Minerve.Un meta-editeur syntaxique

Anne-Marie Vercoustre

INRIA

CENTRE DE ROCQUENCOURT

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

Rapports de Recherche

N° 229

MINERVE
UN META-ÉDITEUR SYNTAXIQUE

Anne-Marie VERCOUSTRE

Juillet 1983

CENTRE DE ROCQUENCOURT

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

Rapports de Recherche

N° 229

MINERVE
UN META-ÉDITEUR SYNTAXIQUE

Anne-Marie VERCOUSTRE

Juillet 1983

MINERVE

Un méta-éditeur syntaxique

Anne-Marie Vercoustre
INRIA
domaine de Voluceau
78153 Le Chesnay

résumé

Minerve est un méta-éditeur syntaxique inspiré de Mentor, dont toutes les fonctions sont paramétrés par le langage de programmation considéré; le langage de commande est défini pour permettre la manipulation d'éléments de programme en terme d'entités syntaxiques imbriquées.
Nous présentons d'abord les principes de réalisation de l'éditeur et du paragrapheur, y compris le traitement des commentaires. Puis nous expliquons comment spécifier l'éditeur pour un langage donné (génération).

abstract

Minerve is a syntax directed editor designed to be language independant : all functions are parametrized with respect to the dedicated programming language; Minerve 's commands permit to select part of programs by using syntactic structures in some expressions.
First, we describe the general design of the editor and the pretty-printer, including the handling of comments; then, we explain how to specify Minerve for your favorite language.

1.introduction

le développement actuel de l'informatique se caractérise par la montée des besoins en logiciel qui deviennent prépondérants par rapport au matériel. Dans cette perspective, l'écriture, la mise au point et la maintenance d'un logiciel ne peuvent se faire sans un ensemble d'outils constituant un environnement cohérent pour le programmeur. le nombre de travaux de recherche , de publications et de réalisations dans ce domaine (génie logiciel) montre l'actualité du problème. L'idée commence à se répandre que l'importance de cet environnement serait prépondérant par rapport au choix du langage de programmation lui-même. Toutefois, on peut noter avec [DON80] que la réalisation, et a fortiori la production automatique, d'environnements de programmation est difficilement envisageable avec un mauvais langage de programmation.

Un des axes de recherche est la conception et la réalisation d'éditeurs orientés structures ayant une connaissance de la syntaxe, éventuellement de la sémantique, du langage de programmation considéré [DON75] [BUR80] [TEI81] .
Dans cet axe, Minerve est un éditeur syntaxique, défini et réalisé dans le cadre d'un environnement de programmation autour de Pascal et Legos [LEG80] sur Mitra 225 (le projet a fait l'objet d'un contrat avec la SEMS, financé par l'ADI; la réalisation est faite en collaboration avec l'université Paul Sabatier de Toulouse).

Son originalité tient principalement dans le fait que c'est un méta-éditeur de programmes, en ce sens que toutes les fonctions d'édition sont indépendantes du langage de programmation utilisé; l'utilisation de Syntax [BOU80] a permi de paramétrer entièrement Minerve par le langage à éditer.
Les avantages d'une telle réalisation , indépendante du langage, sont les suivants:

-- génération d'un même environnement de programmation pour des langages différents: l'usager n'a pas à investir dans l'apprentissage d'un nouvel éditeur pour tout nouveau langage et l'éditeur est une aide à l'apprentissage du langage.

-- génération de paragrapheur (pretty-printer) personnalisé pour une équipe; la présentation des programmes peut être normalisée, facilitant la circulation et la compréhension des

programmes par l'ensemble de l'équipe (aide à la maintenance).

-- réalisation d'un éditeur multi-langages: par exemple, un éditeur pouvant éditer simultanément :
un langage de programmation (ou plusieurs)
un langage de commentaires (ou plusieurs)
un langage d'assertion ...

Minerve n'est pas, actuellement, multi-langages: pour un langage X on génère Minerve X; les commentaires du programme sont édités comme du texte.

-- aide au transport de logiciel: l'analyseur-constructeur est spécifié pour le langage sur la machine source, le décompilateur-paragrapheur est spécifié pour le langage de la machine objet.
la même méthode peut servir d'aide à la traduction d'un langage dans un autre, pourvu que les deux langages ne soient pas sémantiquement trop éloignés.

Nous présentons ici les principes de réalisation de l'éditeur, du paragrapheur et le traitement des commentaires, ainsi que la spécification de l'éditeur pour un langage donné (génération).

2.présentation de l'éditeur

2.1 objectif

Il s'agissait de réaliser un éditeur syntaxique commun aux langages Pascal et Legos (langage dérivé de Pascal, prenant en compte la modularité et les problèmes système) sur Mitra 225 (mini 16 bits). Ayant l'expérience de l'utilisation de Mentor, nous voulions réaliser un tel éditeur, mais indépendant du langage et pouvant tourner avec de petites configurations mémoire.

Nous avons implémenté un mécanisme de mémoire virtuelle et redéfini un éditeur dans lequel les différents processeurs sont paramétrés par le langage:

l'analyseur syntaxique et le constructeur d'arbres sont générés par Syntax, à partir de la grammaire du langage.

Le décompilateur, les commandes permettant la recherche et les transformations d'éléments de programme, sont paramétrés à partir d'une description de la syntaxe abstraite et des schémas de décompilation associés.

Nous ne décrirons pas ici l'ensemble des commandes de l'éditeur, qui ont fait l'objet d'un autre rapport [LUC82].

Nous allons préciser les éléments de la syntaxe abstraite utilisés, ainsi que la façon dont la description de ces éléments est utilisée par les processeurs de l'éditeur.

2.2 syntaxe abstraite

L'éditeur Minerve travaille à partir d'une représentation interne des programmes, sous forme d'arbres dits de syntaxe abstraite. Celle-ci caractérise les arbres qui correspondent à des programmes légaux. La correspondance entre le texte source et l'arbre abstrait est faite, lors de l'analyse syntaxique, par un constructeur d'arbre. La spécification de cet analyseur -constructeur est décrite en 4.1 .

La syntaxe abstraite associée à un langage est définie par un ensemble de règles sur les objets suivants (structures prédéfinies) :

-- des terminaux qui correspondent aux feuilles des arborescences (par exemple pour Pascal : ident, nombre, nil).

-- des opérateurs, ayant un nombre fixe de sous-structures (fils) de types déterminés.

pour indiquer quel type de structure (axiome) doit être construite, à partir du texte qui est lu.

exemples :

k: [stat] &

l'usager indique qu'il va entrer une instruction; le sous-arbre construit sera référencé par le pointeur k.

CH(k,[ident] 'A,B,C')

l'usager veut changer la structure courante, référencée par k, par le sous-arbre construit à partir de la liste d'identificateurs; l'éditeur vérifie, à l'aide des tables des schémas prédéfinis, que la substitution est possible.

Par ailleurs, il est également possible de construire ses programmes en utilisant des squelettes d' arbres crées à partir des schémas prédéfinis, en affectant un nom de schéma à une variable de référence (pointeur).

exemple:

rac : prog ;

Cette commande crée un arbre repéré par le pointeur rac, dont la racine est de type 'prog', et dont les fils sont des feuilles correspondant aux méta-variables du schéma associé à 'prog'.

Cette approche sera particulièrement utile en phase d'apprentissage de la syntaxe abstraite, et pour effectuer une construction descendante des programmes.

2.4 Synonymes.

En plus des structures de la syntaxe abstraite définies précédement, nous avons utilisé dans les schémas de décompilation, des synonymes permettant d'introduire la notion de contexte d'une structure.

exemple :

le schéma d'une instruction 'if' peut être défini comme suit:

```
IF = if $cond then $true-part else $false-part

où :  $cond est synonyme de $exp
      $true-part est synonyme de $stat
      $false-part est synonyme de $stat
```

Par rapport au schéma de décompilation donné en 2.3, on voit que l'introduction du contexte permet de différencier les deux occurrences de 'stat'; de même l'utilisation du synonyme 'cond' permet de distinguer l' 'expression' d'un 'if' (phylum du premier fils), de celle apparaissant, par exemple, dans une structure 'affect'.

La recherche d'un synonyme permet donc de préciser, implicitement le contexte dans lequel on veut trouver une structure; Par contre, une structure ne sera pas différenciée de son synonyme, lorsqu'on voudra effectuer des transformations de programmes: une structure 'true-part' pourra évidemment être échangée avec une 'false-part'.

D'autres processeurs pourront également utiliser les synonymes pour différencier les occurrences d'une même structure abstraite.

2.5 commandes.

Le langage de commande de Minerve permet, non seulement d'effectuer des déplacements élémentaires dans l'arbre abstrait, mais aussi, et surtout, de désigner des parties de programmes en terme d'entités syntaxiques; par exemple : chercher une instruction 'if' dans la procedure TEST, ou trouver la déclaration de la variable A dans le bloc environnant.

