si on a la spécification suivante:

```
abstract syntax
  declaration -> MODE IDLIST;

*  decomp
*     declaration -> $mode $idlist ;        %%
*
  idlist      -> ID + ... ;

*  decomp
*     idlist = $id ... ,          %D,LH,W,M%
*
  bool        ->;

*  decomp
*     bool -> bool      %%
*
  int         ->;

*  decomp
*     int -> int        %%
*
  ref         -> MODE;

*  decomp
*     ref -> ref $mode %%
*

  DECLARATION :: = declaration;
  MODE        :: = bool int ref;
  IDLIST      :: = idlist;
```

La commande **grouper** donnera le résultat suivant:

```
abstract syntax
   declaration -> MODE IDLIST;
   idlist      -> ID + ... ;
   bool        ->;
   int         ->;
   ref         -> MODE;

   DECLARATION :: = declaration;
   MODE        :: = bool int ref;
   IDLIST      :: = idlist;

*decomp
*   declaration -> $mode $idlist ;        %%
*   idlist      = $id ... ,     %D,LH,W,M%
*   bool        -> bool         %%
*   int         -> int          %%
*   ref         -> ref $mode    %%
*
```

La forme ci-dessus parait plus agréable à lire que la précédente; mais la première forme est intéressante lorsque l'on a plusieurs schémas de décompilation associés à un opérateur, en particulier lorsque l'on introduit des schémas synonymes pour décompiler les sous-structures.

3.2. Recherche d'un schéma

Les commandes suivantes permettent de rechercher un schéma à travers les annotations Pretty de l'arbre .

- **nschema**
  Va sur le schéma suivant, en parcourant l'arbre d'annotation en annotation; la commande échoue si l'on est sur le dernier schéma; cette commande est très pratique pour compléter les schémas les uns après les autres, car elle dispense l'utilisateur de la connaissance du nom des annotations et des commandes pour les manipuler.
- **fschema**
  Demande le nom d'une structure (opérateur, liste, synonyme, classe) et va sur le schéma qui lui correspond, **dans le chapitre courant** ; cette procédure échoue si le schéma n'existe pas.

- **ffschema**
  Identique à la procédure **fschema** , mais cherche le schéma désigné en parcourant tout l'arbre .

3.3. Ecriture des schémas complets

3.3.1. Occurrences d'un opérateur

La procedure **opoccur** demande la nom d'un opérateur du langage, et donne la liste des règles (**rule~metal**) de la syntaxe concrète, pour lesquelles cet opérateur apparait dans la description sémantique.
La liste de ces règles est repérable par le pointeur @listopocc.

Cette procédure apporte une aide à l'utilisateur dans l'écriture des schémas de décompilation, car elle permet de visualiser le ou les textes associés à un opérateur donné; ensuite, il est possible de retrouver le ou les contextes de cet opérateur, en utilisant la procédure **occur** de l'environnement ;

3.3.2. Schéma pour un opérateur d'arité fixe

La procédure **insert<n>** permet de compléter un schéma de décompilation, en insérant un élément en nième position (n> = 1); si **n** est omis, l'élément sera inséré **avant** le premier élément; insert<*> permet d'ajouter un élément en dernière position. L'élément à insérer est:

- soit un mot réservé du langage, sous forme d'un identificateur;
  exemples: program if while end

- soit un symbole réservé;
  exemples: : = = < ; (}

- soit le nom d'une structure (phylum,synonyme ou classe), sous forme d'une méta-variable;
  exemples: $exp $stat $stat1 $ifprim $stat_1

- soit un appel de procédure (cf 2.7), avec la syntaxe F(n), où **n** est un entier positif;

3.3.3. Schéma de liste

Les commandes suivantes permettent de compléter ou modifier un schéma de décompilation pour un opérateur de liste; un tel schéma est dit schéma de liste (opérateur **schemaPretty**).

- La procédure **debut** permet de préciser le symbole ou mot réservé servant de délimiteur de début de liste;

  exemple: begin ({

  Si la liste comporte deux (ou plus de deux) mots ou symboles réservés comme délimiteurs de début, il faut les entrer sous forme d'une chaine entre quotes; exemple: 'liste processus'

  Cette procédure échoue si l'on n'est pas sur un schéma de liste.

