N° d'ordre 86

THÈSE

présentée à la

FACULTÉ DES SCIENCES DE L'UNIVERSITÉ DE LILLE

pour obtenir le

Titre de Docteur de Spécialité

(Mathématiques Appliquées)

par

ANDRÉ ARNOLD

Formalisation des Démonstrations Mathématiques

Thèse soutenue le 13 Mars 1968, devant la Commission d'Examen

Monsieur P. BACCHUS	**Président**
Monsieur J. BENABOU	**Examinateur**
Monsieur J.-C. HERZ	**Rapporteur**

FACULTE DES SCIENCES — UNIVERSITE DE LILLE

LISTE DES PROFESSEURS

-oOo-

DOYENS HONORAIRES

MM. H. LEFEBVRE, M. PARREAU

PROFESSEURS HONORAIRES

MM. ARNOULT, BEGHIN, BROCHARD, CAU, CHAPPELON, CHAUDRON, CORDONNIER, DEHEUVELS, DEHORNE, DOLLE, FLEURY, P. GERMAIN, KAMPE DE FERIET, KOURGANOFF, LAMOTTE, LELONG, Mme LELONG, MM. MAZET, MICHEL, NORMANT, PARISELLE, PASCAL, PAUTHENIER, ROIG, ROSEAU, ROUBINE, ROUELLE, WIEMAN, ZAMANSKY

PROFESSEURS

BACCHUS P.	Mathématiques appliquées
BEAUFILS J.P.	Chimie
BONNEMAN P.	Chimie
BECART M.	Physique
BLOCH V.	Biologie et Physiologie Animales
BONTE A.	Sciences de la Terre
BOUGHON P.	Mathématiques purés
BOUISSET S.	Biologie et Physiologie Animales
BOURIQUET R.	Biologie Végétale
CELET P.	Sciences de la Terre
CONSTANT E.	Electronique, Electrotechnique et Automatique
CORSIN P.	Sciences de la Terre
DECUYPER M.	Mathématiques purés
DEDECKER P.	Mathématiques Pures
DEFRETIN R.	Biologie et Physiologie Animales
DEHORS R.	Electronique, Electrotechnique et Automatique
DELATTRE C.	Sciences de la Terre
DELEAU P.	Sciences de la Terre

DELHAYE M.	Chimie
DESCOMBES R.	Mathématiques pures
DURCHON M.	Biologie et Physiologie Animales
FOURET R.	Physique
GABILLARD R.	Electronique, Electrotechnique et Automatique
GLACET C.	Chimie
GONTIER G.	Mathématiques appliquées
HEIM DE BALSAC H.	Biologie et Physiologie Animales
HEUBEL J.	Chimie
HOCQUETTE M.	Biologie végétale
LEBEGUE A.	Botanique
Mme LEBEGUE G.	Physique
LEBRUN A.	Electronique, Electrotechnique et Automatique
Mlle LENOBLE J.	Physique
LIEBAERT R.	Electronique, Electrotechnique et Automatique
LINDER R.	Biologie végétale
LUCQUIN M.	Chimie
MARION E.	Chimie
MARTINOT-LAGARDE A.	Mathématiques appliquées
Mlle MARQUET S.	Mathématiques pures
MENNESSIER G.	Géologie
MONTARIOL F.	Chimie
MONTREUIL J.	Chimie
MORIAMEZ M.	Physique
MOUVIER G.	Chimie
PARREAU M.	Mathématiques pures
PEREZ J.P.	Physique
PHAM MAU QUAN	Mathématiques pures
POUZET P.	Mathématiques appliquées
PROUVOST J.	Sciences de la Terre
SAVARD J.	Chimie
SCHILTZ R.	Physique
SCHALLER F.	Biologie et Physiologie Animales
Mme SCHWARTZ M.H.	Mathématiques pures
TILLIEU J.	Physique
TRIDOT G.	Chimie
VAZART B.	Botanique

VIVIER E.	Biologie et Physiologie Animales
WATERLOT G.	Sciences de la Terre
WERTHEIMER R.	Physique

MAITRES DE CONFERENCES

BELLET J.	Physique
BENABOU J.	Mathématiques pures
BILLARD J.	Physique
BOILLET P.	Physique
BUI TRONG LIEU	Mathématiques pures
CHERRUAULT Y.	Mathématiques pures
CHEVALIER A.	Mathématiques
DERCOURT J.M.	Sciences de la Terre
DEVRAINNE P.	Chimie
Mme DIXMIER S.	Mathématiques
Mme DRAN R.	Chimie
DUQUESNOY A.	Chimie
GOUDMAND P.	Chimie
GUILBAULT P.	Biologie et Physiologie Animales
GUILLAUME J.	Biologie végétale
HENRY L.	Physique
HERZ J.C.	Mathématiques appliquées
HEYMAN M.	Physique
HUARD DE LA MARRE P.	Mathématiques appliquées
JOLY R.	Biologie et Physiologie Animales
LABLACHE-COMBIER A.	Chimie
LACOSTE L.	Biologie végétale
LAMBERT G.	Physique
LANDAIS J.	Chimie
LEHMANN D.	Mathématiques pures
Mme LEHMANN J.	Mathématiques pures
LOUCHEUX C.	Chimie
MAES S.	Physique
METTETAL C.	Zoologie
MONTEL M.	Physique

NGUYEN PHONG CHAU	Mathématiques
PANET M.	Electronique, Electrotechnique et Automatique
PARSY F.	Mathématiques pures
RACZY L.	Physique
SAADA G.	Physique
SEGARD E.	Chimie
TUDO J.	Chimie minérale appliquée
VAILLANT J.	Mathématiques pures
VIDAL P.	Electronique, Electrotechnique et Automatique
Mme ZINN-JUSTIN N.	Mathématiques pures

Je tiens à exprimer mes remerciements à Monsieur BACCHUS, Directeur du Laboratoire de Calcul, qui a bien voulu me faire l'honneur de présider le jury.

Monsieur KUNTZMANN, Directeur de l'Institut de Mathématiques Appliquées de Grenoble m'a donné l'idée de ce travail et en a constamment suivi l'évolution. Je lui suis redevable de bien des encouragements. Qu'il veuille trouver ici l'expression de ma reconnaissance.

C'est sous la direction de Monsieur HERZ que cette thèse a été achevée. Je le remercie vivement de l'intérêt qu'il lui a porté et des précieux conseils qu'il m'a donnés.

Je remercie Monsieur BENABOU d'avoir accepté de faire partie du jury. Avec lui je voudrais remercier aussi, en espérant qu'ils se reconnaîtront, tous ceux qui, d'une manière ou d'une autre, ont favorisé l'aboutissement de ce travail.

Je voudrais plus particulièrement remercier Messieurs HENNERON et GUILLERMINET, élèves ingénieurs à l'Institut Polytechnique de Grenoble, qui, en acceptant de travailler sur le sujet que je leur ai proposé, m'ont permis de bénéficier de leurs remarques et de leurs expériences.

Je remercie également Madame DUSART et tous ceux dont le travail souvent ingrat a permis la réalisation matérielle de cette thèse.

FORMALISATION des DEMONSTRATIONS MATHEMATIQUES

INTRODUCTION

De plus en plus la logique mathématique, dont le but initial est d'exprimer les objets et le raisonnement mathématique dans un langage formel, afin de réduire les démonstrations à un simple calcul, se constitue en branche autonome des mathématiques. Les études des logiciens semblent avoir de moins en moins d'incidences sur le travail habituel du mathématicien. Elles portent davantage sur l'étude des théories et du raisonnement en général que sur l'étude du raisonnement tel qu'il est utilisé dans un quelconque ouvrage de mathématiques. Les systèmes de déduction naturels ont été relativement peu traités.

Dans le domaine de la démonstration automatique, des résultats intéressants ont été obtenus, mais sont difficilement exploitables, entre autres à cause de la difficulté de transcription d'un énoncé en langue naturelle en une formule du calcul des prédicats. D'autre part, la démonstration fournie ne ressemble que de très loin à la démonstration au sens où l'entend un mathématicien. Enfin, les processus de démonstration sont encore purement combinatoires et ne font pratiquement pas intervenir les résultats intermédiaires d'une théorie (définitions, lemmes et théorèmes).

Pour améliorer le rendement de la démonstration automatique et pour la rendre plus proche de la démonstration, il serait bon d'y introduire des heuristiques. Pour ce but particulier et aussi pour des raisons d'ordre pédagogiques ou autres, il est intéressant de connaître de façon plus précise ce qu'est une démonstration. Un travail préalable à cette étude est de formaliser les démonstrations, c'est-à-dire de les écrire dans un langage formel dont la sémantique et la syntaxe sont bien définies (en remarquent que dans la logique mathématique, les deux points de vue syntaxique et sémantique sont pratiquement équivalents : une démonstration syntaxiquement correcte l'est aussi sémantiquement). Un tel langage une fois défini, la vérification d'une

démonstration se réduit à un problème d'analyse syntaxique et peut donc être réalisée sur machine.

Paul Abrahams a proposé un système de formalisation qui est en fait une application du langage LISP au langage mathématique. S'il a l'avantage de se rapprocher davantage de la langue naturelle tant au point de vue de l'écriture qu'à celui de la structure d'une démonstration, il reste encore trop fortement lié à la syntaxe de LISP et demande pour être compris une connaissance préalable de LISP et de son écriture fortement parenthésée.

Le langage que nous proposons ici est indépendant de tout langage de programmation et reste assez près de la langue naturelle pour être assimilable par un mathématicien ayant un minimum de connaissances en calcul des prédicats, et en programmation juste ce qu'il faut pour savoir qu'on ne remplace pas impunément un signe par un autre.

Les expressions mathématiques sont écrites dans un formalisme très voisin de celui du calcul des prédicats. Une démonstration est une suite de lignes, chacune étant une définition, un théorème, ou une expression suivie soit d'une "justification" indiquant comment obtenir cette expression à partir des précédentes, soit par une autre démonstration. Nous introduisons ainsi une structure de bloc à l'intérieur d'une démonstration. D'autres structures de blocs sont utilisées pour la quantification et pour la dérivation sous conditions. Ce travail étant une approche d'un domaine encore très peu exploré, il soulève plus de questions qu'il n'en résout. Nous nous sommes efforcés de cerner et de préciser ces questions en suspens en donnant parfois les directions dans lesquelles il nous semble possible de trouver leur solution.

Comme application immédiate de cette formalisation des démonstrations, nous pensons bien sûr à la vérification automatique, mais nous espérons que ce travail pourra servir d'outil dans l'étude de domaines tels que l'heuristique en démonstration automatique, l'enseignement programmé des mathématiques, la simulation du raisonnement en intelligence artificielle.

Plutôt que de donner "ex abrupto" la définition complète du langage, nous avons préféré introduire progressivement les éléments qu'il comporte, quitte à revenir en arrière par la suite. Dans le chapitre I nous donnons l'allure générale du langage, ses caractéristiques essentielles et les éléments qui permettent de traiter l'aspect logique des mathématiques. Le chapitre II apporte quelques affinements et précise la manière de traiter ce qui est plus spécifiquement mathématique. Dans le chapitre III nous envisageons quelques questions qui ont plusieurs solutions possibles entre lesquelles nous n'avons pas tranché, ou des problèmes non encore résolus. En annexe 1, le langage est redéfini par ses tableaux syntaxiques et en annexe 2 nous donnons quelques aperçus sur l'implémentation du langage en machine.

Chapitre I.- Caractères généraux du langage

CARACTERES GENERAUX du LANGAGE

Le langage défini ici devra nous permettre d'écrire de façon précise les éléments nécessaires à toute démonstration : les expressions mathématiques, les définitions et les théorèmes utilisés au cours de la démonstration, des "justifications" expliquant de quelle façon on passe d'une expression à une autre.

Pour définir la syntaxe du langage nous utiliserons la notation de Backus.

1. Expressions

1.1. Identificateurs.

Nous considérons un alphabet comprenant les lettres latines et grecques, minuscules et majuscules, les chiffres, et tout autre signe dont nous aurons besoin, à l'exclusion de ceux rencontrés par la suite qui jouent un rôle particulier. Nous appellerons identificateurs les mots formés sur cet alphabet.

1.2. Expressions

Les expressions sont construites à partir de formules élémentaires, ou prédicats au moyen de connecteurs logiques $\equiv, \sim, \vee, \wedge, \supset$ et des quantificateurs $\forall$, $\exists$, $\exists!$.

Les priorités que nous avons choisies pour ces opérateurs ne sont pas celles qu'on utilise habituellement en logique. Pour des raisons de commodité nous avons donné à l'opérateur le plus utilisé la plus faible priorité, c'est-à-dire la plus grande portée. Il nous a paru que dans l'usage habituel les deux connecteurs les plus utilisés étaient $\wedge$ et $\supset$. Le premier étant assiocatif, mais pas l'autre, il est intéressant d'avoir $\wedge$ comme connecteur principal.

Par ordre de priorité décroissante les connecteurs sont $\wedge$,$\supset$et $\equiv$, $\vee$, $\sim$. Pour éviter les confusions avec l'usage des logiciens, et pour se rapprocher davantage de l'écriture habituelle, nous remplaçons $\wedge$ par une virgule, par ou, $\supset$ et $\equiv$ par => et <=>.

Ainsi a, b => b ou c se lit logiquement
a∧(b⇒(b∨c))

De façon plus précise, une expression se définit ainsi

<expression>::<sous-formule> / <expression , <sous-formule>
<sous-formule>::<disjonction> / <disjonction> => <disjonction> /
<disjonction> <=> <disjonction>
<disjonction>::= <primaire> / <primaire>/
<disjonction> ou <primaire> /
<disjonction> ou ∿ <primaire>
<primaire>::= (<expression>) / <prédicat> /
<formule quantifiée>
<formule quantifiée>::= <Q><variable> (<formule>)/
<Q><variable><formule quantifiée>
<Q>::=∀/ ∃/ ∃!
<prédicat>::=<objet> est un <identificateur de prédicat>
<arguments> / <Prédicat élémentaire>
<arguments>::=<vide>/[<liste d'objets>]
<liste d'objets>::=<objet> / <liste d'objets>, <objet>
<identificateur de prédicat>::=<identificateur>

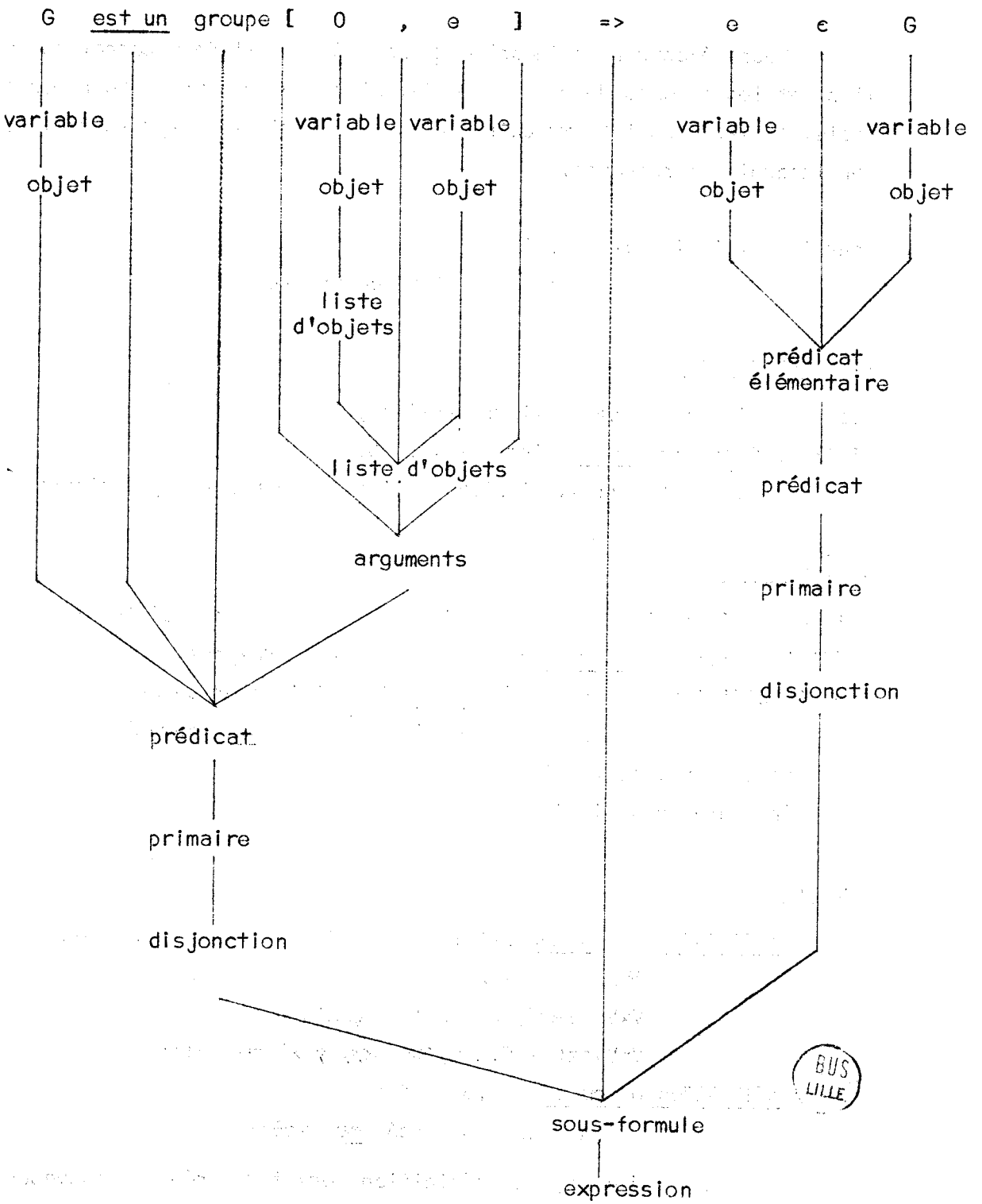
L'objet sera défini ultérieurement (2.2 et II.1.2). Pour le moment il suffit de savoir que c'est une variable (<objet>::=<variable>) ou que c'est une certaine suite de signes contenant des variables.

Un prédicat élémentaire est formé d'objets et d'un symbole relationnel. Les symboles relationnels sont définis dans le système, comme =, ∈, ou apparaissent dans une définition. (cf III.1.2.)

exemple G est un groupe [o,e]= ∀g∀y∀z((x∈G,y∈G,xoy=z)⇒z∈G)

∃F (F est une application biunivoque[P,R])

Comme en logique classique on distinguera les variables liées et les variables libres suivant qu'elles figurent ou non dans la portée d'un quantificateur.


G est un groupe [0 , e] => e e G
variable
objet
variable variable
objet objet
liste d'objets
variable variable
objet objet
prédicat élémentaire
liste d'objets
arguments
prédicat
prédicat
primaire
disjonction
primaire
disjonction
sous-formule
expression

2. Notions utilitaires

Pour démontrer un théorème d'une théorie, il faut connaître les définitions et les axiomes de cette théorie, ainsi que d'autres théorèmes supposés vrais. Il faut aussi "introduire" certaines variables représentant les objets sur lesquels on raisonne.

