Minerve : Manuel de référence

Anne-Marie Vercoustre, Pierre Maurice, Luc Lucrèce

INRIA

CENTRE DE ROCQUENCOURT

Rapports Techniques

N° 17

MINERVE

MANUEL DE RÉFÉRENCE

Luc LUCRÈCE
Pierre MAURICE
Anne-Marie VERCOUSTRE

Institut National
de Recherche
en Informatique
et en Automatique

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

Septembre 1982

M I N E R V E

Manuel de référence

Luc LUCRECE*, Pierre MAURICE*, Anne-Marie VERCOUSTRE**

* INRIA, Toulouse.

** INRIA, Rocquencourt.

Résumé :

MINERVE est un méta-éditeur syntaxique, inspiré de MENTOR. Pour un langage de programmation X donné, un éditeur Minerve-X est généré automatiquement à partir de la grammaire du langage et de sa syntaxe abstraite.

Ce rapport présente les principes de fonctionnement de Minerve ainsi qu'une description complète des commandes de l'éditeur.

Abstract :

MINERVE is a generic MENTOR-like syntactic editor. For a given programming language X, a Minerve-X editor is automatic generated from the grammar and the abstract syntax.

This report gives the main characteristics of Minerve and a complete description of the user commands.

INTRODUCTION

Le méta-éditeur MINERVE a été défini et réalisé dans le cadre du projet LEGOS, dont le but est la réalisation d'un environnement de programmation sous le système MMT2 de la SEMS.

Le projet, financé par l'ADI, est mené en collaboration avec la SEMS et l'Université Paul Sabatier de Toulouse.

MINERVE est un méta-éditeur de programmes, en ce sens que les fonctions d'édition sont indépendantes du langage d'écriture des programmes manipulés ; en outre, l'utilisation du générateur d'analyseurs syntaxiques SYNTAX a permis de paramétrer MINERVE par le langage à éditer et de créer un méta-paragrapheur PARIA, utilisé par l'éditeur.

Largement inspiré de MENTOR, MINERVE se veut toutefois moins puissant, mais surtout adapté à de petites configurations matérielles ; les techniques de réalisation sont différentes dans les deux outils.

Ecrit en Pascal, MINERVE fonctionne actuellement sur IRIS-80 et Mitra/MMT2 ; son transport est aisé, sur tout matériel disposant d'un compilateur Pascal.

Les outils de génération de MINERVE pour un langage donné sont disponibles sous CII-HB 68/Multics et CII-HB/IRIS-80.

I - NOTIONS GENERALES - DEFINITIONS

I.1 - Editeur syntaxique

Un éditeur de textes est un outil permettant la création, et la mise à jour de textes quelconques.

Un éditeur syntaxique a de plus la connaissance de la syntaxe du langage dont il édite les textes. Lorsqu'il s'agit de langages de programmation, c'est un outil permettant la création et la mise au point de programmes (ou d'éléments de programmes).

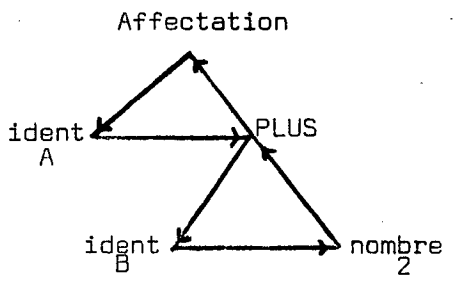
Dans toutes les manipulations qu'il effectue, cet éditeur vérifie que la syntaxe du langage est respectée, en rejetant les commandes, lorsque cette règle n'est pas vérifiée ; plus précisément, l'éditeur effectue une analyse syntaxique de tout programme ou partie de programme qui lui est soumis et ne l'accepte que s'il n'y a pas d'erreur syntaxique ou qu'il est lui-même capable de corriger l'erreur rencontrée (récupération de certains types d'erreurs).

I.2 - Représentation interne des programmes

Après analyse syntaxique, les programmes (ou éléments de programmes) sont conservés sous forme d'arbres par l'éditeur Minerve.

C'est sur cette représentation, dite abstraite, (R.A.) que l'utilisateur va pouvoir effectuer les manipulations qu'il désire.

Exemple : l'instruction : (en Pascal) A := B+2
est représentée par la construction arborescente suivante :



Cette représentation permet de manipuler un programme comme un objet structuré, constitué d'un ensemble d'entités syntaxiques.

Il faut remarquer que l'utilisateur n'a pas à connaître parfaitement la forme de cet arbre interne, car

a) l'éditeur peut lui fournir certains renseignements
b) l'utilisateur peut désigner une partie de programme en nommant l'entité syntaxique qu'il désire.

Exemple : chercher l'ident A
chercher une affectation dans le programme

L'ensemble des entités syntaxiques manipulables dans l'éditeur constitue la syntaxe abstraite associée au langage.

I.3 - Syntaxe abstraite

La construction des arbres précédents est faite, par Minerve, lors de la phase d'analyse syntaxique. La nature et la forme des arborescences à construire sont définies par l'ensemble des règles de syntaxe abstraite caractérisant le langage*.

Exemple : la syntaxe abstraite d'une affectation est la suivante :

Affectation = ⧺VARBL := ⧺EXP

c'est-à-dire qu'un noeud de type affectation est un noeud ayant 2 fils, respectivement de type VARBL (variable) et EXP (expression).

La syntaxe abstraite est définie par :

- un ensemble de terminaux
- un ensemble d'opérateurs
- un ensemble de listes
- un ensemble de classes

---

* C'est à partir de la syntaxe abstraite d'un langage X que des tables sont générées pour Minerve, créant ainsi un éditeur syntaxique (Minerve X) pour le langage considéré.

I.3.1 - Les terminaux

Ils correspondent aux feuilles de toutes les arborescences possibles :

Exemple : %IDENT
%NOMBRE

I.3.2 - Les opérateurs

Ils correspondent aux structures ayant un nombre fixe de sous-structures, de type déterminé.

Exemple : En Pascal, on peut définir la syntaxe abstraite d'un programme comme suit :

PROG = <PROGHEAD><DEFPART> LDCLV LINIT LPROC & PGSTAT

La structure "programme" est ainsi définie comme ayant 6 fils qui sont les sous-structures suivantes :

- PROGHEAD : en-tête du programme
- DEFPART : partie définition (définitions de label, de constante, de type)
- LDCLV : liste de déclarations de variables
- LINIT : initialisation des variables (partie VALUE)
- PGSTAT : corps du programme

I.3.3 - Les listes

Contrairement aux opérateurs, les structures de type liste ont un nombre indéterminé de fils, mais ces fils sont tous de même type.

Exemple : LDCLV = <DCLV> ...

indique que la structure LDCLV est une liste d'éléments de type DCLV (déclaration de variable).

I.3.4 - Les classes

Ce sont des ensembles, non vides, de structures de type terminaux, opérateurs ou listes. Dans une règle de syntaxe abstraite (donc dans la structure correspondante) l'occurrence d'une classe indique quelles sont les sous-structures qui peuvent se trouver à cet emplacement.

Exemple : La classe STAT en Pascal est définie par :

```
STAT :: LSTAT   IF   ASS   CALL   FOR   WHILE
        REPEAT  LABEL_STAT  GOTO  WITH
```

si STAT apparaît dans une règle de syntaxe abstraite cela signifie que dans la structure correspondante (R.A.) ne pourra apparaitre qu'une des sous-structures apparaissant à droite du symbole "::".

Les noms de classes ne correspondent donc pas à des noms de structures pparaissant dans les arbres ou sous-arbres, mais peuvent être utilisés comme les noms des autres structures dans les expressions de recherche.

I.4 - Schémas prédéfinis

I.4.1 - Schémas de décompilation

La connaissance de la syntaxe abstraite est nécessaire à l'utilisateur pour effectuer de manière efficace des manipulations de programmes. L'éditeur fournit une aide pour l'apprentissage de cette syntaxe : ce sont les schémas de décompilation des structures prédéfinies.

Les schémas de décompilation établissent la correspondance entre une structure prédéfinie (de type opérateur ou liste) et une forme de texte source correspondante.

Exemple : IF = `if ⧼EXP then ⧼STAT else ⧼STAT`

est le schéma de décompilation de la structure IF, où ⧼EXP, ⧼STAT, ⧼TAT sont des noms de classes remplaçants les parties de texte associées aux sous-structures correspondantes.

Les schémas de décompilation des structures prédéfinies sont appelés schémas prédéfinis ; ils pourront être utilisés, comme les schémas utilisateurs, comme modèles de recherche (cf. II.).

Les schémas prédéfinis sont le lien entre la syntaxe concrète et la syntaxe abstraite.

Les schémas de décompilation associés aux structures de listes ont une syntaxe mettant en évidence leur caractère spécifique.

Exemple : LSTAT = `begin &STAT end ... ;`

Ce schéma indique que LSTAT est une structure de type liste (indiqué par ...) correspondant à une partie de programme (Pascal) comprise entre 'begin' et 'end' ; les éléments (instructions) étant séparés par des ';'.

I.4.2 - 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 prédéfini du IF peut être défini comme suit :

IF `if &COND then &TRUE_PART else &FALSE_PART`

où : &COND est synonyme de la classe EXP
&TRUE_PART est synonyme de la classe STAT
&FALSE_PART est synonyme de la classe STAT.

L'introduction des synonymes TRUE_PART et FALSE_PART permet de nommer de façon distincte les deux occurrences de structure STAT apparaissant dans un IF ; de même l'utilisation du synonyme COND permet de distinguer l'EXPRESSION apparaissant dans IF de celle apparaissant, par exemple, dans une AFFECTATION.

La recherche d'un synonyme dans une arborescence, contient, implicitement, la précision du contexte dans lequel on veut le trouver.

I.4.3 - Ecriture des schémas prédéfinis

Certaines commandes (PS,TY) permettent d'obtenir des renseignements sur la syntaxe abstraite et les schémas des structures prédéfinies. Pour faciliter l'apprentissage des différents types de structures manipulées, les noms de structures sont imprimées, dans les schémas de décompilation, avec un symbole spécifique :

- les terminaux sont précédés du caractère %
- les opérateurs sont écrits entre < >
- les listes sont précédées du préfixe L_
- les classes sont préfixées par $
- les synonymes sont préfixés par &.