- La procédure **fin** permet de préciser le symbole ou mot réservé servant de délimiteur de fin de liste;

  exemple: end) } 'end loop'

- La procédure **sep** permet de préciser le symbole ou mot réservé servant de séparateur pour les éléments de liste;

  exemple: or , ;

3.3.4. Directives de paragraphage

La procédure **atts** permet de donner les directives de paragraphage pour la structure concernée (cf 2.2).
La procédure échoue si l'on n'est pas sur un schéma de décompilation ou une déclaration de synonyme ou de classe.

3.4. Déclaration de synonymes et les classes

Les synonymes et les classes sont définis en 2.5 et 2.6

- **dclsyn**
  Cette procédure demande le nom du synonyme que l'on veut déclarer, et de quelle structure il est synonyme; la variable Mentol @dclpart1 référence alors la zone de déclarations contenant la déclaration de ce synonyme : c'est une annotation postfixée, nommée ^dclp, attachée à la partie **abs_syn** englobante.

- **dclclasse**
  Cette procédure demande le nom de la classe et la liste des structures (noms d'opérateur ou de synonyme) qui la compose; de même que pour **dclsyn** , la variable @dclpart1 permet d'accéder à la partie de déclaration correspondante.

3.5. Espacement entre symboles

Pour spécifier les espacements entre symboles, on utilise les commandes:

- **blanc0**
  Cette commande demande une liste de symboles que l'on veut précéder d'un espacement.

- **blanc1**
  Cette commande demande une liste de symboles que l'on veut faire suivre d'un espacement.

4. Construction d'un décompilateur

Nous donnons, dans ce chapitre, le mode d'emploi pour créer un décompilateur pour un langage 'Foo', en supposant que le lecteur a déja une certaine connaissance du système Mentor.
Ce chapitre peut être lu indépendemment des autres, de sorte qu'il comprend certaines redites volontaires, en particulier dans les exemples et les commandes de l'environnement Pretty.

4.1. Introduction du langage sous MENTOR

Avant tout, votre langage doit être connu du système Mentor, c'est à dire que vous devez avoir compilé le programme décrivant votre langage; pour cela, se reporter à la documentation [Mel 85]. Les tables décrivant le langage Foo (FOO.t et FOOCODE.t) sont donc créées, mais il n'est pas nécessaire, à cette étape, d'avoir créé les tables d'analyse.

4.2. Programme d'appel du décompilateur

Sous Unix, l'appel au décompilateur, pour le langage Foo, doit être fait explicitement dans le programme SYSDECOMP.po (voir Mentorkit.info), par appel de la procedure **decpretty** , et ceci *quelque soit le nom du* langage.

Sous Multics, l'appel au décompilateur se fait dynamiquement par appel à un programme de nom DECFOO1; ce programme est standard pour tous les langages, et se trouve dans le directory Mentor>info .
Il suffit de recopier DECFOO1.pascal, de substituer partout 'Foo' par le nom de votre langage et de le compiler (ne pas oublier de modifier le nom du fichier).

Si cette étape est réalisée, vous pourrez commencer à tester votre décompilateur, même avant de l'avoir entièrement spécifié.

4.3. Squelettes de schémas de décompilation

Il faut maintenant se mettre sous Mentor (avec les environnements user ou Prettyuser) et charger FOO.po.

A partir des spécifications de votre langage, le système va construire un squelette très précisement, à chaque opérateur de la syntaxe abstraite est attaché un schéma, qui est une annotation postfixée de nom ^schemas.

Ces schémas de décompilation constituent le début des spécifications de décompilation et paragraphage dans le langage PRETTY.