<partie utilitaire>:= <partie soit><liste de définitions>
<axiomes> <liste de théorèmes>

2.1. Introductions

<partie soit>::= <vide> / <introduction>;
<introduction>::= soit <liste de variables>
<liste de variables>::=<variable> / <liste de variables>,<variable>

2.2. Définitions

<liste de définitions>::=<vide>/<liste de définitions><définition>;
<définition>::= définition<partie définie>= def<expression >
<partie définie>::=<variable> est un <identificateur de prédicat>
<paramètres>/<opérateur> compose <paramètres>
<paramètres>::=<vide>/[<liste de variables>]
<opérateur>::=<variable>

exemple

définition R est une relation d'équivalence sur [E] = def
$\forall x(x \in E \Rightarrow xRx)$,
$\forall x \forall y((x \in E, y \in E, xRy) \Rightarrow yRx)$,
$\forall x \forall y \forall z((x \in E, y \in E, z \in E, xRy, yRx) \Rightarrow xRz)$;
définition U compose [A,B] = def
$\forall x(x \in U(A,B) \Leftrightarrow x \in A$ ou $x \in B)$;

Le premier type de définition permet de créer de nouveaux prédicats, le second définit les opérateurs préfixés.

Des objets composés par un opérateur donnent un nouvel objet.

<objet>::=<opérateur> (<liste d'objets>)

2.3 Axiomes

<axiomes>::=<vide>/ h <expression>;

2.4. Théorèmes

<liste de théorèmes>::=<vide> / <liste de théorèmes><théorème>;
<théorème>::= théorème <identificateur de théorème>
<paramètres><hypothèse><conclusion>
<hypothèse>::= <vide> / h <expression>
<conclusion>::= c <expression>

exemple

théorème R1[E,P,R]h ∀x (x∈P => x est une partition de [E]),
∀x(x∈R => x est une relation d'équivalence sur [E])
c ∃f (f est une application biunivoque[P,R]);

2.5. Enoncé

Ensuite il faudra donner l'énoncé du théorème qui sera suivi de sa démonstration.
<programme>::=<partie utilitaire><énoncé><démonstration>

L'énoncé comprendra l'introduction des variables particulières à ce théorème, ses hypothèses et l'expression à démontrer ou conclusion
<énoncé>::= <partie soit><axiomes><expression>

exemple

Soit E, d, r, a, X; h X est une boule[E, d, r, a];
X est un ouvert[E, d]<démonstration>

Soit E, d, r, a; ∀ X (X est une boule [E,d, r, a]<=>
X est un ouvert [E, d]) <démonstration>

3. Démonstration

Une démonstration sera constituée d'une suite d'expressions, chaque expression étant étiquetée pour pouvoir la repérer facilement, et suivie d'une preuve permettant de vérifier qu'on a bien le droit d'écrire cette expression. On se donnera la possibilité d'introduire au cours d'une démonstration de nouvelles définitions et de nouveaux théorèmes. Une preuve pourra être entre autres une démonstration ce qui fait apparaître dans une démonstration une structure récursive dont on examinera plus loin les conséquences.

<démonstration>::= en effet <liste de lignes> cqfd
<liste de lignes>::=<ligne>/<liste de lignes>;<ligne>
<ligne>::=<théorème>/<définition>/<dernière ligne><preuve>/
<étiquette>:<déduction>/
<étiquette>:<généralisation-∀>/
<étiquette>/<généralisation-∃>
<dernière ligne>::=<étiquette>:<expression>
<étiquette>::=<identificateur>
<preuve>::=<démonstration> / par <justification>

Pour qu'une démonstration soit correcte, il faut que l'expression figurant dans ligne juste avant cqfd soit formellement identique à l'expression à démontrer-(c'est-à-dire celle qui précède le en effet correspondant).

Nous dirons que deux expressions sont formellement identiques si elles s'écrivent de la même façon à une substitution près des variables quantifiées.

exemple

∃x (x est un truc [a,x]) est formellement identique à
∃y (y est un truc [a,y])
$x \leqslant y$, $y \leqslant x$ n'est pas f-identique (formellement identique)
à $y \leqslant x$, $x \leqslant y$.

3.1. Déduction

<déduction>::= h <expression>;<liste de lignes>;<dernière ligne>
déduction

Une déduction syntaxiquement correcte a donc la forme suivante :

h f_o;
n_1 : f_1<preuve> ;
---
n_i : f_i <preuve>;
n_{i+1}: f_{i+1} déduction

De plus on doit avoir f_{i+1} f-identique à $\overline{f}_o \Rightarrow \overline{f}_i$

avec $\begin{cases} \overline{f} = f \text{ si } f \text{ est une disjonction} \\ \overline{f} = (f) \text{ dans les autres cas} \end{cases}$

(ceci pour s'assurer que => est bien l'opérateur principal de f_{i+1})

Nous introduisons ici un nouveau type de bloc délimité par h et déduction dont nous verrons plus loin l'utilité.

On utilisera la déduction chaque fois qu'on doit utiliser le théorème de la déduction :

si d'une expression f_o on déduit f_i (ce qui s'écrit $f_o \vdash f_i$ alors $f_o \supset f_i$ est un théorème ($\vdash f_o \supset f_i$).

L'application récursive de ces blocs est aussi légitimée par le théorème de la déduction :

si $f_o, f_1, \ldots, f_n \vdash f_{n+1}$ alors $\vdash f_o \supset (f_1 \supset (\ldots (f_n \supset f_{n+1}) \ldots))$

3.2 Généralisation - ∀

<généralisation-∀>::=<introduction>;<liste de lignes>;
<dernière ligne> généralisation-∀

Une généralisation-∀ est un nouveau type de bloc qui a la forme

soit $x_1, x_2, \ldots, x_n$;

n_1 : f_1 <preuve>;

---

n_i : f_i <preuve>;

n_{i+1} : f_{i+1} généralisation-∀

avec f_{i+1} f-identique à $\forall x_1 \forall x_2 \ldots \forall x_n f^*_i$,

où $\begin{cases} f^* = f \text{ si } f \text{ est une formule quantifiée} \\ f^* = (f) \text{ sinon} \end{cases}$

(ceci évite d'introduire des parenthèses superflues dans une suite de quantificateurs).

L'utilisation de la généralisation-∀ est ce qui se fait habituellement lorsqu'on veut démontrer une formule quantifiée universellement : on démontre la formule pour un x quelconque ... comme elle vraie pour un x quelconque elle est vraie pour tout x.En logique formelle on obtient ce résultat en utilisant la règle de généralisation dans le système de Hilbert et Ackermann ou le schéma AES dans le calcul déductif de Gentzen (On verra plus loin comment la structure de bloc fait intervenir les restrictions sur l'emploi de cette règle).

3.3. Généralisation -∃

<généralisation-∃>::=<détermination>;
<liste de lignes>;<dernière ligne> généralisation-∃

<détermination>::= prenons <variable> défini par
<expression> <existence>/
prenons <variable> déjà défini /
prenons <variable> = <objet>

La détermination permet d'introduire une variable possédant certaines caractéristiques. Le premier type de détermination introduit une variable possédant une certaine propriété; dans ce cas il faut s'assurer qu'une telle variable existe, c'est ce qu'on fait par "existence"

<existence>::=<existence><expression><preuve>

L'expression suivant existence est f-identique à l'expression suivant défini par quantifiée existentiellement.

exemple

Prenons x défini par $x \leqslant y$ existence $\exists x(x \leqslant y)$ <preuve>

Le deuxième type de détermination introduit une variable déjà connue ailleurs.

Le troisième type de détermination permet de donner un nom à un objet.

Une généralisation correcte s'écrit donc

Prenons x .....;

n_i : f_i <preuve>

n_{i+1} : f_{i+1} généralisation-$\exists$ avec f_{i+1} f-identique à $\exists$ x f_i^*

C'est ce qu'on fait dans le raisonnement usuel lorsqu'on utilise la quantification pour démontrer une expression quantifiée existentiellement : on choisit un x particulier (qu'on définit à ce moment là ou qu'on a déjà utilisé) et on démontre qu'il vérifie la propriété en uestion. La généralisation -$\exists$ correspond en logique formelle au 6ème axiome de Hilbert et Ackermann ou au schéma EES de Gentzen.

3.4. Structure de bloc

Cette structure de bloc appliquée à la démonstration nous semble être un des traits les plus caractéristiques du langage que nous définissons ici.

Nous avons donc quatre types de blocs en effet ... cqfd, h... déduction, Soit ... généralisation - $\forall$, Prenons ... généralisation-$\exists$ que nous appellerons respectivement b_0, b_1, b_2 et b_3.

D'après les définitions syntaxiques précédentes, ces blocs ne peuvent jamais se chevaucher, mais ils peuvent s'imbriquer les uns dans les autres.

- Portée des définitions et des théorèmes

Les définitions figurant dans la partie utilitaire sont valides dans tout le reste du "programme"

Lorsqu'une définition apparaît dans un bloc, on la considère valide depuis son apparition jusqu'au cqfd qui termine le bloc où elle apparaît.

La portée des théorèmes est définie par les mêmes règles.

- Portée des variables introduites

On appelle variable introduite une variable figurant dans une introduction, ou derrière prenons, ou devant compose dans une définition.

Si une variable est introduite dans la partie utilitaire ou dans l'énoncé, elle est connue dans tout le reste du programme.

Si une variable est introduite en tête d'un bloc b_2 ou d'un bloc b_3 elle est connue seulement dans ce bloc.

Si une variable est introduite devant compose, elle est connue partout où la définition est valide.

Toute expression figurant dans une démonstration devra répondre aux conditions suivantes :

- tous les identificateurs de prédicats doivent figurer dans la partie définie d'une définition valide.
- toutes les variables doivent être soit quantifiées, soit connues; si l'expression en question figure dans un théorème ou dans une définition, les variables de l'expression peuvent être aussi celles figurant dans la "partie définie" de la définition ou dans les paramètres du théorème.

Ces conditions expriment la nécessité de n'utiliser que des identificateurs que l'on connaît, qu'en mathématiques il faut savoir avec quoi on parle s'il n'est pas nécessaire de savoir de quoi on parle! On retrouve des conditions analogues dans un langage de programmation comme Algol où tous les identificateurs doivent être déclarés à moins qu'ils ne soient paramètres formels de procédures.

Les variables introduites en tête des blocs b_2 sont absolument quelconques. Elles doivent donc être différentes entre elles et différentes de celles introduites en tête des blocs b_3 et de celles introduites dans la partie utilitaire ou dans l'énoncé. En d'autres termes elles ne peuvent pas figurer ailleurs. C'est cette condition qu'on retrouve en logique formelle sur l'emploi de la règle de quantification universelle. En tenant compte de la portée des introductions cela nécessite qu'une expression figurant dans un bloc b_2 ne soit pas utilisable à l'extérieur de ce bloc (à l'exception de la dernière où ces variables sont quantifiées), car alors elle contiendrait une variable non introduite. Ceci permet de réutiliser plusieurs fois une même variable dans des blocs disjoints : comme elles sont locales au bloc, ce ne sont en fait pas les mêmes.

Remarque.-On ne pourra obtenir des expressions du type $\forall x(Px \Rightarrow \forall x(Qx))$ (qui ne sont pas correctes dans certains systèmes de logique formelle), si cela nécessite que la même variable x soit introduite deux fois dans des blocs non disjoints.

De même pour éviter qu'une variable introduite par prenons en tête d'un bloc b_3 soit confondue avec une variable introduite dans un bloc b_2, nous poserons sur l'utilisation des expressions de b_3 les mêmes conditions que pour b_2.

L'en-tête de bloc prenons <variable> déjà défini nous permettra de généraliser existentiellement sur des variables figurant à l'extérieur du bloc, ce qui est parfaitement légitime.

Pour les expressions du bloc b_1 on a aussi les mêmes conditions d'utilisations, mais pour d'autres raisons : toute expression, sauf la dernière est déduite de l'hypothèse, ce qu'il faut faire apparaître quand on utilisera cette formule. Nous le faisons apparaître en l'écrivant explicitement (h => c) dans la dernière expression. Ainsi lorsqu'une formule est "déduite sous conditions" (c'est-à-dire à partir de suppositions), ces suppositions, ou prémisses, figurent soit implicitement, soit explicitement, et on passe de l'implicite à l'explicite en fermant le bloc b_1.

A l'extérieur de ce bloc on ne peut utiliser que la formulation explicite.

Nous conviendrons (mais ceci sans justification théorique) qu'une expression ne peut être utilisée hors du bloc b_0 où elle se trouve, ce qui fait qu'une fois une expression démontrée, on n'aura plus besoin de conserver sa démonstration.

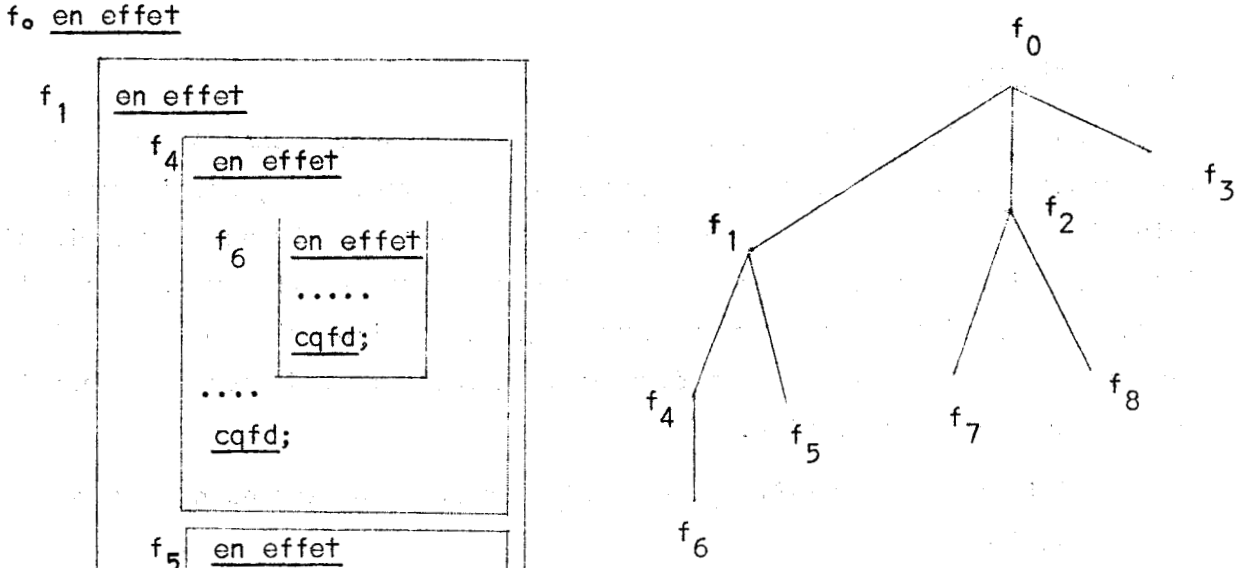
Outre cet intérêt pratique, les blocs b_0 permettent la "structuration" d'une démonstration. Pour un logicien la démonstration est linéaire : on part des axiomes et on aboutit à la formule à démontrer; cette notion de bloc est très peu utilisée en logique [1] et on pourrait fort bien s'en passer dans le langage que nous définissons. Mais il est clair que la structure, l'organisation sont un élément important de toute démonstration.Cette organisation apparaît non seulement par l'utilisation de définitions et de lemmes, mais aussi par le fait qu'à l'intérieur d'une démonstration bien faite on met en évidence les principales étapes du cheminement logique. Une "belle" démonstration n'est pas qu'une démonstration rigoureuse qui satisfait la pensée analytique, mais surtout une démonstration bien construite, bien structurée, et qui s'adresse davantage à la pensée synthétique.

---

(1) La méthode de démonstration par réfutation, dite des tableaux sémantiques de Beth fait apparaître une structure de bloc différente de celle qui apparaît ici. Smullyan dans son article "Analytic natural deduction" (Journal of symbolic logic, juin 65) utilise le bloc h ... déduction.

S'il nous est permis de parler en parabole, disons que pour construire une démonstration, il ne suffit pas de disposer de bons matériaux, il faut encore savoir les agencer intelligemment dans l'édifice. Cet aspect de la démonstration a été peu envisagé et nous osons espérer que ce travail en permettra une meilleure approche expérimentale.

Notons à ce sujet que certains pédagogues se sont attachés à enseigner les mathématiques en mettant en évidence des schémas de démonstration au moyen de graphes. La structure de blocs b_0 que nous utilisons ici est trivialement isomorphe à un graphe arborescent.



Si en plus des blocs b_0, nous considérons aussi les blocs b_1, b_2 et b_3, une démonstration aura une structure plus complexe. Il paraît fort possible que cette structure soit déterminée par la forme de l'expression à démontrer et les axiomes et définitions de la théorie considérée. Si l'on pouvait connaître cette relation de façon plus précise, il est probable que la démonstration automatique pourrait être chose qu'un algorithme combinatoire.

4. Justifications.

Une justification indique comment on a obtenu une expression à partir des formules précédentes repérées par leur étiquette. Ces justifications doivent être régulières, c'est-à-dire que l'expression obtenue est vraie si les expressions desquelles on l'a déduites sont vraies. Il est clair que les expressions repérées par leur étiquette, que pour simplifier nous appellerons expressions étiquetées, doivent être utilisables (cf ci-dessus) à l'endroit où on donne la justification.

Définissons de manière plus précise ce que nous entendons ar expression étiquetée.

Si l'étiquette est suivie d'une expression, (on ne tient pas compte des deux points qui doivent toujours figurer derrière une étiquette) c'est cette expression qui est l'expression étiquetée. Si elle est suivie d'une hypothèse, l'expression étiquetée sera l'expression qui suit le h. Si elle est suivie de prenons <variable> défini par <expression>... ce sera l'expression qui suit défini par. Si elle est suivie de prenons <variable>=<objet> ce sera <variable> =<objet>.Les hypothèses figurant dans la partie utilitaire et dans l'énoncé ne sont pas étiquetées. On les repèrera par deux étiquettes conventionnelles, par exemple 0 et 00.

Nous donnons ci-dessous une liste de justifications. Cette liste n'est ni exhaustive, ni définitive. A l'usage il s'avèrera peut-être que certaines justifications sont inutiles alors qu'il serait intéressant de pouvoir en utiliser d'autres non signalées. Il ne sera pas difficile de modifier cette liste en conséquence.

(J 1)

<justification>::= modus-ponens <étiquette>,<étiquette>

Si la première expression étiquetée est f_1 et la seconde $\overline{f}_1 = \overline{f}_2$, l'expression justifiée est f-identique à f_2

exemple
1 : x ∈ K = x.e = e.x <preuve>;
2 : x ∈ K <preuve>
3 : x.e = e.x par modus-ponens 2,1;

(J 2) <justification>::= conjonction de <liste d'étiquettes>
<liste d'étiquettes>::=<étiquette>/<liste d'étiquettes>,<étiquette>

L'expression justifiée est f-identique à la conjonction des expressions étiquetées, c'est-à-dire à l'expression obtenue en mettant bout à bout ces expressions séparées par des virgules. Il n'y a pas de problème de parenthésage puisque la virgule est l'opérateur de plus faible priorité.

exemple
5 : Hx, Ry <preuve>;
.....
8 : ∀x(Px) <preuve>;
9 : ∀y(Py), Hx, Ry par conjonction de 8,5;

(J 3) <justification>::= cas <liste de nombres entiers> de <étiquette>
<liste de nombres entiers>::=<nombre entier>/
<liste de nombres entiers>, <nombre entier>

Si i, j, k ... sont les nombres entiers de la liste, la formule justifiée est f-identique à la conjonction des $i^{ème}$, $j^{ième}$, $k^{ième}$ ... sous-formules de l'expression étiquetée.

exemple
1 : x ∈ K, y ∈ k, k est un sous-groupe [G,e,T]
x T y = z <preuve>;
....
5 : x T y = z, x ∈ k par cas 4, 1 de 1;
6 : ∀x∀y((x∈E, xRy) => y∈F)
en effet
7 : Soit x,y;
8 : h x ∈ E, xRy;

9 : x R y par cas 2 de 8;

....

Il nous semble évident que ces justifications sont régulières (ce sont des règles, élémentaires ou dérivées, de la logique classique)

(J 4) <justification>::= composition de <liste d'objets> par <opérateur>

La variable considérée est introduite devant compose et représente un opérateur dont la définition est valide. L'expression justifiée est f-identique à l'expression définissante après substitution des objets aux paramètres.

exemples définition U compose [A,B] = def
∀x (x ∈ U(A,B)<=> x ∈ A ou x ∈ B);
....
3 : ∀x(x ∈ U(ρ(A),A) = x ∈ ρ (A) ou x ∈ A)
par composition de ρ(A), A par U;

(J 5) <justification>::= cas <liste de nombres entiers>
de la définition de <étiquette>

L'expression étiquetée est un prédicat défini.

L'expression justifiée est f-identique à la conjonction des $i^{ème}$, $j^{ième}$, $k^{ième}$, ..., sous-formules de l'expression définissant le prédicat (Après avoir substitué aux paramètres les arguments effectifs)

exemple définition x est un inverse y,[E, o,e] = def
x ∈ E, y ∈ E, e est un élément neutre [E,o],
x o y = e, y o x = e;
...
72 : a est un inverse[b, G, T, u] <preuve>;
73 : u est un élément neutre [G, T] par cas 3
de la définition de 72 ;

La régularité de cette justification provient de la régularité de la

justification précédente et du fait qu'un prédicat défini est logiquement équivalent à l'expression définissante. On utilise cette même propriété dans la justification suivante.

(J 6) <justification>::= définition en <liste d'étiquettes>

L'expression justifiée est le prédicat défini par l'expression f-identique à la conjonction des expressions étiquetées (après les substitutions nécessaires)

exemples

définition x est un inverse [y, E, o, e] = def ...
.....
3 : a ∈ E, b ∈ E, u est un élément neutre [E,T],
a T b = u, b T a = u <preuve>
4 : a est un inverse [b,E,T, u] par
définition en 3;

(J 7) <justification>::= application du théorème <identificateur de théorème> <arguments> <référence>

<référence>::=<vide>/<liste d'étiquettes>

On substitue aux paramètres du théorème repéré par son identificateur, les arguments. L'hypothèse (si elle existe) est f-identique à la conjonction des expressions étiquetées. L'expression justifiée est f-identique à la conclusion de ce théorème.

exemple

théorème R1[E,P,R] h ∀x (x∈P => x est une partition de [E],
∀x (x∈r => x est une relation d'équivalence [E] c ∃f (f est une application biunivoque [P,R]);
.....
5 : ∀z (z ∈ k => z est une partition de [G]) <preuve>;
6 : ∀z(z ∈ i => z est une relation d'équivalence [G]) <preuve>;
7 : ∃g (g est une application biunivoque [k,i]) par application du théorème R1[G,K,i] 5,6;

La régularité de cette justification provient de la régularité du modus-ponens et des règles de substitutions.

(J 8) <justification>::= décomposition de <étiquette>

L'expression étiquetée est du type f ou g => h est l'expression justifiée est f => h, g => h.

Nous n'avons pas à nous occuper du parenthésage : f est une disjonction, ainsi que g et h; f => h, g => h est donc une expression bien construite.

(J 9) <justification>::= recomposition de <étiquette>

Cette justification fait exactement l'inverse de la précédente. Là non plus pas de problèmes de parenthésages. L'opérateur principal de f et de g ne pouvant être que le ou, f ou g => h est correcte.

(J 10) <justification>::= implication en <étiquette>

L'expression étiquetée est de la forme a <=> b

L'expression justifiée est alors a => b ou b=> a

(J 11) <justification>::= double implication en <étiquette>

A partir de a => b, b => a on peut écrire a <=> b

(J 12) <justification>::= particularisation de <étiquette> par <liste d'objets>

Tous les objets de la liste doivent être formés de variables connues. Si l'expression étiquetée commence par n quantificateurs universels, la liste d'objets contient moins de n objets.

L'expression justifiée est alors f-identique à l'expression étiquetée à laquelle on a enlevé les premiers quantificateurs et les parenthèses extérieures si la formule n'est plus quantifiée et où les variables correspondantes ont été remplacées par les objets de la liste.

exemple 3 : ∀x∀y∀z((xry, yrz) = xrz) <preuve>;

.....

7 : ∀z((a r b, b r z)) => a r z) par particularisation de 3 par a, b;

Cette règle existe telle quelle en logique formelle.

(J 13) <justification>::= simplification de <étiquette>

Cette justification permet d'enlever les quantificateurs superflus (c'est-à-dire que si on l'enlève, la variable correspondante ne devient pas libre, soit parce qu'elle ne figure pas dans l'expression, soit parce qu'elle est quantifiée par un autre quantificateur) et aussi les parenthèses extérieures si c'est nécessaire.

exemple 1 : ∀i (K est un groupe [e,+]) <preuve>;
2 : K est un groupe [e,+] par simplification de 1;
3 : ∃x∀y ∃x (y est un truc [x]) <preuve>;
4 : ∀u ∃v (u est un truc [v] par simplification de 3;

(J 14) <justification>::= transivité de l'implication <étiquette>, <étiquette>

Cette justification permet d'obtenir $f_1 \Rightarrow f_3$ à partir de $f_1 \Rightarrow f_2$ et de $f_2 = f_3$.

On pourrait obtenir la même chose en utilisant des justifications définies :

désignons par f'_1, f'_2, f'_3 les expressions telles que $\overline{f'_1} = f_1$, $\overline{f'_2} = f_2$ et $\overline{f'_3} \Rightarrow f_3$.

n_1 : $f_1 \Rightarrow f_2$ <preuve>;
n_2 : $f_2 \Rightarrow f_3$ <preuve>;
n_3 : h f'_1;
n_4 f'_2 par modus-ponens n_3, n_1;
n_5 : f'_3 par modus-ponens n_4, n_2;
n_6 : $f_1 \Rightarrow f_3$ déduction;

Cette justification est donc redondante. C'est précisément l'utilisation des règles de déduction redondantes qui distingue le raisonnement naturel de la déduction logique formelle.

(J 15) <justification>::= égalité <étiquette> dans <étiquette>

Nous avons vu que nous pouvons avoir des prédicats élémentaires du type

<objet>=<objet>. Ceci signifie que deux écritures différentes désignent le même objet, et on peut donc substituer l'un par l'autre dans une expression.

La première expression étiquetée est une égalité. L'expression justifiée est f-identique à la seconde expression étiquetée à une substitution près de certaines occurences des objets égaux.

(J 16) <justification>::= identité <étiquette> dans <étiquette>

Cette justification fonctionne de la même façon que la précédente, mais au lieu de substituer des objets égaux, on substitue des expressions logiquement identiques. La première expression étiquetée a donc la forme a<=>b

Quand on fera ces substitutions, il faudra veiller au parenthésage, de façon à ce que les expressions que l'on substitue aient le même opérateur principal, ou plus exactement soient du même type.

exemple 1 : a, b, c <preuve>;
2 : (a, b)<=> d ou <preuve>;
3 : d ou e, c par identité 2 dans 1;

(J 17) <justification>::= contraposition de <étiquette>

L'expression étiquetée est de la forme f => g

L'expression justifiée est de la forme $\tilde{g}$ => $\tilde{f}$

avec $\tilde{g} = \overline{g'}$, si g est de la forme $\sim(g')$

= g' si g est de la forme $\sim g'$, où g' est un prédicat

= $\sim g$ si g est primaire

= $\sim(g)$ si g est une disjonction.

(J 18) <justification>::= l'absurde <étiquette>:<expression>;
<liste de lignes> contradiction

Si l'expression à justifier est f, la première expression qui suit l'absurde est $\hat{f}$,

avec $\hat{f}$= f' si f est de la forme $\sim(f')$, ou de la forme $\sim f'$ où f' est un prédicat

= $\sim f$ si f est un primaire

= $\sim(f)$ sinon

L'expression de la dernière ligne doit être du type g, $\hat{g}$.

Le raisonnement par l'absurde est une règle dérivée de logique formelle qui s'écrit

$\sim x \supset \sim y \wedge y \vdash x$ et qui nécessite le "tiers-exclu"

Ici encore apparaît un nouveau type de bloc dont aucune expression n'est utilisable à l'extérieur pour des raisons bien compréhensibles. Ce type de bloc a une importance bine moins grande que les autres dans la structure d'une démonstration.

Les transformations $g \rightarrow \bar{g}$, $g \rightarrow g^*$, $g \rightarrow \tilde{g}$, $g \rightarrow \hat{g}$ ont toutes le même but, à savoir reparenthéser correctement les expressions obtenues par combinaisons logiques d'autres expressions. Les transformations ne seraient pas nécessaires si les expressions étaient totalement parenthésées, mais alors les expressions auraient une écriture beaucoup plus complexe, leur utilisation serait moins simple dans d'autres cas, et on s'éloignerait de l'écriture habituellement utilisée en mathématiques. Par contre, en mathématiques, les expressions sont souvent insuffisamment structurées et seule une analyse - d'ailleurs très intuitive- du contexte permet de lever les ambiguités.

ex $\forall x, y \in E, x \leqslant y, y \leqslant x$, alors $x = y$ d'où r est antisymétrique et r est une relation d'ordre si $E \subset F$, $\forall F$.

Il est clair que cette formulation est hautement ambigüe.

Les justifications que nous avons signalées sont celles qui apparaissent le plus fréquemment -croyons-nous- dans un raisonnement. Mais cet ensemble de règles n'est pas complet en ce sens qu'il ne permet pas de faire toutes les dérivations logiques qu'il est possible de faire. Par exemple il ne permet pas d'utiliser la définition des connecteurs logiques redondants tels que $\sim A$ ou $B = (A \Rightarrow B)$ ou

$$\sim \exists x\, P(x) \Leftrightarrow \forall x (\sim P(x))$$

Les règles qui nous manquent pour obtenir un système complet apparaissent beaucoup moins souvent et ce serait leur donner trop d'importance que de définir une nouvelle justification pour chacune d'elles. Nous les

grouperons en deux justifications principales; l'une servira quand il faudra traiter l'opérateur de négation

(J 19) <justification>::= négation de <étiquette>

l'autre pour les autres connecteurs logiques

(J 20) <justification>::= calcul logique de <étiquette>.

C'est volontairement que nous ne définissons pas de façon précise leur fonctionnement car nous ne savons pas encore s'il est plus intéressant de les définir sur le modèle des précédentes (mais elles seraient plus complexes) ou d'utiliser des procédures de démonstration automatique rudimentaires. Nous reparlerons plus loin de cette démonstration.

5. Quelques exemples

5.1 Démonstration des axiomes de Hilbert-Ackermann

Pour que ce système de démonstration permette effectivement de démontrer des théorèmes vrais, il ne suffit pas que les justifications soient régulières, il faut qu'il contienne implicitement les axiomes de la logique.

Dans ce qui suit nous noterons les propositions (prédicats sans variables libres) par une lettre majuscule; il n'est pas nécessaire de les définir puisque ce sont des propositions quelconques. Les prédicats à une variable, quelconques eux aussi, seront notés F(x).

```
HA1:  X ou X => X
        en effet 01 : h ∿X;
                 02 : ∿X, ∿X par conjonction de 01,01;
                 03 : ∿(X ou X) par négation de 02;
                 04 : ∿X => ∿(X ou X) déduction;
                 05 :  X ou X => X par contraposition de 04

cqfd;

HA2 : X => X ou Y
   en effet 11 : h ∿(X ou Y);
            12 : ∿X, ∿Y par négation de 11;
```

13 : ∿X, par cas 1 de 12;
14 : ∿(X ou Y) => ∿X déduction;
15 : X => X ou Y par contraposition de 14;

cqfd;

HA3 : X ou Y = Y ou X;
en effet 21 : h∿(Y ou X);
22 :∿y,∿X par négation de 21;
23 :∿y par cas 1 de 22;
24 :∿X par cas 2 de 22;
25 :∿X,∿y par conjonction de 24,23;
26 :∿(X ou Y) par négation de 25;
27 :∿(Y ou X) => ∿(X ou Y) déduction;
28 : X ou Y => Y ou X par contraposition de 27;

cqfd;

HA4 : (X =>Y) => (X ou Z => X ou Y)
en effet 31 : h Z => Y;
32 : Z => X ou Y
en effet 33 : h ∿(X ou Y);
34 : ∿X, ∿Y par négation de 33;
35 : ∿Y par cas 2 de 34;
36 : ∿Y =>∿Z par contraposition de 31
37 : ∿Z par modus-ponens 35,36;
38 : ∿(X ou Y) => ∿Z déduction;
39 : Z => X ou Y par négation de 38
390 : X => X ou Y, Z => X ou Y par conjonction de HA1,39
391 : X ou Z =>X ou Y par recomposition de 390;
392 : (Z => Y) => (X ou X => X ou Y) déduction

cqfd;

HA5 : ∀x(Fx)) => F(y) [On suppose y déjà introduit]

en effet 41 : h ∀x (F(x));

42 : F(y) par particularisation de 41 par y;

43 : ∀x (F(x)) => F(y) déduction

cqfd;

HA6 : F(y) =>∃x (F(x))

en effet 51 : h F(y);

52 : ∃x (F(x))

en effet 53 : prenons y déjà défini;

54 : F(y) par cas 1 de 51;

55 : ∃x (F(x)) généralisation-∃

cqfd;

56 : F(y) => ∃x(F(x)) déduction

cqfd;

5.2. Permutation de quantificateurs

∃x∀y(P(x,y)) => ∀y∃x(P(x,y))

en effet 01 : h ∃x∀y(P(x,y));

02 : Prenons x défini par ∀y (P(x,y)) existence ∃x∀y(P(x,y)) par cas 1 de 01;

03 : Soit y;

04 : Prenons x déjà défini;

05 : P (x,y) par particularisation de 02 par y;

06 : ∃x(P(x,y)) généralisation-∃;

07 : ∀y∃x(P(x,y)) généralisation-∀;

08 : ∃x∀y∃x(P(x,y)) généralisation-∃;

09 : ∀y∃x(P(x,y)) par simplification de 08;

10 : ∃x∀y (P(x,y)) => ∀y∃x(P(x,y)) déduction

cqfd;

5.3. D'après Lewis Carrol

a) h ∀x (x est un canard => ∿x est un danseur),
∀x (x est un officier => x est un danseur),
∀x (x est une volaille => x est un canard);
∀t (t est une volaille => ∿t est un officier)
en effet

1 : Soit t;
2 : h t est une volaille;
3 : ∀x (x est une volaille => x est un canard) par cas 3 de 00;
4 : t est une volaille => t est un canard par particularisation de 3 par t;
5 : t est un canard par modus-ponens 2,4;
6 : ∀x (x est un canard => ∿ x est un danseur) par cas 1 de 00
7 : t est un canard => ∿T est un danseur par particularisation de 6 par t;
8 : ∿t est un danseur par modus-ponens 5,7;
9 : ∀x (x est un officier => x est un danseur) par cas 2 de 00;
10 : t est un officier => t est un danseur par particularisation de 9 par t;
11 :∿t est un danseur => ∿t est un officier par contraposition de 10;
12 :∿t est un officier par modus-ponens 8,11;
13 : t est une volaille => ∿t est un officier déduction;
14 : ∀t (t est une volaille =>∿t est un officier)
généralisation-∀

cqfd ;

b) h ∀x (x est un cheval => x est un animal);
∀x(∃y(y est un cheval, x est une tête de [y]))=>
y(y est un animal, x est une tête de [y]))
en effet

1 : Soit x; 2 : h ∃y(y est un cheval, x est une tête de [y]));
3 : ∃y(y est un animal, x est une tête de [y]))

en effet

4 : Prenons y défini par y est un cheval, x est une tête de [y]
existence ∃y(y est un cheval, x est une tête de [y])
par cas 1 de 2;

5 : y est un cheval par cas 1 de 4;

6 : x est une tête de [y] par cas 2 de 4;

7 : y est un cheval => y est un animal par particularisation de
00 par y;

8 : y est un animal par modus-ponens 5,7;

9 : y est un animal, x est une tête de [y] par conjonction de 8,6;

10 : ∃y (y est un animal, x est une tête de [y])généralisation-∃
cqfd;

11 : ∃y (y est un cheval, x est une tête de [y]) =>
∃y (y est un animal, x est une tête de [y]) déduction;

12 : ∀x (∃y(y est un cheval, x est une tête de [y]) =>
∃y (y est un animal, x est une tête de[y])) généralisation-∀
cqfd

Chapitre II. - Compléments et suppléments

Pages

COMPLEMENTS et SUPPLEMENTS

Dans le premier chapitre nous avons vu les caractères généraux du langage. Les justifications que nous avons données sont d'ordre purement logique et ne permettent pas de traiter le formalisme habituel courant.

Nous allons d'abord rajouter à ce langage ce qui permettra de traiter la théorie des ensembles.

1. Formalisme ensembliste

1.1 Ensembles

<objet>::=<ensemble>

<ensemble>::= {<liste d'objets>}/ Ø /

{<variable>: <expression>}

L'expression servant à définir un ensemble ne peut contenir comme seule variable non connue que celle qui figure avant les deux points.

Nous savons par la théorie des ensembles que pour qu'une relation puisse effectivement définir un ensemble (formé des objets qui vérifient cette ralation) il faut qu'elle soit collectivisante. En particulier il suffit que les objets qui vérifient la propriété appartiennent à un ensemble déjà connu. Il nous semble que c'est toujours le cas puisqu'on ne se préoccupe pratiquement jamais de cette question. Si à l'usage il se révélait nécessaire d'en tenir compte on pourrait définir dans ce cas un ensemble par {<variable> ϵ <ensemble>:<expression>}

1.2. Relation d'appartenance

Le signe ϵ fait partie des symboles de base. C'est un symbole relationnel qui permet d'obtenir des prédicats élémentaires.

<prédicat élémentaire>::=<objet>ϵ<objet>)

(Le signe = permet aussi d'obtenir des prédicats élémentaires :

<prédicat élémentaire>::=<objet>=<objet>

En fait il ne nous paraît pas nécessaire de préciser que l'objet suivant ϵ est du type "ensemble" car c'est insuffisant pour s'assurer qu'on a bien le droit d'écrire x ϵ X (par exemple; si x ϵ Y et X = $\mathcal{P}$(Y), X est bien

un ensemble et x ∈ X n'est pas correcte). Nous reviendrons plus loin sur cette question.

1.3. Justifications

Lorsqu'un objet appartient à un ensemble défini par une relation, il est évident que cet objet vérifie la relation. Réciproquement, si un objet vérifie une relation, il appartient à l'ensemble défini par cette relation. Enfin il nous est apparu qu'on avait rarement des expressions de la forme y ∈ {y : P(y)} mais beaucoup plus souvent Y = {y : P(y)} ... y ∈ Y

C'est pourquoi quand on se réfèrera à un ensemble, la première expression étiquetée sera toujours de la forme <variable>=<ensemble>

<justification>::= appartenance à l'ensemble <étiquette>,<étiquette>

De X = {x : Px} et y ∈ X on déduit Py.

<justification>::= définition de l'ensemble <étiquette>,<étiquette>

De X = {x : Px} et Py on déduit y ∈ X

exemple

1 : Prenons X = {x : Px};
2 : y ∈ X <preuve>;
3 : Py par appartenance à l'ensemble 1,2;
4 : y ∈ x par définition de l'ensemble 1,3;
.....

Si on a effectivement une expression y ∈ {x : Px} on pourra se ramener au cas précédent en ouvrant un bloc Prenons X ={x : Px} puis on substituera X à {x : Px}, ou l'inverse suivant la justification, ensuite on généralisera existentiellement sur X et on pourra enlever ce quantificateur puisque X ne figurera pas dans l'expression quantifiée.

exemple

1 : x ∈ {y : Py} <preuve>;
2 : Prenons Y = {y : Py};
3 : x ∈ Y par égalité 2 dans 1;
4 : Px par appartenance à l'ensemble 2,3;

5 : ∃Y(Px) généralisation-∃

6 : Px par simplification de 5;

dans l'autre cas

1 : Px ‹preuve›;

2 : Prenons Y = {y : Py};

3 : x ∈ Y par définition de l'ensemble 2,1;

4 : x ∈ {y : Py} par égalité 2 dans 3;

5 : ∃Y(x ∈{y : Py}) généralisation-∃

6 : x ∈ {y : Py} par simplification de 5;

Encore une fois si notre hypothèse se révélait fausse, il conviendrait de modifier en conséquence ces justifications.

1.4. Théorèmes

Il est commode aussi, pour éviter d'avoir un trop grand nombre de justifications, de raisonner sur les ensembles en utilisant des théorèmes

exemple

théorème égalité d'ensembles [A,B] h ∀x (x∈A<=>x∈B) c A = B ;

2. Applications - Suites - N-Uplets

On peut considérer une application comme un opérateur en ce sens que si f est une application et x un objet, f(x) est encore un objet qui possède certaines propriétés particulières. Mais ici on s'intéresse davantage à l'application elle-même qu'à sa valeur.

On utilisera donc le prédicat défini par :

définition f est une application [E,F] = def

∀x(x∈E => f(x) ∈ F),

∀x(x∈E => ∃! y(y = f(x)));

Ce prédicat sera spécial puisque non seulement il donne la définition de l'application mais il permet d'écrire f(x). On ne le fera donc pas figurer dans les définitions, on le supposera défini dans le métalangage.

Les deux prédicats suivants sont dans le même cas

définition X est une suite [I,E] = def
$\forall i(i \in I \Rightarrow X(i) \in E)$
$\forall i(i \in I \Rightarrow \exists! a(a = X(i)))$;

Ce prédicat est le même que le précédent mais avec la différence que I doit obligatoirement être un ensemble de nombres entiers.

X est un n-uplet [p]
nous assure que pour tout entier i compris entre 1 et p on peut écrire X(i)

exemple

définition X compose E,F = def
$\forall x(x \in X(E,F) \Leftrightarrow$
(x est un n-uplet [2], $x(1) \in E$, $x(2) \in F$));

3. Traitement du quantificateur $\exists!$

Le quantificateur $\exists!$ ne fait pas partie du formalisme logique classique mais apparaît fréquemment en mathématiques. L'expression
$\exists!x(Px)$ est une abréviation pour
$\exists x(Px)$, $\forall x \forall y$ ((Px,Py) => x = y);

On a donc $\exists!x(Px) \vdash \exists x(Px)$
et pour éviter une justification spéciale pour faire cette dérivation nous poserons les deux règles suivantes :

1. Si une expression figurant devant en effet contient $\exists$ et que l'expression devant cqfd lui est f-identique avec en plus $\exists$ remplacé par $\exists!$ la démonstration est correcte.

2. Chaque fois qu'on applique une justification on peut remplacer $\exists!$ par $\exists$ dans l'expression justifiée.

exemple 1 : Prenons x défini par Px<existence>
$\exists x$ (Px) en effet $\exists!$ (Px) <preuve>
cqfd;.....

2 : y e E, z e E, ∃!x(x≤ y, x≤ z) <preuve>;
3 : ∃x (x ≤ y, x ≤z) par cas 3 de 2;

Il est bien évident qu'on ne peut pas faire l'inverse car alors on ajouterait des propriétés sans les avoir démontrées.

<justification>::= unicité en <étiquette>

Cette justification permet d'écrire ∀x∀y((Px,Py) => x = y) à partir de ∃!x(Px)

Il n'y a pas ici de problèmes de parenthésages mais il faudra faire attention à ce que la nouvelle variable y ne figure pas dans Px car alors on quantifierait une variable qui doit être libre.

Réciproquement, à partir de ∃x(Px) et ∀x∀y((Px,Py) => x = y) on peut déduire ∃!x(Px) par
<justification>::= existence unique en <étiquette>,<étiquette>

Il n'est pas possible d'utiliser un bloc particulier pour ce quantificateur parce que l'unicité ne se conserve pas forcément au cours d'une dérivation

∃x(Px), ∀x (Px => Qx) ⊢— ∃x (Qx)

mais de ∃!x(Px), ∀x (Px => Qx) on ne peut pas déduire ∃! x (Qx)

De même ∃x (Px,Qx) ⊢— ∃x(Px, ∃x (Qx)

∃! x (Px,Qx) ∃! x(Px),∃! (Qx)

4. Réduction du nombre de lignes d'une démonstration

Bien que l'utilisation des justifications, qui sont pratiquement des règles dérivées de la logique formelle permette de réduire le nombre de pas nécessaires à une démonstration (c'est-à-dire d'étapes permettant de passer d'une expression à une autre en appliquant une seule règle), une démonstration formalisée est encore très longue proportionnellement à sa longueur lorsqu'elle est écrite en langue naturelle. Il faut donc encore condenser l'écriture formalisée, réduire le nombre de pas tout en évitant d'augmenter le nombre de justifications.

Une première méthode pour réduire la longueur d'une démonstration consiste à faire ce qu'on fait dans la pratique : ne signaler que les points importants de la démonstration, l'intuition du lecteur faisant le reste. Nous reviendrons au chapitre suivant sur cet aspect de la question. Pour le moment nous nous intéressons à un lecteur dénué d'intuition (une machine!) à qui il faut préciser tous les détails.

4.1 Condensation

Nous avons vu dans le chapitre précédent que certaines justifications avaient un fonctionnement relativement complexe. Au paragraphe précédent nous avons signalé que tout une partie du traitement du quantificateur $\exists!$ se faisait de façon implicite lorsqu'on testait la f-identité de 2 formules. De même, certaines justifications en utilisent d'autres implicitement (par exemple application du théorème utilise la conjonction)

On peut utiliser le même principe pour d'autres justifications s'il se trouve que plusieurs justifications sont très souvent groupées. En particulier chaque fois qu'on applique la généralisation, on appliquera aussi la simplification ce qui évitera de généraliser sur des variables qui n'existent pas dans l'expression.

4.2. Générateurs d'expressions

On peut considérer la plupart des justifications comme des applications univoques de $\mathcal{E}^*$ dans $\mathcal{E}$ où $\mathcal{E}$ est l'ensemble des expressions ; et $\mathcal{E}^* = \emptyset \cup \bigcup \mathcal{E}^n$. $n > 0$. On pourra donc composer ces applications, et au lieu d'obtenir une expression en plusieurs lignes, on pourra l'obtenir en une seule ligne au moyen de justifications composées

exemple au lieu de

3 : $x \leq y \Rightarrow x \in H$, $x \leq y$, $y \in H$ <preuve>;
4 : $x \leq y$ = $x \in H$ par cas 1 de 3;

5 : x ≤ y par cas 2 de 3;

6 : x ∈ H par modus-ponens 5,4;

nous écrirons

3 : x ≤ y => x ∈ H, x ≤ y, y ∈ H <preuve>;

4 : x ∈ H par modus-ponens (cas 2 de 3), (cas 1 de 3);

Pour définir rigoureusement une justification composée nous allons modifier légèrement la grammaire du langage de la façon suivante :

Nous allons grouper les justifications en deux, celles qui sont univoques et celles qui ne le sont pas. Les seules justifications multivoques définies sont : égalité ...en
identité ...en
implication en
calcul logique
négation de

c'est-à-dire celles qui à partir des mêmes expressions permettent d'obtenir plusieurs expressions différentes (1)

<justification>::=<justification univoque>/
<justification multivoque>

A une étiquette est associée univoquement une expression(cf I.4) et nous posons

<générateur d'expression>::= <étiquette>/
(<justification univoque>)

et dans la définition syntaxique des justifications nous remplaçons partout étiquette par générateur d'expression .

Dans l'explication du fonctionnement de ces justifications nous ne nous réfèrerons plus aux expressions étiquetées mais aux expressions générées. Dans le cas particulier où le générateur d'expression est une étiquette, on retrouvera la même chose que précédemment.

L'utilisation des justifications composées permet de réduire le nombre de lignes d'une démonstration en ne faisant pas figurer certaines expressions

(1) Nous ajouterons aussi à cette liste la justification définition en (qui n'est pas multivoque) parce qu'il vaut mieux faire figurer explicitement le prédicat défini pour le retrouver plus facilement dans la liste de définitions, et la justification l'absurde ... contradiction qui est formée de façon différente.

intermédiaires, mais il reste indispensable de signaler toutes les étapes du raisonnement qui permet d'obtenir une expression. La justification composée est donc une simplification de l'écriture d'une démonstration qui nous rapproche davantage de l'écriture habituelle d'une démonstration où, même si on précise la règle utilisée, on n'écrit pas forcément le résultat (ex. en appliquant le théorème x puis le théorème y et en utilisant la définition z on obtient...). Mais elle n'est pas une simplification de la démonstration elle-même.

5. Génération de théorèmes

Lorsqu'on a démontré un théorème, il est important de pouvoir conserver son énoncé pour pouvoir le réutiliser sans être obligé de le redémontrer à chaque utilisation. C'est pour avoir cette possibilité que nous introduisons une nouvelle extension au langage. Cette même extension nous permettra aussi de démontrer plusieurs théorèmes dans un même "programme".

<programme>::= <partie utilitaire> <liste de travaux>

<liste de travaux>::=<travail>/<liste de travaux>;<travail>

<travail>::=<énoncé><démonstration>

Pour conserver un théorème que l'on vient de démontrer, on utilisera l'instruction

conserver <identificateur de théorème>

qui sera placé entre le cqfd qui termine la démonstration et le point virgule qui suit ce cqfd (si le programme n'est pas terminé)

Plus rigoureusement il faut dans la définition syntaxique de la démonstration (I.3) remplacer cqfd par <fin de démonstration> avec

<fin de démonstration>::= cqfd / cqfd conserver <identificateur de théorème>

L'instruction conserver identificateur rangera dans la liste de théorèmes la séquence

théorème <identificateur> <paramètres> <hypothèse> <conclusion>

où la conclusion est c suivie de l'expression qu'on a démontrée et les paramètres et l'hypothèses sont respectivement les variables et les hypothèses qui ont servi pour la démonstration.

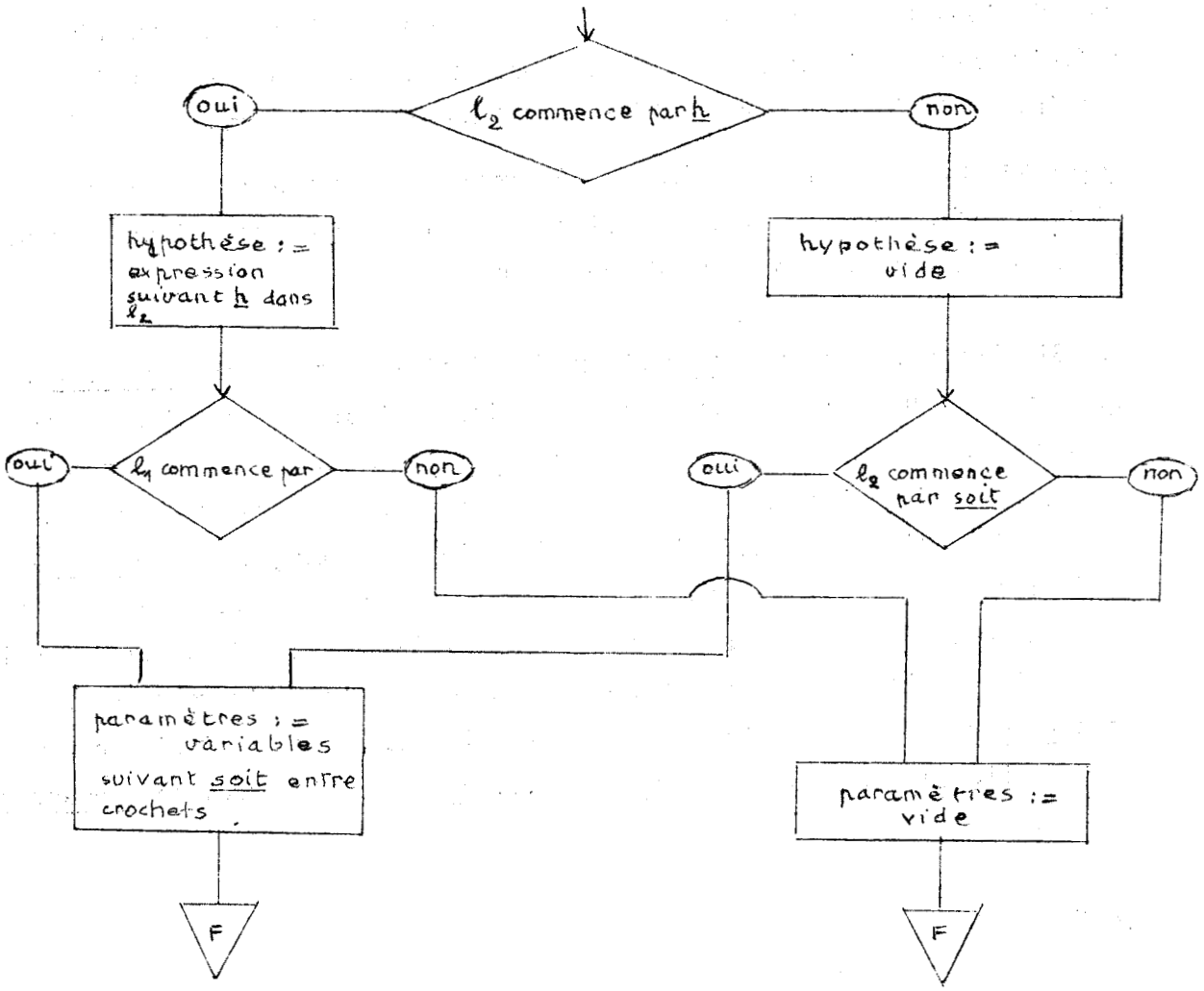
De façon plus précise il faut considérer 2 cas selon que l'instruction

conserver figure ou non à l'intérieur d'un bloc en effet ... cqfd (ce qui est facile à déterminer)

1. Si elle figure au plus haut niveau les paramètres sont les variables figurant dans l'introduction de l'énoncé. Pour les retrouver on utilise l'algorithme suivant :

on désigne par l_1 et l_2 les deux séquences entre ; précédent l'expression à démontrer

; XXXXX ; XXXXX ; expression en effet .... cqfd conserver



On supprime ensuite celles des séquences I_1 et I_2 qui ont servi pour former les paramètres et l'hypothèse du théorème.

Il n'y a pas de différences entre les introductions et hypothèses figurant dans l'énoncé d'un théorème à démontrer et celles figurant dans la partie utilitaire. Mais pour que l'algorithme précédent nous donne les axiomes de la partie utilitaire il faudrait que le théorème à démontrer n'ait pas d'hypothèses, pas de variables introduites et que la liste de théorèmes soit vide, ce qui est hautement improbable. Pour que l'on obtienne les variables introduites dans la partie utilitaire il faudrait des conditions encore plus restrictives (pas de variables introduites, pas d'axiomes, de théorèmes et de définitions). Ceci est pratiquement impossible. Pour plus de sécurité il est possible de faire figurer un théorème "bidon" dans la liste de théorèmes.

remarque si plusieurs théorèmes à démontrer figurent dans le même programme l'étiquette conventionnelle 00 désigne toujours l'hypothèse figurant dans l'énoncé du théorème en cours de démonstration.

2. Si l'instruction conserver figure à l'intérieur d'un bloc en effet ... cqfd, on prendra comme paramètres toutes les variables introduites entre l'expression à démontrer et le premier en effet qui précède, et comme hypothèse, la conjonction de toutes les hypothèses et de toutes les expressions servant à préciser une variable (dans les lignes commençant par prenons). Variables et hypothèses seront rangées dans l'ordre où elles apparaissent dans la démonstration. Comme en rangeant le théorème on fait figurer explicitement les variables et les hypothèses desquelles dépend la conclusion, conserver ferme tous les blocs correspondants. Le théorème ainsi conservé n'est valide que dans le bloc en effet ... cqfd où l'instruction conserver figure.

exemple

```
1 : P(x,y,z)
en effet
2 : Soit u; 3 : h Q(u);
4 : Prenons v = Op(u);
5 : R(u,v) en effet
```

```
..... cqfd conserver truc;  théorème truc [u,v]
      génère                h Q(u), v = Op(u)
                            c R(u,v)
17 : Soit α, β; 18 : h Q (α),  β= Op(α);
19 : R (α,β) par application du théorème truc [α,β] 18;
.....
```

Remarque 1 :

A cause de la façon dont on prend les paramètres et les hypothèses d'un théorème à conserver, toutes les démonstrations de ces théorèmes doivent figurer en tête de bloc, sinon on risque de perdre des hypothèses et des introductions utiles par la suite

Remarque 2

Lorsqu'une ligne du type prenons x défini par Px .... ou prenons x = y figure dans la partie d'une démonstration où on va prendre les paramètres et les hypothèses d'un théorème à conserver, on fermera le bloc ouvert par cette ligne sans avoir généralisé existentiellement sur la variable introduite (en d'autres termes, on ne s'est intéressé qu'au fait que cette variable possédait une certaine propriété). On pourra donc remplacer

```
n : prenons x défini par Px
                                  par
n : Soit x; n bis : h Px;
et  n : prenons x = y
                                  par
n : Soit x, nbis : h x = y;                (1)
```

(1) On ne tiendra pas compte d'une variable figurant dans prenons variable déjà définie puisqu'on la retrouvera ailleurs

Dans la pratique on retrouve la même chose : "prenons le triangle ABC tel que AB = AC et montrons que $\hat{B} = \hat{C}$" peut être remplacé par "tout triangle ayant deux côtés égaux a les angles opposés égaux"

6. Ecemples

6.1 L'ensemble des points d'un espace topologique est un ouvert.

Soit voisinage, C;[1]

définition x est un point = def ∃F (F est un voisinage, x ∈ F);

définition F est un voisinage de [x] = def F est un voisinage, x ∈ F;

définition F est une classe de points = def ∀z(z ⊂ F=>z est un point);

définition x est un point intérieur à [F] = def F est une classe de points, ∃G(G est un voisinage de [x] , G ∈ F;

définition F est un ouvert = def F est une classe de points,
∀x(x ∈ F => x est un point intérieur à [F]);

définition ∩ compose [A,B] = def ∀x(x ∈ ∩(A,B)<=> (x ∈ A, x ∈ B));

h $\forall x \forall F_1 \forall F_2$((($F_1$ est un voisinage de [x], F_2 est un voisinage de x =>
∃G(G ∈ ∩(F_1,F_2),
G est un voisinage de [x])),
∀x∀y((x est un point, y est un point,∿x = y) =>
∃F ∃G(F est un voisinage de [x], G est un voisinage de [y],
∩(F,G) = ∅));

théorème de l'ensemble vide [E] ⊂ E = ∅ <=> ∿ ∃x (x ∈ E);

théorème définition de l'inclusion [A,B] ⊂ A C B <=>
∀x (x ∈ A => x ∈ B);

Soit P; h P={x : x est un point};

P est un ouvert

en effet

1 : P est une classe de points

---

(1)
- La façon d'introduire les identificateurs élémentaires d'une théorie sera discutée au chapitre III
- Axiomatique et définitions tirées de R. Carnap :"Introduction to symbolic logic and its applications" - New York 1958.

en effet

11 : Soit x;

12 : h x ∈ P;

13 : x est un point par appartenance à l'ensemble 00,12;

14 : x ∈ P => x est un point déduction;

15 : ∀x(x ∈ P => x est un point) généralisation-∀;

16 : P est une classe de points par définition en 15

cqfd;

2 : ∀x (x ∈ P => x est un point intérieur à [P])

en effet

21 : Soit x;

22 : h x ∈ P;

23 : x est un point intérieur à [P]

en effet;

231 : Prenons F défini par F est un voisinage, x ∈ F existence ∃F (F est un voisinage, x ∈ F) en effet 2310 : x est un point par appartenance à l'ensemble 00,22; 2311 : ∃F (F est un voisinage, x ∈ F) par cas 1 de la définition de 2310 cqfd;

232 : F est un voisinage de [x] par définition en 231;

233 : F ⊂ P

en effet

2331 : Soit z;

2332 : h z ∈ F;

2333 : prenons F déjà défini;

2334 : F est un voisinage, z ∈ F par conjonction de (cas 1 de 231), 2332;

2335 : ∃F (F est un voisinage, z ⊂ F) généralisation-∃;

2336 : z est un point par définition en 2335;

2337 : z ∈ P par définition de l'ensemble 00,2336

2338 : z ∈ F => z ∈ P déduction;

2339 : ∀ z (z ∈ F => z ∈ P) généralisation-∀;

2340 : F ∈ P <=> ∀x(x ∈ F<=> x ∈ P) par application du théorème définition de l'inclusion [F,P];

2341 : ∀x(x ∈ F => x ∈ P) => F ⊂ P par implication en 2340;

2342 : F ∈ P par modus-ponens 2239,2341

cqfd;

234 : F est un voisinage de [x] , F ⊂ P par conjonction de 232,233;

235 : ∃F (F est un voisinage de [x], F ⊂ P) généralisation-∃;

236 : P est une classe de points, ∃F (F est un voisinage de [x], F ⊂ P) par conjonction 1, 235;

237 : x est un point intérieur à [P] par définition en 236

cqfd;

238 : x ∈ P => x est un point intérieur à [P] déduction;

239 : ∀x (x ∈ P => x est un point intérieur à [P]) généralisation-∀

cqfd;

3 : P est un ouvert par définition en (conjonction de 1,2);

cqfd;

6.2 La relation d'ordre strict est transitive.

définition r est une relation d'ordre [E] = def

∀x (x ∈ E => x r x);

∀x∀y((x ∈ E, y ∈ E, x r y, y r x) => x = y))

∀x∀y∀z((x ∈ E, y ∈ E, z ∈ E, x r y, y r z) => x r z);

définition $\bar{r}$ est un ordre strict associé à [r,E] = def

r est une relation d'ordre [E],

∀x∀y((x ∈ E, y ∈ E) => (x $\bar{r}$ y<=> ∀x r y, ¬x = y)));

Soit E, r, $\bar{r}$; h $\bar{r}$ est un ordre strict associé à [r, E];

∀x∀y∀z((x ∈ E, y ∈ E, z ∈ E, x $\bar{r}$ y, y $\bar{r}$ z) => x $\bar{r}$ z);

en effet

1 : Soit x, y, z;

2 : h x ∈ E, y ∈ E, z ∈ E, x $\overline{r}$ y, y $\overline{r}$ z;

3 : (x ∈ E, y ∈ E, z ∈ e, x r y, y r z) => x r z par particularisation de
(cas 3 de la définition de (cas 1 de la définition de 00))
par x, y, z;

4 : ∀x∀y((x ∈ E, y ∈ E) => (x $\overline{r}$ y <=> (x r y, ∿x = y)))
par cas 2 de la définition de 00;

5 : x $\overline{r}$ y = (x r y, ∿x = y) par implication en
(modus-ponens (cas 1,2 de 2),
(particularisation de 4 par x,y));

6 : y $\overline{r}$ z => (y r z, ∿y = z) par implication en
(modus-ponens (cas 2,3 de 2), (particularisation de 4 par y,z));

7 : (x r z, ∿x = z) => x $\overline{r}$ z par implication en
(modus-ponens(cas 1,3 de 2), (particularisation de 4 par x,z));

8 : x r y, ∿x = y par modus-ponens (cas 4 de 2),5;

9 : y r z, ∿y = z par modus-ponens (cas 5 de 2), 6;

10 : x r z par modus-ponens (conjonction de (cas 1,2,3 de 2),
(cas 1, de 8), (cas 1 de 9)), 3;

11 : ∿x = z par l'absurde 111 : x = z;
112 : z r y par égalité 111 dans
(cas 1 de 8);
113 : y = z par modus-ponens
(conjonction de (cas 2,3 de 2), (cas 1 de 9),112),
(particularisation de (cas 2 de la définition de
(cas 1 de la définition de 00)) par y,z);
114 : y = z, ∿y=z par conjonction 113,
(cas 2 de 9) contradiction;

12 : x $\overline{r}$ z par modus-ponens (conjonction de 10,11), 7;

13 : (x ∈ E, y ∈ E, z ∈ E, x $\overline{r}$ y, y $\overline{r}$ z) => x $\overline{r}$ z déduction;

14 : ∀x∀y∀z((x ∈ E, y ∈ E, z ∈ E, x $\overline{r}$ y, y $\overline{r}$ z) => x $\overline{r}$ z))
généralisation-∀

cqfd;

Chapitre III.- Possibilités d'amélioration

Pages

Chapitre III

POSSIBILITES d'AMELIORATION

A part quelques points encore incomplètement précisés, tels que l'introduction des prédicats élémentaires d'une théorie (ex x est un voisinage en topologie), et dont la formalisation complète dépend étroitement de la façon dont le langage sera traité en machine, le langage que nous avons défini permet effectivement de formaliser des démonstrations d'une manière qui se rapproche davantage de la pratique courante.

A titre d'exemple voici deux démonstrations de la proposition suivante :

soient deux opérations + et . telles que

1) $x + x.y = x$ pour tous x et y.
2) $0 + x = x$ pour tout x

alors $0.x = 0$ pour tout x

La première démonstration utilise la méthode de réfutation de Robinson.

1) ∿P x y u, ∿S x u v, E x v
2) S o x x
3) P o b c
4) ∿E o c
5) ∿S o c v, E o v 3, 1; o/x, b/y, c/u
6) E o c 5,2; c/x, c/v
7) □ 4,6

La seconde démonstration utilise le langage défini précédemment

Soit +, x,0; h ∀x∀y (x + x y = x),
∀x(0+ x = x);
∀x (0 x = 0)
en effet
1 : Soit y;
2 : 0 + 0 y = 0 par particularisation de (cas 1 de 00) par 0,y);
3 : 0 + 0 y = 0 y par particularisation de (cas 2 de 00) par 0 y;
4 : 0 y = 0 par égalité 3 dans 2;
5 : ∀y (0 y = 0) généralisation-∀
cqfd

Cependant ce langage ne nous permet pas encore de formaliser toutes les possibilités d'écriture et de raisonnement dont dispose le mathématicien. C'est ce que nous allons examiner dans les deux parties suivantes.

1. Aspects à développer

1.1. Notion de f-identité

La notion de f-identité est très restreinte puisqu'il faut que deux expressions soient écrites de la même façon , à l'exception des variables quantifiées, ce qui est vraiment le moins qu'on puisse exiger.

A l'autre extrémité nous avons l'identité logique qui n'est pas utilisable non plus puisque la question de savoir si deux expressions du calcul des prédicats sont identiques est indécidable. (Sinon le problème de la démonstration automatique serait résolu : si T est un théorème, T ≡ (a⊃a)!)

Entre les deux, nous pouvons utiliser l'identité de calcul des propositions (le calcul des propositions est inclus dans le calcul des prédicats). Ici le problème est décidable et l'on connaît plusieurs méthodes pour le résoudre dont la plus élémentaire est de former les tables de vérité.

Utiliser l'identité propositionnelle nous permettant de supprimer la règle calcul logique ne nous permettrait pas de tester des identités aussi simples que $x(Px)$ $Vx(Px)$.

D'autre part, il ne faut pas perdre de vue le principe qui nous guide : se rapprocher autant que possible du travail d'un mathématicien. Nous considérons comme identiques deux expressions qu'un mathématicien considère comme telles.

C'est pourquoi nous pensons qu'il conviendrait soit de définir un système formel générant à partir d'une expression donnée un ensemble d'expressions qui lui serait identiques et tel que le problème de l'appartenance à une classe d'équivalence soit décidable, soit de trouver un algorithme permettant de tester l'identité "restreinte" de deux expressions.

Il faudrait naturellement que ces méthodes soient simples et rapides pour qu'elles soient réellement intéressantes, et d'autre part qu'elles recouvrent au moins les justifications négation de et calcul logique.

Signalons qu'il nous semble qu'il y a deux points de vue pour aborder cette question : un point de vue logique si l'on définit à l'intérieur du calcul logique une relation d'identité restreinte; l'autre point de vue probablement plus fécond, et plus général que le précédent, est syntaxique, et revient à considérer cette question comme cas particulier du problème du mot.

1.2. Déclarations de types

Nous avons signalé déjà qu'il se posait certains problèmes relatifs à l'instruction d'identificateurs qui ne sont ni des identificateurs de prédicats définis ni des variables ou des opérateurs préfixés. Dans certains cas ces problèmes ne se posent qu'en fonction de la façon dont on

programme sera traité en machine, dans d'autres ces problèmes sont liés au formalisme et il convient de les regarder de plus près.

a) prédicats élémentaires

Ils sont de deux sortes

1 - <variable> est un <identificateur de prédicats> <paramètres> mais où le prédicat en question est un prédicat de base dans la théorie et n'a pas à être défini

2 - <objet> <identificateur> <objet> où l'identificateur représente un symbole relationnel de la théorie; son apparition dans des définitions ou des théorèmes permet de le traiter.

Les identificateurs "voisinage" et "c" figurant dans l'exemple 11.6.1 correspondent respectivement à ces deux cas.

On peut les introduire, comme nous l'avons fait par soit ce qui revient simplement à signaler que ce ne sont pas des identificateurs inconnus (ou lors du passage en machine, à leur attribuer un code interne).

Si l'on exécute une analyse syntaxique plus précise il est probable qu'il faille être plus rigoureux. Pour le prédicat de base, puisqu'il n'a pas de définitions, il semble naturel de le ranger dans la liste des définitions avec une partie définissante vide :

exemple définition x est un voisinage = def;

Le symbole relationnel peut, lui, être utilisé comme variable (cf la relation d'ordre en 11.6.2) et doit donc nécessairement être introduit par soit mais on peut rajouter dans les hypothèses un "méta-prédicat" indiquant la nature de cet identificateur.

exemple Soit C ; h C est un symbole relationnel;
ou
définition r est une relation d'ordre [E] = def
r est un symbole relationnel, ...;

Cette utilisation de méta-prédicats nous semble intéressante car on pourrait l'utiliser aussi dans les prédicats "est une application","est un n-uplet" où le méta-prédicat

"est un symbole fonctionnel"

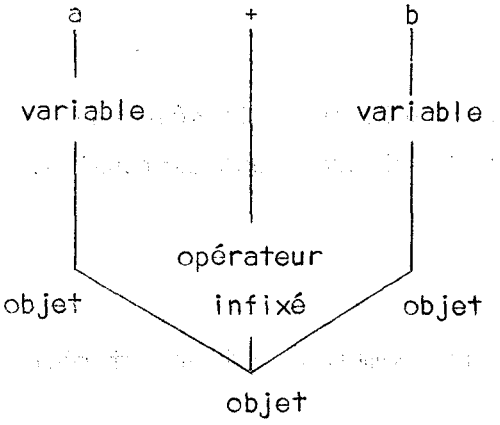
permettrait d'écrire une séquence du type f(n). Le prédicat "est une application" pourrait alors figurer dans la liste de définitions :

définition f est une application [E,F] = def
f est un symbole fonctionnel,
∀x(x ∈ E => f(x) ∈ F),
∀x(x ∈ E => ∃!y(y = f(x)));

b) opérateurs infixés

Le problème qui se pose ici est analogue à celui du symbole relationnel avec des complications provenant de la notion de priorité d'opérateurs

Au premier abord, il apparaît suffisant, pour qu'une séquence du type a + b = c soit correcte que + ait été introduit précédemment, et que c'est seulement dans le cas où l'analyse syntaxique est plus détaillée qu'il faille utiliser un méta-prédicat de façon à ce que la séquence a + b soit analysée de la façon suivante :



Mais si un objet est formé à partir de plusieurs opérateurs infixés il va se poser des problèmes de parenthésage de priorité en particulier lors des substitutions

si a. b = e
et a = c + d

le résultat, après substitution d'objets égaux devra être (c + d).b = e et non c + d.b = e si l'opération . est prioritaire sur l'opération +.

Remarquuons au passage que cette notion de priorité d'opérateurs ne figure pratiquement jamais explicitement dans les textes mathématiques. Le parenthésage se fait de façon intuitive. Seulement dans le cas d'une opération associative on signale parfois "qu'on pourra donc enlever les parenthèses"

Il faudra donc introduire des méta-prédicats qui indiquent qu'un identificateur est un opérateur infixé avec sa priorité par rapport aux autres opérateurs et que les objets formés à partir de tels opérateurs soient traités en conséquence.

Dans la nouvelle version d'Algol, il est possible d'introduire des opérateurs dont on fixe arbitrairement la priorité; il doit être possible de s'en inspirer pour résoudre cette question.

De manière plus générale, l'utilisation de méta-prédicats semble être une méthode intéressante pour introduire certaines notations mathématiques particulières.

L'écriture des métas-prédicats de la même manière que les prédicats ordinaires permet d'unifier l'écriture d'une expression.

Nombres

Si nous distinguons les nombres (entiers et réels) des autres objets

mathématiques c'est qu'ils sont souvent traités de façon intuitive.

Il existe bien des constructions axiomatiques des nombres mais lorsqu'un mathématicien utilise ces nombres il ne se réfère pas à leurs axiomes mais à un ensemble de propriétés supposées connues (sauf si bien sûr il s'occupe par exemple de théories arithmétiques ou de la topologie de R). Il faudrait déterminer empiriquement cet ensemble de propriétés, qui peut ne pas être constant. Ainsi lorsqu'on écrit :

$$x_{i+j} = f(x_i, x_j)$$

on n'a pas besoin des mêmes choses que si l'on écrit

"Les solutions de l'équation $x^p = n$ avec n pair et p premier inférieur à q- —"

Il nous faudrait donc bâtir un système qui permette de traiter les nombres de la même façon que dans les mathématiques usuelles et qui tienne compte aussi de leur écriture symbolique. Là encore c'est l'étude d'exemples qui permettra de dégager quelles sont les propriétés des nombres et de la numération habituellement utilisés.

C'est à ce problème des nombres que nous rattacherons celui du fini et de l'infini. Ces deux notions sont, elles aussi, utilisées de manière intuitive :

"Si E est un ensemble fini et A une suite infinie d'éléments de E alors il existe au moins un élément de E qui apparaît une infinité de fois dans A".

Il existe en logique formelle des axiomes d'infinité, mais indépendamment du fait qu'ils appartiennent à un calcul du second ordre, ce qui importe, comme dans le cas des nombres, c'est d'utiliser non pas les axiomes, mais les propriétés intuitives.

1.3. Récurrence

Le problème de la récurrence, dans les définitions et dans les raisonnements dépend évidemment du traitement du nombre, mais provient aussi du fait qu'on utilise des nombres indéterminés (ce qui se mani-

feste en mathématiques par le point de suspension : Soit x_1; x_2 .... x_n)

Définition récursives

Essayons par exemple de définir E^n, où E est un ensemble

Nous supposons connu l'opérateur X (produit cartésien, cf II.2)

définition P compose [n,E] = def

n = 2 => P(n,E) = X(E,E),

n >2 => P(n,E) = X(P(n-1,E),E);

d'après la définition du produit cartésien

x ∈ P(n,E) signifie

x est un n-uplet [2] , x (1) ∈ P(n-1,E), x(2)∈ E

et il faut recommencer avec x(1) ∈ P(n-1,E), mais comme n est indéterminé, on ne pourra pas arrêter la récurrence, et à moins d'introduire de façon formelle les points de suspension, on ne pourra pas donner la forme exacte d'un élément de P(n,E).

Dans ce cas particulier on pourrait définir la puissance cartésienne d'une autre façon

définition P compose [n,E] = def

∀x(x ∈ P<=> (x est un n-uplet [n],

∀i((1≤i,i≤n) => x(i) ∈ E)));

mais on se heurterait à des problèmes similaires lorsque par exemple on voudrait démontrer que

X(P(n,E), P(m,E)) = P(n + m, E)

Raisonnement par récurrence

Nous ne voulons pas parler ici du raisonnement par induction qui est un axiome de la logique formelle du second ordre avec fonction successeur

∀P((P(o) ∧ ∀x(P(x) = P(s(x)))) => ∀x(P(x))).

et que nous pouvons utiliser sous forme de théorèmes

ex théorème d'induction sur la longueur du mot [E]

h ∀x(l(x) = 0 => x ∈ E),

$$\forall_n \text{ (n est un entier, } \forall x\ (I(x) = n \Rightarrow x \in E) \Rightarrow \forall y\ (I(y) = n + 1 \Rightarrow y \in E))$$
$$\subset \forall n \text{ (n est un entier, } \forall x, (I(x) = n \Rightarrow x \in E));$$

Nous voulons parler des raisonnements qui font apparaître les points de suspension

ex Lorsqu'on veut démontrer que deux bases finies d'un espace vectoriel ont même dimension on pose $V_1, V_2 \ldots V_p$ et

$V'_1, V'_2 \ldots V'_q$ les deux bases $(p>q)$;

on a $V'_1 = f(V_1, V'_2 \ldots V'_q)$

On obtient une nouvelle base

$V_1, V'_2 \ldots V'_q$

même chose pour $V'_2 = f(V_1, V_2, V'_3 \ldots V'_q)$

etc...

$V_p = f(V_1, V_2 \ldots V_q)$ impossible

Ce mode de raisonnement n'utilise pas exactement l'induction telle qu'elle est définie ci-dessus mais une "généralisation inductive finie" qu'il faut arriver à formaliser.

On retrouvera des problèmes de cet ordre chaque fois qu'on manipulera des suites finies, des opérateurs finement itérés etc...

1.4. Théories du second ordre

Le langage que nous utilisons ne nous permet de traiter les expressions et les déductions que dans la logique du 1^{er} ordre (c'est-à-dire où seules les variables sont substituables ou quantifiables). Or il existe en mathématique des expressions et des règles de déduction qui sont du second ordre; nous venons de voir que c'est le cas pour le raisonnement par induction qui peut s'appliquer quelle que soit la propriété en question et que nous ne pouvons appliquer que pour chaque propriété particulière.

Il en est de même pour la notion de famille et même de suite. Les éléments d'une famille peuvent avoir en commun de vérifier une certaine propriété.

ex $\mathcal{F}$ est une famille de sous-groupe de G indicée par I signifie que pour tout i c I l'élément correspondant de la famille est un sous-groupe de G.

Si nous définissons un prédicat est une famille, nous serons obligés de faire des substitutions d'expressions.

Evidemment nous pouvons tourner la difficulté de deux façons :

- pour chaque famille particulière utiliser un prédicat spécial. Par exemple dans le cas ci-dessus définir le prédicat "est une famille de sous-groupes".
- considérer que toutes les propriétés en question sont collectivisantes et prendre comme relation définissant la famille, l'appartenance à un ensemble défini antérieurement.

ex G' = {x : x est un sous-groupe de [G]}

G est une famille de sous-groupes de G s'écrira alors G est une famille [G', i]
et le prédicat est une famille sera défini par définition X est une famille [Y,Z] = def

X est un symbole fonctionnel;
$\forall z\ (z \in Z = X(z) \in Y)$;

Cette méthode a un désavantage, c'est que toutes les propriétés ne sont pas collectivisantes (par exemple on ne parle pas de l'ensemble des groupes, mais de la classe des groupes).

D'autre part, pour avoir plus de généralité il nous semble qu'il faille quand même étudier la possibilité de susbtituer des propriétés c'est-à-dire d'introduire dans ce langage des expressions et raisonnements

du second ordre.

2. Simplification du raisonnement

Le principal inconvénient de la formalisation des démonstrations est actuellement la longueur presque repoussante d'une démonstration formalisée , à cause de la nécessité de signaler toutes les étapes du raisonnement. Nous avons déjà proposé une méthode de réduction, mais comme nous l'avons vu, nous sommes encore obligé de raisonner pas à pas.

L'utilisation d'une notion d'identité plus large (cf 1.1) permettra probablement de supprimer certaines étapes, mais ce sera encore insuffisant; l'idéal serait de pouvoir se rapprocher de ce qui se fait en mathématiques où on ne signale que les points les plus importants de la démonstration.

2.1. Démonstration "procédurale"

Ce qualificatif de procédural vient de l'analogie que nous allons établir avec les langages de programmation.

Nous pouvons considérer les justifications comme des instructions d'un langage de programmation. Lorsque dans un programme, on retrouve plusieurs fois la même séquence d'instructions, on les groupe en procédures.

De même en démonstration, il est possible de retrouver plusieurs fois la même suite de justifications (par exemple les lignes 5,8 et 6,9 dans II.6.2). Il serait intéressant alors de les grouper pour former une "procédure" ce qui allègerait l'écriture d'une démonstration. Dans la pratique de la démonstration c'est ce qui se passe lorsque l'on dit : "De la même façon on montre que ...", "En utilisant le même raisonnement..."

De manière plus générale, il semble que selon la théorie mathématique que l'on traite, on utilise plus souvent telle ou telle forme de raisonnement.

Une procédure de raisonnement n'est pas exactement analogue à la génération de théorèmes quoique ces deux notions soient relativement voisine (un théorème étant une forme particulière d'une règle de dérivation). Mais surtout leurs fonctions dans le raisonnement sont différentes; la démonstration procédurale devrait avoir une importance beaucoup plus grande lors de l'utilisation de méthodes heuristiques.

2.2. Calcul algébrique

Comme cas particulier important de la démonstration procédurale, nous pensons au calcul algébrique qui concerne les expressions de la forme <objet>=<objet> où les objets sont formés avec ces opérateurs. La dérivation de telles expressions peut se faire au moyen d'un formalisme purement logique, mais il serait plus intéressant et plus naturel de les traiter de façon particulière. Il nous semble que la méthode à prendre est de considérer à l'intérieur du langage des "sous-systèmes" formels qui auraient un traitement propre. Ces sous-systèmes seraient définis par le langage; les méta-prédicats donnant la priorité des opérateurs pourraient être utilisés à cet effet.

Des systèmes formels permettant de traiter des égalités sont déjà étudiés. Le plus connu est le λ-calcul de Church.

2.3. Utilisation du contexte

De même que nous sommes obligés de signaler toutes les étapes du raisonnement, nous sommes obligés aussi de signaler tous les objets auxquels on se réfère, alors qu'en démonstration habituelle ça n'est pas le cas : l'intuition et le contexte permettent au lecteur de préciser les imprécisions de l'auteur.

Exemple

En toute rigueur la définition d'une boule dans un espace métrique dépend de l'espace et de sa métrique, du centre de la boule et de son rayon, c'est pourquoi nous utilisons le prédicat B est une boule [E, d, a, ρ] et chaque fois que nous voudrons indiquer qu'une variable donnée est une boule, nous devons utiliser ce prédicat à 4 arguments.

Par contre, lorsque dans un texte mathématique un auteur a introduit un espace métrique et qu'il dit qu'une variable est une boule, il ne précise pratiquement jamais que cette boule est définie par rapport à l'espace métrique en question, et même il ne précise pas son centre et son rayon. Il faudrait donc arriver à formaliser cette pratique et à pouvoir se dispenser de faire figurer dans un prédicat les arguments qui sont bien définis par le contexte. Il serait également intéressant que dans ce cas particulier l'hypothèse "B est une boule" introduise en même temps deux variables qui seraient son centre et son rayon.

De manière générale on devrait pouvoir supprimer parmi les arguments d'un prédicat ceux qui sont définis par le contexte, et ceux qui sont totalement indéterminés, l'écriture du prédicat permettant de les introduire.

Cette modification représentant aussi un allègement considérable de l'écriture . Par exemple nous pourrions nous dispenser d'écrire les 8 arguments d'un espace vectoriel (l'addition vectorielle et son élément neutre, la multiplication scalaire, le corps, ses deux opérations et ses deux éléments neutres, et encore nous ne distinguons pas entre une structure et son ensemble support!).

D'autres références au contexte devraient aussi permettre de réduire certaines justifications :

- ne pas préciser avec quels arguments on applique un théorème
- ne pas préciser avec quels arguments on particularise une expression quantifiée universellement

- ne pas préciser à partir de quelles expressions on applique une justification donnée (surtout quand il s'agit de l'expression précédente)
- application implicite de définitions simples etc...

Mais ici nous commençons à aborder des problèmes de recherche heuristique que nous allons regarder en détail dans le paragraphe suivant.

2.4. Heuristiques

Enfin, la seule méthode qui permettra réellement de formaliser les démonstrations telles qu'elles sont écrites dans les textes mathématiques, c'est-à-dire en formalisant seulement le langage mais en n'apportant aucune précision à ce qu'à effectivement écrit l'auteur est d'utiliser des heuristiques.

Nous entendons par là que les étapes qui manquent dans la démonstration devront être trouvées par la machine non pas en utilisant des algorithmes combinatoires, mais par des processus utilisant au maximum toutes les informations disponibles (forme de la théorie, forme de l'expression à justifier etc...). En particulier, l'utilisation des théorèmes de la théorie et des procédures de raisonnement devrait avoir beaucoup plus d'importance.

Une fois connue la stratégie à adopter, son application e devrait pas causer de grosses difficultés. Par contre le problème de la détermination de cette stratégie est beaucoup plus important.

Ce problème se décompose en deux parties qui sont d'ailleurs étroitement liées.

1 - Quelles sont les heuristiques utilisées en démonstration?

2 - Etant donné une certaine situation quelle est l'heuristique qui a le plus de chance de mener au résultat?

Il nous semble que pour pouvoir répondre précisément à ces questions, il faudrait disposer d'une masse importante de renseignements expérimentaux qu'il faudrait ensuite classer et analyser. Nous pensons que ce travail serait beaucoup plus facile à entreprendre si les démonstrations étaient formalisées et c'est une des utilisations possibles du langage que nous étudions.

Une meilleure connaissance de la partie heuristique, ou intuitive, d'une démonstration pourrait avoir un très grand intérêt dans le domaine de l'enseignement : on pourrait mettre en évidence plus rationnellement la marche à suivre pour démontrer telle ou telle propriété. D'autre part il serait peut-être possible, justement en fonction de la difficulté de déterminer l'heuristique convenable de formaliser la notion plus ou moins subjective de difficulté d'une démonstration en celle, encore plus relative, d'évidence.

Des programmes de démonstration automatique utilisant des méthodes heuristiques ont déjà été réalisés par Gelernter pour la géométrie euclidienne élémentaire, par Slagle pour l'intégration formelle. Pitrat dans sa thèse utilise des méthodes heuristiques, mais seulement pour le calcul propositionnel.

3. Etude formelle du langage .

Le but essentiel de ce travail était de définir un langage qui permette de formaliser l'écriture des démonstrations et nous nous sommes principalement attachés à son contenu : c'est-à-dire aux notions mathématiques utilisées en démonstration et à la manière de les formaliser. Nous avons relativement peu envisagé ce langage comme système formel autonome. Il serait souhaitable de l'étudier encore plus systématiquement d'un point de vue formel en tant que système logique et en tant que langage formel.

3.1. Aspect logique

Nous avons montré sans nous y attarder que les justifications utilisées étaient régulières et cohérentes d'une part avec la pratique de la démonstra-

tion, d'autre part avec des systèmes logiques formels.

On pourrait considérer ce langage comme un système de logique formelle et en étudier les propriétés (régularité des règles, non contradiction etc..) de la même façon que pour le calcul des prédicats classiques ou que pour les calculs de Gentzen.

Pour commencer il faudrait, à notre avis, trouver une autre manière de décrire ce système afin de mettre davantage en évidence sa structure logique; la méthode que nous avons employée mettant davantage en évidence les propriétés syntaxiques.

Une fois ce travail préliminaire réalisé, nous croyons que nous obtiendrons un formalisme assez semblable à celui de Gentzen ou à celui de Suppes, avec toujours cette différence importante qui est l'utilisation d'une structure de bloc, et il serait possible alors d'étudier formellement le système obtenu. Comme nous avons constamment essayé de rester le plus près possible des méthodes naturelles de déduction, nous aurions de nouveaux éléments pour étudier formellement la déduction naturelle.

3.2. Aspect syntaxique

Nous avons également tenu compte en définissant ce langage qu'il était destiné à être traité en machine. Nous avons donc essayé de faire en sorte que la syntaxe soit la plus simple possible. En particulier nous avons fait en sorte que l'analyse soit le plus possible déterministe c'est-à-dire que dans la plus grande majorité des cas, la lecture de l'unité syntaxique suivante suffise à déterminer l'état de l'automate interpréteur.

Signalons aussi que la vérification d'une démonstration par la machine peut se faire en un seul passage, en même temps que l'analyse syntaxique : il suffit lorsqu'on arrive sur la catégorie syntaxique <justification> de tester que l'expression justifiée est bien celle qui est écrite. Ce test étant lui-même une question d'analyse s ntaxique, la vérification en machine d'une démons-

tration devient un problème purement syntaxique, ce à quoi on pouvait s'attendre puisqu'on connaît déjà l'équivalence entre les points de vue sémantique et syntaxique en logique formelle (théorèmes de Gödel, de Tarski, de Henkin aux environs de 1930.

Ici aussi il devrait être possible de trouver une formulation du langage qui fasse mieux apparaître le caractère purement syntaxique d'une démonstration.

4. Applications

Enfin, en plus de ces travaux sur le langage lui-même, nous entrevoyons des domaines de recherche où il servira d'outil. Nous les avons déjà signalés au passage; nous les citerons ici en guise de conclusion.

- mise en évidence de la structure d'une démonstration; étude du rapport de cette structure avec celle de la théorie mathématique considérée et celle du théorème à démontrer
- étude des processus heuristiques de démonstration et du mécanisme de l'intuition mathématique; recherche systématique de la meilleure stratégie.
- connaissance plus approfondie de la déduction naturelle
- utilisation de ces nouvelles connaissances et peut-être du langage lui-même dans des domaines tels que la démonstration automatique, l'enseignement des méthodes mathématiques, l'enseignement programmé des mathématiques.

Annexe I

TABLEAUX SYNTAXIQUES

Les tableaux syntaxiques qui suivent correspondent au langage tel qu'il est défini dans les chapitres I et II, auquel nous avons rajouté celles des extensions envisagées au chapitre III qui nous ont semblé suffisamment précises pour pouvoir être utilisées dès maintenant.

La plus grande partie de ces tableaux a été écrite par Messieurs HENNERON et GUILLERMINET dans le cadre d'un projet d'élèves-ingénieurs de 3ème année.

Tableau I

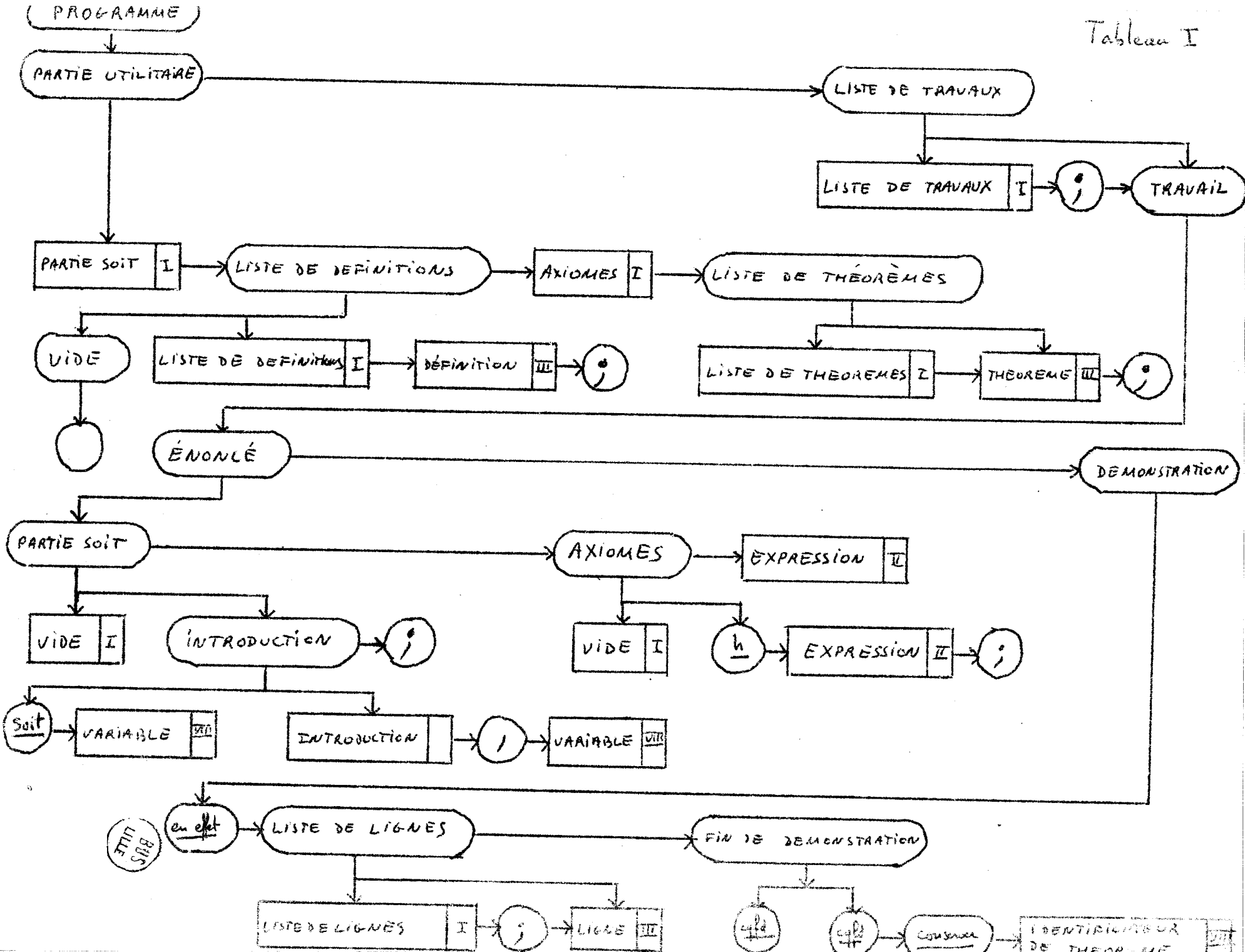

PROGRAMME
PARTIE UTILITAIRE
LISTE DE TRAVAUX
LISTE DE TRAVAUX I
;
TRAVAIL
PARTIE SOIT I
LISTE DE DEFINITIONS
AXIOMES I
LISTE DE THÉORÈMES
VIDE
LISTE DE DEFINITIONS I
DÉFINITION II
;
LISTE DE THEOREMES I
THEOREME II
;
ÉNONCÉ
DEMONSTRATION
PARTIE SOIT
AXIOMES
EXPRESSION II
VIDE I
INTRODUCTION
;
VIDE I
h
EXPRESSION II
;
Soit
VARIABLE VII
INTRODUCTION
,
VARIABLE VIII
BUS LIBRE
en effet
LISTE DE LIGNES
FIN DE DEMONSTRATION
LISTE DE LIGNES I
;
LIGNE III
cqfd
cqfd
Conseq
IDENTIFICATEUR

Tableau II

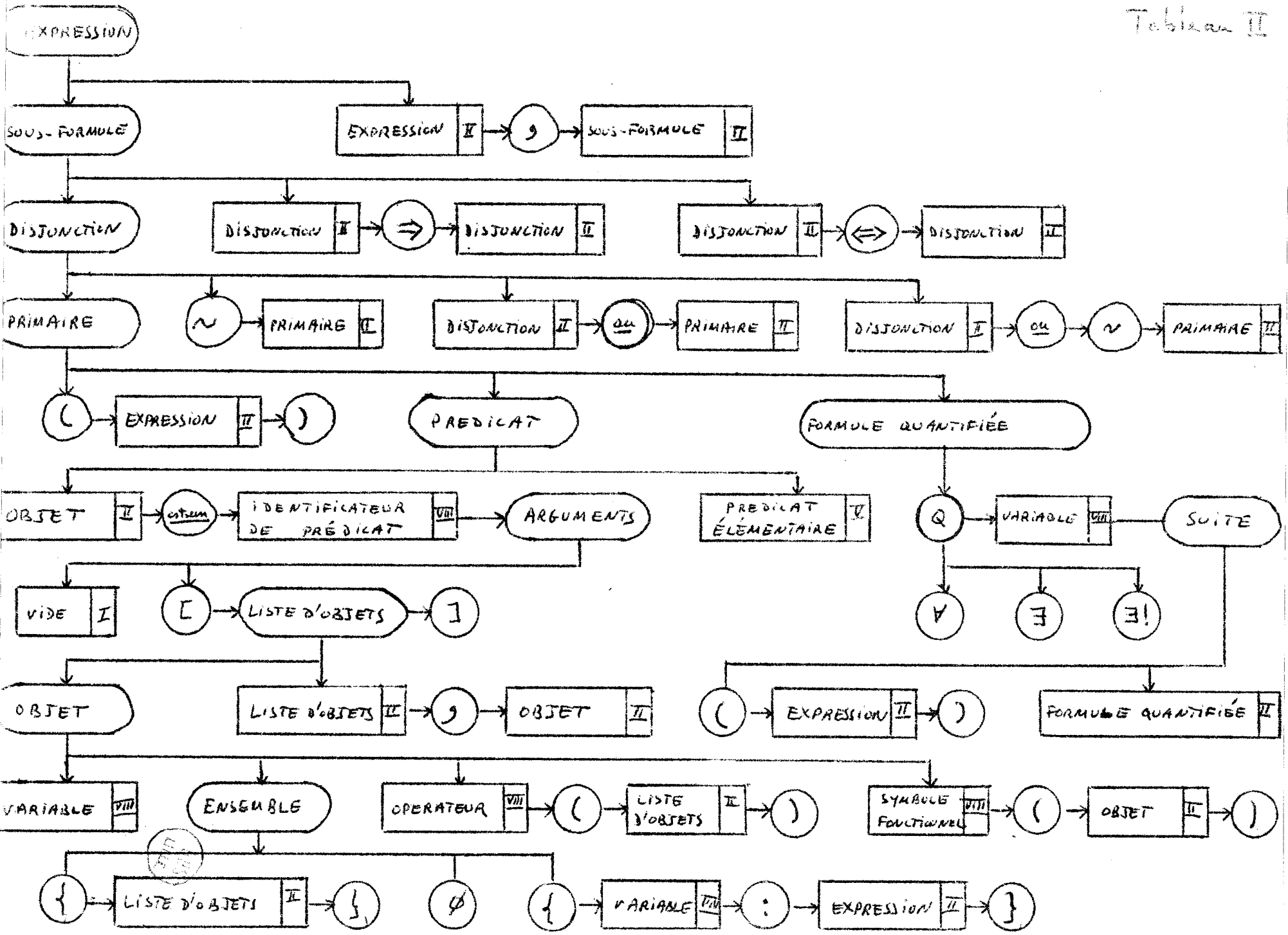

EXPRESSION
SOUS-FORMULE
EXPRESSION II
,
SOUS-FORMULE II
DISJONCTION
DISJONCTION II
⇒
DISJONCTION II
DISJONCTION II
⇔
DISJONCTION II
PRIMAIRE
~
PRIMAIRE II
DISJONCTION II
ou
PRIMAIRE II
DISJONCTION II
ou
~
PRIMAIRE II
(
EXPRESSION II
)
PREDICAT
FORMULE QUANTIFIÉE
OBJET II
estun
IDENTIFICATEUR DE PRÉDICAT VIII
ARGUMENTS
PREDICAT ÉLÉMENTAIRE V
Q
VARIABLE VIII
SUITE
VIDE I
[
LISTE D'OBJETS
]
∀
∃
∃!
OBJET
LISTE D'OBJETS II
,
OBJET II
(
EXPRESSION II
)
FORMULE QUANTIFIÉE II
VARIABLE VIII
ENSEMBLE
OPERATEUR VIII
(
LISTE D'OBJETS II
)
SYMBOLE FONCTIONNEL VIII
(
OBJET II
)
{
LISTE D'OBJETS II
}
∅
{
VARIABLE VIII
:
EXPRESSION II
}

Tableau III

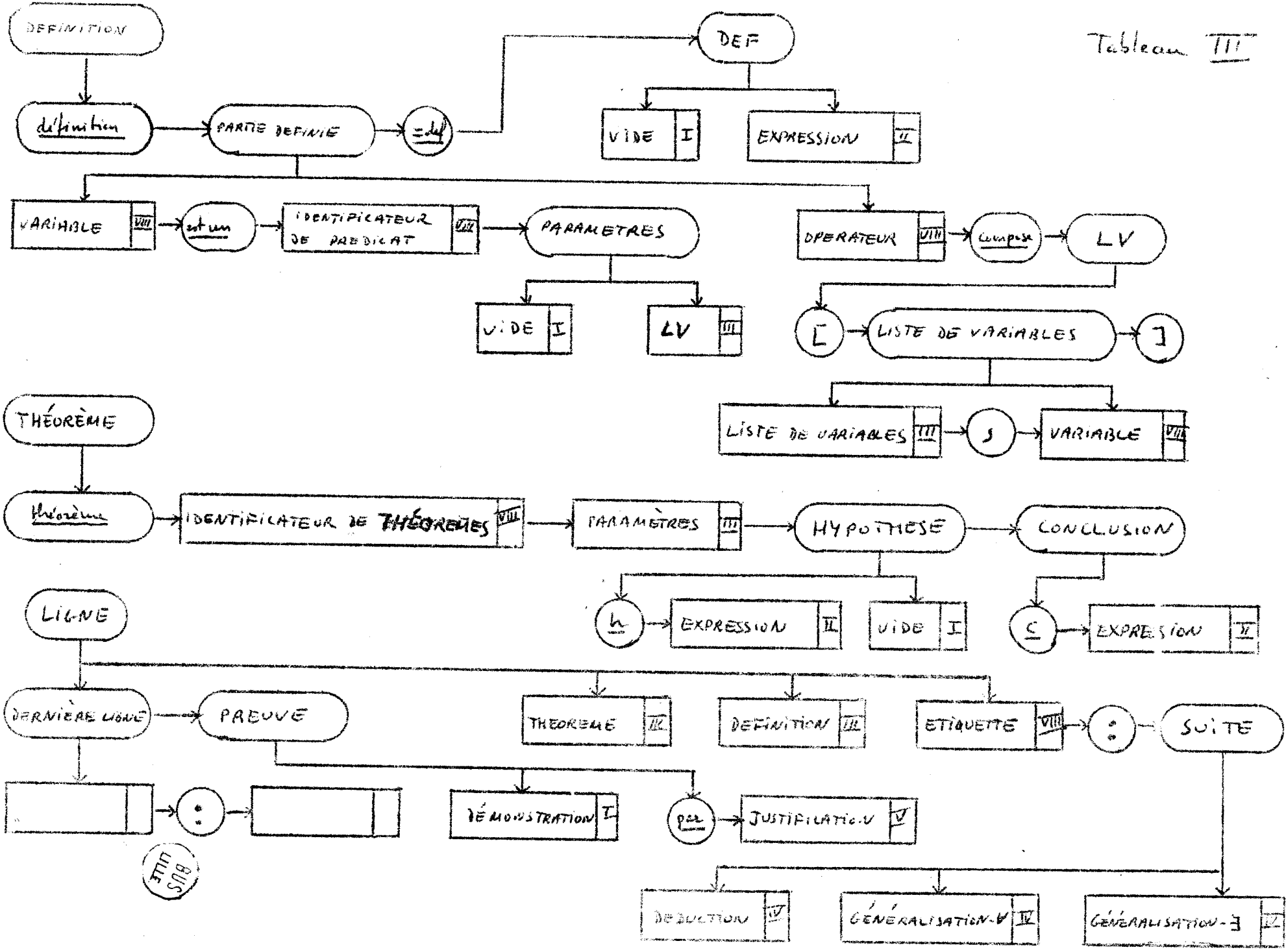

DEFINITION
définition
PARTIE DEFINIE
=def
DEF
VIDE I
EXPRESSION II
VARIABLE VIII
soit un
IDENTIFICATEUR DE PREDICAT VIII
PARAMETRES
VIDE I
LV III
OPERATEUR VIII
compose
LV
[
LISTE DE VARIABLES
]
LISTE DE VARIABLES III
;
VARIABLE VIII
THÉORÈME
théorème
IDENTIFICATEUR DE THÉORÈMES VIII
PARAMÈTRES III
HYPOTHESE
CONCLUSION
h
EXPRESSION II
VIDE I
C
EXPRESSION II
LIGNE
DERNIÈRE LIGNE
PREUVE
THEOREME II
DEFINITION II
ETIQUETTE VIII
:
SUITE
:
DÉMONSTRATION I
par
JUSTIFICATION V
fin sub
DEDUCTION V
GÉNÉRALISATION-∀ V
GÉNÉRALISATION-∃ V

Tableau IV

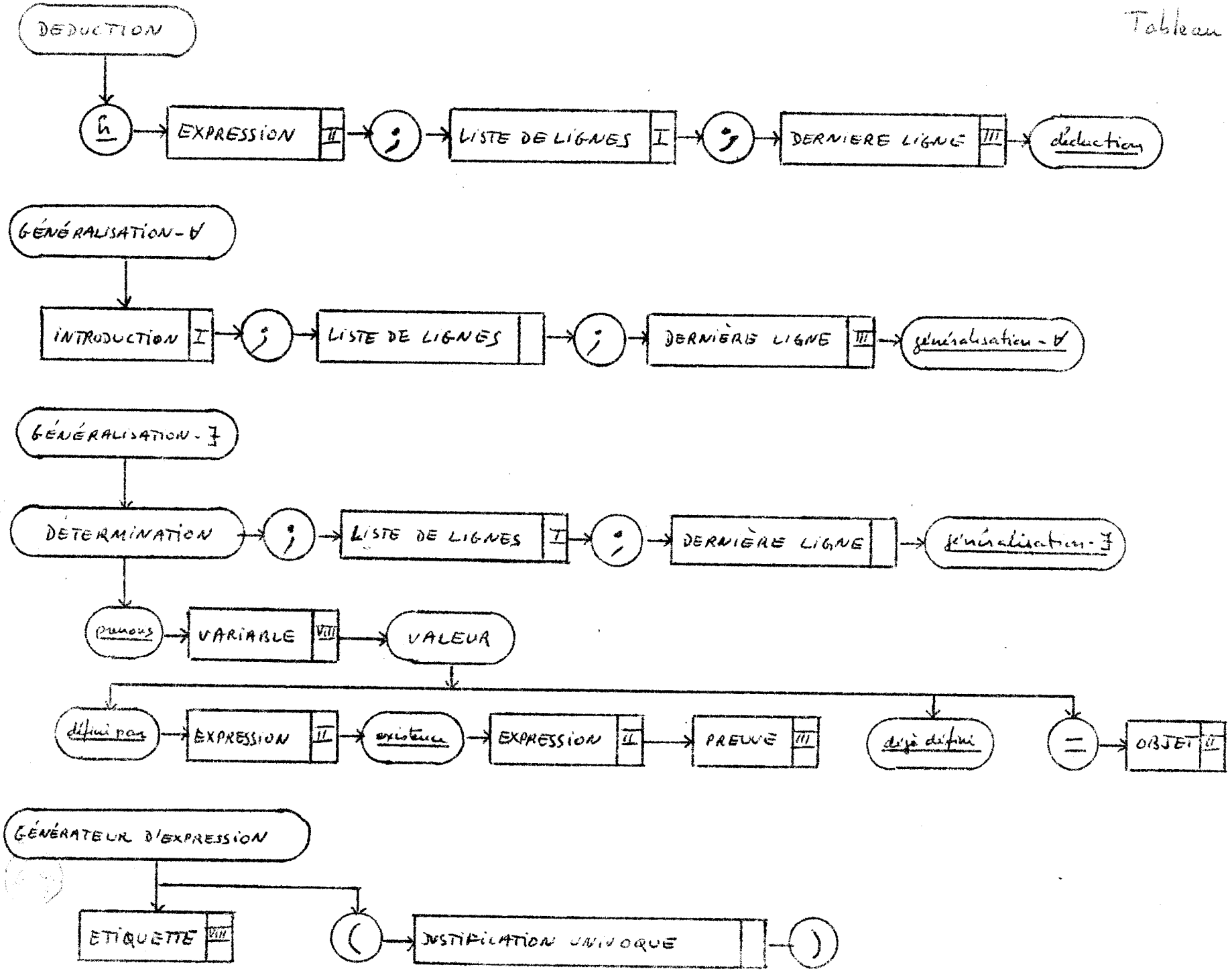
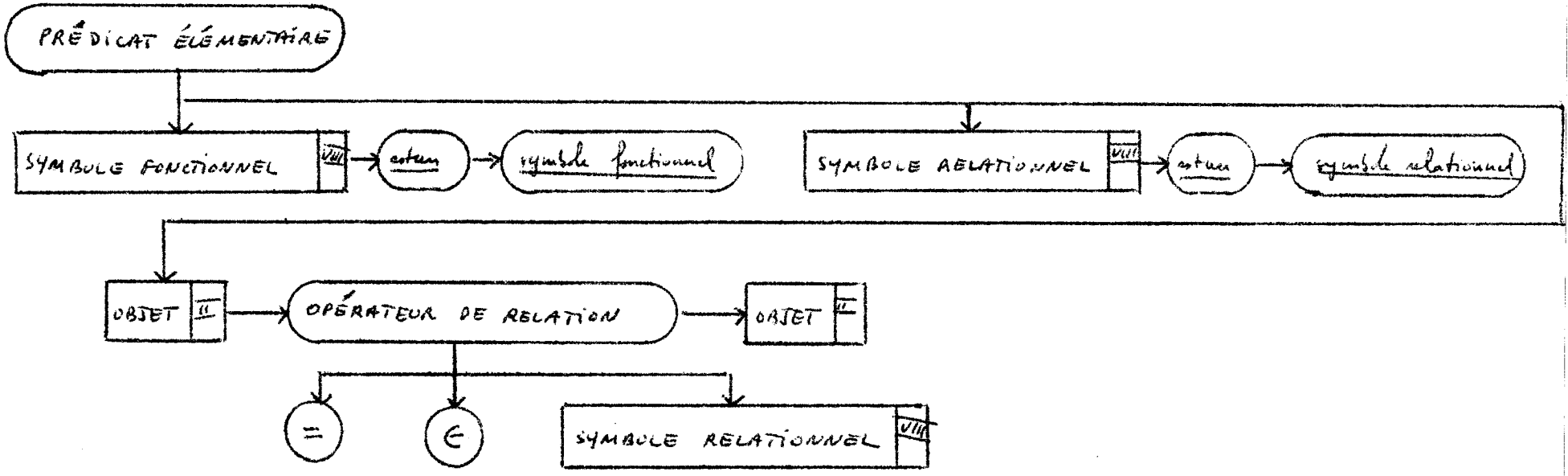
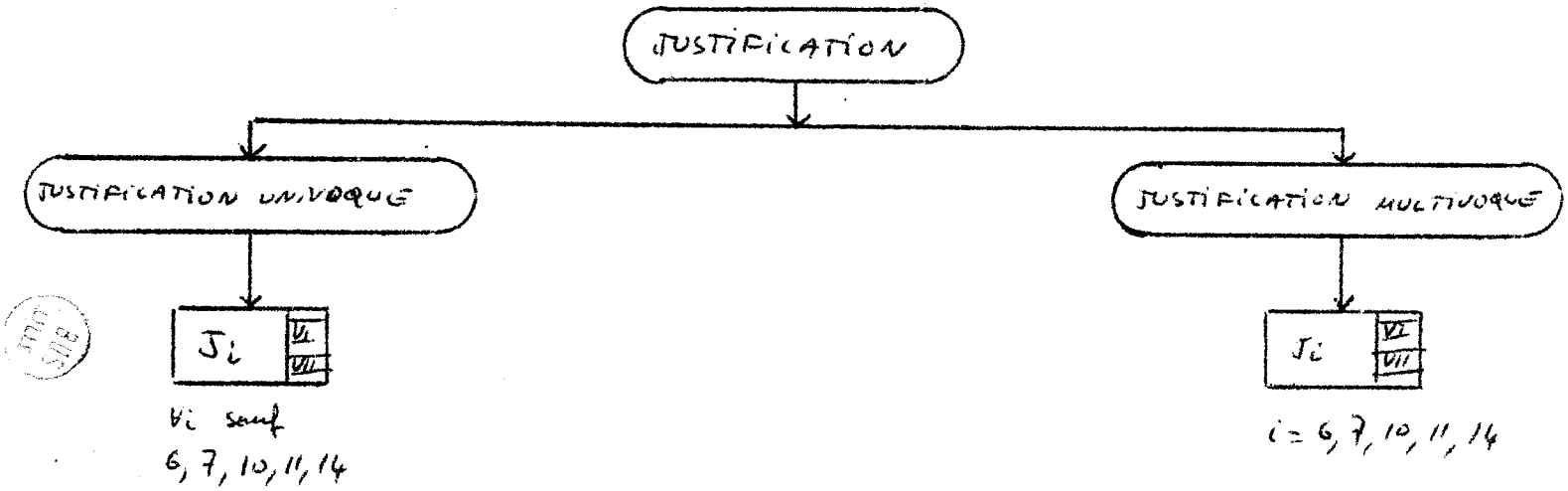

DÉDUCTION
h
EXPRESSION II
;
LISTE DE LIGNES I
;
DERNIÈRE LIGNE III
déduction
GÉNÉRALISATION - ∀
INTRODUCTION I
;
LISTE DE LIGNES
;
DERNIÈRE LIGNE III
généralisation - ∀
GÉNÉRALISATION - ∃
DÉTERMINATION
;
LISTE DE LIGNES I
;
DERNIÈRE LIGNE
généralisation - ∃
prenons
VARIABLE VIII
VALEUR
défini par
EXPRESSION II
existence
EXPRESSION II
PREUVE III
déjà défini
=
OBJET II
GÉNÉRATEUR D'EXPRESSION
ETIQUETTE VIII
(
JUSTIFICATION UNIVOQUE
)

Tableau V


PRÉDICAT ÉLÉMENTAIRE
SYMBOLE FONCTIONNEL
VIII
entier
symbole fonctionnel
SYMBOLE RELATIONNEL
VIII
entier
symbole relationnel
OBJET
II
OPÉRATEUR DE RELATION
OBJET
II
=
∈
SYMBOLE RELATIONNEL
VIII


JUSTIFICATION
JUSTIFICATION UNIVOQUE
JUSTIFICATION MULTIVOQUE
Ji
VI
VII
Vi sauf
6, 7, 10, 11, 14
Ji
VI
VII
i = 6, 7, 10, 11, 14

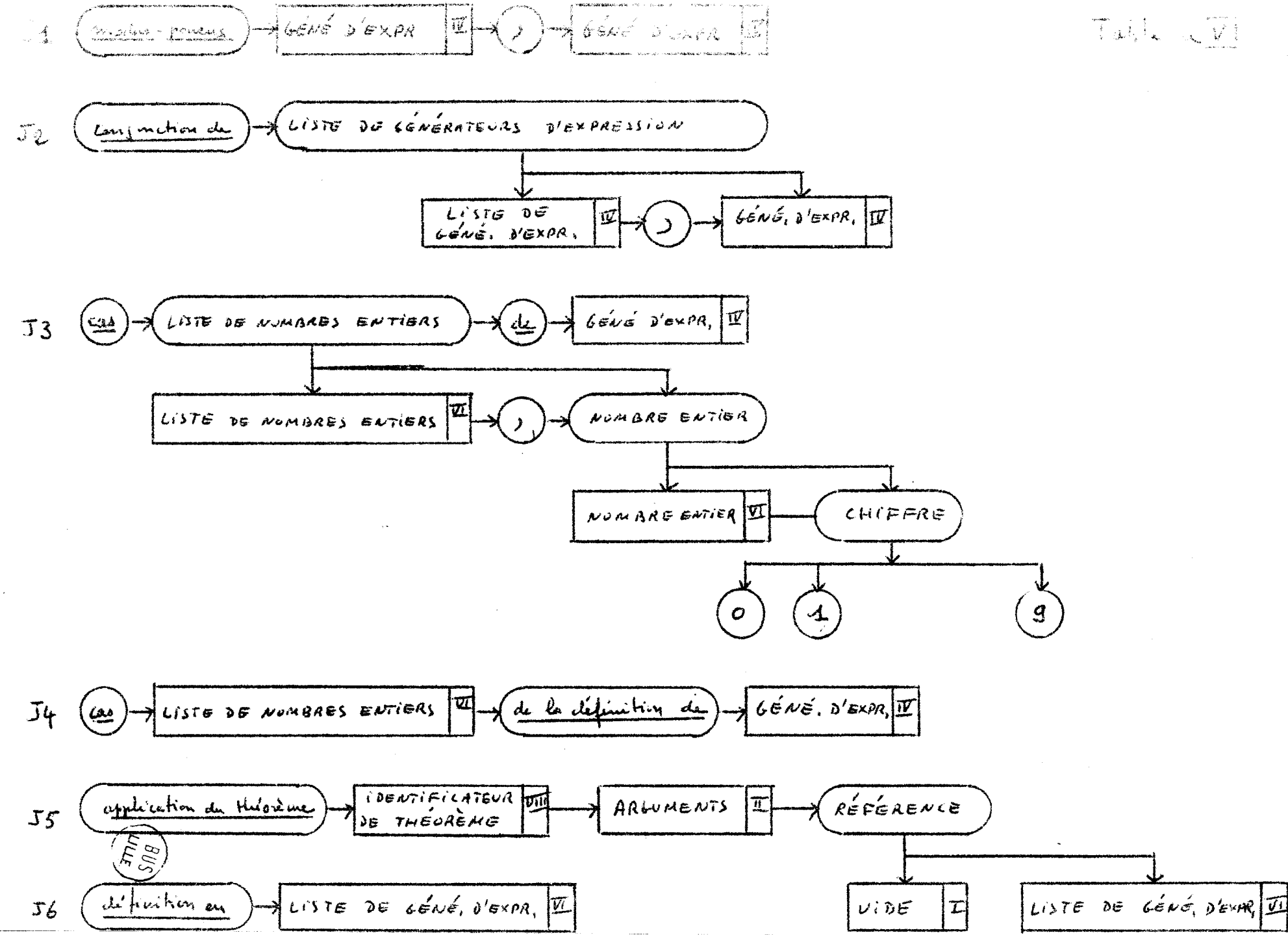

J1
GÉNÉ D'EXPR IV
)
GÉNÉ D'EXPR IV
Table VI
J2
LISTE DE GÉNÉRATEURS D'EXPRESSION
LISTE DE GÉNÉ. D'EXPR. IV
)
GÉNÉ. D'EXPR. IV
J3
LISTE DE NOMBRES ENTIERS
GÉNÉ D'EXPR. IV
LISTE DE NOMBRES ENTIERS VI
)
NOMBRE ENTIER
NOMBRE ENTIER VI
CHIFFRE
0
1
9
J4
LISTE DE NOMBRES ENTIERS VI
GÉNÉ. D'EXPR. IV
J5
application du théorème
IDENTIFICATEUR DE THÉORÈME VIII
ARGUMENTS II
RÉFÉRENCE
BUS
J6
LISTE DE GÉNÉ. D'EXPR. VI
VIDE I
LISTE DE GÉNÉ. D'EXPR. VI

Tableau VII

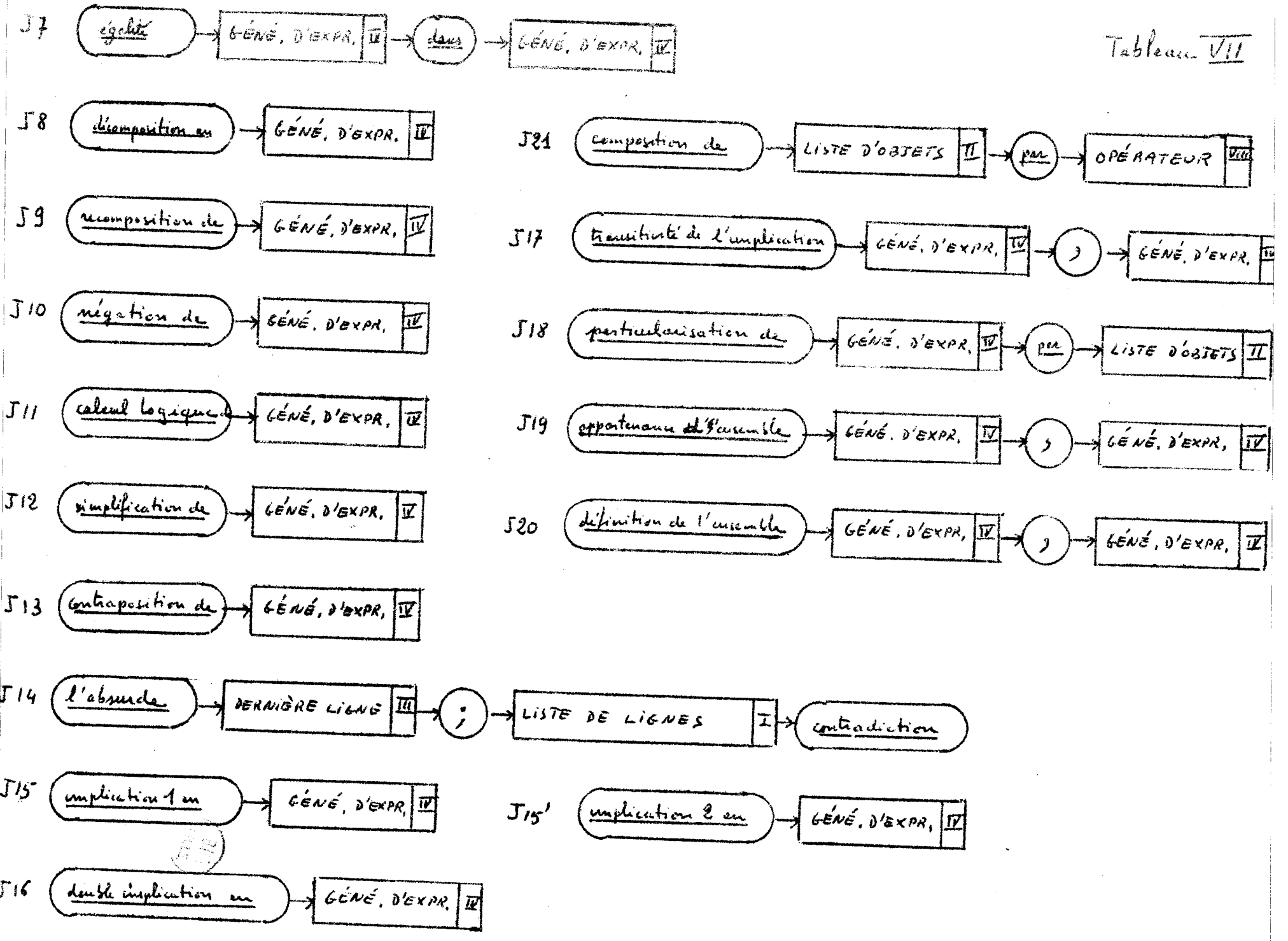
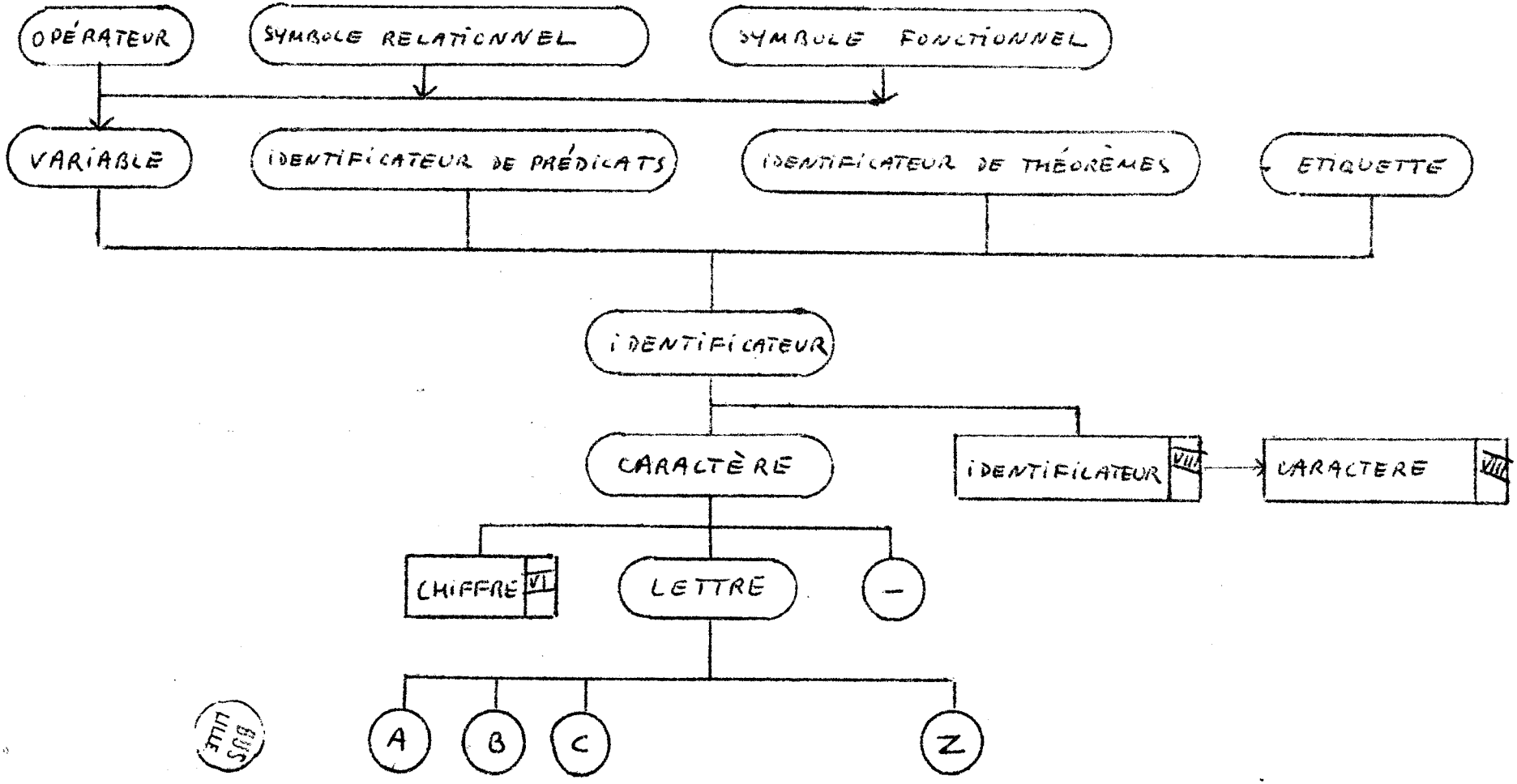

J7
égalité
GÉNÉ. D'EXPR. IV
dans
GÉNÉ. D'EXPR. IV
J8
décomposition en
GÉNÉ. D'EXPR. IV
J9
recomposition de
GÉNÉ. D'EXPR. IV
J10
négation de
GÉNÉ. D'EXPR. IV
J11
calcul logique
GÉNÉ. D'EXPR. IV
J12
simplification de
GÉNÉ. D'EXPR. IV
J13
contraposition de
GÉNÉ. D'EXPR. IV
J14
l'absurde
DERNIÈRE LIGNE III
;
LISTE DE LIGNES I
contradiction
J15
implication 1 en
GÉNÉ. D'EXPR. IV
J15'
implication 2 en
GÉNÉ. D'EXPR. IV
J16
double implication en
GÉNÉ. D'EXPR. IV
J21
composition de
LISTE D'OBJETS II
par
OPÉRATEUR VIII
J17
transitivité de l'implication
GÉNÉ. D'EXPR. IV
)
GÉNÉ. D'EXPR. IV
J18
particularisation de
GÉNÉ. D'EXPR. IV
par
LISTE D'OBJETS II
J19
appartenance à l'ensemble
GÉNÉ. D'EXPR. IV
)
GÉNÉ. D'EXPR. IV
J20
définition de l'ensemble
GÉNÉ. D'EXPR. IV
)
GÉNÉ. D'EXPR. IV

Tableau VIII



Annexe 2

INTERPRETATION DU LANGAGE

Comme nous l'avons déjà signalé, une des applications possibles du langage de formalisation des démonstrations est la vérification automatique. Le travail de la machine consistera à lire le programme et à vérifier que chaque expression est écrite correctement, que les variables et prédicats qui y figurent sont respectivement introduits et définis et que la preuve donnée (démonstration ou justification) permet bien d'écrire cette expression. Cette vérification se fait séquentiellement ; en même temps le programme interpréteur se charge de ranger les définitions et théorèmes, de traiter les introductions de variables, les hypothèses, les ouvertures et fermetures de blocs.

Le programme interpréteur utilisera plusieurs piles pour ranger

1) les variables introduites
2) les définitions
3) les expressions déjà vérifiées et les hypothèses
4) les théorèmes
5) les expressions en cours de vérification (ceci dû à la récursivité de la démonstration)
6) les justifications à effectuer (définition récursive des justifications)

Les 5 premières piles seront chargées et déchargées lors des ouvertures et fermetures de bloc par l'interpréteur -qui fonctionnera lui-même de manière récursive- ce qui assure qu'une variable introduite en tête d'un bloc, ou une

expression figurant à l'intérieur sont bien locales à ce bloc puisqu'à l'extérieur elles ne figureront plus dans les piles correspondantes. De même pour les définitions et les théorèmes.

Il ne nous paraît pas nécessaire de faire, avant l'interprétation proprement dite, une analyse syntaxique détaillée d'une démonstration, car on peut estimer que l'interpréteur est en même temps automate de reconnaissance du langage, ceci sera d'autant plus facile que ce langage nous semble être en grande partie déterministe. L'analyse syntaxique des expressions est une analyse syntaxique classique et ne semble pas devoir présenter de difficultés particulières. (Ce sera probablement la partie non déterministe de l'analyse).

On aura probablement intérêt à effectuer les justifications récursivement: à partir des expressions étiquetées on génère l'expression résultat et on teste sa f-identité à l'expression justifiée - dans le cas des justifications univoques ; pour celles qui sont multivoques il faudra au contraire partir de l'expression justifiée.

Une part importante du travail d'interprétation consistera donc à vérifier que deux expressions sont f-identiques. A ce sujet se pose la question de la représentation interne des expressions.

Après codage des éléments constitutifs, on peut les représenter soit sous forme linéaire, telles qu'elles sont écrites dans une démonstration, soit sous forme arborescente (aux noeuds : des opérateurs ; branches terminales : soit des identificateurs, soit des prédicats). Selon la justification à appliquer l'une ou l'autre forme serait plus commode à manipuler. Peut-être faudrait-il envisager des solutions mixtes où certaines parties de l'expression seraient écrites linéairement, d'autres développées en arborescence lorsque cela serait nécessaire.

Messieurs HENNERON et GUILLERMINET travaillent actuellement à l'implémentation du langage de démonstration sur IBM 7044. Il est raison-

nable de penser que d'ici peu, on disposera donc de moyens permettant d'une part de tester de façon plus efficace le langage que nous avons défini, d'autre part de franchir une nouvelle étape dans cette approche expérimentale de la démonstration mathématique.

BIBLIOGRAPHIE GENERALE

Abrahams, Paul W — *Machine verification of mathematical proof*
Doctoral dissertation MIT (1963)

Gentzen, G. — *Recherches sur la déduction logique*
traduit par Feys et Ladrière
Paris 1955

Hilbert et Ackermann — *Principles of mathematical logic*
New York (1950)

Kuntzmann, J — *La foire aux symboles (inédit)*

Suppes, P. — *Introduction to logic*
Princeton (1957)

- En ce qui concerne la démonstration automatique nous renvoyons aux ouvrages et articles signalés dans la thèse de Mme H. NGOA

- Sur quelques unes des questions soulevées au chapitre III on trouvera des approches particulières dans les ouvrages déjà cités et dans

Dupraz, M. — *Utilisation de l'algèbre de Boole en logique mathématique*
Thèse 3^{e} cycle Grenoble (1966)

Kondo, M. et Murata, H — *On proof retrival : problem solving machines*

Maurer, W.D.

Computer experiments in finite algebra
Com. ACM 9/8 Août 1966

Wos, Robinson, Carson, Shalla

The concept of demodulation in theorem proving
Journal ACM 14/4 -- Octobre 1967