Exemple : IF = IF &COND then &TRUE_PART else &FALSE_PART
LSTAT = begin $STAT end ... ;
PROGHEAD = program %ident &LEXTERN ;

Pour un synonyme, l'éditeur donne comme renseignements :

- le nom de la structure dont il est synonyme
- le schéma de décompilation associé, s'il en a un spécifique (seulement possible pour un synonyme de liste).

Exemple : LEXTERN == L_LIDENT (%IDENT) ... , %synonyme de LIDENT%

II - MANIPULATIONS ELEMENTAIRES

Le but d'un éditeur syntaxique est de permettre la création, la mise au point et la transformation de programmes en respectant, à chaque étape, la syntaxe du langage de programmation.

Un programme sera donc manipulé, non comme un simple texte, mais comme un ensemble d'objets structurés sur lesquels on peut effectuer certaines opérations.

II.1 - Les Commandes

Les commandes sont l'expression des désirs et interrogations de l'utilisateur, vis à vis du système Minerve.

Elles permettent d'agir sur les objets connus de ce dernier et d'effectuer certaines manipulations.

Les commandes sont analysées par Minerve, puis interprétées une à une, tant qu'aucune anomalie n'est rencontrée.

II.1.1 - Programme éditeur

Un programme éditeur est une suite de commandes séparée par ";" et terminées par le caractère "fin de ligne".

Généralement un programme éditeur est interprété ligne à ligne. Cependant des commandes itératives permettent son extension sur plusieurs lignes.

Un programme peut être constitué d'une seule commande .

II.1.2 - Exécution des commandes

Les commandes d'un programme sont exécutées l'une après l'autre, dans l'ordre ou elles ont été écrites.

Toute anomalie est signalée par un message. Dans ce cas, toute l'exécution est interrompue : les commandes restantes sont perdues.

Le contrôle est rendu à l'utilisateur ("?").

Remarque : Le rang de la commande qui échoue est indiqué dans le message d'erreur.

II.1.3 - Commandes

Elles sont de quatre types :

a) Commandes d'affectations de pointeurs
b) Appels de procédures prédéfinies
c) Commandes itératives
d) Commandes externes

et elles peuvent agir

- soit à l'intérieur de MINERVE (a,b,c)
- soit sur un contexte englobant le système (d)

II.2 - Objets manipulés

On désigne ainsi l'ensemble des entités reconnues et traitées par l'éditeur MINERVE et qui sont (sous certaines réserves) accessibles à l'utilisateur. Ce sont les objets manipulés par le langage de l'éditeur.

Ils sont répartis suivant les six types suivants :

1) Espaces de travail,
2) Fichiers,
3) Structures,
4) Repères,
5) Textes,
6) Expressions.

II.2.1 - Espaces de travail

Au cours d'une session de travail, MINERVE met à la disposition de l'usager une zone dans laquelle il va pouvoir effectuer ses manipulations, sur des éléments de programmes (sur des programmes).

Tous les objets crées sont répertoriés et conservés dans cet espace de travail (Working Space).

Lorsqu'il termine ou interrompt une session de travail, l'utilisateur peut conserver l'intégralité du WS ; ceci lui permettra ultérieurement de se replacer dans le même contexte de travail (sauvegarde de tous les pointeurs).

Un WS est désigné par un identificateur -la liste des WS peut être fournie par le système-. Un WS sauvegardé est donc accessible tant qu'il n'est pas détruit explicitement par commande.

Si l'usager n'a pas désigné explicitement un WS, le système sauvegarde les travaux de la session dans un WS standard nommé ENCAS.

Les commandes agissant sur les WS sont préfixées par le caractère ">" (cf. III).

Exemple : la commande : `> CWS(TOTO)` permet de créer un WS de nom TOTO.

II.2.2 - Les Fichiers

Pour permettre la liaison avec un environnement plus général que lui-même, MINERVE offre la possibilité de créer des fichiers externes et d'y accéder.

Ces fichiers contiennent des programmes, sous forme de texte ou sous forme de structures ; ils peuvent avoir été créés par MINERVE ou par un système extérieur.

Les fichiers sont désignés par des identificateurs, mais contrairement aux WS, aucun répertoire de fichiers n'est conservé par MINERVE.

II.2.3 - Les Structures

Comme nous l'avons vu au Chapitre I, un programme utilisateur est construit par l'éditeur comme un objet structuré en sous-structures correspondant à des entités syntaxiques. Celles-ci, définies par la syntaxe abstraite, sont appelées structures prédéfinies du langage considéré.

Ce sont les noms et les éléments de ces structures qui servent à désigner et manipuler les parties de programmes.

Exemple : l'instruction PASCAL

```
A[i] := x+2
```  (1)

correspond à la structure prédéfinie 'assignment-statement' (ASS) dont le schéma est

```
ASS → ØVARBL := ØEXP
```  (2)

L'utilisateur pourra manipuler l'instruction (1) comme une structure composée de deux éléments :

- une variable (désignée par VARBL)
- une expression (EXP).

L'expression x+2 de cette instruction pourra être accédée

- comme la partie EXP de la structure ASS
- comme le 2ème fils du noeud ASS
- ou de façon plus spécifique, par le nom de la structure particulière rencontrée (et appartenant à la classe EXPRESSION) : ici la structure PLUS.

A l'intérieur de chaque espace de travail (WS), des morceaux de programmes, sous forme de structures, peuvent être crées, manipulés, modifiés, sauvegardés ou détruits par des commandes de l'éditeur.

II.2.4 - Les Repères

Les structures prédéfinies sont désignées par des identificateurs

Exemple : ASS, VARBL, EXP, PLUS.

De même les structures de l'utilisateur (programmes ou éléments de programmes) sont repérées par des noms de pointeurs.

Le nom d'un pointeur est déclaré implicitement par l'utilisateur par une commande d'affectation de pointeur (commande ':', cf. III).

Les noms de pointeurs sont sauvegardés dans le WS courant ; ils doivent être différents des noms des structures prédéfinies.

Il existe des pointeurs prédéfinis, dont les noms sont aussi protégés, et qui ne peuvent être détruits par l'usager.

Exemple : Le pointeur K est le pointeur courant :

- il est implicitement déplacé par certaines commandes,
- il peut être omis dans certaines commandes, qui sont ainsi abrégées (pointeur par défaut).

La liste des noms de pointeurs créés dans un WS, (avec le type d'objets qu'ils repèrent) est obtenue par la commande : [>NAMES].

II.2.5 - Textes

Outre les structures qui sont des éléments de programmes analysés, MINERVE permet aussi de manipuler des chaînes de caractères.

Ces chaînes de caractères, ou textes, peuvent être affectées à des variables dîtes de type texte

Exemple : T := 'UNTEXTE'.

Les variables de type texte sont créées, conservées, détruites de manière analogue aux pointeurs sur des structures.

Des opérateurs permettent de manipuler les textes

Exemple : Concaténation.

Certaines directives, valables pour les structures, le sont aussi pour les variables de type texte (cf. Commandes).

II.2.6 - Expressions

Les déplacements et les recherches dans une structure sont exprimés à l'aide des repères et de plusieurs opérateurs fournis par l'éditeur (cf. Opérateurs).

Les combinaisons de repères et d'opérateurs déterminent les expressions du langage de l'éditeur.

L'écriture d'une expression est, pour l'utilisateur, le moyen de désigner à MINERVE le noeud d'une structure sur lequel (ou a partir duquel) il veut opérer une manipulation.

Le résultat de l'évaluation d'une expression est similaire à une adresse, désignant un noeud d'une structure.

Si cette adresse est vide, la commande ne sera pas exécutée et un message est émis.

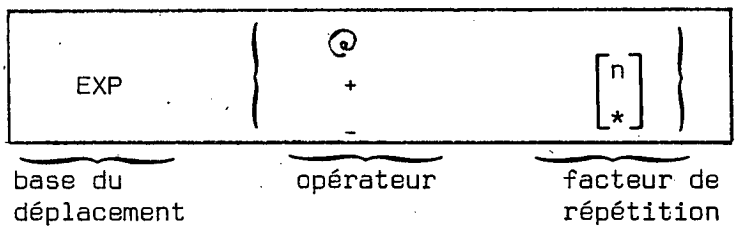
II.3 - Les Opérateurs

Les opérateurs correspondent à des commandes élémentaires de manipulation des objets décrits ci-dessus.

II.3.1 - Opérateurs de déplacement

Il s'agit des opérateurs permettant le déplacement dans une structure, considérée comme un arbre.

La syntaxe est la suivante :



- La base

EXP est une expression du langage de commande permettant de calculer une adresse.

Cette adresse désigne, dans une structure, le noeud sur lequel va agir l'opérateur.

Remarques : 1) - Si le repère initial est K il peut être omis.

2) - Ces opérateurs ne s'appliquent pas à une variable désignant un texte.

- l'opérateur

Il précise le sens du déplacement à effectuer

@ : déplacement vers le père
\+ : déplacement vers la droite
\- : déplacement vers la gauche

Remarque : Les opérateurs peuvent être combinés dans la même commande ; on obtient alors une expression.

• le facteur de répétition

Il indique le nombre de fois où l'opération doit être effectuée

n = valeur entière, par défaut n=1

* : indique que l'opération doit être effectuée autant de fois que la structure le permet.

Exemple : P@* déplace le pointeur P de la position actuelle à la racine de la structure.

Lors d'une telle opération de déplacement, le pointeur prédéfini K est également déplacé (sauf si le déplacement échoue).

Lorsqu'un déplacement ne peut être effectué :

- un message d'erreur est imprimé
- l'exécution de la commande est interrompue.

II.3.2 - Opérateurs de recherche

Dans ce qui précède, un élément d'une structure était atteint en décrivant le chemin à parcourir à partir d'un repère initial.

Ici l'élément à atteindre sera désigné par une caractéristique particulière.

La syntaxe d'utilisation de ces opérateurs est la suivante :

- EXP est une expression désignant un élément de structure à partir duquel va s'effectuer la recherche (repère de base).
- critère est la caractéristique de la structure à rechercher.
- l'opérateur '.' (POINT 1) demande une recherche de niveau 1 dans une structure (niveau des fils).
- l'opérateur '..' (POINT 2) permet d'explorer une structure et ses sous-structures à n'importe quel niveau (cf. Exemples).

a)Opérateur POINT 1

EXP •	nom de structure prédéfinie	(a)
	repère utilisateur	(b)
	entier n	(c)
	*	(d)

- le critère a) correspond à la recherche, dans la structure de base, d'une structure correspondant à la structure prédéfinie indiquée.
- le critère b) correspond à la recherche d'une structure analogue à celle désignée par le repère.
- les critères c) et d) permettent de se positionner sur le $n^{ième}$ (resp. dernier) fils de la structure de base.