Pour créer l'ossature des schémas de décompilation, il faut utiliser la commande **prompt** de l'environnement Pretty.

exemple : (en Asple)

Si la spécification d'un programme ASPLE, en , est la suivante:

```
chapter 'programs in ASPLE'
  rules
    <Program>   ::= BEGIN <Dcl_Train> <Stm_Train> #END ;
      program(<Dcl_Train>,<Stm_Train>)
  abstract syntax
    program     -> DECLS STMS;
end chapter;
```

La procédure **prompt** crée le squelette de schéma de décompilation pour l'opérateur **program** , comme suit :

```
chapter 'Programs in ASPLE'
  rules
    <Program>   ::= BEGIN <Dcl_Train> <Stm_Train> #END ;
      program(<Dcl_Train>,<Stm_Train>)
  abstract syntax
    program     -> DECLS STMS;

*    decomp
*      program -> $decls $stms    %%
*
    end chapter;
```

Le schéma créé en annotation apparait comme un commentaire dans le programme (lignes précédées de *). Le schéma créé permet seulement l'appel récursif du décompilateur sur les deux fils de l'opérateur **program** , sans indications de paragraphage,

puisque la liste de directives de paragraphage, qui peut apparaître entre les deux symboles '%', est vide.

4.4. Ecriture des schémas

Les squelettes de schéma de décompilation doivent ensuite être complétés, c'est à dire qu'il faut leur ajouter les mots réservés du langage, ainsi que les directives de paragraphage. On se déplace facilement de schéma en schéma en utilisant la procédure **nschema** de l'environnement Pretty.

Pour éditer les schémas, il n'est pas nécessaire de connaitre la syntaxe du langage Pretty; on peut :

- soit utiliser l'environnement Pretty défini dans la section 3
- soit utiliser le mode guidé par menus [Mig 85]

exemple:

Le schéma complet d'un **program** en Asple pourra être le suivant:

```
decomp
  program -> begin $decls $stms end     %B,W%
```

Dans ce schéma, les mots résérves *begin* et *end* ont été ajoutés au schéma initial. Les directives de paragraphage, entre les symboles '%', ont l'effet suivant:

- B (**bloc**): signifie qu'un programme Asple (correspondant à l'opérateur **program**) commence et termine sur une nouvelle ligne.
- W (**weight**): permet l'holophraste des sous-structures (**decls, stms**) de l'opérateur **program**

Autre exemple:

Pour une structure de liste comme **stms** (liste d'instructions), le schéma complet sera le suivant:

```
abstract syntax
  stms        -> STATEMENT * ... ;

*  decomp
*     stms = $statement ... ;     %LV,I,B,W%
*
```

Le symbole ... indique, dans le langage Pretty, qu'il s'agit d'une liste.
Le *point virgule* qui apparait après ce symbole est le séparateur des éléments de la liste.

Les nouvelles directives qui apparaissent dans cet exemple ne s'appliquent qu'aux listes; ce sont:

- LV (**liste verticale**), dont la signification est évidente
- I (**indente**): signifie que les éléments de la liste apparaissent décalés à droite, par rapport à la marge courante.

Une fois les schémas de décompilation complétés, et un minimum de directives de paragraphage spécifiées (quelques directives **bloc** et surtout les directives **LV** ou **LH**), on peut commencer à tester son décompilateur.
Pour cela, on doit générer les tables du décompilateur.

4.5. Création des tables de décompilation

Pour créer les tables de décompilation pour la première fois, il faut se positionner tout en haut de l'arbre. La création des tables (compilation des annotations Pretty) se fait par la commande **cpretty.**

- S'il n'y a pas d'erreur, on peut essayer tout de suite de décompiler un document dans le langage Foo: par exemple, passez dans le langage Foo, et appelez le mode menu.

- S'il y a des erreurs, il faudra recommencer l'exécution de cpretty, après avoir effectué les corrections nécessaires. Les messages d'erreurs sont décrits en Annexe 2.

S'il n'y a pas de message d'erreur, mais que le paragraphage, ou la décompilation du document Foo ne vous satisfait pas, il faut modifier les schémas ou les directives de paragraphage. Il est alors possible de recompiler (commande **cpretty**) uniquement le ou les schémas modifiés; pour cela, il faut être, dans l'arbre , sur un **operator** , une partie **abs_syn** ou un **chapt** ; tous les schémas contenus dans le sous arbre sont recompilés.
Toutefois, dans l'implémentation actuelle, les tables sur fichier ne seront mises à jour qu'en lancant la commande **cpretty tout en haut de l'arbre.**

Attention
Lorsque la mise au point de votre décompilateur est terminée, ne pas oublier de lancer une compilation Pretty générale, (ni de sauvegarder le programme Métal FOO.po) avant de sortir de Mentor (on peut utiliser la commande .sauve).