Essentiellement, nous avons étendu les possibilités de recherche selon un schéma, prédéfini ou non, existant dans Mentor (pattern-matching) :
Nous avons fait de cette fonction un opérateur (symbole '..'), pouvant être utilisé dans des expressions de recherche, et dont les opérandes sont des structures prédéfinies, des synonymes, ou des repères sur des schémas définis par l'utilisateur.
Nous avons introduit également un autre opérateur ('.'), qui effectue une recherche horizontale (niveau 1 des fils), ce qui permet de sélectionner différemment les éléments, et d'optimiser certaines recherches. Enfin l'opérateur '@' permet de remonter dans le programme jusqu'à une structure d'un type donné.

Pour plus de détails sur les expressions de recherche, on pourra se référer à [LUC82].

exemple : (avec la syntaxe abstraite de Pascal décrite en annexe)

si le pointeur courant K se trouve sur le noeud racine d'un programme, il est possible d'accéder aux éléments du programme, de niveau 1 par :

```
K.HDPROG     (* on référence l'entête du programme *)
K.DEFPART    (*accès aux listes de label, const, type  *)
K.DCLPART    (* accès à la liste de déclarations *)
K.LPROC      (* accès à la liste de procédures *)
K.BODY       (* accès au corps    *)
K.LSTAT      (*   idem       *)
```

Remarques:
Les expressions K.BODY et K.LSTAT permettent toutes les deux d'accéder au corps du programme, puisque 'body' est synonyme de 'lstat'.
Les même commandes avec '..' donneraient le même résultat, sauf K.LSTAT (un noeud de type 'lstat' pouvant être rencontré avant le corps du programme, dans une procedure interne); mais les recherches coûteraient plus cher, puisqu'elles s'effectueraient de haut en bas et de gauche à droite, jusqu'aux feuilles.

les éléments de niveau inférieur seront accédés par :

```
K..IF        (* pour chercher le premier if-statement *)
             (* dans l'ordre textuel *)

K.BODY..IF   (*premier if dans le corps principal *)


@PROC          (*pour remonter à la
               procédure englobante *)
```

Les noms des éléments référencés sont évidemment dépendants langage, mais le mécanisme est général.

Pour rechercher une procedure de nom LIRE, on pourra utiliser, par exemple, la séquence suivante:

```
NOM : [IDENT] 'LIRE'          le pointeur NOM repère
                              un sous-arbre réduit à
```

```
                                  l'identificateur LIRE
                                  ( schéma utilisateur )

K : @*.LPROC                      pour remonter, de n'importe
                                  où, à la liste de procédure
                                  de plus haut niveau

loop                              itère sur les éléments de
                                  liste
   on EQ(K..ident,NOM)            compare le nom de la procédure
                                  avec NOM
   (Y:K;exit)
end
```

Remarque.
La définition du schéma NOM pour préciser le nom serait inutile en utilisant un mécanisme de nommage des structures tel que celui décrit dans [COU78].

Si l'on compare avec Mentor, notre langage de commande se situe, du point de vue utilisation, comme un intermédiaire entre Mentol, pour les commandes de déplacements élémentaires, et un ensemble de macros effectuant des recherches, vérifications et transformations syntaxiques.
En ce qui concerne les vérifications et transformations sémantiques des programmes, on est ramené à écrire un environnement de programmation dépendant langage; ceci pourra se faire en utilisant des fichiers de commandes, et les commandes de répétition (loop et scan) avec test.

3.présentation du paragrapheur.

Dans un éditeur construit autour d'un arbre de syntaxe abstraite, le paragrapheur n'est pas seulement un outil de mise en forme des programmes (pretty-printer); il s'agit d'abord de restituer un texte source à partir de la forme interne des programmes (décompilation).
Le décompilateur-paragrapheur de Minerve travaille à partir des tables associées aux schémas prédefinis, et de directives de paragraphage attachées à chaque structure. Les spécifications précises seront présentées en 4.

L'idée de base est que la décompilation d'une structure ne devrait pas dépendre du contexte où elle apparait; en effet, si deux entités s'écrivent différemment, elles devraient correspondre logiquement à des concepts différents. Evidemment, ce principe est rarement vérifié, d'une part parce que les langages ne sont pas tous bien définis, d'autre part parce que le choix de la syntaxe abstraite correspond souvent à des critères plus sémantiques que syntaxiques.
Quant à la mise en forme des structures, elle peut dépendre non seulement du contexte défini par la syntaxe, mais du programme lui-même (coupure de lignes, alignement vertical).
Nous avons donc introduit des mécanismes permettant de tenir compte de ces problèmes.

3.1 fonctions et directives

-- Le décompilateur travaille en effectuant un parcours de l'arbre, avec consultation en parallèle des tables associées à chaque structure (type du noeud rencontré) et est appelé récursivement sur chaque fils : c'est un interpréteur de tables contenant la description des schémas prédéfinis. Le paragraphage s'effectue en utilisant des directives de mise en forme attachées à chaque structure abstraite. ces directives sont interprétées au début et à la fin de la décompilation de chaque sous structure.

exemple 1 :

Si le schéma de décompilation de la structure 'lstat' est le suivant :

```
lstat = begin   $stat   end   ... ;    %B,LV,I%
```

Les directives associées (données entre '%') sont les suivantes:

B	(BLOC)	la structure tient, seule, sur un ensemble de lignes
LV		c'est une liste verticale
I	(INDENTE)	les éléments sont décalés à droite par rapport aux symboles de parenthésage

Avec ces directives, la mise en forme d'une liste d'instruction sera la suivante :

```
(*positionnement debut de ligne*)
begin
(*indentation de la marge*)
    a:=x+2;
    b:=a;
(*restitution de la marge*)
end
(*retour ligne *)
```

exemple 2 :

En Pascal, si l'on souhaite présenter une liste de constantes de la façon suivante:

```
CONST
     MAXRES     = 10;
     MAXNTERM   = 10;
     MAXOP      = 9;
```

la structure 'lcdcl', correspndant a une liste de déclaration de constantes, doit être spécifiée de la façon suivante:

```
LCDCL  =  CONST $DCLCST  ... ; %LV,J,A%
```

La directive d'alignement 'A' est interprétée au niveau de la liste, et la valeur d'alignement calculée est utilisée au nivau des éléments de liste, ce qui permet d'effectuer un traitement plus global qu'en utilisant des tabulations .

On trouvera en 4.3 la liste des directives existantes.

L'utilisation de tables et de directives interprétées permet de mettre en facteur l'information : on a plutot des modèles de

paragraphage que des actions directement exécutables.
Ceci permet de faire certains traitements sophistiqués, dépendants du programe lui-même, par exemple, l'alignement vertical d'un symbole dans une liste.
Un autre avantage de cette approche est qu'il est facile de permettre à l'usager de modifier interactivement les directives; on pourrait, par exemple, afficher un menu pour chaque structure prédéfinie.

-- Les espacements entre symbole (un espace, ou pas d'espace) sont définis par des directives attachées aux symboles terminaux , hors du contexte où ils apparaissent. Par défaut, les mots réserves (begin,program etc...) sont précedés et suivis d'un blanc, alors que les symboles ([, (, + etc ...) sont écrits sans espacement. Les espacements standards pour les symboles peuvent être modifiés, en spécifiant une liste de symboles non précédés d'un blanc ('non collés devant') et une liste de symboles suivis d'un blanc ('non collés derriere ').

Toutefois,il semble que cette approche ne soit pas assez souple, un même symbole étant souvant utilisé dans des contextes très différents; nous envisageons donc de spécifier cette directive au niveau du schéma où apparait le symbole lui-même.

-- La possibilité d'avoir un effet de zoom (holophraste) est particulièrement intéressante en mode interactif, puisque généralement le programme ne tient pas sur l'écran. Le niveau de détail apparaissant sur l'écran varie avec la valeur de l'holophraste. les parties non détaillées sont remplacées par les symboles '#' pour une structure de type operateur, et '...' pour un liste (les symboles ou mots de parenthésage sont conservés).

exemple

```
program test(...);
    var ... ;
begin
    ...
end.
```

Le listing holophrasté d'un programme peut servir également à des fins de documentation : listing des entêtes de procedures, avec paramètres et commentaires.
La façon dont s'effectue l'holophraste est specifiée par la directive 'poids' attachée à une structure. Ceci permet d'avoir

une visibilité non homogène sur des éléments de même niveau dans l'arbre abstrait; on préférera certainement obtenir :

```
var struct          : # ;
    natelem         : # ;
    setval,curval   : # ;
```

plutôt que :

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

La première forme pourra être obtenue en donnant un 'poids' ('W') sur les structures appartenants au phylum 'type '.

-- les priorités des opérateurs

il s'agit de pouvoir rétablir , à la décompilation des expressions, les parenthèses nécessaires pour rendre cohérentes les représentations interne et textuelle d'une expression.

A chaque opérateur sera donc associé une directive 'priorité', spécifiée par R(n) (par défaut, la priorité n vaut zero); l'utilisation des priorités évite d'avoir deux schémas de décompilation pour chaque opérateur, avec ou sans parenthèses), dépendants de l'opérateur du père. (ce problème disparait, si, au niveau de la syntaxe abstraite, on choisit d' "aplatir" les expressions, c'est à dire d'avoir un noeud 'expression' terminal, auquel est associé le texte de l'expression, toutes les parenthèses étant conservées).