Exemples :

Si S repère une structure correspondant au morceau de programme Pascal suivant :

```
if A=1 then
   begin
   x := x+1 ;
   y := 3 ;
   APPEL(x,y) ;
   end ;
```

et si le schéma prédéfini de la structure IF est le suivant :

IF = if &COND then &TRUE-PART else &FALSE-PART

alors :

S.COND — repère la sous-structure associée à A=1, cette recherche echouerait si S ne référençait pas une structure IF

S.1, S.EXP, S.EQU — repèrent la même sous-structure car :
&COND est synonyme de ØEXP
EQU appartient à la classe ØEXP

S.TRUE-PART, S.2, S.STAT, S.LSTAT — repèrent le bloc (begin...end)

S.* S.3 S.FALSE-PART	repèrent la structure vide correspondant à ØFALSE-PART
S.4	ne repère rien et provoque une erreur.

Si Y repère une structure correspondant à APPEL(X,Y) alors
S.LSTAT.Y repère la 3ème instruction de la partie vraie de S.

Une telle recherche avec repère sera surtout utilisée :

- si Y repère une structure contenant des méta-variables, cf.(II.4).
- dans des commandes itératives avec contrôle, cf.(III.2).

Remarques :

1) - La recherche s'effectue toujours à un niveau inférieur à celui du repère de départ.

Exemple : dans l'exemple précédent

S.IF échoue.

2) - Le repère initial d'une expression (de base) doit toujours être un repère utilisateur.

C'est-à-dire que : IF.COND est interdit.

3) - Les repères prédéfinis ne peuvent être utilisés comme intermédiaires dans une expression de recherche.

Exemple : S.LSTAT.K est interdit

K peut évidemment être le repère initial de l'expression.

Une recherche qui échoue provoque :

- l'impression d'un message d'erreur :

POINT1 ECHOUE pour a) et b)
FILS ECHOUE pour c) et d)

- l'interruption de l'exécution de la commande.

b) Opérateur POINT 2

EXP • •	{ nom de structure prédéfinie repère utilisateur }

- Recherche d'un élément dans une combinaison de structures.
- L'Opérateur POINT 2 correspond à une recherche selon l'ordre textuel, de l'élément de programme désigné par l'expression de base.

Exemple :

- Si S repère la structure

```
BEGIN
  IF A=1 THEN
        BEGIN
        X := 1
        A := A+1
        END ;
  X := X+1 ;
END ;
```

alors S.ASS désignera l'affectation X := X+1
et S..ASS désignera X := 1

mais aussi S.IF.ASS n'aboutit pas

alors que S.IF..ASS désigne X :=1.

- De même si Y repère l'identificateur X

S.Y échoue
S..Y désigne la partie gauche de l'affectation X := 1
ou S.ASS.Y, S.ASS..Y } désignent la partie gauche de X := X+1.

- Une recherche qui échoue provoque
  - l'impression du message '*POINT 2 ECHOUE*'
  - l'interruption de l'exécution.

Les remarques faites sur l'opérateur POINT 1 restent ici valables.

II.3.3 - Opérateurs sur les textes

Deux opérateurs peuvent être utilisés pour la manipulation des textes.

1) Concaténation

TXT | TXT

Permet de construire une nouvelle chaîne de caractères à partir de deux autres.

TXT désigne soit :

- une chaîne de caractères (ex. : 'TEXTEUN')
- soit une variable de type TEXTE (cf. Objets manipulés)
- soit le symbole &, indiquant que l'on veut rentrer le texte à la console.

2) Restitution de texte

EXP !

Fournit, à partir d'une structure désignée par une expression (cf. expressions), un texte paragraphé dans le langage édité.

EXP peut être une expression quelconque de parcours de structure.

Le résultat de l'opération est de type TEXTE.

II.4 - Méta-variables et schémas de recherche

Nous avons repris la notion de recherche selon un schéma, telle qu'elle existe dans l'éditeur MENTOR, ainsi que l'existence de méta-variables.

II.4.1 - Les Méta-variables

Les structures construites par l'éditeur peuvent contenir des éléments non spécifiés et remplacés par des méta-variables.

Exemple : if A>0 then ØI1 else ØI2

correspond à une structure dont les instructions ne sont pas spécifiées, et désigne donc toutes les instructions IF dont la condition est 'A>0' et qui ont une partie ELSE.

Dans une structure prédéfinie, le nom des sous-structures joue de façon identique le rôle de méta-variables.

Exemple : if ØCOND then ØTRUE_PART else ØFALSE_PART

II.4.2 - Recherche selon un schéma

Nous avons vu précédement (opérateurs POINT 1 et POINT 2) qu'une structure prédéfinie pouvait être utilisée comme critère de recherche.

L'utilisateur a de même la possibilité de construire des schémas contenant des méta-variables, de les repérer par des pointeurs et de les utiliser dans des opérations de recherche

Exemple :

Si S désigne la structure

```
IF A>0 THEN
    BEGIN
    X := X+1
    Y := 2
    END ;
```

et Z désigne la structure de type ASS

```
X := ØV
```

l'expression S..Z désignera la sous-structure

```
X := X+1 .
```

Effet de bord :

Lorsqu'une recherche, mettant en cause une méta-variable, aboutit, le système crée (ou met à jour) un pointeur utilisateur de même nom et qui repère la structure correspondante.

Exemple : dans l'exemple précédent,

Le pointeur V est créé et repère la sous-structure correspondant à X+1.

Cet effet de bord se produit également lors de la recherche d'une structure prédéfinie.

Exemple : si Y référence

```
begin
  X := Z+1
end ;
```

Y.ASS référence l'instruction X := Z+1

Le pointeur VARBL référencera alors X

et

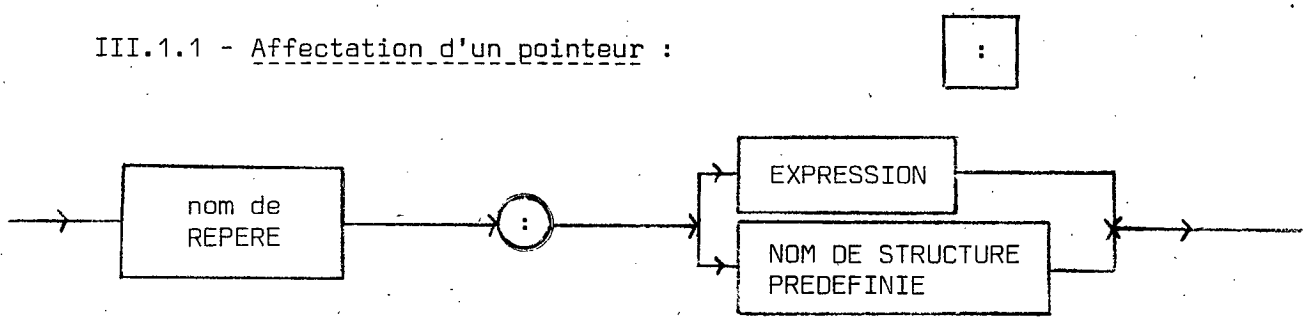
le pointeur EXP référencera l'expression Z+1.

III - DESCRIPTION DES COMMANDES

Ce chapitre donne une description complète des commandes permettant de manipuler les objets décrits précédemmment.

III.1 - Commandes d'édition de programmes

III.1.1 - Affectation d'un pointeur :



Le résultat de la commande est le positionnement du REPERE sur la structure désignée en partie droite.

1) Nom de REPERE

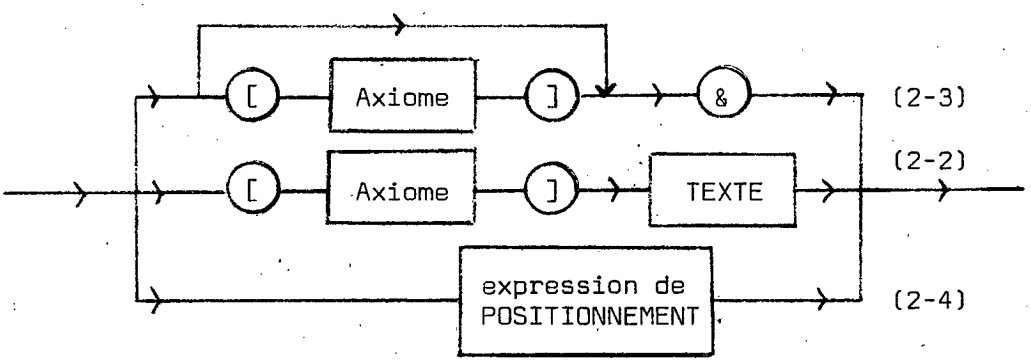
Le nom de repère peut être soit un identificateur fourni par l'usager, soit le repère prédéfini K qui peut être omis.

Exemple : TOTO : exp.

Exemple : K : exp.

Si ce repère usager n'existait pas, il est créé, sinon il est mis à jour.

2) Affectation d'une EXPRESSION



2-1) Axiomes

Lorsqu'on désire entrer, dans l'éditeur, un élément de programme, il faut indiquer le type de la structure qui doit être créée (type de la racine ou axiome).

Ce type est un nom de structure prédéfinie de la syntaxe abstraite du langage édité.

Remarque :

MINERVE ne permet, pour un langage donné, qu'un nombre limité de points d'entrée (c'est-à-dire que toutes les structures prédéfinies ne sont pas des axiomes).

Exemple : K : [STAT] & permet d'entrer une instruction PASCAL
K : [PROG] & permet d'entrer un programme
K : [TYP] & permet d'entrer une description de type

Par contre, on ne peut indiquer que l'on veut entrer une instruction IF ou une déclaration de tableau, qui ne sont pas des axiomes.

2-2) Affectation directe

L'utilisateur écrit, immédiatement après l'axiome, le texte programme qu'il veut introduire.