Attention encore
Si votre syntaxe abstraite change ultérieurement, il faudra refaire une compilation Pretty après la compilation ; s'assurer également que les schémas restent cohérents avec l'arité des opérateurs .

5. Restrictions et mise en garde

5.1. Spécifications

Tout d'abord, pour reprendre la terminologie de [Mel 83b], on peut indiquer que Pretty est plus adapté pour décompiler des langages à structures syntaxiques fortes (langages de programmation ou de spécification) que des langages à structures syntaxiques faibles (comme le texte d'un paragraphe en langage naturel).

Même dans le cas de langages à structures syntaxiques fortes, nous avons vu, à propos du traitement des priorités des opérateurs que les mécanismes existants ne permettent pas toujours de spécifier un décompilateur pour n'importe quel langage, *et n'importe quel choix de syntaxe abstraite.*
Actuellement, un schéma écrit en Pretty ne peut tenir compte que du contexte hérité, car un opérateur ne peut avoir plusieurs schémas dépendant des opérateurs de ses fils.
En pratique, un mécanisme de "pattern-matching" serait vraiment utile pour décompiler un opérateur de facon différente selon que certaines de ses sous-structures sont vides ou non (les synonymes ne permettent de traiter que des cas particuliers).

exemple:(Spectre)
On voudrait pouvoir spécifier la décompilation suivante:

```
decomp
    type_abstr -> abstrait  $obbase_1  $void     %%
    type_abstr -> abstrait ( $opbase_1 $exp_1 ) %%
```

5.2. Décompilateur multi-langages

Le décompilateur Pretty utilise son propre display et communique avec celui de Mentor pour l'écriture d'un caractère et les retours à la ligne; ceci pose certains problèmes de synchronisation des displays, en particulier pour les marges, lorsqu'un décompilateur Pretty est appelé par un décompilateur programmé (et réciproquement), pour décompiler des annotations dans un langage différent.

Naturellement les décompilateurs Pretty savent se synchroniser entre eux aussi bien que les décompilateurs programmés.

6. Conclusion

Le méta-décompilateur Pretty a déja été utilisé pour construire des décompilateurs pour les langages :

- Pascal, Legos, Spectre, LTR2 sous Minerve,
- Pascal, , Spectre, Asple, Pretty sous Mentor.

Sous l'environnement Mentor-Pretty la mise au point interactive du décompilateur est assez aisée (même si le temps passé augmente vite avec la taille du langage), car les directives de paragraphage correspondent à des modèles de mise en page facilement visualisables pour l'utilisateur; avec cette approche, il serait d'ailleurs facile de permettre à l'utilisateur de modifier sa mise en page interactivement, en utilisant des menus ou des îcones.

Un effort reste à faire dans l'utilisation d'un meilleur display, pour permettre une mise en page optionnelle, selon la taille du texte à écrire; par exemple, il faudrait, comme dans Ceyx [Ber 85], définir des listes mixtes, c'est à dire se décompilant en liste horizontale si tous les éléments peuvent tenir sur la même ligne, en liste verticale sinon.