3.2 contexte

Nous avons vu que les critères de choix de la syntaxe abstraite peuvent conduire à représenter par un même noeud abstrait des entités syntaxiques différentes. Ceci arrive souvent pour les listes de mêmes éléments, mais parenthésées par des symboles différents.

L'utilistation des synonymes dans les schémas de décompilation permet de mémoriser le contexte d'appel de la structure et de restituer le texte qui convient. Lors du parcours de l'arbre, il est gardé trace du synonyme dans le schéma courant, et c'est le schéma associé au synonyme qui sera utilisé pour décompiler le fils correspondant.

Dans la réalisation actuelle, il n'est possible de définir un schéma de décompilation que pour les synonymes de listes. Le

mécanisme général, implanté dans [LEB83] permet, par exemple, de restituer les parenthèses dans les expressions, à la place du mécanisme de priorité.

Toutefois, l'utilisation des synonymes ne permet pas de traiter les cas où la décompilation d'une structure dépend d'un de ses descendants, à un niveau quelconque; C'est à dire lorsqu'il faut effectuer du "pattern-matching" sur le sous-arbre pour déterminer le contexte de décompilation. Une solution est de disposer de plusieurs schémas de décompilation et de sélectionner celui qui convient selon le résultat du pattern-matching [DON83]. Cette approche, combinée avec l'utilisation des synonymes pour sauvegarder le contexte du père, permet de résoudre le problème bien connu du if-then-else ambigu de pascal (dangling else); cependant elle conduit à une spécification assez lourde pour l'utilisateur.
Nous avons préféré introduire la possibilité de faire appel, lors de la décompilation, à des procédures, prédéfinies ou définies par l'utilisateur, permettant de traiter tous les cas considérés comme des exceptions au traitement général.
L'appel de fonction est contenu dans le schéma de décompilation de la structure, avec la syntaxe F(n) , où n est le numero de la procédure à appeler.

exemple :

IF = if $cond then $true-part F(12) else $false-part

La décompilation d'un 'if', selon ce schéma, se fera avec un appel de procédure après la décompilation du deuxième fils ('true-part').
la procedure appelée devra effectuer le "pattern-matching" suivant:

"tant que le noeud courant est un 'if-statement' dont le troisième fils est vide, prendre le père comme noeud courant; si ce noeud courant est un 'if' ayant une partie 'false-part' non vide, alors 'else ' devra être écrit (dangling else), dans le schéma initial, sinon ce n'est pas nécessaire ".

Cette approche évite de mémoriser le contexte à partir du père, et permet de n'avoir qu'un schéma de décompilation par structure, ce qui est suffisant à 95% [W0081].
Pour écrire ses fonctions, l'utilisateur peut manipuler des variables booléennes prédéfinies permettant d'activer ou d'inhiber le traitement standard.

L'appel de procedures prédéfinies est utilisé actuellement pour traiter l'espacement entre symboles dépendant du contexte; on trouve par exemple, pour Pascal, la spécification suivante :

```
COL = $LCST : F(12) $STAT   %%
```

l'appel de fonction prédéfinie F(12) permet d'écrire un blanc après le symbole ':' , dans ce contexte particulier;

En fait, le mécanisme d'appel de fonctions permet à l'utilisateur de spécifier et de réaliser tout traitement particulier dépendant du programme, ou rendu nécessaire par le langage ou le choix de la syntaxe abstraite. Mais l'utilisation n'en est raisonnable que si cela reste l'exception.

3.3 coupures de ligne

Les mécanismes de mise en page décrits précédement correspondent au paragraphage logique entièrement défini par les entités syntaxiques ; Lorsque la ligne logique à écrire est plus longue que la ligne physique, il faut déclencher un mécanisme de coupure de ligne.
Dans Minerve, le décompilateur logique et le processeur d'écriture effective (display) sont découplés, donc modifiables de façon indépendante;
Dans l'implémentation actuelle, le mécanisme de coupure est particulièrement simple, puisqu'une ligne est coupée lorsque l'unité lexicale suivante ne tient pas sur la ligne physique. Un marqueur de marge permet de mettre en évidence une coupure par une indentation particulière : soit la marge courante augmentée de deux blancs, soit la position de la colonne courante en entrée d'une structure ayant la directive 'marqueur'.

exemple:

Pour Pascal, avec les spécifications suivantes:

```
HDPROG   =  program $IDENT  $LEXTERN  ;    %%
LEXTERN  =  (  $IDENT  )      ...  ,       %LH,M%
```

La décompilation de l'entête de programme, en cas de coupure de ligne, sera :

```
program parag(outfile,printchar,printchaine,retourligne,
              ligneblanche,posmarge,printres,coupure,indent,
              initparag,nilnode,chcomment,comprint,memoire);
```

la structure prédéfinie 'l-extern' a été spécifiée avec la directive 'marqueur' ('M'), ce qui, à la décompilation, permet de mémoriser une marge de coupure.

A notre avis, un mécanisme de coupure même simple, ne perturbe pas beaucoup l'aspect général du texte . Naturellement, pour ceux qui regardent le listing à la loupe, des mécanismes plus sophistiqués devraient être implémentés en utilisant un "tampon" de sortie; par exemple, transformer certaines listes horizontales trop longues en listes verticales; ou bien simuler l'algorithme décrit dans [COU78] ,qui détermine une coupure dans la structure courante, en la parcourant par niveaux horizontaux, jusqu'à trouver une coupure posible entre deux sous-structures.

3.4 gestion des commentaires

Dans les langages de programmation actuels, la syntaxe des commentaires est réduite au minimum : c'est généralement du texte en langage naturel, placé presque n'importe où dans le programme, et encadré par un ou deux symboles particuliers. Le refus de leur donner une syntaxe plus riche entraine un certain nombre de problèmes dans l'utilisation des commentaires par le programmeur, et leur gestion par des outils de manipulation automatique.

1.qu'est-ce qu'un commentaire ?

Les commentaires sont utilisés dans les programmes pour remplir différentes fonctions:

- commenter une instruction, une partie de programme, soit parce que le programmeur a rencontré une difficulté, soit pour pallier un manque de précision du langage.
  C'est ce que nous appellerons documentation interne du programme.

- documenter le programme (documentation externe).
  il s'agit de préciser les spécifications externes du programme (fonctions, interface avec l'exterieur) ou les différentes versions (date, réalisateurs ...)

- supprimer momentanément des parties de programme, dans le cas où le compilateur ne comporte pas d'instructions de compilation conditionnelle, ou d'outil d'assistance à l'exécution (trace).

- établir des propriétés devant être vérifiées en ce point (assertions).

- spécifier des parties de programme non encore écrites.

Clairement, ces commentaires correspondent à des fonctionnalités différentes, et devraient être traités différemment, voire par des processeurs différents.

2.que veut-on commenter ?

La position d'un commentaire n'étant ni précisée, ni limitée par la syntaxe des langages, c'est le lecteur du programme source qui

doit deviner à quoi se rattache l'information:
veut-on commenter ce qui suit, ce qui précède, l'entité locale, ou une plus englobante ? l'usager ne peut qu'essayer d'interpréter des renseignement rudimentaires, comme : mise en page par rapport au texte programme, ou contenu du commentaire lui-même. Par ailleurs, la position des commentaires pourra dépendre des habitudes ou des normes définies par une équipe.

Un outil de traitement automatique des commentaires n'aura pas, au niveau du texte source, de moyen de savoir exactement ce que le programmeur veut commenter. Lorsque les commentaires sont attachés directement dans l'arbre de syntaxe abstraite, c'est le programmeur qui décide à quelle structure ou sous-structure se rapporte l'information; la situation semble donc moins critique lorsqu'on travaille toujours sous l'éditeur, mais il semble difficile de renoncer définitivement à communiquer avec le monde exterieur par l'intermediaire d'un texte source.

3.impression du commentaire

La façon dont l'utilisateur souhaite formater les commentaires dépend souvent de la nature de ceux-ci. Dans certain cas, il voudra imposer sa propre mise en page (tableaux, documentation externe) , dans d'autres cas il souhaitera que les commentaires soient alignés avec les éléments de programme ; enfin certains commentaires pourront être rajoutés automatiquement par le décompilateur (exemple : date de mise à jour, nom de procedure courante).

4.traitements effectués

Au niveau de l'éditeur, un commentaire est caractérisé par sa nature (texte, documentation, message d'erreurs du compilateur ...) et un ensemble de directives de mise en page. Certaines de ces directives sont , par défaut, celles qui sont associées à la nature du commentaire. Actuellement, ces aspects sont simplement ébauchés dans Minerve.

Au niveau du texte source, nous n'avons pas modifié la syntaxe des commentaires (pas de directives dans les commentaires).
Pour rattacher un commentaire lors de la construction de l'arbre abstrait, nous utilisons une heuristique basée sur les principes suivants :

- rattachement à la structure de plus haut niveau possible; lors de la construction d'un sous-arbre, le noeud "père" crée

hérite du commentaire préfixé de son premier fils; les commentaires pré et post fixés attachés aux autres fils restent attachés à ceux-ci.