Exemple : TOTO : [STEP] 'I DOWNTO J' ;

Le texte est analysé comme une structure STEP et la représentation qui est créée est accessible par le pointeur TOTO.

2-3) Affectation différée

Si le texte est remplacé par le caractère &, le système rend le contrôle à l'usager, qui doit alors entrer son texte.

Dans ce cas, il est possible de retarder la donnée de l'axiome.

Exemple : ? TOTO : &
[? % demande d'axiome %
STEP] % réponse usager %
? I DOWNTO J % entrée de texte à la demande du système %

Dans ce cas, avant de demander à lire du texte, le système se met en attente de la donnée d'un axiome.

L'oubli de la donnée de l'axiome ou la donnée d'une structure prédéfinie qui n'est pas un axiome provoque une erreur.

La non-conformité entre l'axiome et le texte entré provoque une erreur.

En cas d'erreur, la commande n'est pas exécutée.

La fin de l'entrée du texte se fait pas la rencontre du caractère de fin de données, c'est-à-dire sur MITRA, en tapant %EOD.

2-4) Positionnement

Dans les cas 2-2 et 2-3, la commande d'affectation s'accompagnait de la création d'une nouvelle structure. On peut également affecter des repères à des parties de structures existantes, en utilisant les expressions de déplacement définies précédemment.

Exemple : TOTO : S .. IF ;

Le repère TOTO va repérer la première instruction IF de la structure repérée par S.

3) Affectation d'un nom de structure prédéfinie

On peut affecter à un repère le nom d'une structure prédéfinie.
Il y aura alors création d'une représentation arborescente dont le type sera celui précisé par le nom et dont les fils ne seront pas spécifiés (métavariables).

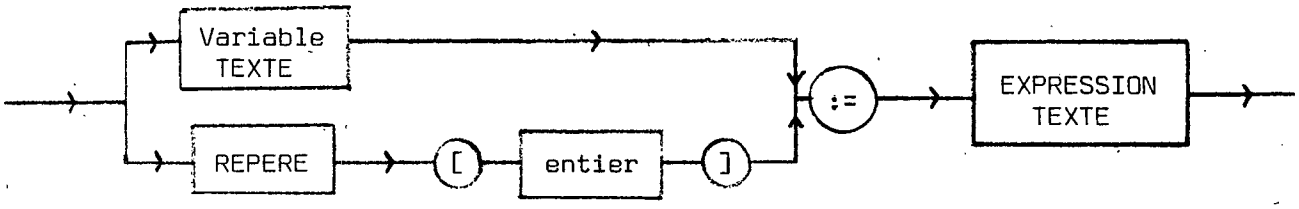
Exemple : TOTO : IF

crée un arbre de type IF dont les trois fils (&COND, &TP, &FP) sont de type métavariable (cf. métavariables).

Ultérieurement, l'utilisateur peut remplacer les métavariables par des éléments de programme du langage édité.

Remarque : Cette affectation n'est pas limitée aux seuls axiomes, mais peut être effectuée avec tous les schémas prédéfinis de type opérateur ou liste. Par contre, les schémas de type synonyme ou classe (cf. Ch. I) ne peuvent intervenir dans une telle affectation.

III.1.2 - Affectation d'une variable de type TEXTE : :=



permet soit d'affecter une chaîne de caractères (TEXTE) à une variable, soit de l'associer à un élément d'une structure.

1) Variable TEXTE

C'est une variable dont le nom est fourni par l'utilisateur et qui est conservée par l'éditeur.

Elle permet de désigner une chaîne de caractères.

Le comportement de l'éditeur vis-à-vis des noms de variables textes est le même que pour les noms de repères : création, conservation, affectation, suppression.

Exemple : TX := 'CECI EST UN TEXTE'

2) Textes associés aux structures

Un certain nombre d'informations, sous forme de texte, peuvent être associées aux représentations arborescentes des programmes (ou éléments de programmes) du langage édité. Ce sont :

1- les commentaires préfixes
2- les commentaires postfixes
3- les parties de texte non analysées
4- les indicateurs d'erreurs de compilation.

Pour accéder à ces informations, on utilise la notation

REPERE [entier]

Exemple : Dans la structure repérée par S :

S[1] accède au commentaire préfixe de la structure S
S[2] au commentaire postfixe.

3) Expression Texte

Un texte peut être exprimé de plusieurs manières ;

a- explicitement :

'VOILA UN TEXTE'

b- par une variable préalablement affectée à un texte

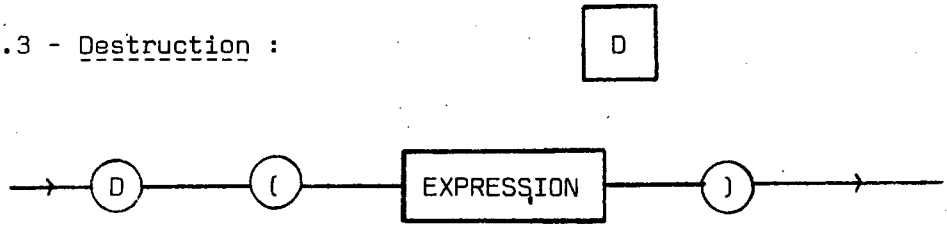
c- par une combinaison d'opérateurs et de variables (cf. opérateurs sur les textes) dont le résultat est de type TEXTE.

Exemples : ? T1 := 'VOILA' ; T2 := 'UN TEXTE' ;
? T3 := T1|T2 %concaténations%
? T4 := 'CECI EST'|T2
? S[1] := 'COMM. PREF'
? T5 := S..IF ! %restitution%

Remarque : Les combinaisons d'opérateurs ne peuvent être utilisées qu'en partie droite d'une affectation. La partie gauche doit nécessairement faire apparaître un nom de variable ou de repère, et un seul.

Exemple : ? S[1] := S..IF[2] est autorisé
mais
? S..IF[2] := S[1] est interdit

III.1.3 - Destruction :



Destruction d'une structure ou suppression d'un nom de pointeur/variable texte.

1) Destruction d'une variable

La commande de destruction permet d'éliminer le nom de variable, en paramètre, de la liste des noms connus de l'éditeur.

Exemple : ? D (TØTØ) rend inconnu l'identificateur TOTO.

Les structures désignées par un repère détruit ne sont pas effacées. Le pointeur prédéfini DUMP repère le sommet de la structure désignée. La même commande peut être utilisée pour la destruction des noms de variables texte, mais il n'y a pas de sauvegarde associée.

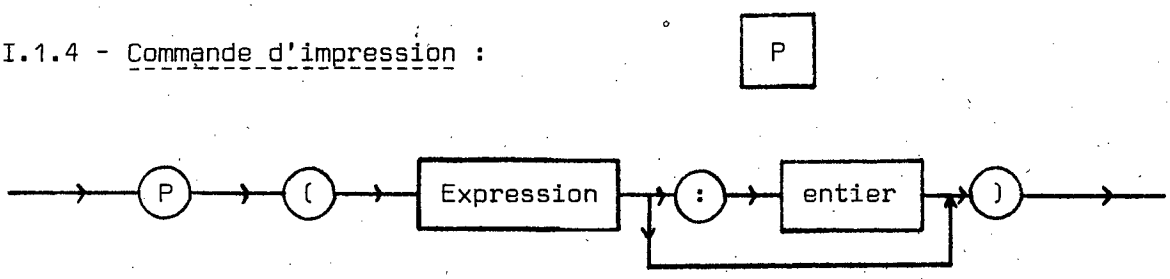
2) Destruction d'une structure

Si l'expression, en paramètre de la commande D, exprime une recherche ou un déplacement dans une structure, le résultat de la destruction est une déconnexion de la structure désignée par cette expression.

```
Exemple : ? Y : S.2.1
          ? D(S.2)        % le 2ème fils de S est déconnecté de la
                            structure S %
```

La sous-structure déconnectée existe toujours et peut être accédée, soit par des pointeurs usagers précédemment positionnés, soit par le pointeur prédéfini DUMP.

III.1.4 - Commande d'impression :



1) Impression de chaînes de caractères

Si le type de "expression" est TEXTE, la commande P provoque la sortie, à la console, de la chaîne de caractères désignée.

Exemple :
```
? P(TX)
VOILA UN TEXTE
? P(S[1])
% impression du commentaire préfixé de S s'il existe %
? P('DIRECT')
DIRECT
```

NB : Dans le cas des chaînes de caractères, seule la forme P(expression) est autorisée.

2) Impression de structures

Si "expression" désigne une structure, celle-ci est restituée sous forme de texte source.

Le texte fourni est paragraphé, c'est-à-dire exprimé sous une forme agréable à lire, mettant en évidence les structures du langage.

Exemple : Si l'on rentre l'instruction Pascal :
```
? X : [STAT] 'IF A>B THEN BEGIN A := A+1 ;
                B := 0 END ELSE WRITE (A)'
? P(X)    % impression paragraphée %
IF A>B THEN
    BEGIN
    A := A+1 ;
    B := 0
    END
ELSE WRITE (A)
```

3) Précision de l'impression (holophraste)

Dans le cas d'impression de structures, il est possible de préciser un degré de finesse dans cette écriture. La syntaxe est la suivante :

```
P(expression : entier)
```

où la valeur de l'entier permet de préciser le niveau de détail souhaité dans l'impression de la structure.

NB : Cet entier correspond à un certain niveau dans la structure, relatif à son sommet, bien que le niveau n'augmente pas systématiquement de 1 lorqu'on passe au fils.

Exemple : en reprenant l'exemple du paragraphe précédent

```
? P(X:2)  donne
IF A>B then
    BEGIN
    # ; #
    END
ELSE WRITE (A)
```

Il existe une valeur courante du degré de précision qui est prise par défaut.

La valeur courante peut être modifiée par commande (cf. HOL).

Remarques : L'impression de la structure repérée par le pointeur prédéfini K peut être simplifiée, sous la forme

```
P
```

% sans paramètre %

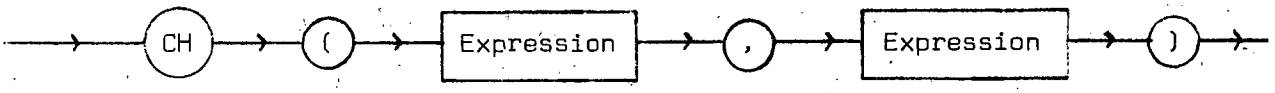
ou

```
P(:entier)
```

si l'on veut préciser la finesse d'impression

La fonction P ne modifie pas la valeur du pointeur courant K.