7. Annexe 1 : Spécifications Pretty pour Pascal

```
abstract syntax
    ident   -> implemented as IDENTIFIER;
•    decomp
•        ident -> $ident %Y%
•
    meta    -> implemented as IDENTIFIER;
•    decomp
•        meta -> $ $meta f(11) %Y%
•
    intcst  -> implemented as INTEGER;
•    decomp
•        intcst -> $intcst %Y%
•
    alfacst -> implemented as STRING;
•    decomp
•        alfacst -> f(19) #'' $alfacst #'' %T%
•
    charcst -> implemented as CHAR;
•    decomp
•        charcst -> #'' $charcst #''  %T%
•
    nil     -> implemented as SINGLETON;
•    decomp
•        nil -> nil     %%
•
    hexcst  -> implemented as STRING;
•    decomp
•        hexcst -> $hexcst %%
•
    realcst -> implemented as STRING;
•    decomp
•        realcst -> $realcst  %%
•
    metasym -> implemented as STRING;
•    decomp
•        metasym -> external  %%
•
    pascal  -> implemented as SINGLETON;
•    decomp
•        pascal -> external %%
•
    forward -> implemented as SINGLETON;
•    decomp
•        forward -> forward  %Y%
•
    line    -> implemented as STRING;
•    decomp
•        line -> $line  %T%
•
    setof   -> LELEM;
•    decomp
•        setof -> [ $lelem ]      %%
•
    not     -> EXP;
•    decomp
•        not -> not $exp          % r(7)%
•
    uplus   -> EXP;
•    decomp
•        uplus -> + $exp          % r(4)%
•
    uminus  -> EXP;
•    decomp
•        uminus -> - $exp2        % r(4)%
•        minus2 -> ( $exp - $exp3 )       % r(5)%
•        plus2  -> ( $exp + $exp )        % r(5)%
•
    unref   -> VARBL;
•    decomp
•        unref -> $varbl ^        %%
•
    hexa    -> EXP;
•    decomp
•        hexa -> $exp hexa        %%
•
    def     -> IDENT;
•    decomp
•        def -> def $ident        %%
•
    funcpar -> LDEFID;
•    decomp
•        funcpar -> function $ldefid      %%
•
    varpar  -> LDEFID;
•    decomp
•        varpar -> var $ldefid    %%
•
    ref     -> IDENT;
•    decomp
•        ref -> ^ $ident          %%
•
    packed  -> TYP;
•    decomp
•        packed -> packed $typ    %%
•
    file    -> TYP;
•    decomp
•        file -> file of $typ     %W%
•
    set     -> SPLTYP;
•    decomp
•        set -> set of $spltyp    %W%
•
    procpar -> LIDENT;
•    decomp
•        procpar -> procedure $lident     %W%
•
    goto    -> INTCST;
•    decomp
•        goto  -> goto $intcst    %M%
•        goto2 -> goto $intcst    %M.J%
    eqlc    -> IDENT CST;
•    decomp
•        eqlc -> $ident = $cst    %%
•
    eqlt    -> IDENT TYP;
•    decomp
•        eqlt -> $ident = f(12) $typ1     %%
•
    eqlv    -> IDENT VALU;
•    decomp
```

```
        eqlv -> $ident : $valu1  %%

interv -> EXP EXP;
 decomp
        interv -> $exp .. $exp   %W%

eql    -> EXP EXP;
 decomp
        eql -> $exp = $exp       % r(2)%

lss    -> EXP EXP;
 decomp
        lss -> $exp < $exp       % r(3)%

gtr    -> EXP EXP;
 decomp
        gtr -> $exp > $exp       % r(3)%

neq    -> EXP EXP;
 decomp
        neq -> $exp <> $exp      % r(2)%

leq    -> EXP EXP;
 decomp
        leq -> $exp <= $exp      % r(3)%

geq    -> EXP EXP;
 decomp
        geq -> $exp >= $exp      % r(3)%

#in    -> EXP EXP;
 decomp
        in -> $exp in $exp       % r(2)%

intdiv -> EXP EXP;
 decomp
        intdiv -> $exp div $exp  % r(6)%

mod    -> EXP EXP;
 decomp
        mod -> $exp mod $exp     % r(6)%

div    -> EXP EXP;