- on favorise le préfixé par rapport au postfixé, c'est à dire qu'un commentaire se rattache plutot à ce qui suit qu'à ce qui précède. Cette stratégie est la plus naturelle et sera généralement satisfaisante pour les commentaires commençant sur une nouvelle ligne (ceux qui commentent des blocs ou des unités syntaxiques pas trop petites); par contre, pour certains langages (Ada par exemple), il sera impossible d''écrire de façon naturelle un commentaire postfixé associé aux éléments d'une liste.

exemple:

```
A:=  1   ;    -- comment1
B:=  2        -- comment2
          ;
```

comment1 est préfixé de la deuxieme instruction
comment2 est postfixé de cette instruction, mais la position du ';' n'est pas satisfaisante.

- effectuer un traitement des commentaires compatible avec la décompilation pour éviter le déplacement de ceux-ci dans le texte source, à la suite de décompilations et d'analyses répétées : il s'agit d'obtenir la stabilité, d'ordre 1 ou 2, pour les commentaires , dans l'opération analyse-décompilation.

4.génération des tables

4.1 principes généraux

La génération des tables de Minerve a été faite en utilisant de façon intensive Syntax[SYN81], générateur d'analyseur syntaxique disponible sur Multics (écrit en PL1).
Nous n'avons pas défini un langage de spécification d'arbre comme dans [MEL82], mais utilisé directement la possibilité d'appel d' actions sémantiques existant dans Syntax; la spécification est moins explicite , mais l'écriture, pour l' usager, simplifiée.
Syntax a été utilisé à la fois pour générer l'analyseur-constructeur de Minerve, et pour réaliser le générateur Paria, qui produit des tables à partir d'une description de la syntaxe abstraite (dans le langage DATA).

La génération croisée (Multics-Mitra) produit des tables pour des interpréteurs (analyseurs lexicaux et syntaxiques) qui ont été réécrits en Pascal-Mitra225. Le reste de l'éditeur a été également écrit en Pascal.
Par ailleurs, nous avons évité de dupliquer les informations communes à différents processeurs. Ceci permet , outre d'économiser la place mémoire, d'assurer "de facto" la cohérence de ces données.

Les données générées sont produites sous forme de "constantes" et "value" Pascal, à inclure dans le code. Cette approche, qui permet d'ajuster la taille des tables sans utiliser l'allocation dynamique, est justifiée sur petite machine et dans une optique où l'éditeur est généré pour chaque langage. En contrepartie, l'éditeur doit être recompilé pour tout nouveau langage, et ne permet pas de passer d'un langage à un autre dans une même session.

4.2 analyseur-constructeur d'arbres

L'analyseur-constructeur d'arbres est généré par Syntax à partir des descriptions lexicale et syntaxique du langage, ainsi que des numéros d'actions sémantiques attachées à chaque règle; chaque numero se décompose en deux parties : le numéro de l'action à effectuer et le code du noeud à construire.
Les actions sont complètement indépendantes du langage, et sont exécutées sur chaque réduction lors de l'analyse (ascendante) du programme; l'arbre est construit des feuilles vers la racine, par les actions suivantes :

action 1 (crée feuille)
crée une feuille ayant le code specifié.

action 2 (crée noeud)
crée un noeud à partir des noeuds précédement crées correspondant aux non- terminaux apparaissant en partie droite de la règle.

exemple

<array> = array <l-index> of <type> ;2020

le noeud 'array', de code 20, sera crée, et aura comme fils les noeuds 'l-index' et 'type' précédemment crées.

remarque. Il n'y a pas de possibilité de réorganisation des sous-arbres précédement crées, ni de définition de construction récursive, ce qui peut contraindre à modifier la grammaire.

action 3 (crée liste)
action 4 (mise en liste)
ces deux actions permettent respectivement de créer une liste et de lui ajouter un élément.

exemple :

```
<lstat> = <stat> #;          ;3110
<lstat> = <lstat> #; <stat>  ;4110
```

la première action crée la liste 'lstat', de code 110, la deuxième ajoute un élément à la liste précédement créée, sauf si ce dernier est un élément vide pour lequel aucun noeud n'a été crée .

action 5 (cree vide)
crée un noeud vide, de code 0

action 7 (synthétise)
Cette action ne crée pas de nouveau noeud, mais "remonte" le noeud correspondant au non-terminal en partie droite sur le non-terminal en partie gauche, en effectuant éventuellement un renommage.

exemple :

<l-const> = const <const-list> ;7000

<l-extern> = (<l-ident>) ;7150

dans le deuxième exemple, la liste 'l-ident' est "renommée" 'l-extern' .

remarque.

il n'est pas nécessaire de spécifier une action sur les règles de la forme :
<A> = <B> ;
si on ne veut pas créer de noeud pour le non-terminal A .

action 8 (crée axiome)
Cette action termine la construction d'arbre; elle n'effectue pas, normalement, de création de noeud, la racine ayant été créee à l'étape précédente; toutefois, il faut vérifier que tous les sous-arbres construits ont bien été rattachés à l' arbre de sommet "racine", ce qui peut ne pas être le cas s'il y a eu récupération d'erreur.

Nous devons préciser ici la façon dont sont spécifiés les axiomes du langage qui correspondent aux morceaux de programme pouvant être construits indépendamments.
L'analyseur produit par Syntax n'étant pas multi-axiomes, nous avons rajouté des règles à la grammaire; ces règles contiennent des pseudo- terminaux qui permettent de sélectionner l'état initial de l'analyseur; l'éditeur connait la nature du texte à analyser (indiquée par l'usager) et passe donc comme paramètre le pseudo-terminal correspondant.
La syntaxe des règles de grammaire correspondant aux axiomes est la suivante :
exemple :

```
<axiome>      =  _lstat   <stat-l>      ;8000
<axiome>      =  _prog    <pascal>      ;8000
<axiome>      =  _lvdcl   <l-vardcl>    ;8000
```

'lstat', 'prog', 'lvdcl' sont des noms de structures de la syntaxe abstraite, qui seront dites axiomes.

Enfin précisons que les actions décrites ci-dessus contiennent également les heuristiques de traitement des commentaires, dont nous avons parlé précédement.

4.3 syntaxe abstraite et paragrapheur

les tables contenant la description de la syntaxe abstraite sont générées par le constructeur à partir de spécifications écrites dans le langage DATA (Annexe 2).
Les données sont les suivantes :

- noms et code des structures prédéfinies (terminaux, opérateurs, listes)
- descriptions des phyla (classes)
- définitions des synonymes
- description des schémas prédéfinis des terminaux et opérateurs
- description des schémas pour les listes et synonymes de liste
- liste des symboles devant être précédés ou suivis d'un blanc

remarque: les axiomes ne sont pas spécifiés explicitement, mais déduits par le générateur : ce sont les structures ayant un nom correspondant aux pseudo-terminaux de la grammaire.

Chaque description peut être suivie d'une liste de directives, que nous allons préciser .

directives générales

RC	(retour)	: provoque un retour ligne après la décompilation de la structure
ll	(ligne)	: la structure doit commencer sur une nouvelle ligne

B	(bloc)	: la structure tient seule sur un ensemble de lignes (équivalant aux deux directives précédantes)
LB		: impression d'une ligne blanche avant la décompilation de la structure
J	(indente)	: la marge courante est indentée au début de la décompilation de la structure et restituée à la fin
P	(positionne)	: la marge est positionnée à la colonne courante
M	(marqueur)	: positionne une marge de coupure à la colonne courante
W	(poids)	: la profondeur augmente de 1 en entrant dans la structure
R(n)	(priorité)	: 'n' est la priorité de l'opérateur; ne concerne que les structures intervenant dans les expressions

directives spécifiques des listes

LV	(liste verticale)	: les éléments de le listes doivent être alignés verticalement
LH	(liste horizontale)	: les éléments de la listes sont sur la même ligne
I	(indentera)	: les éléments sont décalés par rapport aux délimiteurs (équivalant à J, s'il n'y a pas de délimiteurs)
D	(délimiteurs)	: les délimiteurs doivent être imprimés même si la liste est vide
H		: le premier élément est sur la même ligne que ce qui précède
A	(aligné)	: provoque l'alignement du premier symbole rencontré dans le schéma des éléments de liste

Avec ces directives, on a un certain nombre de modèles de paragraphage mettant en évidence l'imbrication des parties de programme; pour les listes le nombre de modèles est sans doute insuffisant: on augmenterait les possiblités de spécification, en utilisant pour celles-ci un schéma de décompilation avec deux éléments, de la forme :

```
lstat = begin $stat1 #; %stat2 end
```

'stat1' représentant le premier élément et 'stat2' les suivants.

Par ailleurs, nous n'avons pas de directives correspondant à un paragraphage orienté texte (tabulation, positionnement en colonne n, nouvelle page); de telles directives pourraient être ajoutées, mais il serait plus simple de les placer directement dans les schémas de décompilation, comme dans [TEI81].