III.1.5 - Substitution de structure : CH



Remplace la structure désignée par la première expression, par celle désignée par la seconde.

Exemple :
```
? T : [STAT] 'IF A = 1 THEN B ELSE C'
? E : [EXP] 'A>0'
? CH(T.COND, E)
? P(T)                          % impression %
IF A>0 THEN B
ELSE C
? P(T.1) ; P(T.COND)
A>0
A>0
```

L'expression substituée est déconnectée de la structure, mais accessible par le pointeur DUMP.

La structure contenant la deuxième expression (substituante) n'est pas modifiée. Une recopie de la structure substituante est effectuée, si nécessaire (c'est-à-dire si c'est une sous-structure).

La partie substituante peut être entrée, en cours d'exécution, à la console. Dans ce cas, on peut utiliser les notations

(1) `CH (exp1, &)`

ou

(2) `CH (exp1, [axiome] &)`

Remarque : Le symbole & ne peut pas apparaître dans la 1ère expression.

L'éditeur vérifie que la substitution est syntaxiquement cohérente, c'est-à-dire que la structure substituante est la même, ou appartient à la même classe, que la structure substituée.

III.1.6 - Echange de structures : EC

Permute les structures désignées par les deux expressions. Elles doivent être de même type ou de type équivalent. L'échange peut être effectué :

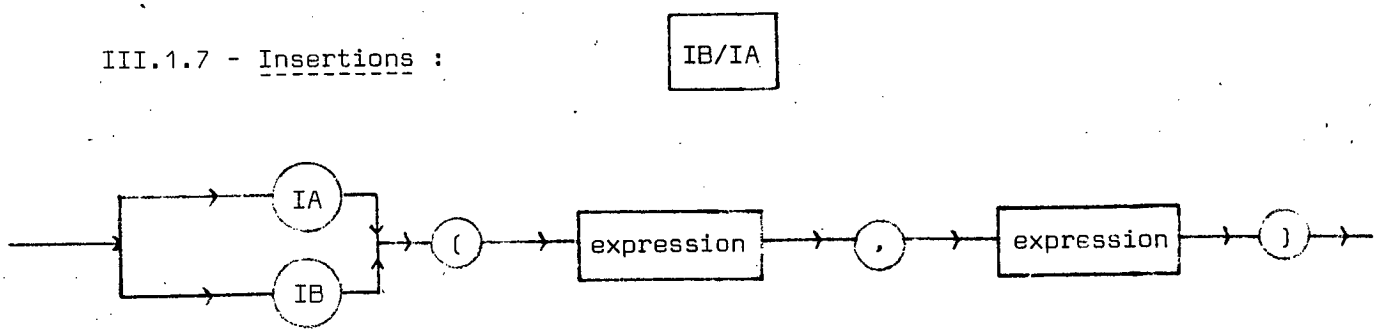
a) dans une même structure englobante,
b) entre deux structures non liées.

Exemple :
```
? T : [STAT] 'IF A = 1 THEN B ELSE C'
? Z : [STAT] 'BEGIN A := 0 ; WRITE (B) END'
? EC (T.2, T.FP)          % cas a) %
? P(T)
  IF A = 1 THEN C
  ELSE B
? EC (T.TP, Z.1)          % cas b) %
? P(T)
  IF A = 1 THEN A := 0
  ELSE B
? P(Z)
  BEGIN
  B ;
  WRITE (B)
  END
```

La commande EC n'effectue pas de création ou de copie de structure, mais modifie seulement les connexions entre structures.

En conséquence, la commande EC ne permet pas l'utilisation d'entrée différée (avec le symbole &).

III.1.7 - Insertions :



Insère la structure désignée par la seconde expression avant (IBefore) ou après (IAfter) la structure désignée par la première expression.

Une insertion ne peut se faire que dans une structure LISTE ; la première expression doit donc désigner un élément d'une liste ; la deuxième expression peut désigner :

a) un élément de liste,
b) une liste.

La deuxième expression peut aussi désigner une entrée différée :

c) IA (exp, &),
d) IA (exp, [axiome] &).

```
Exemple 1 : ? T : [LIDENT] 'A,B,C'
            ? S : [LIDENT] 'X,Y'
            ? IA (T.*, S.2)
            ? P(T)
              A,B,C,Y

Exemple 2 : ? IB (S.1, [LIDENT] &)
              [LIDENT]
                 U,T,V
            ? P(S)
              U,T,V,X,Y
```

III.1.8 - Changement du degré de précision pour l'impression de structures :

HOL

Le degré de précision de l'impression d'une structure (ou holophraste) est fixé, par l'éditeur, à une valeur courante (cf. commande P).

Néanmoins l'utilisateur peut en changer la valeur, de manière temporaire, au moyen de la commande HOL (entier) .

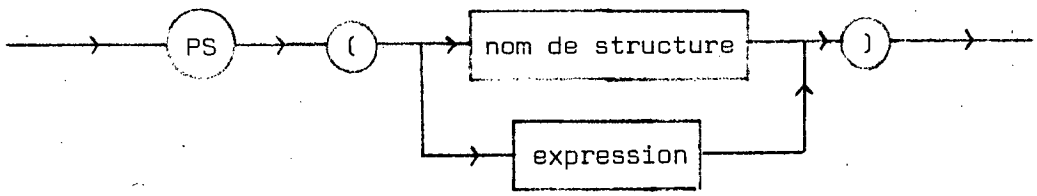
La valeur courante devient alors "entier", jusqu'à la prochaine commande HØL.

Le paramètre "entier" peut être positif ou nul. Le degré de précision est proportionnel au paramètre.

Pour obtenir la totalité du texte, il suffit de préciser une valeur "assez grande" de "entier" (par exemple 100).

Remarque : Le degré de précision n'a de sens que dans le cas d'une structure. La commande HOL est donc sans effet sur l'impression des variables de type TEXTE.

III.1.9 - Impression des schémas prédéfinis : PS



Permet d'obtenir le schéma de décompilation des structures prédéfinies. La structure prédéfinie peut être précisée :

a) par son nom (ex : PROG, ASS, IF, STAT),

b) par une expression de positionnement dans une structure de l'usager.

Exemple 1 : ? PS (DIV) % opérateur de division en Pascal %
DIV = & EXP1 / & EXP2
? PS (LEXP)
LEXP = ∅ EXP % donne le type des éléments de la liste %

Exemple 2 : Si S est un repère désignant la déclaration d'une procédure Pascal, alors S.1 est l'en-tête de cette procédure, renseignement que l'on obtient par :

? PS (S.1)
PROCHEAD = procedure & REFID & LPARAM

L'impression des schémas prédéfinis fournit à l'utilisateur :

a) des renseignements sur la nature des sous-structures qu'il manipule (exemple ci-dessus),

b) une aide à l'apprentissage de la syntaxe abstraite du langage.

Exemple : ? PS (IF)
IF = IF & COND THEN & TP else & FP

indique que la structure IF est représentée par un arbre ayant trois fils (COND, TP, FP) dont la nature peut être obtenue de nouveau avec la commande PS.

Des noms prédéfinis peuvent être utilisés pour obtenir des informations sur les schémas prédéfinis.

a) liste des axiomes (cf. Axiomes) : la liste des points d'entrée de l'analyseur (du langage édité) est obtenue par l'expression :

```
PS (AXIOMES)
```

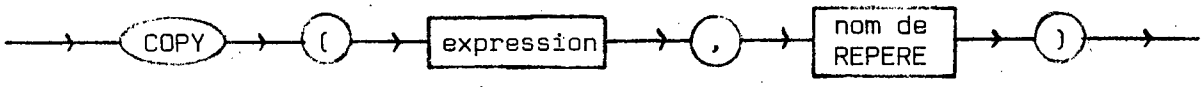
b) liste des schémas prédéfinis : permet d'obtenir l'impression de toute la syntaxe abstraite (structure prédéfinie) du langage :

```
PS (SCHPRD)          % SCHémas PRéDéfinis %
```

c) liste des opérateurs PS(OPERS)

- Classes PS(CLASSES)
- Listes PS(LISTES)
- Terminaux PS(TERMINX)
- Synonymes PS(SYNONS)
- Réservés PS(RESERVES)

III.1.10 - Duplication de structure : COPY



La structure désignée par "expression" est dupliquée et repérée par "nom de REPERE". La commande peut échouer pour deux raisons :

a) le déplacement décrit par l'expression est impossible,

b) l'éditeur ne dispose pas d'assez de place pour une création.

Les deux cas font l'objet d'un message d'erreur approprié.

Le traitement du "nom" est identique à celui effectué en cas d'affectation de structure (cf. commande ":").

Le repère prédéfini K ne peut être utilisé, dans une commande COPY, pour désigner le résultat. En particulier la forme élidée

```
COPY (expression)
```

provoque une erreur.

Exemple :

```
? S :  STAT  'A := B + C'
? COPY (S.2, Z)
? P(Z)                          % impression %
  B + C
?
```

III.2 - Commandes de contrôle

Ces commandes permettent de contrôler, d'interrompre ou d'effectuer itérativement des commandes selon la valeur de certaines conditions.

L'utilisateur peut donc écrire de véritables programmes de commandes, ce qui peut être utile, par exemple, pour effectuer des transformations systématiques sur un programme ou un ensemble de programmes.

L'utilisation effective d'un tel langage de commande suppose qu'il soit possible de soumettre à l'exécution un fichier de commandes.

III.2.1 - Interruption d'une liste de commandes : USER

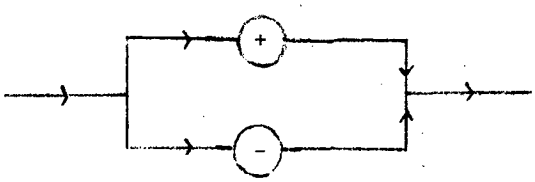
La commande USER permet d'interrompre l'exécution d'un programme éditeur pour rendre le contrôle à l'utilisateur.

Celui-ci pourra alors entamer un nouveau dialogue par l'introduction d'une liste de commandes.

Il devra enfin indiquer à l'éditeur la reprise de l'exécution après le traitement de l'interruption.