 decomp
        div -> $exp / $exp       % r(6)%

mult   -> EXP EXP;
 decomp
        mult -> $exp * $exp      % r(6)%

plus   -> EXP EXP;
 decomp
        plus -> $exp + $exp      % r(5)%

minus  -> EXP EXP;
 decomp
        minus -> $exp - $exp3    % r(5)%

or     -> EXP EXP;
 decomp
        or -> $exp or $exp       % r(3)%

and    -> EXP EXP;
 decomp
        and -> $exp and $exp     % r(5)%

index  -> VARBL LEXP;
 decomp
        index -> $varbl [ $lexp1 ]       %%

dot    -> VARBL IDENT;
 decomp
        dot -> $varbl #. $ident  %%

format -> EXP EXP;
 decomp
        format -> $exp : $exp    %%

range  -> CST CST;
 decomp
        range -> $cst :. $cst    %%

caserc -> CASETG LCOLRC;
 decomp
        caserc -> case $casetg of $lcolrc end        %B,W%

array  -> LTYP TYP;
 decomp
        array -> array f(12) $ltyp of $typ %W%

dectag -> IDENT IDENT;
 decomp
        dectag -> $ident : $ident        %%

colrc  -> LCST LFIELD;
 decomp
        colrc  -> $lcst : f(18) $lfield1  %%
        lfield1 = ( $field ) ... ;       %W.H.LV%

decl   -> LOCAL TYP;
 decomp
        decl -> $local : f(12) $typ1     %%

upstep -> EXP EXP;
 decomp
        upstep -> $exp to $exp   %%

dwnstep -> EXP EXP;
 decomp
        dwnstep -> $exp downto $exp      %%

repeat -> LSTAT EXP;
 decomp
        repeat  -> repeat $repeatls until $exp      %W%
        repeatls = $stat ... ;  %LV,J.W.B%
         repeat2  ->  f(18)  repeat  $repeatls  until  $exp
%W%

ass    -> VARBL EXP;
 decomp
        ass  -> $varbl := $exp1  %M%
        ass2 -> $varbl := $exp1  %M.J%

call   -> IDENT LEXP;
 decomp
        call  -> $ident $lexp    %M%
        call2 -> $ident $lexp    %M,J%
```

```
#case   -> EXP LCOLON;
 decomp
     case -> case $exp of $lcolon end  %B%
     case2 -> f(18) case $exp of $lcolon end      %W%

colon   -> LCST STAT;
 decomp
     colon -> $lcst : f(11) $stat2      %%

while   -> EXP STAT;
 decomp
     while -> while $exp do $stat2     %W%
     while2 -> f(18) while $exp do $stat2        %W%

with    -> LVARBL STAT;
 decomp
     with -> with $lvarbl do $stat2    %W%
     with2 -> f(18) with $lvarbl do $stat2       %W%

labstat -> INTCST STAT;
 decomp
     labstat -> $intcst : f(11) $stat2  %B%

times   -> INTCST CST;
 decomp
     times -> $intcst * $cst  %%

prog    -> DEFID LIDENT;
 decomp
     prog -> program $defid f(3) $lident ;       %RC%

proc    -> DEFID LPARAM;
 decomp
     proc -> procedure $defid f(3) $lparam ;     %LB%

lelem   -> ELEM + ... ;
 decomp
     lelem = $elem ... ,      %LH,M%

label   -> INTCST + ... ;
 decomp
     label = label $intcst ; ... ,      %LH,H%

lexp    -> EXP + ... ;
 decomp
     lexp  = ( $exp ) ... ,   %LH,W,M,H%
     lexp1 = $exp ... ,       %LH,M,W%

lident -> IDENT + ... ;
 decomp
     lident  = ( $ident ) ... ,        %LH,M,W,H%
     lident1 = $ident ... ,   %LH,M,W%

lcst    -> CST + ... ;
 decomp
     lcst = $cst ... ,        %LH%

ldefid -> DEFID + ... ;
 decomp
     ldefid = $defid ... ,    %LH%

var     -> DECL + ... ;
 decomp
     var = var $decl ; ... ;  %LV,I,W,A,H%

ltyp    -> SPLTYP + ... ;
 decomp
     ltyp = [ $spltyp ] ... , %LH,M,W,H%

lfield -> FIELD + ... ;
 decomp
     lfield = record $field end ... ;   %LV,W,I,A,P%

lcolrc -> COLRC + ... ;
 decomp
     lcolrc = $colrc ... ;    %LV,J,W.B%

lstat   -> STAT + ... ;
 decomp