4.3 Vérifications des données

Le générateur Paria effectue certaines vérifications au niveau de la description des données :

- unicité des déclarations de noeud
- existence des déclarations de nom de noeuds rencontrés dans les schémas
- existence d'un schéma de décompilation pour les opérateurs et les listes
- utilisation, dans les schémas, de symboles et mots réservés appartenant bien au langage (table commune avec l'analyseur). si le symbole n'est pas connu, un message l'indiquera, mais le symbole sera conservé , pour permettre des traductions d'un langage dans un autre.

Les spécifications du constructeur d'arbre et de l'éditeur étant faites, par l'usager, de façon indépendante, la cohérence entre les arbres construits et leur description abstraite n'est pas assurée. On peut évidemment écrire des vérificateurs de cohérence a posteriori; mais l'écriture et la mise au point des spécifications , pour un langage de la taille d'un langage de programmation, resterait un travail assez long et fastidieux. Une solution, en cours de réalisation [LEB83] est de disposer d'un outil permettant de générer, tout ou partie des spécifications de la syntaxe abstraite, à partir de la grammaire et des spécifications de construction d'arbre.

5.Conclusions

L'éditeur Minerve a déja été générê et utilisê pour les langages suivants :

- Pascal
- Legos
- Pascal-s (Pascal étendu défini par la SEMS)
- LTR ([YOU82])
- Spectre (langage de spécification défini par l'Universite de Toulouse)

L'experience a montrê que les spécifications de Minerve sont suffisantes et adaptées à une famille de langage assez large, et permettent d'obtenir facilement et rapidement un outil d'aide à la programmation et à l'apprentissage du langage. Toutefois, pour effectuer des transformations sophistiquées de programmes, ou pour des langages complexes, Minerve offre moins de possibilitê que Mentor, dont le langage de spécification (Metal) est plus complet.

Cependant, compte tenu des objectifs visés, nous n'envisageons pas d'enrichir les spécifications actuelles, mais plutot d'aller dans le sens d'une plus grande simplicitê de l'interface usager et d'une automatisation de l'écriture des spécifications.

ANNEXES

Nous donnons en Annexe 1, les spécifications de syntaxe abstraite et de paragraphage pour le langage Pascal.
En Annexe 2, on trouve la grammaire de Pascal et les actions de constructions d'arbre; ces deux annexes correspondent aux spécifications complètes pour Minerve-pascal.
En Annexe 3, nous donnons la grammaire du langage de spécification DATA.

ANNEXE 1

syntaxe abstraite et directives pour pascal

```
<<
 (* LES NOMS ENTRE '%' SONT LES VRAIS NOMS DES STRUCTURES *)

CONSTRES
      NILL=1;%NIL%

CONSTTERM
      IDENT    = 2 ;
      META     = 8 ;
      INTCST   = 3 ;
      ALPHA    = 5 ;%STRING%
      HEXCST   = 6 ;
      REALCST  = 4 ;%REAL%
      DEFID    = 31 ;

CONSTOP
      NOTT     = 69;%NOT%
      UPLUS    = 62;
      UMINUS   = 63;
      UNREF    = 35;
      HEXF     = 73;
      FUNCPAR  = 26;
      VARPAR   = 25;
      REF      = 21;
      PACKEDT  = 17;%PACKED%
      FILET    = 20;%FILE%
      SETT     = 19;%SET%
      PROCPAR  = 27;
      VALPAR   = 28;%LOCPAR%
      GOTOT    = 43;%GOTO%
      EQLC     = 12;%DCLCST%
      EQLT     = 13;%DCLTYP%
      EQLV     = 32;%INIT%
      RANGE    = 16;
      EQU      = 72;
      LSS      = 54;
      GTR      = 57;
      NEQ      = 53;
      LEQ      = 55;
      GEQ      = 56;
      INT      = 58;%IN%
      INTDIV   = 66;
      MODT     = 67;%MOD%
```

```
      DIVT     = 65;%DIV%
      MULT     = 64;
      ORT      = 61;%OR%
      ANDT     = 68;%AND%
      PLUS     = 59;
      MINUS    = 60;
      INDEX    = 33;
      DOT      = 34;
      FORMAT   = 39;
      RANGEXP  = 37;
      CASERC   = 24;
      ARRAYT   = 18;%ARRAY%
      DECTAG   = 22;
      COLONRC  = 23;%COLRC%
      DCLVAR   = 14;
      UPSTEP   = 51;%TO%
      DWNSTEP  = 52;%DOWN%
      REPEATT  = 47;%REPEAT%
      ASS      = 41;
      CALL     = 42;
      FCALL    = 36;
      CASEST   = 44;%CASE%
      COL      = 50 ;
      WHILET   = 46 ;%WHILE%
      WITHT    = 49;%WITH%
      LABST    = 40;
      TIMES    = 38;
      PROG     = 10;
      PROC     = 15;
      PROCHD   = 29;
      HDPROG   = 11;
      FORST    = 48;%FOR%
      IFST     = 45;%IF%
      FUNCHD   = 30;
      PCST     = 70;
      MCST     = 71;
      DEFPART  = 74;
      RECORD   = 75;
      BLOCK    = 76;

CONSTLIST
      CMLAB    = 101;%LLAB%
      CMEXP    = 114;%LEXP%
      CMIDENT  = 100;%LIDENT%
      CMCONST  = 108;%LCST%
      CMDEFID  = 113;%LDEFID%
      SMDECL   = 109;%LVDCL%
```

```
    SMFIELD = 105;%LFIELD%
    SMCLNRC = 107;%LRCCOL%
    SMSTAT  = 112;%LSTAT%
    SMCOLON = 117;%LCOL%
    CMVAR   = 118;%LVRBL%
    SMPARAM = 111;%LPARAM%
    SMEQLC  = 102;%LCDCL%
    SMEQLT  = 103;%LTDCL%
    LIX     = 104;
    SMLEQLV = 106;%LINIT%
    LEXP1   = 115;%LARG%
    LEXP2   =116;%LELEM%
    LPROC   =110;
    LVAL    = 119;
    LSYMB   =120;

CLASSES

 STAT     :: REPEAT ASS CALL CASE WHILE LABST FOR IF WITH
             FCALL  GOTO LSTAT                                  %%
 SPLTYP   :: IDENT LSYMB RANGE                                  %NV%
 STRCTYP  :: ARRAY RECORD SET FILE                              %NV%
 TYP      :: SPLTYP STRCTYP PACKED REF                          %NV%
 UNCST    :: NIL STRING INTCST REAL HEXCST                      %%
 VARBL    :: IDENT INDEX UNREF DOT                              %NV%
 LOCAL    :: LIDENT VARPAR                                      %NV%
 PARAM    :: LOCPAR FUNCPAR PROCPAR                             %%
 EXP      :: EQU NEQ LSS LEQ GEQ GTR IN PLUS MINUS OR UPLUS
             UMINUS DIV INTDIV MOD AND MULT NOT LELEM FCALL
             UNCST VARBL                                        %NV%
 STEP     :: TO   DOWN                                          %NV%
 CST      :: IDENT INTCST HEXCST REAL NIL PCST MCST             %NV%
 ELEM     :: EXP RANGEXP                                        %%
 VALU     :: IDENT UNCST LVAL                                   %NV%
 VAL      :: IDENT UNCST TIMES                                  %%
 ARG      :: EXP FORMAT                                         %%
 TITLE    :: PROCHD FUNCHD                                      %NV%
 BODY     :: BLOCK IDENT                                        %NV%
 CASETG   :: IDENT DECTAG                                       %NV%
 REFID    :: IDENT DEFID                                        %NV%


SYNONYMES

(*  SYNONYMES INTRODUITS POUR DIFFERENCIER 2 META-VARIABLES *)
```

```
    DEFPART = $LLAB  $LCDCL $LTDCL                %LL%
    HDPROG  = PROGRAM  $REFID  $LEXTERN  ;  % NV,W %
    BLOCK   = $DEFPART $LVDCL $LPROC $PCSTAT   %LL%


(*DECOMPILATION DES LISTES        *)

LISTES
    LFIELD =  $FIELD  ... ;          %B,W,J,LV%
    LFIELD1=  $FIELD  ... ;          %W,LV,H,J%
    LSTAT  = BEGIN  $STAT  END ... ; %B,J,LV,W%
    LEXP   =  $EXP             ... , %LH,NV,W,D%
    LIDENT = $IDENT            ... , %LH,NV%
    LSYMB  = ( $IDENT )        ... , %LH,W,NV%
    LCOL   = $COL              ... ; %B,J,A,W,LV,NV%
    LELEM  = [ $ELEM     ]     ... , %LH,W,D%
    LCST   = $CST              ... , %LH,NV%
    LDEFID = $REFID            ... , %LH,NV%
    LRCCOL = $COLRC            ... ; %LL,J,A,LV,W,NV%
    LIX    = [ $SPLTYP ]       ... , %LH,W,NV%
    LVRBL  = $VARBL            ... , %LH,NV,W%
    LPARAM = ( $PARAM )        ... ; %LH,W%
    LVAL   = ( $VAL )          ... , %LH,W%
    LCDCL  =  CONST $DCLCST    ... ; %LL,LV,A,W,I,J,SF%
    LTDCL  =  TYPE  $DCLTYP    ... ; %LL,LV,W,A,I,J,SF%
    LLAB   =  LABEL $INTCST  ; ... , %LL,W,LH,J%
    LVDCL  =  VAR   $DCLVAR    ... ; %LL,W,LV,A,I,J,SF,H%
    LINIT  =  VALUE $INIT      ... ; %LL,W,LV,A,I,J,SF%
    LPROC  = $PROC             ...   %LL,W,LV,J%
    LARG   = (  $ARG  )        ... , %W,LH%
    LEXP1  =  ( $EXP )         ... , %W,LH%
    LSTAT1  =  $STAT           ... ; % LV,B,J,W%
    LEXTERN = ( $IDENT )       ... , %LH,W%
    PGSTAT  =  BEGIN $STAT END ... ; %LB,LV,LL,J,W,D,N%
    PCSTAT  =  BEGIN $STAT END ... ; %LB,LV,LL,J,W,D,N%


(*ESPACEMENT DES TERMINAUX *)

DEVNONCOLLE
      :  DERNONCOLLE
      :
 ENDATA
```