Reprise de l'exécution

Deux commandes indiquent à l'éditeur le comportement à avoir en fin d'interruption programmée :



1) Reprise en séquence

L'utilisation de la comande (+) demande à l'éditeur de reprendre l'exécution, à la première directive suivant la commande USER.

```
Exemple : ? C1 ; C2 ; USER ; C3 ; C4
            exécution de C1
            exécution de C2
          ?                          % reprise en main %
            C5 ; C6 ; +
            exécution de C5
            exécution de C6
                                     % retour en séquence %
            exécution de C3
            exécution de C4
          ?                          % fin de séquence %
```

Si l'usager ne précise pas le mode de reprise, l'éditeur le lui demande avant de poursuivre.

```
Exemple : ? C1 ; USER ; C2
            exécution de C1
          ? C3
            exécution de C3
          * SORTIE USER =' ? *          % message %
          ? +                           % réponse %
```

2) Abandon de la liste de commandes

On peut, après une interruption USER, abandonner les commandes non exécutées. On utilise pour cela la commande (-)

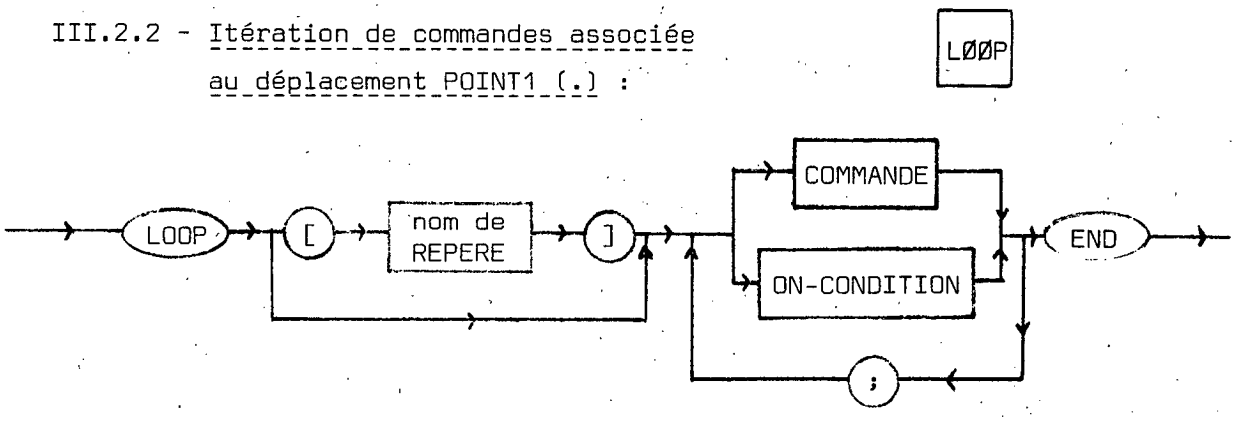
```
Exemple : ? C1 ; USER ; C2 ; C3
            exécution de C1
          ? C4 ; -
            exécution de C4
          ?                             % fin de séquence %
```

Répétition de USER

La commande USER peut être utilisée plusieurs fois

- soit dans une même liste de commandes,
- soit pendant l'interruption provoquée par une autre commande USER.

III.2.2 - Itération de commandes associée au déplacement POINT1 (.) :



La structure désignée par "nom de REPERE" est parcourue à un niveau horizontal. Seuls les fils sont accédés à chaque étape et on leur applique l'action décrite par la COMMANDE (ou la CONDITION).

Exemple : ? S : [STAT]'FOR I := 1 TO 10 DO J := J * (J+1)'

```
? LOOP [S]    P(S)    END ;
I                                   % premier fils du FOR %
1 TØ 10                             % deuxième fils %
J := J * (J+1)                      % troisième fils %
?                                   % fin de la boucle %
```

La variable de contrôle (nom de repère) sert, à chaque itération, à désigner l'élément de la structure atteint par l'éditeur.

En fin d'itération, la valeur initiale du repère est restaurée, même si une des commandes intermédiaire échoue.

Exemple : ? S : [LIDENT] 'A, B, C, D'

```
? LOOP [S]  P(S)  END ; P(S)
    A
    B
    C
    D
   A, B, C, D
?
```

Si le nom de repère est omis, la commande s'exécute sur la structure désignée par K.

LOOP ... END ; est équivalent à LOOP [K] ... END ;

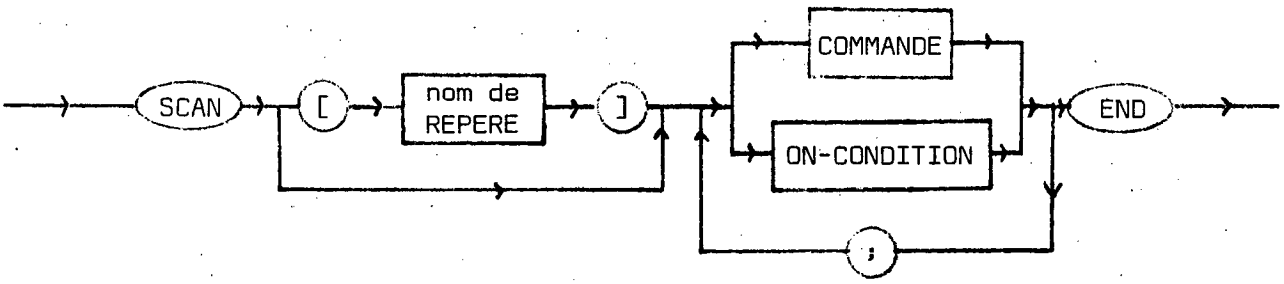
Contrairement aux autres commandes, les commandes itératives peuvent être entrées sur plusieurs lignes.

Le symbole "END" indique la fin de la séquence.

ON-CONDITION : cf. ON, III.2.4 .

III.2.3 - Itération de commandes correspondant au parcours textuel :

SCAN



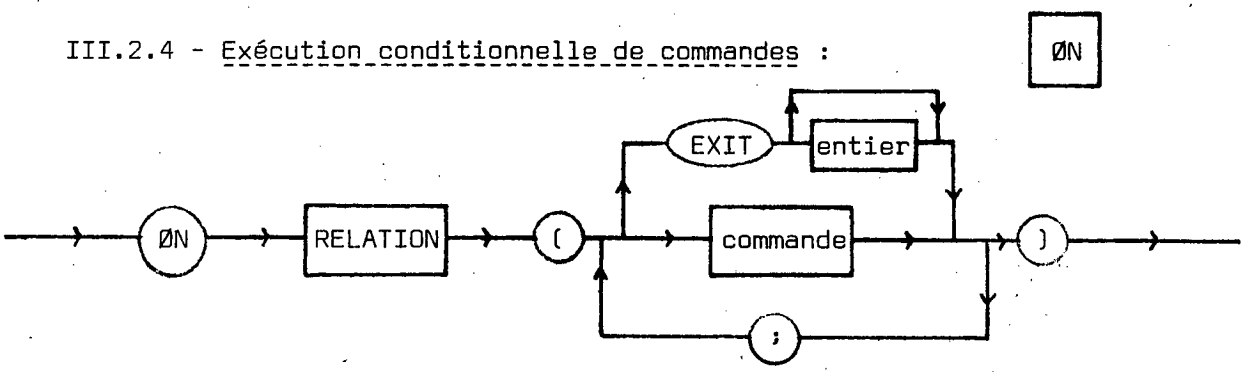
La structure désignée par le repère est parcourue entièrement de gauche à droite et de bas en haut (ceci correspond à un parcours textuel du programme).

La liste des commandes est appliquée à chaque élément de la structure.

Exemple : ? S : [STAT] 'IF A > B THEN XX := X + 1'
? SCAN [S] P(S) END
A
B
A > B
XX
X
1
X + 1
XX := X + 1

Toutes les remarques faites pour la commande LOOP s'appliquent à SCAN.

III.2.4 - Exécution conditionnelle de commandes :



1) ON-CONDITION

Si la relation fournit un résultat vrai, la suite de commandes, entre parenthèses, est exécutée.

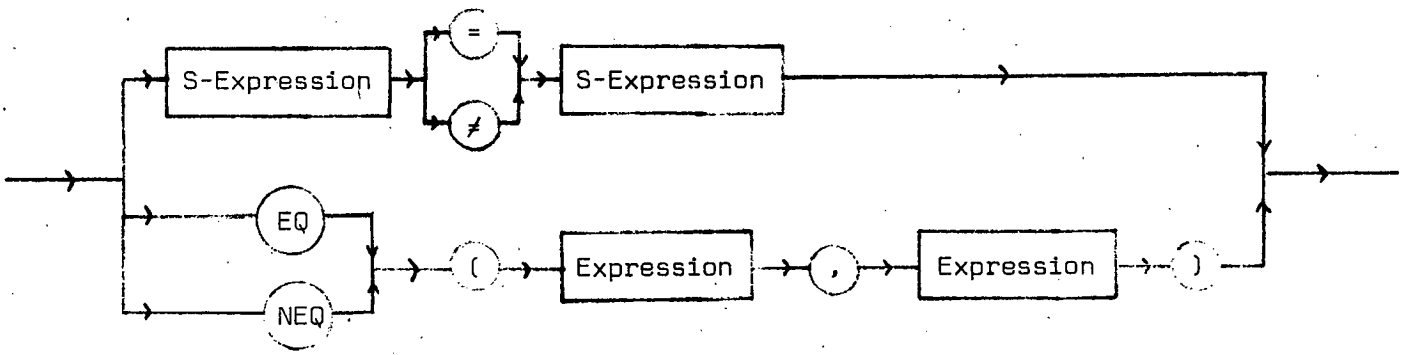
Sinon l'éditeur va exécuter la commande qui suit.

Un élément de la liste de commandes peut être :

- soit une commande normale,
- soit une instruction EXIT, de sortie impérative de boucle.

Une condition ON ne peut apparaître que dans une commande itérative (LØØP ou SCAN).

2) RELATION



2-1) S-Expression

Désigne :

- un nom de repère, de variable texte,
- un type explicite ou calculé par la directive TY (cf. TY).

Exemple : ? ... ON S = Y (.....) (1)
? ... ON TY(S) = LSTAT (.....) (2)

soit (1) si S et Y repèrent le même objet (et non des copies identiques) ;
(2) si le type de la structure S est une liste d'instructions.

2-2) EQ, NEQ

Comparent des sous-structures repérées par des expressions de recherche ou de déplacements.

Exemple : ? ... ON EQ (S.1, Y..IF) .....
% si le premier fils de S est identique au premier IF de Y %
? .. ON NEQ (S!, 'TEX') .....
% si le texte restitué, à partir du repère S, est différent de la chaine de caractères %

2-3) EXIT

La commande (EXIT n) demande la sortie impérative de "n" boucles (LOOP ou SCAN) imbriquées.

Si l'entier "n" est omis, sa valeur par défaut est 1.

Si la valeur de "n" est supérieure au nombre de boucles, la sortie est effectuée immédiatement après la boucle la plus externe. Cela ne provoque pas d'erreur.

Exemple :

```
LOOP [S1]
 ...
   LOOP [S2]
    ...
      SCAN [S3]
       ...
         EXIT j        <=
          ...
      END
   (1) ...
   END
(2) ...
END
(3) ...
```

Sorties effectuées suivant la valeur de "j" :

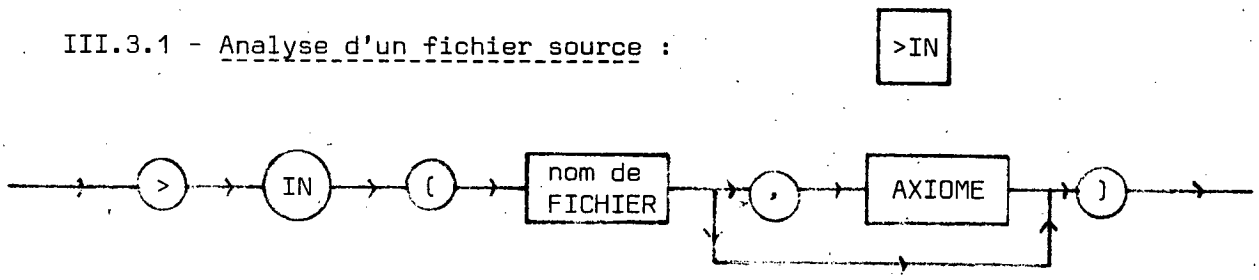
EXIT 1 ⟹ (1)
EXIT 2 ⟹ (2)
EXIT 3 ⟹ (3)
EXIT ⟹ (1)
EXIT 100 ⟹ (3)

Remarque : La commande EXIT n'est possible que dans une condition ON.

III.3 - Commandes externes

Ces commandes permettent la liaison avec les fichiers externes à l'éditeur, ainsi que la gestion des espaces de travail.

III.3.1 - Analyse d'un fichier source :



Le fichier désigné est externe à MINERVE et contient du texte source (programme ou partie de programme).

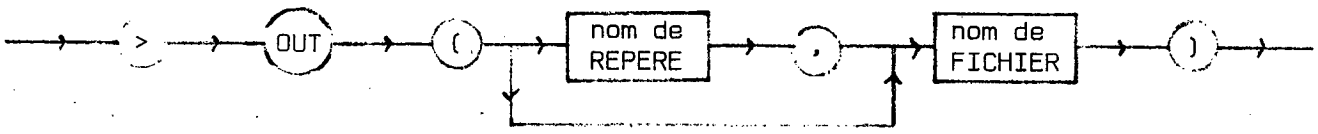
Ce texte est alors analysé en tant que structure du type "axiome" et, si l'analyse est correcte, ou le rattrapage d'erreur possible, une structure est construite. Cette structure sera repérée par le pointeur K.

Exemple : ? > IN (TEST1, LIDENT)

Le texte source contenu dans le fichier TEST1 est analysé en tant que liste d'identificateurs.

Dans le cas ou l'axiome est programme (PROG pour Pascal), celui-ci peut être omis dans la commande.

III.3.2 - Création de fichier : >OUT



La structure repérée par "nom de REPERE" est décompilée et copiée sous forme de texte (en langage édité) paragraphé sur le fichier externe "nom de FICHIER".

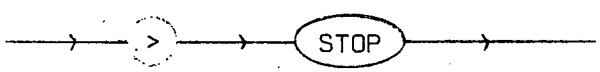
Le fichier est créé par MINERVE.

Le nom de repère peut être omis. Dans ce cas, c'est la structure repérée par K qui est décompilée.

Exemple : ? >OUT (S, FICH1)
décompile la structure S sur le fichier FICH1

? >OUT (FICH2)
décompile la structure repérée par K

III.3.3 - Fin de session : >STOP



Cette commande provoque la sortie du système.

Les structures et repères créés pendant la session de travail sont sauvegardés dans le WS actif (par défaut ENCAS).

III.3.4 - Changement de WS actif : >UWS

>UWS (nom de WS)

Le WS actif est sauvegardé ; le WS de nom indiqué devient le WS actif.

Exemple : >UWS (ENCAS) rend accessible ENCAS

..... } manipulations de ENCAS

.....

>UWS (TOTO) ENCAS est sauvegardé ; TOTO est accessible

..... } manipulations de TOTO

.....

>STOP sauvegarde dans TOTO

III.3.5 - Création de WS : >CWS

>CWS (nom de WS)

Le WS actif est sauvegardé ; un nouveau WS est créé, de nom indiqué ; le nouveau WS est une copie du WS actif ; le nouveau WS devient le WS actif.

Exemple :

```
.....
.....
.....
>UWS (TOTO)
.....
.....   } manipulations de TOTO
>CWS (TATA)      TOTO est sauvegardé ;
                 TATA est créé (identique à TOTO)
.....
.....   } manipulations de TATA
.....
>CWS (TATA)      inutile
.....
.....
>STOP            sauvegarde dans TATA
```

Remarque : >CWS (ENCAS) est interdit ;
>CWS (ID) détruit le WS de nom ID si celui-ci existait déjà.

Exemples : - modification d'un WS existant

```
>UWS (TOTO)
.....
.....
.....
>STOP
```

- création d'un WS à partir d'un WS existant

```
>UWS (TOTO)
>CWS (TATA)
.....
.....   } modifications
>STOP
```

III.3.6 - Commandes diverses

```
>CLEAR
```

Le WS actif est sauvegardé ; ENCAS devient WS actif, mais il est vide comme en début de session.

Exemple :

```
>UWS (TOTO)
.....
.....
>CLEAR          TOTO est sauvegardé
>CWS (TATA)     TATA est créé, mais il est vide
.....
.....
>STOP           sauvegarde dans TATA
```

Remarque : CLEAR est équivalent à

```
{ >STOP
{  appel de l'éditeur
```

```
>WSN
```

Imprime les noms des WS de l'usager.

```
>NAMES
```

Imprime les noms des repères et des variables de type texte du WS actif.

Exemple : nettoyage d'un WS

```
>UWS (TOTO)
>NAMES
.....
.....   } opérations de nettoyage
>STOP
```

>DWS (nom de WS)

Détruit le WS indiqué ; s'il s'agit du WS actif, ENCAS devient WS actif.

ENCAS ne peut être détruit.

>MOVE (nom de repère, nom de WS)

Copie la structure indiquée, du WS actif dans le WS indiqué ; le nom de repère ne doit pas exister dans le WS d'arrivée.

Exemple : Création d'un WS à partir de morceaux d'un autre

1ère solution
```
>UWS (TOTO)
>CWS (TATA)
.....
.....      } effacements de repères
>STOP        TOTO n'est pas modifié
```

2ème solution
```
>UWS (TOTO)
>MOVE (X, TATA)
.....
.....      } copies de structures de TOTO dans TATA
```

>LEGOS (X)

Cette commande provoque l'appel du compilateur LEGOS sur la structure repérée par X :

- le repère X doit appartenir au WS courant ;
- le type de la structure doit appartenir à la classe UNIT :

```
UNIT :: MODULE
        INTERFACE
        PREFIXE.
```

III.4 - Fonctions prédéfinies

III.4.1 - Extraction d'une sous-liste : SL

SL (EXP, fdl)

Construit une liste à partir d'une autre, pré-existante.

EXP est une combinaison de pointeurs et d'opérateurs (cf. EXPRESSION) désignant un élément d'une liste.

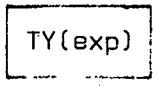
fdl permet d'indiquer la longueur de la sous-liste désirée avec une des trois formes suivantes :

a) fdl = i
   l'entier i précise la longueur de la sous-liste.

b) fdl = *
   la liste est construite en prenant tous les éléments à partir de celui désigné par "EXP".

c) fdl = PTR
   le pointeur PTR, qui doit être préalablement positionné dans la liste, sert à désigner la fin de la sous-liste.

III.4.2 - Type d'une expression : 



Permet d'accéder au nom de la structure désignée par "exp".

Exemple : Si X repère une structure PROGRAM

TY (X.1) = PROGHD

TY (X.*) = LSTAT

NB : TY ne peut être utilisé que

- dans une expression (P(TY(X.1)) ;

ou - dans une ON-CONDITION

ex : ON X.4 = TY(Y.*).

ANNEXE

Liste des commandes et opérateurs.

Liste des commandes

1 - Commandes d'affectation

:	Affectation d'un pointeur sur une structure
:=	Affectation d'une variable de type texte

2 - Commandes et opérateurs d'édition

D	Suppression d'une structure
P	Impression
CH	Substitution de structure
EC	Echange de deux structures
IA	Insertion après une structure
IB	Insertion avant une structure
COPY	Duplication de structure
SL	Extraction d'une sous-liste
HOL	Changement d'holophraste
PS	Impression de schéma prédéfini
TY	Type d'une structure

3 - commandes et opérateurs itératifs

LOOP	Boucle sur une structure
SCAN	Boucle sur une combinaison de structures
EQ	Comparaison de structures
NEQ	Comparaison de structures
.=	Comparaison de variables
≠	Comparaison de variables
ON	Condition de sortie de boucle
EXIT	Sortie de boucles
USER	Contrôle donné à l'usager
+	retour USER en séquence
-	retour USER avec abandon des commandes suivantes

4 - Commandes externes

>IN	Analyse de source sur un fichier externe et création de RA
>OUT	impression sur fichier de texte source paragraphé
>STOP	Fin de session
>WSN	Liste des noms de WS
>NAMES	Liste des variables du WS actif
>CLEAR	Nettoyage du WS actif
>UWS	Activation d'un WS
>CWS	Création d'un WS
>DWS	Suppression d'un WS
>MOVE	Recopie d'une structure d'un WS dans un autre
>LEGOS	Appel au compilateur LEGOS

SYNTAXE ABSTRAITE DE PASCAL

structures predefinies et schemas predefinis

terminaux

```
NIL             =   NIL
IDENT           =   %IDENT
INTCST          =   %INTCST
REAL            =   %REAL
STRING          =   '   %STRING   '
HEXCST          =   '   %HEXCST   '   X
META            =   $   %META
```

operateurs

```
PROG            =   <HDPROG>  <DEFPART>  L-LVDCL  L-LINIT  L-LPROC  &PGSTAT  .
HDPROG          =   PROGRAM  $REFID  &LEXTERN  ;
DCLCST          =   %IDENT  =  $CST
DCLTYP          =   %IDENT  =  $TYP
DCLVAR          =   L-LDEFID  :  $TYP
PROC            =   $TITLE  ;  $BODY  ;
RANGE           =   &CST1  ..  &CST2
PACKED          =   PACKED  $STRCTYP
ARRAY           =   ARRAY  L-LIX  OF  $TYP
SET             =   SET  OF  $SPLTYP
FILE            =   FILE  OF  $SPLTYP
REF             =   @  %IDENT
DECTAG          =   &IDENT1  :  &IDENT2
COLRC           =   L-LCST  :  (  &LFIELD1  <CASERC>  )
CASERC          =   CASE  $CASETG  OF  L-LRCCOL
VARPAR          =   VAR  L-LIDENT
FUNCPAR         =   FUNCTION  L-LIDENT  :  %IDENT
PROCPAR         =   PROCEDURE  L-LIDENT
LOCPAR          =   $LOCAL  :  %IDENT
PROCHD          =   PROCEDURE  $REFID  L-LPARAM
FUNCHD          =   FUNCTION  $REFID  L-LPARAM  :  %IDENT
DEFID           =   DEF  %IDENT
INIT            =   %IDENT  =  $VALU
INDEX           =   $VARBL  [  L-LEXP  ]
DOT             =   $VARBL  .  %IDENT
UNREF           =   $VARBL  @
FCALL           =   %IDENT  &LEXP1
RANGEXP         =   &EXP1  ..  &EXP2
TIMES           =   %INTCST  *  $CST
FORMAT          =   &EXP1  :  &EXP2
LABST           =   %INTCST  :  $STAT
ASS             =   $VARBL  :=  $EXP
CALL            =   %IDENT  L-LARG
GOTO            =   GOTO  %INTCST
CASE            =   CASE  $EXP  OF  L-LCOL  END
IF              =   IF  &COND  THEN  &TP  ELSE  &FP
WHILE           =   WHILE  $EXP  DO  $STAT
REPEAT          =   REPEAT  &LSTAT1  UNTIL  $EXP
FOR             =   FOR  %IDENT  :=  $STEP  DO  $STAT
WITH            =   WITH  L-LVRBL  DO  $STAT
COL             =   L-LCST  :  $STAT
TO              =   &EXP1  TO  &EXP2
DOWN            =   &EXP1  DOWNTO  &EXP2
NEQ             =   &EXP1  <>  &EXP2
LSS             =   &EXP1  <  &EXP2
LEQ             =   &EXP1  <=  &EXP2
GEQ             =   &EXP1  >=  &EXP2
GTR             =   &EXP1  >  &EXP2
IN              =   &EXP1  IN  &EXP2
PLUS            =   &EXP1  +  &EXP2
MINUS           =   &EXP1  -  &EXP2
OR              =   &EXP1  OR  &EXP2
UPLUS           =   +  $EXP
UMINUS          =   -  $EXP
MULT            =   &EXP1  *  &EXP2
DIV             =   &EXP1  /  &EXP2
INTDIV          =   &EXP1  DIV  &EXP2
MOD             =   &EXP1  MOD  &EXP2
AND             =   &EXP1  AND  &EXP2
NOT             =   NOT  $EXP
PCST            =   +  $CST
MCST            =   -  $CST
EQU             =   &EXP1  =  &EXP2
HEXF            =   $EXP  HEXA
DEFPART         =   L-LLAB  L-LCDCL  L-LTDCL
RECORD          =   RECORD  L-LFIELD  <CASERC>  END
BLOCK           =   <DEFPART>  L-LVDCL  L-LPROC  &PGSTAT
```

listes

```
LIDENT      =  %IDENT                ... ,
LLAB        =
               LABEL %INTCST ,       ... ,
LCDCL       =
               CONST
                   <DCLCST>         ... ;
LTDCL       =
               TYPE
                   <DCLTYP>         ... ;
LIX         =  [ $SPLTYP ]
LFIELD      =
               &FIELD               ... ;
LINIT       =
               VALUE
                   <INIT>           ... ;
LRCCOL      =
               <COLRC>              ... ;
LCST        =  $CST                 ... ;
LVDCL       =
               VAR
                   <DCLVAR>         ... ;
LPROC       =
               <PROC>               ...
LPARAM      =  ( $PARAM )           ... ;
LSTAT       =
               BEGIN
               $STAT                ... ;
               END
LDEFID      =  $REFID               ... ,
LEXP        =  $EXP                 ... ,
LARG        =  ( $ARG )             ... ,
LELEM       =  [ $ELEM ]            ... ,
LCOL        =
               <COL>                ... ,
LVRBL       =  $VARBL               ... ,
LVAL        =  ( $VAL )             ... ,
LSYMB       =  ( %IDENT )           ... ,
```

classes

```
STAT        :: REPEAT  ASS  CALL  CASE  WHILE  LABST  FOR  IF
               WITH  FCALL  GOTO  LSTAT

SPLTYP      :: IDENT  LSYMB  RANGE
STRCTYP     :: ARRAY  RECORD  SET  FILE
TYP         :: SPLTYP  STRCTYP  PACKED  REF
UNCST       :: NIL  STRING  INTCST  REAL  HEXCST
VARBL       :: IDENT  INDEX  UNREF  OUT
LOCAL       :: LIDENT  VARPAR
PARAM       :: LOCPAR  FUNCPAR  PROCPAR
EXP         :: EQU  NEQ  LSS  LEQ  GEQ  GTR  IN  PLUS  MINUS  OR
               UPLUS  UMINUS  UNCST  VARBL
               DIV  INTDIV  MOD  AND  MULT  NOT  LELEM  FCALL
STEP        :: TO  DOWN
CST         :: IDENT  INTCST  HEXCST  REAL  NIL  FCST  MCST
ELEM        :: EXP  RANGEXP
VALU        :: IDENT  UNCST  LVAL
VAL         :: IDENT  UNCST  TIMES
ARG         :: EXP  FORMAT
TITLE       :: PROCHD  FUNCHD
BODY        :: BLOCK  IDENT
CASETG      :: IDENT  DECTAG
REFID       :: IDENT  DEFID
```

synonymes

```
EXP1        == EXP
EXP2        == EXP
CST1        == CST
CST2        == CST
IDENT1      == IDENT
IDENT2      == IDENT
COND        == EXP
TP          == STAT
FP          == STAT
FIELD       == DCLVAR
LSTAT1      == LSTAT
LEXTERN     == LIDENT
PGSTAT      == LSTAT
PCSTAT      == LSTAT
LEXP1       == LEXP
LFIELD1     == LFIELD
```

REFERENCES

J. Cellier

"Un éditeur de programmes pour mini-ordinateur"

Thèse de 3ème cycle - UPS/Toulouse 79.

A.M. Couhault-Vercoustre

"Editeur syntaxique LIS et constructeur. Vers une construction automatique d'éditeur syntaxique"

Rapport IRIA-Sesori.

A.M. Couhault-Vercoustre, P. Maurice

"Spécification d'un éditeur syntaxique pour les langages Pascal et Legos"

Rapport interne INRIA/Projet LEGOS - Juin 80.

V. Donzeau-Gouge, G. Huet, G. Kahn, B. Lang, J.J. Levy

"A structure oriented program editor : a first step toward computer assisted programming"

International Computing Symposium, North Holland Publishing Co - 75.

V. Donzeau-Gouge, G. Huet, G. Kahn, B. Lang

"Programming environments based on structured editors : the Mentor experience"

Rapport INRIA n° 26 - Juillet 1980.

Equipe LANGAGES et TRADUCTEURS

"Le système SYNTAX - Manuel d'utilisation"

Rapport IRIA-LABORIA - 1977.

P.H. Feiler, R. Medina-Mora

"An incremental programming environment"

5th International Conference on Software Engineering - San Diego 81.

C.W. Fraser

"Syntax directed editing of several data structures"

SIGPLAN Notices, Vol. 16, n° 6 - June 1981.

W.J. Hansen

"Creation of hierarchic text with a computer display"

Thèse - Stanford University 71.

G. Huet, G. Kahn, P. Maurice

"Environement de programmation Pascal. Manuel d'utilisation"

Rapport IRIA-Sesori 1977.

J.W. Lewis

"ALBE/P : Language Neutral Form"

5th International Conference on Software Engineering - San Diego 81.

L. Lucrèce

"Aide au développement de logiciel. Une réalisation pour le langage Pascal"

Thèse de 3ème cycle - UPS/Toulouse 79.

B. Mélèse

"Mentor : l'environnement Pascal"

Rapport INRIA n° 5 - Octobre 1981.

M. Morris, M.D. Schwartz

"The design of a language-directed editor for block-structured languages"

SIGPLAN Notices, vol. 16, n° 6 - June 1981.

O. Strömfors, L. Jonesco

"The implementation and experiences of a structure-oriented text editor"

SIGPLAN Notices, vol. 16, n° 6 - June 1981.

T. Teitelbaum, T.H. Reps

"The Cornell program synthesizer : a syntax-directed programming environment"

CACM - vol. 24, n° 9 - September 1981.

W. Teitelman

"Interlisp Reference Manual"

XEROX, Palo Alto Research Center - 1974.

A. Van Dam, D.E. Rice

"On line text editing : a survey"

ACM, Computing Surveys, Vol. 3, n° 3 - Sept. 71.

J.R. Wood

Imprimé en France

par

l'Institut National de Recherche en Informatique et en Automatique