     lstat   = begin $stat end ... ;   %LV,W,J,LL,D%
     lstat2  = begin $stat end ... ;   %LV,W,D%
     stat_syn -> f(18) $stat3 %%

lcolon  -> COLON + ... ;
 decomp
     lcolon = $colon ... ;    %LV,W,J,B%

lvarbl  -> VARBL + ... ;
 decomp
     lvarbl = $varbl ... ,    %LH,M%

lparam  -> PARAM + ... ;
 decomp
     lparam = ( $param ) ... ;          %LH,W,M,H%

lval    -> VAL + ... ;
 decomp
     lval = ( $val ) ... ,    %LH,W.H%

const   -> EQLC + ... ;
 decomp
     const = const $eqlc ; ... ;        %LV.W,I.H.A,B%

type    -> EQLT + ... ;
 decomp
     type = type $eqlt ; ... ;          %LV.B.W,H,A,I%

value   -> EQLV + ... ;
 decomp
     value = value $eqlv ; ... ;        %LV,H,W,I,A%

lzone   -> ZONE + ... ;
 decomp
     lzone = $zone ...        %LV.J.W.B%

lline   -> LINE + ... ;
 decomp
     lline = $line ...        %LV%

lchar   -> CHARCST + ... ;
 decomp
     lchar = $charcst ... ,   %LH%

for     -> IDENT STEP STAT;
 decomp
     for  -> for $ident := $step do $stat2        %W%
     for2 -> f(18) for $ident := $step do $stat2  %W%
```

```
    if      -> EXP STAT STAT;
*    decomp
*        if  -> if $exp then $stat2 f(10) else $stat2 %W%
*        if2 -> f(18) if $exp then $stat2 f(10) else $stat2
%W%
*
    func    -> DEFID LPARAM IDENT;
*    decomp
*        func -> function $defid f(3) $lparam f(4) : $ident ;
%LB%
*
    block   -> TITLE LZONE BODY;
*    decomp
*        block -> $title $lzone $body $title1 $1      %B%
*        prog1 -> #.    %%
*        modul1 -> #.   %%
*        proc1 -> ;     %%
*        func1 -> ;     %%
*        meta1 -> ;     %%
*
    module  -> IDENT;
*    decomp
*        module -> module $ident ;          %RC%
*
    export  -> DECL + ... ;
*    decomp
*        export = export $decl ; ... ;      %LV,I,W,A,H%
*
    import  -> DECL + ... ;
*    decomp
*        import = import $decl ; ... ;      %LV,W,I,A,H%
*
    lexport -> IDENT + ... ;
*    decomp
*        lexport = exporte $ident $ ... ,   %LH,W,I%
*
    limport -> IMPITEM + ... ;
*    decomp
*        limport = import $impitem $ ... ;  %LV,W,I,A%
*
    impitem -> ALFACST LIDENT;
*    decomp
*        impitem -> $alfacst : $ident1      %%
*
```

```
*synonymes
*    ident1  :: ident       %%
*    repeatls :: lstat      %%
*    lfield1  :: lfield     %%
*    lexp1    :: lexp       %%
*    typ1     :: typ        %W,M%
*    exp1     :: exp        %W%
*    valu1    :: valu       %W%
*    case2    :: case       %%
*    with2    :: with       %%
*    for2     :: for        %%
*    if2      :: if %%
*    repeat2  :: repeat     %%
*    while2   :: while      %%
*    call2    :: call       %%
*    goto2    :: goto       %%
*    ass2     :: ass        %%
*    lstat2   :: lstat      %%
*    modul1   :: module     %%
*    prog1    :: prog       %%
*    proc1    :: proc       %%
*    func1    :: func       %%
*    meta1    :: meta       %%
*    minus2   :: minus      %%
*    plus2    :: plus       %%
*classes
*    stat3  :: = lstat2 %%
*    title1 :: = proc1 func1 modul1 prog1 meta1 %%
*    spltyp :: = range ident ident1 %%
*    exp2   :: = minus2 plus2 %%
*    exp3   :: = minus2 %%
*synonymes
*    stat_syn :: stat3      %%
*classes
*    stat2 :: = stat_syn if2 for2 ass2 call2 while2 repeat2
*            case2 with2 goto2 %%
*
*
*devnoncolle
*    :
*    =
*
*dernoncolle
*    :
*    :
*    ,
*
```