ANNEXE 2

grammaire et actions de construction d'arbre abstrait pour pascal

```
<AXIOME>        =   _PROG      <PASCAL>                          ;8000
<AXIOME>        =   _TYP       <TYPE>                            ;8000
<AXIOME>        =   _STAT      <STAT-LIST>                       ;8000
<AXIOME>        =   _LCDCL     <CONST-DECL-LIST>                 ;8000
<AXIOME>        =   _LTDCL     <TYPE-DECL-LIST>                  ;8000
<AXIOME>        =   _LVDCL     <VAR-DECL-LIST>                   ;8000
<AXIOME>        =   _LRCCOL    <VARIANT-LIST>                    ;8000
<AXIOME>        =   _LPROC     <PROC-DECL-LIST>                  ;8000
<AXIOME>        =   _LINIT     <VALUE-LIST>                      ;8000
<AXIOME>        =   _LPARAM    <PARAM-SECTION-LIST>              ;8000
<AXIOME>        =   _LEXP      <INDEX-LIST>                      ;8000
<AXIOME>        =  _LDEFID     <LDEFID>                          ;8000
<AXIOME>        =  _LIDENT     <ID-LIST>                         ;8000
<AXIOME>        =  _LLAB       <LABEL-LIST>                      ;8000
<AXIOME>        =  _LFIELD     <LFIELD>                          ;8000
<AXIOME>        =  _CASERC     <VARIANT-PART>                    ;8000
<AXIOME>        =  _LCST       <CASE-LABEL-LIST>                 ;8000
<AXIOME>        =  _LVAL       <L-VAL>                           ;8000
<AXIOME>        =  _LCOL       <CASE-LIST>                       ;8000
<AXIOME>        =  _LVRBL      <VARIABLE-LIST>                   ;8000
<AXIOME>        =  _LIX        <INDEX-TYPE-LIST>                 ;8000
<AXIOME>        =  _LARG       <ARGUMENTS*>                      ;8000
<AXIOME>        =  _LELEM      <ELEMENT-LIST*>                   ;8000
<AXIOME>        =  _DECTAG     <TAG-FIELD>                       ;8000
<AXIOME>        =  _TITLE      <PROC-HEAD>                       ;8000
<AXIOME>        =  _HDPROG     <PROG-HEAD>                       ;8000
<AXIOME>        =  _STEP       <STEP>                            ;8000
<AXIOME>        =  _IDENT      <IDENT>                           ;8000
<AXIOME>        =  _INTCST     <NOMBRE>                          ;8000
<AXIOME>        =  _NIL        <NIL>                             ;8000
<AXIOME>        =  _STRING     <STRING>                          ;8000
<AXIOME>        =  _HEXA       <HEXA>                            ;8000
<AXIOME>        =  _REAL       <REEL>                            ;8000
<PASCAL>        =  <PROG-HEAD> <DEF-PART> <VAR-PART>
                    <VALUE-PART>  <PROC-PART>   <BODY>          ;2010
<NIL>           =          NIL                                  ;1001
<IDENT>         =          %ID                                  ;1002
<NOMBRE>        =          %NATUREL                             ;1003
<REEL>          =          %REEL                                ;1004
<HEXA>          =          %HEXA                                ;1006
<STRING>        =          %STRING                              ;1005
<IDENT>         =          %META                                ;1008
```

```
<EXTERNES>          =                                                    ;5100
<EXTERNES>          =       (  <ID-LIST>  )                              ;7000
<ID-LIST>           =       <IDENT>                                      ;3100
<ID-LIST>           =       <ID-LIST>  , <IDENT>                         ;4100
<DEF-PART>          =  <LABEL-PART>  <CONST-PART> <TYPE-PART>            ;2074
<PROG-HEAD>         =    PROGRAM <DEFID>  <EXTERNES>  #;                 ;2011
<BODY>              =       <STAT-PART>                                  ;
<BODY>              =                                                    ;1000
<LABEL-PART>        =                                                    ;5101
<LABEL-PART>        =       LABEL  <LABEL-LIST>  #;                      ;7000
<LABEL-LIST>        =       <LABEL>                                      ;3101
<LABEL-LIST>        =       <LABEL-LIST>  ,  <LABEL>                     ;4101
<LABEL>             =       <NOMBRE>                                     ;
<CONST-PART>        =                                                    ;5102
<CONST-PART>        =        CONST  <CONST-DECL-LIST>                    ;7000
<CONST-DECL-LIST>=          <CONST-DECL>                                 ;3102
<CONST-DECL-LIST>=          <CONST-DECL-LIST>   <CONST-DECL>             ;4102
<CONST-DECL>        =       <IDENT>  =   <CONSTANT>  #;                  ;2012
<CONSTANT>          =       <UNSIGNED-NUMBER>                            ;
<CONSTANT>          =   +   <UNSIGNED-NUMBER>                            ;2070
<CONSTANT>          =   -   <UNSIGNED-NUMBER>                            ;2071
<CONSTANT>          =       <IDENT>                                      ;
<CONSTANT>          =   +   <IDENT>                                      ;2070
<CONSTANT>          =   -   <IDENT>                                      ;2071
<CONSTANT>          =       <NIL>                                        ;
<CONSTANT>          =       <STRING>                                     ;
<UNSIGNED-NUMBER>=          <NOMBRE>                                     ;
<UNSIGNED-NUMBER>=          <REEL>                                       ;
<UNSIGNED-NUMBER>=          <HEXA>                                       ;
<TYPE-PART>         =                                                    ;5103
<TYPE-PART>         =       TYPE  <TYPE-DECL-LIST>                       ;7000
<TYPE-DECL-LIST>    =       <TYPE-DECL>                                  ;3103
<TYPE-DECL-LIST>    =       <TYPE-DECL-LIST>   <TYPE-DECL>               ;4103
<TYPE-DECL>         =       <IDENT>  =   <TYPE>  #;                      ;2013
<TYPE>              =       <SIMPLE-TYPE>                                ;
<TYPE>              =       <STRUCT-TYPE>                                ;
<TYPE>              =       <POINTER-TYPE>                               ;
<SIMPLE-TYPE>       =       <SCALAR-TYPE>                                ;
<SIMPLE-TYPE>       =       <SUBRANGE-TYPE>                              ;
<SIMPLE-TYPE>       =       <IDENT>                                      ;
<SCALAR-TYPE>       =       (  <SYMB-LIST>  )                            ;7000
<SYMB-LIST>         =       <IDENT>                                      ;3120
<SYMB-LIST>         =       <SYMB-LIST>   ,   <IDENT>                    ;4120
<SUBRANGE-TYPE>     =       <CONSTANT>  ..  <CONSTANT>                   ;2016
<STRUCT-TYPE>       =       <UNPACK-TYPE>                                ;
<STRUCT-TYPE>       =       PACKED  <UNPACK-TYPE>                        ;2017
<UNPACK-TYPE>       =       <ARRAY-TYPE>                                 ;
```

```
<UNPACK-TYPE>      =      <RECORD-TYPE>                              ;
<UNPACK-TYPE>      =      <SET-TYPE>                                 ;
<UNPACK-TYPE>      =      <FILE-TYPE>                                ;
<ARRAY-TYPE>       =     ARRAY [ <INDEX-TYPE-LIST> ]
                                 OF  <COMPONENT-TYPE>               ;2018
<INDEX-TYPE-LIST>=       <INDEX-TYPE>                               ;3104
<INDEX-TYPE-LIST>=       <INDEX-TYPE-LIST> ,  <INDEX-TYPE>          ;4104
<INDEX-TYPE>       =     <SIMPLE-TYPE>                              ;
<COMPONENT-TYPE>   =     <TYPE>                                     ;
<RECORD-TYPE>      =     RECORD
                               <LFIELD>  <VARIANT-PART>
                         END                                        ;2075
<LFIELD>           =     <RECORD-SECTION>                           ;3105
<LFIELD>           =     <LFIELD>  #;  <RECORD-SECTION>             ;4105
<RECORD-SECTION>   =     <ID-LIST>  :   <TYPE>                      ;2014
<RECORD-SECTION>   =                                                ;5000
<VARIANT-PART>     =     CASE     <TAG-FIELD>
                             OF  <VARIANT-LIST>                     ;2024
<VARIANT-PART>     =                                                ;5024
<TAG-FIELD>        =    <IDENT> :  <IDENT>                          ;2022
<TAG-FIELD>        =    <IDENT>                                     ;
<VARIANT-LIST>     =     <VARIANT>                                  ;3107
<VARIANT-LIST>     =     <VARIANT-LIST>  #;  <VARIANT>              ;4107
<VARIANT>          =     <CASE-LABEL-LIST>  :
                       ( <LFIELD>   <VARIANT-PART>  )               ;2023
<VARIANT>          =                                                ;
<CASE-LABEL-LIST>=       <CONSTANT>                                 ;3108
<CASE-LABEL-LIST>=       <CASE-LABEL-LIST>  ,  <CONSTANT>           ;4108
<SET-TYPE>         =     SET OF  <SIMPLE-TYPE>                      ;2019
<FILE-TYPE>        =     FILE OF  <TYPE>                            ;2020
<POINTER-TYPE>     =     @ <IDENT>                                  ;2021
<VAR-PART>         =                                                ;5109
<VAR-PART>         =     VAR  <VAR-DECL-LIST>                       ;7000
<VAR-DECL-LIST>    =     <VAR-DECL>                                 ;3109
<VAR-DECL-LIST>    =     <VAR-DECL-LIST>   <VAR-DECL>               ;4109
<VAR-DECL>         =     <LDEFID>  :   <TYPE>  #;                   ;2014
<VALUE-PART>       =                                                ;5106
<VALUE-PART>       =     VALUE   <VALUE-LIST> #;                    ;7000
<VALUE-LIST>       =     <INIT>                                     ;3106
<VALUE-LIST>       =     <VALUE-LIST>  #;  <INIT>                   ;4106
<INIT>             =     <IDENT>  =  <VALEUR>                       ;2032
<VALEUR>           =     <UNSIGNED-CONST1>                          ;
<VALEUR>           =     (  <L-VAL>  )                              ;7000
<L-VAL>            =     <VAL>                                      ;3119
<L-VAL>            =     <L-VAL>  ,  <VAL>                          ;4119
<VAL>              =     <UNSIGNED-CONST1>                          ;
<VAL>              =     <NOMBRE>  *  <UNSIGNED-CONST1>             ;2038
```

```
<ELEMENT-LIST>   =     <ELEMENT-LIST>  ,  <ELEMENT>          ;4116
<ELEMENT>        =     <EXPRESSION>                          ;
<ELEMENT>        =     <EXPRESSION>  ..  <EXPRESSION>        ;2037
<CALL>           =     <IDENT>  <ARGUMENTS*>                 ;2042
<ARGUMENTS*>     =   (   <L-ARGUMENTS>  )                    ;7000
<ARGUMENTS*>     =                                           ;5115
<L-ARGUMENTS>    =     <ACTUALBIS>                           ;3115
<L-ARGUMENTS>    =     <L-ARGUMENTS>  ,  <ACTUALBIS>         ;4115
<ACTUALBIS>      =     <EXP>                                 ;
<ACTUALBIS>      =     <FORMAT>                              ;
<ACTUALBIS>      =     <FORMAT>  :  <EXP>                    ;2039
<FORMAT>         =     <EXPRESSION>  :  <EXP>                ;2039
<EXP>            =     <EXPRESSION>                          ;
<EXP>            =     <HEXF>                                ;
<HEXF>           =     <EXPRESSION>  HEXA                    ;2073
<GOTO-STAT>      =     GOTO  <LABEL>                         ;2043
<CLOSED-STAT>    =     <COMPOUND-STAT>                       ;
<CLOSED-STAT>    =     <WITH-CLOSED>                         ;
<CLOSED-STAT>    =     <CASE-STAT>                           ;
<CLOSED-STAT>    =     <CLOSED-IF>                           ;
<CASE-STAT>      =     CASE <EXPRESSION> OF
                           <CASE-LIST>  END                  ;2044
<CASE-LIST>      =     <CASE-ELT>                            ;3117
<CASE-LIST>      =     <CASE-LIST>  #;       <CASE-ELT>      ;4117
<CASE-ELT>       =     <CASE-LABEL-LIST>  :   <STAT>         ;2050
<CASE-ELT>       =                                           ;5050
<CLOSED-IF>      =     <IF-THEN> <CLOSED-STAT1>
                       ELSE <CLOSED-STAT1>                   ;2045
<IF-THEN>        =     IF  <EXPRESSION>  THEN                ;7000
<CLOSED-STAT>    =     <WHILE-CLOSED>                        ;
<CLOSED-STAT>    =     <REPEAT-STAT>                         ;
<CLOSED-STAT>    =     <FOR-CLOSED>                          ;
<WHILE-CLOSED>   =     <WHILE-DO>  <CLOSED-STAT1>            ;2046
<WHILE-DO>       =     WHILE  <EXPRESSION>  DO               ;7000
<REPEAT-STAT>    =     REPEAT  <STAT-LIST>
                         UNTIL  <EXPRESSION>                 ;2047
<FOR-CLOSED>     =     FOR  <IDENT>  :=  <STEP>  DO
                          <CLOSED-STAT1>                     ;2048
<STEP>           =     <EXPRESSION> TO <EXPRESSION>          ;2051
<STEP>           =     <EXPRESSION> DOWNTO <EXPRESSION>      ;2052
<WITH-CLOSED>    =     <WITH-DO>  <CLOSED-STAT1>             ;2049
<WITH-DO>        =     WITH  <VARIABLE-LIST>  DO             ;7000
<VARIABLE-LIST>  =     <VARIABLE>                            ;3118
<VARIABLE-LIST>  =     <VARIABLE-LIST> , <VARIABLE>          ;4118
<OPEN-STAT>      =     <WHILE-OPEN>                          ;
<OPEN-STAT>      =     <FOR-OPEN>                            ;
<OPEN-STAT>      =     <WITH-OPEN>                           ;
```

```
<OPEN-STAT>       =       <IF-THEN> <CLOSED-STAT1>
                          ELSE  <OPEN-STAT1>                        ;2045
<OPEN-STAT>       =    <IF-THEN> <CLOSED-STAT1> <ELSE-VIDE>         ;2045
<OPEN-STAT>       =    <IF-THEN>  <OPEN-STAT1> <ELSE-VIDE>          ;2045
<ELSE-VIDE>       =                                                 ;5000
<WHILE-OPEN>      =       <WHILE-DO>  <OPEN-STAT1>                  ;2046
<FOR-OPEN>        =       FOR  <IDENT>  :=  <STEP>  DO
                              <OPEN-STAT1>                          ;2048
<WITH-OPEN>       =       <WITH-DO>  <OPEN-STAT1>                   ;2049
$
BEGIN
VAR
TYPE
CONST
$
```

ANNEXE 3

grammaire du langage DATA

```
<AXIOME>      = #<< <DATA>                                                    ;
<DATA>        = <CONSTANTES> <CLASSES> <SYN> <SHEMAS> <LISTES>
                <TERM> ENDATA ;100
<CONSTANTES>= <CONSTRES> <CONSTERM> <CONSTOP> <CONSTLIST>                     ;
<CONSTRES>    = CONSTRES <L-DECLRES>                                          ;
<L-DECLRES>   = <DECLRES>                                                     ;
<L-DECLRES>   = <L-DECLRES> <DECLRES>                                         ;
<DECLRES>     = <IDENT> #= <NOMBRE> #; <COMMENT>                              ;3
<CONSTERM>    = CONSTTERM <L-DECLTERM>                                        ;
<L-DECLTERM>= <DECLTERM>                                                      ;
<L-DECLTERM>= <L-DECLTERM> <DECLTERM>                                         ;
<DECLTERM>    = <IDENT> #= <NOMBRE> #; <COMMENT>                              ;4
<CONSTOP>     = CONSTOP <L-DECLOP>                                            ;
<L-DECLOP>    = <DECLOP>                                                      ;
<L-DECLOP>    = <L-DECLOP> <DECLOP>                                           ;
<DECLOP>      = <IDENT> #= <NOMBRE> #; <COMMENT>                              ;5
<CONSTLIST>   = CONSTLIST <L-DECLLIST>                                        ;
<L-DECLLIST>= <DECLLIST>                                                      ;
<L-DECLLIST>= <L-DECLLIST> <DECLLIST>                                         ;
<DECLLIST>    = <IDENT> #= <NOMBRE> #; <COMMENT>                              ;6
<IDENT>       = %IDENT                                                        ;1
<COMMENT>     =  % %IDENT %                                                   ;7
<COMMENT>     =                                                               ;
<NOMBRE>      = %NOMBRE                                                       ;2
*
*
<SYN>         = SYNONYMES <SYNLIST>                                           ;
<SYN>         =                                                               ;
<SYNLIST>     = <SYNLIST> <SYNO>                                              ;
<SYNLIST>     = <SYNO>                                                        ;
<SYNO>        = <IDENT> :: <RENAME>    % <ATTR> %                            ;9
<RENAME>      = %IDENT                                                        ;8
*
<CLASSES>     = CLASSES <L-CLASSES>                                           ;
<L-CLASSES>   = <DEFCLASSE>                                                   ;
<L-CLASSES>   = <L-CLASSES>  <DEFCLASSE>                                      ;
<DEFCLASSE>   = <IDENT> :: <SETVAL>              % <ATTR> %                    ;10
<SETVAL>      = %IDENT                                                        ;11
<SETVAL>      = <SETVAL>  %IDENT                                              ;12
*
*
<SHEMAS>      = DECOMP <L-SHEMAS>                                             ;
<L-SHEMAS>    =  <SHEMAFIXE>                                                  ;
<L-SHEMAS>    =  <L-SHEMAS> <SHEMAFIXE>                                       ;
```

```
<SHEMAFIXE> = <IDENT>  #=  <SETELEM>  % <ATTR>  %          ;13
<SETELEM>   =  <ELEM>                                     ;14
<SETELEM>   =  <SETELEM> <ELEM>                           ;15
<ELEM>      =  <RESERVE>                                  ;
<ELEM>      =  <SYMBOL>                                   ;
<ELEM>      =  %STRUCT                                    ;26
<ELEM>      =      F  (  %NOMBRE  )                       ;18
<RESERVE>   = %IDENT                                      ;16
<SYMBOL>    = #=                                          ;17
<SYMBOL>    =  #;                                         ;17
<SYMBOL>    =  #<                                         ;17
<SYMBOL>    =   :                                         ;17
<SYMBOL>    =  >                                          ;17
<SYMBOL>    =  ,                                          ;17
<SYMBOL>    =  .                                          ;17
<SYMBOL>    =  [                                          ;17
<SYMBOL>    =  ]                                          ;17
<SYMBOL>    =  (                                          ;17
<SYMBOL>    =  )                                          ;17
<SYMBOL>    =  @                                          ;17
<SYMBOL>    =  +                                          ;17
<SYMBOL>    =  -                                          ;17
<SYMBOL>    =  *                                          ;17
<SYMBOL>    =  /                                          ;17
<SYMBOL>    = :=                                          ;17
<SYMBOL>    =  /=                                         ;17
<SYMBOL>    =  **                                         ;17
<SYMBOL>    =  #<<                                        ;17
<SYMBOL>    =  #<>                                        ;17
<SYMBOL>    =  >>                                         ;17
<SYMBOL>    =  =>                                         ;17
<SYMBOL>    =  ->                                         ;17
<SYMBOL>    =  #$                                         ;17
<SYMBOL>    =  '                                          ;17
<SYMBOL>    =  "                                          ;17
<SYMBOL>    =  ^                                          ;17
<SYMBOL>    =  &                                          ;17
<SYMBOL>    = #<=                                         ;17
<SYMBOL>    =  >=                                         ;17
<SYMBOL>    =  ..                                         ;17
<SYMBOL>    =  ##                                         ;17
<SYMBOL>    =                                             ;17
<SYMBOL>    =  ?                                          ;17
*
*
<LISTES>     = LISTES <L-DECLISTE>                        ;
<L-DECLISTE>= <DECLISTE>                                  ;
```

```
<L-DECLISTE>=  <L-DECLISTE> <DECLISTE>                                  ;
<DECLISTE>  = <IDENT> #= <DEBUT> <NATELEM> <FIN> <SEP>
                % <ATTR> %                                             ;19
<DEBUT>     =   <RESERVE>                                              ;20
<DEBUT>     =   <SYMBOL>                                               ;20
<DEBUT>     =                                                          ;20
<FIN>       =  <RESERVE>                                               ;21
<FIN>       =   <SYMBOL>                                               ;21
<FIN>       =                                                          ;21
<SEP>       =   ...                                                    ;47
<SEP>       =   ...    <RESERVE>                                       ;23
<SEP>       =   ...    <SYMBOL>                                        ;23
<NATELEM>   =   %STRUCT                                               ;22
*
*
<TERM>      = <NON-COLLE-DEVANT> <NON-COLLE-DERR>                      ;
<NON-COLLE-DEVANT> = DEVNONCOLLE <L-SYMB>                              ;24
<NON-COLLE-DEVANT> =                                                   ;
<NON-COLLE-DERR>   = DERNONCOLLE  <L-SYMB>                             ;25
<NON-COLLE-DERR>   =                                                   ;
<L-SYMB>    =   <SYMBOL>                                               ;27
<L-SYMB>    =   <L-SYMB> <SYMBOL>                                      ;27
*
*
<ATTR>      =                                                          ;
<ATTR>      = <L-DIR>                                                  ;
<L-DIR>     = <DIR>                                                    ;
<L-DIR>     =  <L-DIR> , <DIR>                                         ;
<DIR>       =  B                                                       ;28
<DIR>       = LL                                                       ;29
<DIR>       = RC                                                       ;30
<DIR>       = LB                                                       ;31
<DIR>       = I                                                        ;32
<DIR>       = E                                                        ;33
<DIR>       = P                                                        ;34
<DIR>       = W                                                        ;35
<DIR>       = N                                                        ;36
<DIR>       = LV                                                       ;37
<DIR>       = LH                                                       ;38
<DIR>       = LU                                                       ;39
<DIR>       = J                                                        ;40
<DIR>       = H                                                        ;41
<DIR>       = D                                                        ;42
<DIR>       = A                                                        ;44
<DIR>       = M                                                        ;45
<DIR>       = R ( %NOMBRE )                                            ;49
<DIR>       = NV                                                       ;46
```

Références

ACM'81 Proceedings of the ACM SIGPLAN SIGOA symposium on text Manipulation, Portland, Oregon (USA), june 8-10, 1981, vol.16 no.6.

BOU 80 P.Boullier, "Generation automatique d'analyseurs syntaxiques, avec rattrapage d'erreurs", Journees francophones sur la production assistee de logiciel, Geneve, janvier 1980.

BUR 80 H.Burkhart, J.Nievergelt, "structure-oriented editors", in [ACM81] p.164-181.

COU 78 A.M.Couhault-Vercoustre, "vers une construction automatique d"editeur syntaxique ", Rapport Iria-Sesori, septembre 1978.

DON 82 V.Donzeau-Gouge, "Les raisons des choix dans la définition formelle du langage Ada", thèse d'etat, Paris VII, juillet 82.

DON 75 V.Donzeau-Gouge, G.Huet, G.Kahn, B.Lang, J.J.Levy, "A Structured Oriented Program Editor", Proceedings in the International Computing Symposium, North Holland Publishing Company 1975.

DON 79 V.Donzeau-Gouge, G.Huet, G.Kahn, B.Lang, "Introduction au systeme Mentor et à ses aplications", Journees francophones sur la certification du logiciel. Genève, Janvier 79.

DON 83 V.Donzeau-Gouge, G.Kahn, B.Lang, B.Mélèse, "Outline of a tool for document manipulation", submitted at the 9th World Computer Congress, version préliminaire, communication personnelle.

LEB 81 E.R.Lebon,
"réalisation d'un paragrapheur pour Syntax",
rapport de DEA, Paris VI ,septembre 81.

LEB 83 E.R.Lebon,
"réalisation d'un décompilateur d'arbre
abstrait spécifié à l'aide de la
grammaire paragraphée"
thèse de 3ème cycle, a paraitre .

LEG 80 J.L Bouchenez,M.Loyer,P.Maurice,F.Prusker,
J.C.Sogno,A.M.Vercoustre,
"le système Legos, environnement de
programmation sur Mitra 225 "
Journées Bigre, Rennes 1980.

MEL 81 B.Mélèse,
"Mentor,L'environnement Pascal",
I.N.R.I.A,Rapport technique no.5, Octobre 1981.

Mel 82 B.Mélèse,
"Métal, un langage de spécification pour le
système Mentor",
A paraitre dans TSI, Technique et Science
de l'informatique (AFCET),octobre 1982.

LUC 79 L.Lucrèce,
"aide au développement de logiciel.
Une réalisation pour le langage Pascal"
Thèse de 3ème cycle - UPS/Toulouse 79.

LUC 82 L.Lucrèce, P.Maurice, A.M.Vercoustre,
"Minerve,manuel de référence ",
I.N.R.I.A, Rapport technique no.17, sept 1982.

SYN 82 "le système Syntax:manuel d'utilisation
et de mise en oeuvre",
disponible dans la documentation MULTICS
de l'INRIA.

TEI 81 T.Teitelbaum, T.H Reps,
"The Cornell program synthesizer :
a syntax directed programming environnement",
CACM - vol.24, no.9 - septembre 1981.

WOO 81 Steven R.WOOD,
"Z-the 95% program editor,
in [ACM81].

YOU 82 A.Ben Youssef Naouar,
"réalisation d'un editeur de programmes LTR ",
mémoire de DEA, UPS-Toulouse 1982.

Imprimé en France

par

l'Institut National de Recherche en Informatique et en Automatique