8. Annexe 2 : Message d'erreur

Les messages d'erreur du compilateur **cpretty** sont les suivants:

- 'compilation -Pretty refusée : arbre attendu'
  La compilation **cpretty** ne peut être lancée que sur un arbre ;
- 'le langage "Foo" n'est pas connu du système Mentor'
  Les tables du langage "Foo" n'ont pas été générées ou ne sont pas trouvées;
- 'la génération Pretty ne peut se faire sur cette zone '
  La commande cpretty ne peut être lancée que sur un sous-arbre correspondant aux opérateurs **language, abs_syn, operator, chapt** ;
- ' "maxnom" trop petit '
  La dimension de table "maxnom" est trop petite; voir la maintenance !;
- 'la stucture "x" doit avoir un schéma de liste '
  "x" est un opérateur de type liste ou un synonyme de liste, et ne peut avoir un schéma de type fixe;
- ' "x" n'est pas une structure de liste ou synonyme de liste '
  "x" est un opérateur de type fixe, ou synonyme de phylum ou de classe, et doit avoir un schéma de type fixe;
- 'attention: le schema de "x" a déja été défini'
  Si une structure a plusieurs schémas de décompilation, seul le dernier est pris en compte;
- 'attention: la structure "x" n'a pas été définie '
  Il peut s'agir d'une faute d'orthographe dans le nom d'un opérateur ou d'un phylum, ou d'un synonyme ou classe non déclaré;
- 'attention: la structure "x" a déjà été définie '
  Le nom du synonyme ou de la classe déclarée existe déjà;
- 'directive de paragraphage pour le synonyme "x" non autorisée ici, utilisez un schéma;'
  Une directive de paragraphage est utilisée dans la déclaration d'un synonyme, alors qu'il faut spécifier cette directive dans un schéma associé au synonyme;
- 'la classe "x" ne peut avoir de directive de paragraphage, utilisez un synonyme '
  Une classe ne peut pas avoir de schéma de décompilation, ni de directives de paragraphage;

Bibliographie

[Ber 85] G.Berry, J.M.Hullot, *CEYX Version 5;III: VPRINT, le composeur de Ceyx,* Rapport Technique No.46, INRIA, février 1985

[Des 84] T.Despeyroux, *Executable Specification of Static Semantics,* International Symposium "Semantics of Data Types" Lectures Notes in Computer Science, No 173, Springer Verlag

[Don 83] V.Donzeau-Gouge, G.Kahn, B.Lang, B.Mélèse, E.Morcos, *Outline of a tool for document manipulation,* IFIP, septembre 1983, Paris

[Don 84a] V.Donzeau-Gouge, B.Lang, B.Mélèse, *Practical Applications of a Syntax Directed Program Manipulation Environment,* Proceedings of the 7th Int. Conf. on Soft. Eng., Orlando, Florida, March 1984

[Don 84b] V.Donzeau-Gouge, G.Kahn, B.Lang, B.Mélèse, *Documents Structure and Modularity in Mentor,* ACM SIGSOFT/SIGPLAN Soft. Eng. Symp. on Practical Software Development Environments, Pittsburgh, April 1984

[Fei 81] P.H.Feiler, R.Medina-Mora, *An incremental programming environment,* IEEE Trans. on Soft. Eng. SE-7, No 5, pp.472-481, Sept 1981

[Kah 83] G.Kahn, B.Lang, B.Mélèse, E.Morcos, *: a formalism to specify formalisms,* Science of Computer Programming, North Holland, Vol. 3, No. 2, 151-188, Aug 1983

[Luc 82] L.Lucrèce, P.Maurice, A.M.Vercoustre, *Minerve: Manuel de référence,* Rapport Technique No.17,INRIA, Septembre 1982

[Mel 82] B.Mélèse, *Métal, un langage de spécification pour le système Mentor,* Technique et Science Informatique (AFCET), Vol. 1 No 4, Juillet-Aout 1982

[Mel 83a] B.Mélèse, *Mentor Rapport: Manipulation de textes structurés sous Mentor,* Rapport Technique No. 23, INRIA, Avril 1983

[Mel 83b] B.Mélèse, *Edition structurée, édition non structurée,* Rapport de recherche No. 253, INRIA, novembre 1983

[Mel 85] B.Mélèse, V.Migot, D.Verove, *The Mentor-V5 Documentation,* Rapport Technique No.43, INRIA, Janvier 1985

[Mig 85] V.Migot, *Saisie de texte interactive sous Mentor,* Rapport INRIA, Novembre 1985

[Ver 83] A.M.Vercoustre, *Minerve: un Méta-éditeur syntaxique,* Rapport de Recherche No.229, INRIA, Juillet 1983

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique