### UNIVERSITE DE LIEGE FACULTE DES SCIENCES APPLIQUEES

## LE CONCEPT "HOLON" ET SON APPLICATION A LA PROGRAMMATION

PAR

P.A. de MARNEFFE

Ingénieur civil Assistant à l'Université de Liège

OCTOBRE 1975

THESE PRESENTEE EN VUE DE L'OBTENTION DU GRADE DE DOCTEUR EN SCIENCES APPLIQUEES DE L'UNIVERSITE DE LIEGE

### UNIVERSITE DE LIEGE FACULTE DES SCIENCES APPLIQUEES

# LE CONCEPT "HOLON" ET SON APPLICATION A LA PROGRAMMATION

PAR

P.A. de MARNEFFE

Ingénieur civil Assistant à l'Université de Liège

OCTOBRE 1975

THESE PRESENTEE EN VUE
DE L'OBTENTION DU GRADE
DE DOCTEUR EN
SCIENCES APPLIQUEES
DE L'UNIVERSITE DE LIEGE

LE CONCEPT "HOLON"

ET

SON APPLICATION A LA PROGRAMMATION

P.A. de MARNEFFE Service d'Informatique (Prof. D. RIBBENS.) Faculté des Sciences Appliquées Université de Liège.

#### **ARMERSES**

Noue distrone experimen touts notice gentitude our pursonner qui suivent.

A Maneigur le Professour D. Bibbens pour na patiente supereleinn. Ere incutivalles gonesillat en suragemente contributrent confessous à l'aphènement de le trosail.

A Mentioure Lee Professione M. Linzman et A. Berthine dont in participation on would de thick name fue d'une stip priciouse:

A Monsieur le Professeur M. Leroy paus use monbreness et stimelantes discussions que nous aidénent à clarifies et simplifier notré approahe du sujet.

A Monoteur la Profession E. Mission que, par que infliques franches en directos, nestifia servaina Jeanes de moire démarche.

parine and series with the series of the ser

et à ma mère

A tous cour qui oni su l'amabilité de lire partininé de nos rapporte techniques et de nous prodiquer constile, orisignes ou encouragements; tout particuliérateunt à Ventieure les Projesseures 6.8. Dilette, D.E. Fruit et à Constiles d'American & Civinstile et un Dr. L. Bendamen.

du Dr. M. Michards qui nous a fait apprinter une conception britamises de l'informations.

A Mesticura A. Cristoford at Each fri an pour l'aide qu'ille nous est espected dans la marchéstes des éprances, à Madame C. Schoitz pour la freppe du taste et à Mondiaur C. Dispars pour le dessin des figures.

\$1. "Light but not Lagge" of Michaeles, was downed never as intiguing at any approach worst.

#### REMERCIEMENTS

Nous désirons exprimer toute notre gratitude aux personnes qui suivent.

A Monsieur le Professeur D. Ribbens pour sa patiente supervision. Ses inestimables conseils et encouragements contribuèrent amplement à l'achèvement de ce travail.

A Messieurs les Professeurs M. Linsman et A. Danthine dont la participation au comité de thèse nous fut d'une aide précieuse.

A Monsieur le Professeur H. Leroy pour ses nombreuses et stimulantes discussions qui nous aidérent à clarifier et simplifier notre approche du sujet.

A Monsieur le Professeur E. Milgrom qui, par ses critiques franches et directes, rectifia certains écarts de notre démarche.

Au Dr. Harlan Mills et à tous les membres de son équipe pour leur accueil cordial. Leur expérience nous aida à découvrir les problèmes soulevés par l'application "en vraie grandeur" de la "programmation structurée".

Au F.N.R.S. et à l'I.B.M.-Belgium qui nous ont donné la possibilité de séjourner à l'I.B.M. Federal Systems Division.

A tous ceux qui ont eu l'amabilité de lire certains de nos rapports techniques et de nous prodiguer conseils, critiques ou encouragements; tout particulièrement à Messieurs les Professeurs E.W. Dijkstra, D.E. Knuth et A. Danthine, à Monsieur M. Sintzoff et au Dr. P. Henderson.

Au Dr. M. Richards qui nous a fait apprécier une conception britannique de l'informatique.

A Messieurs A. Cristofori et Banh Tri An pour l'aide qu'ils nous ont apportée dans la correction des épreuves, à Madame C. Schmitz pour la frappe du texte et à Monsieur C. Diepart pour le dessin des figures.

Et, "last but not least", à Béatrice, mon épouse, pour sa patience et son support moral.

#### RESUME

Les problèmes inhérents à la conception de grands programmes ont justifié l'intérêt de l'étude de la méthodologie de la programmation. On ignore toutefois si les principes méthodologiques peuvent influencer la conception des langages de programmation, en effet toute méthodologie, par ses principes et règles, tend à imposer à la structure d'un programme des restrictions qui entraînent une certaine inefficacité d'exécution.

D'une tentative de conciliation des deux points de vue, résulte la conception d'un langage de programmation "holon" qui impose une structure précise aux programmes mais qui tient compte des impératifs d'efficacité par une nette séparation entre opération "primitive" et "terminale", par l'utilisation d'une "unité de programme", le "holon", à laquelle ne correspond systématiquement aucune méthode d'implémentation, et par l'exploitation des propriétés de la structure.

## TABLE DES MATIERES

CHAPI	TRE I : De l'idée d'un programme au programme.	
1.	Aperçu historique	I.1.
	Engineering et Software Engineering	I.2.
	Langages et méthodologies de programmation	I.3.
	Caractéristiques d'un langage de développement de programmes	I.5.
	2.1. Rôle du distotest	
CHAPI	TRE II : Définition du langage HPL.	
1.	Introduction	II.1.
2.	Principes de conception du HPL	II.1.
3.	Eléments fondamentaux du HPL	11.3.
4.	Règles de construction d'un programme-holon	11.3.
5.	Structure du coprs de holon	II.5.
	5.1. Relation d'enclenchement	11.5.
	5.2. Réalisation effective d'un effet-net	II.6.
	5.3. Enclenchement séquentiel	II.6.
	5.4. Enclenchement en parallèle et non-déterminisme	II.7.
6.	Les instructions-holon	11.8.
	6.1. Méthodes de description des instructions-holon	11.8.
	6.2. Instructions-holon de sélection	II.8.
	6.3. Instructions-holon d'itération	II.11.
	6.4. Justification du choix des instructions-holon	II.12.
7.	Syntaxe du nom de holon	II.12.
8.	Structure d'un programme-holon	II.14.
	8.1. Conditions de production effective d'un effet-net	II.14.
	8.2. Cas du holon récursif	II.15.
9.	Structure du holon booléen	II.17.
	9.1. Structure générale du holon booléen	II.17.
	9.2. Restrictions pour les instructions-holon d'itération	II.19.
	9.3. Holon booléen récursif	II.20.
	9.4. Justification de la structure du holon booléen	II.20.
10.	Programmes et arbres finis	II.22.
	10.1. Rappel	II.22.
	10.2. Arbres orientés et instructions de contrôle	II.23.
	10.3. Programmes séquentiels et arbres ordonnés	II.26.
	10.4. Programmes-holon et arbres binaires	II.26.
11.	Hiérarchisation des instructions de contrôle	II.27.
12.	Les langages terminaux	II.28.
	12.1. Définition d'une opération terminale	II.28.
	12.2. Forme généralisée d'une opération terminale	II.29.
	12.3. Syntaxe d'une opération terminale	II.29.
	12.4. Définition d'un langage terminal	II.31.
	12.5. Syntaxes externe et interne	II.31.
13.	. Modèle d'exécution d'un programme HPL	II.32.
	13.1. Eléments du modèle	II.32.
	13.2. Sous-arbres binaires correspondant aux éléments du langage-holon	II.33.
	13.3. Règle en cas d'occurrences multiples d'un holon	II.34.
	13.4. Exécution d'un programme-holon	11.35.
	13.5. Arbre binaire de l'algorithme de traversée	II.37.
14	. Propriétés d'un programme HPL	11.38.
	14.1. Propriété d'exécution	11.38.
	14.2. Détermination pratique de la propriété d'exécution	II.40.

	14.3. Propriété de "durée de vie" d'un holon	II.41.
	14.4. Détermination pratique de la durée de vie	II.41.
	14.5. Propriété de récursivité d'un holon	II.43.
15.	Limitation de la nature des paramètres	II.46.
CHAPI	TRE III : Logique du Datatest.	
1.	Introduction	III.1.
2.	Domaine LR et datatest	III.1.
	2.1. Rôle du datatest	III.4.
3.	Le Postest	111.5.
	3.1. Efficacité du datatest	III.5.
	3.2. Hypothèses sur les données d'un programme	III.6.
	3.3. Exemple de fractionnement d'un datatest	III.6.
4.	Syntaxe du Datatest et du Postest	III.8.
	4.1. Justification d'une forme syntaxique particulière	III.8.
	4.2. Description de la syntaxe du datatest	III.9.
	4.3. Evaluation d'un datatest	III.10.
	4.4. Description de la syntaxe du postest	III.11.
5.	Syntaxe complète d'un holon	III.11.
	Modifications d'un datatest (ou d'un postest)	III.12.
	6.1. Groupe des inhibitions	III.12.
	6.2. Groupe des insertions	III.13.
	6.3. Groupe des remplacements	III.14.
	6.4. Cas des opérations terminales	III.14.
	6.5. Influence des modifications sur la structure d'un programme	III.14.
7.	. Influence de la méthodologie de programmation	III.15.
	. Interruption du programme dans une instruction-holon "case"	III.15.
	Rôle du cospilateur	
CHAP:	ITRE IV : Environnement et Objets.	
1	. Types, objets et objets abstraits	IV.1.
	1.1. Types et objets	IV.1.
	1.2. Création d'un type	IV.1.
	1.3. Objets "abstraits"	IV.2.
2	. Création et raffinement des objets abstraits	IV.2.
	2.1. Ensemble des objets et ensemble des types	IV.2.
	2.2. Ordre "create"	IV.3.
	2.3. Ordre "refine"	IV.3.
3	. Procédés de structuration	IV.5.
	3.1. Array	IV.5.
	3.2. Sequence	IV.6.
	3.3. File	IV.6.
4	. Syntaxe de "create" et de "refine"	IV.6.
	4.1. Create	IV.6.
	4.2. Refine	IV.7.
	4.3. Règles de construction	IV.7.
5	. Construction de l'environnement d'un programme	IV.8.
	5.1. L'ordre "declare"	IV.8.
	5.2. La portée	IV.9.
	5.3. Spécificateur de portée	IV.10.
	5.4. Initialisation	IV.10.
	5.5. Variable-index	IV.12.
(	5. Syntaxe des identificateurs de variable	IV.12.
	7. Structure "récursivement complète" et pointeurs	IV.14.
	. Seructure recursivement complete et pointeurs	

8.1. Mode de transmission des paramètres	IV.15.
8.2. Disjonction des paramètres	IV.16.
8.3. Identificateurs "généraux"	IV.16.
9. Questions diverses	IV.16.
9.1. Commentaires dans les ordres "declare", "refine", "create"	IV.16.
9.2. L'option "external"	IV.16.
CHAPITRE V : Sémantique Mathématique du HPL.	
2 to langue de programmation ou compe un artefact	
1. Introduction	V.1.
2. Sémantique Mathématique	V.1.
2.1. Exemple introductif	V.1.
2.2. Sémantique et résolution de problèmes	V.2.
3. Sémantique du Langage "X"	V.2.
3.1. Ensemble des états	V.2.
3.2. Syntaxe du langage "X"	V.3.
3.3. Commandes et expressions	V.4.
3.4. Opérations mathématiques sur les fonctions	V.4.
3.5. Sémantique des "expressions"	V.5.
3.6. Sémantique des "commandes"	V.7.
3.7. Itération et récursion	V.7.
3.8. Sémantique du datatest	V.9.
3.9. Sémantique d'un programme-holon	V.9.
4. Correspondance entre le langage "X" et le "HPL"	V.10.
5. Avantage de la sémantique mathématique	V.11.
CHAPITRE VI : Compilation et exécution d'un programme-holon.	
1. Introduction	
1. Introduction	VI.1.
2 Pêlo du comilator	*** *
2. Rôle du compilateur	VI.1.
3. Requêtes au compilateur	VI.1.
3. Requêtes au compilateur 3.1. New program	VI.1. VI.2.
3. Requêtes au compilateur 3.1. New program 3.2. Purge	VI.1. VI.2. VI.2.
3. Requêtes au compilateur 3.1. New program 3.2. Purge 3.3. Access	VI.1. VI.2. VI.2. VI.2.
3. Requêtes au compilateur 3.1. New program 3.2. Purge 3.3. Access 3.4. Append	VI.1. VI.2. VI.2. VI.2. VI.2.
3. Requêtes au compilateur 3.1. New program 3.2. Purge 3.3. Access 3.4. Append 3.4.1. Cohérence de structure	VI.1. VI.2. VI.2. VI.2. VI.2. VI.3.
3. Requêtes au compilateur 3.1. New program 3.2. Purge 3.3. Access 3.4. Append 3.4.1. Cohérence de structure 3.4.2. Cohérence d'environnement	VI.1. VI.2. VI.2. VI.2. VI.2. VI.3. VI.3.
3. Requêtes au compilateur 3.1. New program 3.2. Purge 3.3. Access 3.4. Append 3.4.1. Cohérence de structure 3.4.2. Cohérence d'environnement 3.5. Erase	VI.1. VI.2. VI.2. VI.2. VI.2. VI.3. VI.3. VI.5.
3. Requêtes au compilateur 3.1. New program 3.2. Purge 3.3. Access 3.4. Append 3.4.1. Cohérence de structure 3.4.2. Cohérence d'environnement 3.5. Erase 3.6. Change	VI.1. VI.2. VI.2. VI.2. VI.2. VI.3. VI.3. VI.5. VI.7.
3. Requêtes au compilateur 3.1. New program 3.2. Purge 3.3. Access 3.4. Append 3.4.1. Cohérence de structure 3.4.2. Cohérence d'environnement 3.5. Erase 3.6. Change 3.7. Synthetise	VI.1. VI.2. VI.2. VI.2. VI.2. VI.3. VI.3. VI.5. VI.7.
3. Requêtes au compilateur 3.1. New program 3.2. Purge 3.3. Access 3.4. Append 3.4.1. Cohérence de structure 3.4.2. Cohérence d'environnement 3.5. Erase 3.6. Change 3.7. Synthetise 3.8. Execute	VI.1. VI.2. VI.2. VI.2. VI.3. VI.3. VI.5. VI.7. VI.8.
3. Requêtes au compilateur 3.1. New program 3.2. Purge 3.3. Access 3.4. Append 3.4.1. Cohérence de structure 3.4.2. Cohérence d'environnement 3.5. Erase 3.6. Change 3.7. Synthetise 3.8. Execute 3.9. Analyseur et synthétiseur	VI.1. VI.2. VI.2. VI.2. VI.3. VI.3. VI.5. VI.7. VI.8. VI.8.
3. Requêtes au compilateur 3.1. New program 3.2. Purge 3.3. Access 3.4. Append 3.4.1. Cohérence de structure 3.4.2. Cohérence d'environnement 3.5. Erase 3.6. Change 3.7. Synthetise 3.8. Execute 3.9. Analyseur et synthétiseur 4. Le synthétiseur	VI.1. VI.2. VI.2. VI.2. VI.3. VI.3. VI.5. VI.7. VI.8. VI.8. VI.8.
3. Requêtes au compilateur 3.1. New program 3.2. Purge 3.3. Access 3.4. Append 3.4.1. Cohérence de structure 3.4.2. Cohérence d'environnement 3.5. Erase 3.6. Change 3.7. Synthetise 3.8. Execute 3.9. Analyseur et synthétiseur 4. Le synthétiseur 4.1. Choix d'une méthode d'implémentation pour un holon	VI.1. VI.2. VI.2. VI.2. VI.3. VI.3. VI.5. VI.7. VI.8. VI.8. VI.8.
3. Requêtes au compilateur 3.1. New program 3.2. Purge 3.3. Access 3.4. Append 3.4.1. Cohérence de structure 3.4.2. Cohérence d'environnement 3.5. Erase 3.6. Change 3.7. Synthetise 3.8. Execute 3.9. Analyseur et synthétiseur 4. Le synthétiseur 4.1. Choix d'une méthode d'implémentation pour un holon 4.2. Détermination de la longueur du code d'un holon	VI.1. VI.2. VI.2. VI.2. VI.3. VI.3. VI.5. VI.7. VI.8. VI.8. VI.8. VI.8.
3. Requêtes au compilateur 3.1. New program 3.2. Purge 3.3. Access 3.4. Append 3.4.1. Cohérence de structure 3.4.2. Cohérence d'environnement 3.5. Erase 3.6. Change 3.7. Synthetise 3.8. Execute 3.9. Analyseur et synthétiseur 4. Le synthétiseur 4.1. Choix d'une méthode d'implémentation pour un holon 4.2. Détermination de la longueur du code d'un holon 4.3. Implémentation des modifications d'environnement	VI.1. VI.2. VI.2. VI.2. VI.3. VI.3. VI.5. VI.7. VI.8. VI.8. VI.8. VI.8. VI.8. VI.9.
3. Requêtes au compilateur 3.1. New program 3.2. Purge 3.3. Access 3.4. Append 3.4.1. Cohérence de structure 3.4.2. Cohérence d'environnement 3.5. Erase 3.6. Change 3.7. Synthetise 3.8. Execute 3.9. Analyseur et synthétiseur 4. Le synthétiseur 4.1. Choix d'une méthode d'implémentation pour un holon 4.2. Détermination de la longueur du code d'un holon 4.3. Implémentation des modifications d'environnement 5. Possibilités d'optimisation du code	VI.1. VI.2. VI.2. VI.2. VI.3. VI.3. VI.5. VI.7. VI.8. VI.8. VI.8. VI.8. VI.8. VI.9. VI.10.
3. Requêtes au compilateur 3.1. New program 3.2. Purge 3.3. Access 3.4. Append 3.4.1. Cohérence de structure 3.4.2. Cohérence d'environnement 3.5. Erase 3.6. Change 3.7. Synthetise 3.8. Execute 3.9. Analyseur et synthétiseur 4. Le synthétiseur 4.1. Choix d'une méthode d'implémentation pour un holon 4.2. Détermination de la longueur du code d'un holon 4.3. Implémentation des modifications d'environnement 5. Possibilités d'optimisation du code 5.1. Résolution des chaînes de "resultis"	VI.1. VI.2. VI.2. VI.2. VI.3. VI.3. VI.5. VI.7. VI.8. VI.8. VI.8. VI.8. VI.8. VI.9. VI.10. VI.11.
3. Requêtes au compilateur 3.1. New program 3.2. Purge 3.3. Access 3.4. Append 3.4.1. Cohérence de structure 3.4.2. Cohérence d'environnement 3.5. Erase 3.6. Change 3.7. Synthetise 3.8. Execute 3.9. Analyseur et synthétiseur 4. Le synthétiseur 4.1. Choix d'une méthode d'implémentation pour un holon 4.2. Détermination de la longueur du code d'un holon 4.3. Implémentation des modifications d'environnement 5. Possibilités d'optimisation du code	VI.1. VI.2. VI.2. VI.2. VI.3. VI.3. VI.5. VI.7. VI.8. VI.8. VI.8. VI.8. VI.8. VI.9. VI.10. VI.11.
3. Requêtes au compilateur 3.1. New program 3.2. Purge 3.3. Access 3.4. Append 3.4.1. Cohérence de structure 3.4.2. Cohérence d'environnement 3.5. Erase 3.6. Change 3.7. Synthetise 3.8. Execute 3.9. Analyseur et synthétiseur 4. Le synthétiseur 4.1. Choix d'une méthode d'implémentation pour un holon 4.2. Détermination de la longueur du code d'un holon 4.3. Implémentation des modifications d'environnement 5. Possibilités d'optimisation du code 5.1. Résolution des chaînes de "resultis" 5.2. Déplacement des déclarations	VI.1. VI.2. VI.2. VI.2. VI.3. VI.3. VI.5. VI.7. VI.8. VI.8. VI.8. VI.8. VI.8. VI.9. VI.10. VI.11.
3. Requêtes au compilateur 3.1. New program 3.2. Purge 3.3. Access 3.4. Append 3.4.1. Cohérence de structure 3.4.2. Cohérence d'environnement 3.5. Erase 3.6. Change 3.7. Synthetise 3.8. Execute 3.9. Analyseur et synthétiseur 4. Le synthétiseur 4.1. Choix d'une méthode d'implémentation pour un holon 4.2. Détermination de la longueur du code d'un holon 4.3. Implémentation des modifications d'environnement 5. Possibilités d'optimisation du code 5.1. Résolution des chaînes de "resultis" 5.2. Déplacement des déclarations 5.3. Transformation des holons récursifs 6. Machine virtuelle pour le HPL	VI.1. VI.2. VI.2. VI.2. VI.3. VI.3. VI.5. VI.7. VI.8. VI.8. VI.8. VI.8. VI.9. VI.10. VI.11. VI.11.
3. Requêtes au compilateur 3.1. New program 3.2. Purge 3.3. Access 3.4. Append 3.4.1. Cohérence de structure 3.4.2. Cohérence d'environnement 3.5. Erase 3.6. Change 3.7. Synthetise 3.8. Execute 3.9. Analyseur et synthétiseur 4. Le synthétiseur 4.1. Choix d'une méthode d'implémentation pour un holon 4.2. Détermination de la longueur du code d'un holon 4.3. Implémentation des modifications d'environnement 5. Possibilités d'optimisation du code 5.1. Résolution des chaînes de "resultis" 5.2. Déplacement des déclarations 5.3. Transformation des holons récursifs 6. Machine virtuelle pour le HPL	VI.1. VI.2. VI.2. VI.2. VI.2. VI.3. VI.3. VI.5. VI.7. VI.8. VI.8. VI.8. VI.8. VI.8. VI.9. VI.10. VI.11. VI.11. VI.11. VI.11.
3. Requêtes au compilateur 3.1. New program 3.2. Purge 3.3. Access 3.4. Append 3.4.1. Cohérence de structure 3.4.2. Cohérence d'environnement 3.5. Erase 3.6. Change 3.7. Synthetise 3.8. Execute 3.9. Analyseur et synthétiseur 4. Le synthétiseur 4.1. Choix d'une méthode d'implémentation pour un holon 4.2. Détermination de la longueur du code d'un holon 4.3. Implémentation des modifications d'environnement 5. Possibilités d'optimisation du code 5.1. Résolution des chaînes de "resultis" 5.2. Déplacement des déclarations 5.3. Transformation des holons récursifs 6. Machine virtuelle pour le HPL  CHAPITRE VII : Le Holon. 1. Introduction	VI.1. VI.2. VI.2. VI.2. VI.2. VI.3. VI.3. VI.5. VI.7. VI.8. VI.8. VI.8. VI.8. VI.9. VI.10. VI.11. VI.11. VI.11. VI.11. VI.11. VI.11. VI.11.
3. Requêtes au compilateur 3.1. New program 3.2. Purge 3.3. Access 3.4. Append 3.4.1. Cohérence de structure 3.4.2. Cohérence d'environnement 3.5. Erase 3.6. Change 3.7. Synthetise 3.8. Execute 3.9. Analyseur et synthétiseur 4. Le synthétiseur 4.1. Choix d'une méthode d'implémentation pour un holon 4.2. Détermination de la longueur du code d'un holon 4.3. Implémentation des modifications d'environnement 5. Possibilités d'optimisation du code 5.1. Résolution des chaînes de "resultis" 5.2. Déplacement des déclarations 5.3. Transformation des holons récursifs 6. Machine virtuelle pour le HPL	VI.1. VI.2. VI.2. VI.2. VI.2. VI.3. VI.3. VI.5. VI.7. VI.8. VI.8. VI.8. VI.8. VI.8. VI.9. VI.10. VI.11. VI.11. VI.11. VI.11.

CHAPITRE VIII : Exemples de programmes.	
1. Introduction	VIII.1
2. Premier exemple	VIII.1
3. Deuxième exemple	VIII.6
CHAPITRE IX : Conclusions.	
1. "System analysis" et "programming"	IX.1.
2. Le langage de programmation vu comme un artefact	IX.1.
3. Une (tentative de) définition de la programmation	IX.2.
4. La programmation comme une discipline	IX.3.
ANNEXE A : La "programmation structurée" de E.W. Dijkstra.	
A. Taille d'un programme	A.1.
B. Exactitude d'un programme	A.1.
C. Abstraction et instructions "structurées"	A.2.
D. "Data structures" abstraites	A.3.
Commentaires sur la programmation structurée	A.3.
1. Exactitude et efficacité	A.3.
2. Programmes structurés et problèmes à résoudre	A.4.
3. Data structures "abstraites"	A.5.
4. Conclusions	A.8.

Références citées dans le texte.

#### CHAPITRE I

#### DE L'IDEE D'UN PROGRAMME AU PROGRAMME.

#### I. APERCU HISTORIQUE

Nous allons exposer brièvement ce qui nous semble être l'évolution dans la manière de considérer le rôle des langages de programmation; nous distinguerons, un peu arbitrairement, trois périodes : pragmatique, théorique et méthodologique.

Les premières machines à calculer électroniques furent le résultat des efforts des spécialistes de l'analyse numérique afin de disposer d'un moyen sûr et rapide pour mener à bien leurs opérations de calcul. Le point de vue de ces pionniers de l'informatique vis-à-vis des langages de programmation était à l'origine purement "pragmatique"; en effet, vu le domaine d'application, un "langage" de programmation n'était point nécessaire; il existait seulement une méthode de "codage" en instructions-machine d'un algorithme développé selon les méthodes de raisonnements mathématiques propres à l'analyse numérique. La phase de "codage" de l'algorithme était une opération difficile et éprouvante mais d'un niveau d'invention inférieur à celui de l'élaboration de l'algorithme mathématique. Il en résulta : d'une part, la conception de langages qui permettaient de s'éloigner des particularités des répertoires d'instructions-machine en se rapprochant de la forme mathématique des opérations de calcul d'un algorithme; le FORTRAN fut, sans conteste, le premier résultat notable dans ce domaine.

D'autre part, le programmeur récupéra l'initiative qui lui était refusée dans la conception de l'algorithme, par la création de toute une série de "trucs" de programmation basés sur une connaissance des particularités du "hardware" des machines et destinés à améliorer les performances de la version codée.

A la période (1) précédente où le langage de programmation est assimilé à un "instrument" utile pour passer d'un algorithme à sa version codée, succéda une période "théorique" où l'accent fut mis sur l'étude des algorithmes d'un point de vue formel : les éléments syntaxiques de base pour construire un algorithme sont inventoriés et étudiés sans tenir compte de la signification de l'algorithme. Cette étude procéda d'une salutaire réaction contre le point de vue pragmatique qui avait généré des langages imposant des restrictions arbitraires à la représentation externe d'un algorithme, restrictions dont l'origine provenait de difficultés ou de particularités d'implémentation, (par exemple, les restrictions imposées en FORTRAN à la valeur de la variable "contrôlée" d'une boucle "DO").

Ce point de vue fut très bien exprimé par le concept d'Algol Généralisé de A. van Wijngaarden [vWi63]: un langage ne doit pas être surchargé de règles syntaxiques qui définissent des textes significatifs, mais doit comporter au contraire les éléments fondamentaux des algorithmes, la possibilité de les composer librement et d'exprimer d'une manière concise un algorithme.

Ces principes furent appliqués par N. Wirth dans le langage EULER [Wir66]; il en résulta - selon les termes mêmes de son auteur [Wir74] - : "une machine de Turing de haut-niveau où il était difficile et presque impossible de détecter une erreur de logique dans le programme".

Le résultat le plus fécond de cette période fut probablement l'étude intensive des grammaires, des méthodes de définition de la syntaxe et de l'analyse syntaxique; les réalisations majeures furent l'ALGOL 60 et le LISP 1.5.

Cette période s'acheva dans une certaine confusion: d'une part, la conception de nouveaux langages fut confiée à des "comités"; on oubliait que le résultat du travail d'un comité est rarement la somme des talents des individus qui le composent, mais plus fréquemment leur plus petit commun dénominateur; d'autre part, la disponibilité d'engins capables de traiter de grandes quantités de données à grande vitesse suscita une multitude d'idées d'application dont les difficultés de réalisation et les nombreux échecs engendrèrent ce qu'on appela la "software crisis".

Le point de vue méthodologique fut proposé par E.W. Dijkstra [Dij68,Dij72] ; on peut le résumer

<sup>(1)</sup> Le terme "état d'esprit" serait plus adéquat que "période".

comme suit : il importe de déceler les schémas de raisonnement fondamentaux appliqués par le programmeur lors de la conception d'un programme et non les éléments fondamentaux des algorithmes. La recherche de ces principes fondamentaux devrait nous amener à une conception plus rationnelle des langages de programmation et à ce que l'objet du travail des inventeurs de langages de programmation soit pratiquement la spécification de la structure du "hardware" des ordinateurs.

On peut comparer cette attitude à l'attitude théorique, en envisageant, par exemple, la récursion : d'un point de vue théorique, on introduira la récursion comme résultant des règles générales de substitution de paramètres dans le texte d'un programme; d'un point de vue méthodologique, la récursion sera une technique de résolution d'un problème où l'on transforme progressivement le cas donné en un cas dont la solution est connue.

La recherche de ces schémas fondamentaux de raisonnement est justifié par la supposition qu'ils doivent être en réalité peu nombreux et que leur détermination pourra fournir une base cohérente pour l'enseignement de l'art de la programmation. Dijkstra appela globalement l'ensemble de ces idées "programmation structurée" [Dij72]; les interprétations multiples données à ce terme ont introduit malheureusement une certaine confusion; dans l'annexe A, nous donnons une brève description de la "programmation structurée", suivie de commentaires.

Cette troisième période n'a pas encore donné le jour à un langage de programmation basé sur ce point de vue. (1)

#### 2. ENGINEERING ET SOFTWARE ENGINEERING

Le terme "Software Engineering" fut forgé dans les années 60 par analogie avec les termes "Electrical Engineering", "Mechanical Engineering", etc... Par "Software Engineering", on entend la discipline de conception et de production de programmes en général de "grande taille". Il existe une différence fondamentale entre le "Software Engineering" et les "Engineerings classiques": la nature "immatérielle" de l'objet produit. Il en résulte que les méthodes de conception et de travail utilisées en "engineering classique" ne sont pas vraiment adaptées à la programmation.

La spécification primordiale d'un engin à construire en "engineering classique" est la définition du comportement de cet engin dans les conditions d'utilisation normale; lorsque le principe du schéma général de l'engin est découvert, les caractéristiques et dimensions de ses composants sont déterminées par l'application de modèles approximatifs de la réalité, c'est-à-dire que l'on suppose que le comportement des matériaux (au sens large) qui composent l'engin, est identique au comportement prédit par un modèle mathématique; (2) cette attitude est d'autant plus légitime que les caractéristiques des éléments utilisés dans la réalisation pratique ne sont jamais connues qu'approximativement et varient en général à l'usage. Il importe bien entendu que le choix du modèle soit correct, c'est-à-dire que le modèle choisi ne néglige aucun des éléments qui agissent d'une manière significative sur le comportement de l'engin.

Le programmeur ne peut utiliser un modèle approximatif car le résultat de son travail est un automate passant par une série d'états, tous également importants pour garantir la production du résultat attendu; tout au plus, peut-il utiliser des "abstractions", dans le sens qu'il peut concevoir le programme en termes d'opérations définies "par ce qu'elles font" et décider ultérieurement "comment elles le font"; l'introduction de ces "abstractions" a pour but d'être extrêmement précis dans la définition des opérations et non d'être vague.

La distinction entre "modèle approximatif" et "abstraction" est importante, car ce qu'on a coutume d'appeler "période de tests" ne peut fournir en programmation des informations aussi précieuses qu'en "engineering classique"; en effet, dans le cas d'un engin construit sur base d'un modèle approximatif, les tests effectués sur la réalisation concrète permettent de vérifier :

- 1) si les aspects négligés par le modèle sont effectivement négligeables,
- 2) si le modèle reflète suffisamment l'influence de l'usage de l'engin sur son comportement;

<sup>(&</sup>lt;sup>1</sup>) Des études sont cependant en cours, notamment au M.I.T. avec le langage CLU.5 de Barbara Liskov

et à Newcastle-upon-Tyne avec le système TOPD de Peter Henderson. (<sup>2</sup>) Il en est ainsi quand on suppose que "l'air se comporte comme un gaz parfait", que "la transformation est adiabatique", que "les phénomènes transitoires sont négligeables", etc...

d'autre part, les différents environnements susceptibles d'accueillir l'engin sont en nombre limité et leurs influences peuvent généralement être testées d'une manière exhaustive. Dans le cas d'un programme, les trois possibilités précédentes n'existent pas : il n'y a ni phénomènes négligeables - car tous les états de l'automate sont d'égale importance -, ni "usure", et le nombre d'états possibles est tellement élevé - même pour les automates les plus simples (1) - qu'une vérification exhaustive de ceux-ci est impossible.

D'autre part, le comportement des engins est généralement "observable" pendant leur fonctionnement en exploitation; les dispositifs d'observation et de contrôle sont généralement sans effet notable sur le rendement de l'engin. Au contraire, tout contrôle d'un programme en cours d'exécution implique la mise en service d'opérations de vérification qui seront une charge importante en temps-machine et occupation de la mémoire. La seule issue est une étude complète des propriétés de l'automate représenté par le programme afin de limiter au maximum le système de contrôle.

#### 3. LANGAGES ET METHODOLOGIE DE PROGRAMMATION

Le but de la méthodologie de la programmation est de donner au programmeur des règles qui lui permettent de décider de la nature et de l'ordre des décisions qu'il prend lors de la conception d'un programme; celle-ci est considérée comme étant essentiellement un processus évolutif transformant progressivement l'idée d'un programme en un programme exécutable. La question est de savoir dans quelle mesure ce caractère évolutif de la conception d'un programme doit apparaître dans les langages de programmation. Il y a trois manières de répondre à cette question :

- a) les langages actuels possèdent suffisamment d'éléments pour faire apparaître ce caractère si on le désire;
- b) il faut compléter les langages par une espèce de méta-langage qui tient compte du caractère évolutif; on utilise ce méta-langage dans la phase de conception qui est suivie d'une phase de "codage" pendant laquelle le programme est traduit du méta-langage en un des langages de programmation actuels;
- c) le langage de programmation utilisé doit refléter dans sa structure le caractère évolutif de conception d'un programme.

Dans la première réponse, on considère que quelle que soit la forme des raisonnements proposés par la méthodologie, le programme qui en résultera sera toujours un "flow-chart" qui pourra être exprimé dans un des langages actuels.

La deuxième réponse procède d'une étude plus fouillée de la méthodologie et d'une volonté de la mettre en pratique avec les moyens actuels. Le promoteur de cette forme de programmation est Harlan D. Mills de l'IBM Federal Systems Division. Il utilise comme méta-langage une forme de "flow-charting" obéissant à des règles sévères quant au choix des éléments qui peuvent apparaître dans les "flow-charts" [Mil72,Mil75]. Outre une présentation des idées de Dijkstra sous une forme plus mathématique utilisant la notion de fonction, le grand mérite de la méthode de Mills est d'avoir pu servir de point de départ à une nouvelle forme d'organisation du travail des programmeurs : le "Chief Programmer Team" [Bak72a,Bak72b]. En fait, en partant de l'outil "cher" à de nombreux programmeurs (le "flow-chart") et en le modifiant de manière adéquate, Mills a réussi à imposer aux groupes qui programment selon le "Chief Programmer Team" une méthodologie de travail qu'ils auraient difficilement acceptée si la présentation eut été tout autre (celle de Dijkstra, par exemple).

Toutefois, la méthode comporte deux inconvénients :

a) il est nécessaire de transformer le programme développé dans le méta-langage en un - ou plusieurs (2) langage utilisable sur l'équipement. Cette opération de traduction est de nature presque mécanique et il semblerait plus logique de confier celle-ci à un'algorithme plutôt qu'au programmeur.

<sup>(1)</sup> Un simple additionneur de nombres binaires de 32 chiffres possède  $2^{64}$  états différents; si on peut tester un état par seconde (un an =  $2^{28}$  secondes), il faut  $10^{10}$  ans !

<sup>(2)</sup> Plusieurs langages si certaines parties du programme développé se rapportent à des opérations de l'Operating Systems ou du Linkage Editor; celles-ci sont codées en Job Control Language.

b) le programme, tel qu'il résulte de la conception, possède certaines propriétés qu'un compilateur pourrait utiliser pour améliorer la qualité et l'efficacité du code généré; ces propriétés concernent la nature arborescente du programme qui est composé par une succession de fonctions emboîtées les unes dans les autres. Lorsque le programme est traduit du méta-langage en PL/I par exemple, le compilateur PL/I ignore bien entendu la nature "structurée" du programme, et dans sa phase d'analyse syntaxique il entreprend de "désembrouiller" la structure du programme qui existe clairement dans la version écrite en méta-langage. La détection d'une forme de structure dans un programme sert de base aux méthodes d'optimalisation les plus poussées du code généré, il serait donc plus intéressant de ne point perdre les informations que contient la version en méta-langage. (1)

Malgré ces deux inconvénients, l'utilisation d'un méta-langage est,à notre avis, la manière la plus sûre d'entreprendre et de mener à bien un projet de grande dimension, tant qu'on ne dispose pas directement d'un langage qui nous permette d'appliquer directement la méthodologie qui a notre préférence.

#### Remarques :

- 1. Puisque les programmes conçus par méta-langage sont finalement transcrits en langage classique, on se demande s'il est correct de faire une telle distinction entre les deux méthodes. En fait, la distinction est très nette, car si les langages sont apparemment identiques, ils ne le sont pas en réalité : en effet, les utilisateurs du langage classique gardent une conception "combinatoire" de la syntaxe du langage, ne rejetant a priori aucune association grammaticalement correcte des éléments syntaxiques; les utilisateurs d'un méta-langage restreignent considérablement l'ensemble des programmes syntaxiquement corrects qu'ils sont prêts à composer. Si on caractérise un langage non plus par sa seule syntaxe, mais par le sous-ensemble des programmes "admissibles" par un programmeur, on voit immédiatement que de part et d'autre, on n'utilise pas le "même" langage.
- 2. Il est fréquemment avancé que les résultats du "Chief Programmer Team", notamment au point de vue de l'accroissement de la productivité du programmeur, ne seraient pas dûs à la méthodologie employée mais plutôt à l'organisation administrative qui permet de concentrer tout l'effort de la programmation sur un seul individu (le chief-programmer) qui est assisté des programmeurs de "support" et qui est seul à prendre les décisions importantes dans la création du programme. L'argument est que quelle que soit la méthodologie utilisée, la productivité serait accrue parce que toutes les distractions administratives sont éliminées et que les communications entre les individus sont réduites [Abt75]. Personnellement, ayant eu l'occasion d'observer sur place le fonctionnement du système, il nous semble que le côté administratif n'explique pas tout, car en réalité, en plus de la concentration des efforts sur un petit groupe, il y a le fait plus important que tous les membres du groupe expriment leurs programmes dans le même méta-langage; il en résulte un "style" commun. Un aspect important de la méthode, rarement cité, est que chaque pièce de programme est lue par au moins deux personnes qui en certifient l'exactitude par rapport aux spécifications; si le "style" n'était pas commun, cette opération prendrait énormément de temps. Le "chief programmer team" ne prouve rien au point de vue de l'efficacité méthodologique de la "programmation structurée", mais il montre qu'il faut une "communauté" méthodologique parmi les membres d'une équipe de conception. Rétrospectivement, comme la majorité des idées efficaces, elle paraît évidente, et d'une certaine manière elle se rattache à la pratique courante des bureaux d'études où il n'est pas permis que chacun utilise son propre système d'unités de mesure ou ses propres symboles non-standards pour dessiner schémas et plans; mais il est nécessaire qu'existe une méthode "standard" efficace et facile à enseigner aux membres de l'équipe.

Les deux inconvénients cités disparaissent si le méta-langage est transformé en un langage de programmation, mais dans ce cas, on court le risque de se trouver en possession d'un outil aux possibilités insuffisantes. Le méta-langage peut toujours accepter quelques modifications temporaires; en fait le méta-langage est un langage de "conception" auquel succédera l'utilisation d'un langage "d'implémentation". Un langage basé sur des considérations méthodologiques doit donc pouvoir refléter exactement tous les aspects de la méthodologie utilisée; par exemple, le méta-langage de "flow-chart"structuré" du Dr. Mills permet de représenter d'une manière adéquate le raffinement des opérations d'un programme structuré, mais il ne pourvoit en rien au raffinement des données. Vu les rectrictions introduites par un langage d'origine méthodologique, il est nécessaire que la portée de celles-ci soit clairement évaluée afin de les justifier.

<sup>(1)</sup> On peut comparer ce cas à l'analogue suivant : un compilateur qui produit une version écrite en assembleur du programme à compiler; lorsque ce programme écrit en assembleur est traité par le programme d'assemblage, celui-ci reconstitue des tables de symboles presque semblables aux tables utilisées par le compilateur.

Un langage de developpement de programmes est un langage de programmation qui permet au programmeur d'exprimer toutes les étapes du développement d'un programme et qui l'aide à transformer l'idée d'un programme en un programme concret.

A notre avis, un langage de ce type devrait posséder les sept caractéristiques suivantes :

- permettre de représenter les différentes phases de l'évolution d'un programme; le programmeur doit avoir la possibilité d'exprimer sous forme d'une ébauche de programmes toutes les abstractions qu'il utilise, aussi bien les abstractions d'opérations que les abstractions de structures de données;
- permettre d'exprimer chaque décision prise, de la documenter complètement et de décrire explicitement les conséquences de celle-ci;
- 3) posséder une structure logique qui permette d'utiliser une méthode de vérification des programmes; par vérification, on comprend aussi bien une vérification a priori du programme lors de sa construction que des opérations de vérification a postériori, même pendant l'exécution;
- 4) les domaines d'application du langage doivent être aussi variés que possible; il faut veiller spécialement à ce que certaines applications ne soient pas nettement défavorisées par rapport à d'autres;
- 5) aider le programmeur dans les opérations de modification d'un programme, et permettre une "modularisation" aisée d'un grand programme ("module" signifiant tâche confiée à un programmeur);
- 6) générer un code-machine efficace; en d'autres termes, la structure du langage doit permettre de réaliser une optimalisation du code généré;
- 7) la définition du langage doit être simple et peu volumineuse; prendre connaissance de la syntaxe et de la sémantique doit être possible en un temps suffisamment court, la documentation étant brève mais complète.

Dans les chapitres suivants, nous traitons de la conception et de la définition d'un langage de ce type : le "holon programming language". Ce langage possède pratiquement, quoique de manière inégale, les sept caractéristiques proposées.

Nous prendrous un nutre point de déficientementation manuel des fondame

#### CHAPITRE II

#### DEFINITION DU LANGAGE HPL.

#### I. INTRODUCTION

Dans ce chapitre, nous donnons une description d'un langage de programmation par holons (en abrégé : HPL, Holon Programming Language).

La syntaxe et la sémantique de ce langage seront décrites d'une manière informelle; un modèle abstrait de l'exécution d'un programme écrit en HPL est décrit dans ce chapitre; le chapitre V contient une description de la sémantique mathématique du langage.

#### 2. PRINCIPES DE CONCEPTION DU HPL

Les langages de programmation sont destinés à exprimer des algorithmes; le problème initial de la conception d'un langage de programmation est l'ignorance totale des algorithmes qui seront "programmés" dans le langage. On sait que tout algorithme peut être exprimé par un "flow-chart"; la conséquence de cette constatation est visible dans la structure des langages actuels de progammation (ALGOL, FORTRAN, PL/1, PASCAL,...): tous les mécanismes élémentaires du "flow-chart" (la succession, le test, le branchement) auxquels des variations ou des "facilités" syntaxiques ont été adjointes, sont présents dans leurs syntaxes; par exemple, la notion de "procédure" est définie et expliquée selon la "règle de copie" qui est fondamentalement une opération "syntaxique"; le cas le plus aigu d'accumulation de facilités syntaxiques est probablement le PL/1.

Il résulte de cette manière de concevoir les langages de programmation que l'ensemble des programmes qu'on peut écrire par leur intermédiaire comprend tous les programmes utilisables et, en outre, une multitude de programmes "absurdes". En fait, dans la phrase précédente nous appelons "programme" toute combinaison des règles syntaxiques d'un langage, ce qui est manifestement une usurpation du sens du mot; elle est aussi illégitime que de qualifier de "démonstration" tout texte écrit selon les règles de la grammaire !

Nous prendrons un autre point de départ que celui des mécanismes fondamentaux du "flow-chart" pour développer le langage HPL. Une base de départ adéquate est une définition plus réaliste du terme "programme" :

un "programme" est la description d'une séquence d'opérations primitives dont l'exécution produit un effet-net.

Par cette définition, nous insistons sur la production effective d'un effet-net par un "programme". Nous nous assignons comme but la conception d'un langage de "programmation", c'est-à-dire d'une méthode de description (et également de construction) d'objets qui sont des "programmes".

La tâche d'un programmeur est la conception d'un programme dont l'effet-net correspond à l'effetnet exigé par les spécifications du programme; cette conception est,en fait, le choix, parmi tous les objets que peut générer l'application des règles syntaxiques du langage, de l'objet dont l'effetnet correspond à l'effet-net spécifié. Nous pouvons décider de n'incorporer dans le langage HPL que des règles syntaxiques susceptibles d'aider le programmeur dans le construction d'un "programme".

Pour qu'un programme produise effectivement un effet-net, il est nécessaire que cet effet-net résulte d'une exécution qui dure un temps fini; en d'autres termes, que l'exécution du programme "se termine". Cette condition appliquée à la réciproque du théorème de König sur les arbres orientés infinis donne une description presque complète du principe de conception du langage HPL.

Le théorème de König établit que :

dans tout arbre orienté infini dont chaque noeud a un degré fini, il existe un "chemin infini issu de la racine".  $^{(1)}$ 

<sup>(</sup>¹) König exprime également ce théorème comme suit : "si l'on suppose que la race humaine ne disparaîtra jamais, il existe un homme, actuellement en vie, dont la lignée ne s'éteindra jamais".

La réciproque (en fait, la "contrapositive") que nous utilisons a été énoncée par D.E. Knuth [Knu68, page 382]:

"si un algorithme se divise périodiquement en un nombre fini de sous-algorithmes, et si chaque chaîne de sous-algorithmes se termine, alors l'algorithme se termine".

Cette "contrapositive" du théorème de König va nous servir de guide pour définir les grandes lignes de la conception du HPL :

- 1) la syntaxe représentera essentiellement des divisions périodiques d'un même algorithme;
- 2) chaque chaîne de sous-algorithmes doit se terminer; pour garantir cette propriété, la structure de la syntaxe permettra de détecter et d'éliminer les sous-algorithmes dont la structure ne peut correspondre à celle d'un algorithme qui se termine; quant aux chaînes non-éliminées, il faudra établir les conditions qui garantiront qu'elles se terminent;
- 3) finalement, il reste à établir que l'effet-net produit par l'algorithme est bien l'effet-net escompté.

Ces trois principes de base délimitent la part de l'activité de conception d'un programme qui est automatique et celle qui reste humaine :

La division d'un algorithme en sous-algorithmes admet une multitude de solutions, même si on limite les choix de divisions à la combinaison d'un nombre restreint d'éléments primitifs; le choix d'une division résultera d'une véritable "invention" et non d'une simple sélection mécanique. Cette "invention" est du ressort du programmeur, mais un mécanisme peut vérifier constamment si les "inventions" du programmeur obéissent aux règles fixées pour la division de l'algorithme et rejeter les structures de division qui ne pourront jamais correspondre à un programme.

D'autre part, si la structure de l'algorithme vérifie les conditions nécessaires à l'existence d'un "programme" (ces conditions sont de nature purement syntaxique), le système automatique peut également indiquer les conditions que doit vérifier toute tentative d'exécution du programme pour que celui-ci produise un effet-net. On verra ultérieurement que ces conditions porteront sur l'existence éventuelle de valeurs booléennes déterminées pour certaines expressions de l'algorithme; à partir de la structure syntaxique du texte du programme, un système automatique peut préciser quelles sont les expressions en cause et quelle est la valeur booléenne que toute exécution doit produire pour chacune de ces expressions afin de garantir que le texte de l'algorithme est un "programme".

Le programmeur doit conduire les raisonnements nécessaires pour convaincre de la production effective de ces valeurs . En pratique, il arrivera à des affirmations conditionnelles, remplaçant une (ou des) condition, située par le système automatique en un endroit précis de l'algorithme, par une condition sur l'état de l'environnement située en ament de la première condition.

Ensuite, le programmeur doit montrer que l'ensemble des effets-nets produits par les exécutions de l'algorithme correspondent aux effets-nets spécifiés; s'il n'en est point ainsi, le programmeur peut soit renforcer (rendre plus restrictives) les conditions sur l'état de l'environnement afin d'éliminer les effets-nets indésirables qui appartiennent à l'ensemble des effets-nets produits, soit modifier la division de l'algorithme. Cette dernière décision peut également résulter d'une analyse des performances de l'algorithme qui aurait révélé des possibilités d'amélioration de l'efficacité de l'exécution du programme (moins d'opérations élémentaires) ou de diminution de la quantité des ressources mobilisées pour l'exécution (moins d'espace mémoire).

Toute conception d'un programme implique que le programmeur accomplisse ces tâches d'invention et d'analyse; le langage HPL est conçu en tentant de tenir compte des obligations du programmeur et d'automatiser au maximum les tâches qui peuvent l'être. La partie "automatique" de la conception d'un programme reste relativement minime; pourtant, les possibilités proposées par le HPL sont de loin supérieures à tout ce qu'on peut obtenir, à l'heure présente, d'une implémentation du FORTRAN, de l'ALGOL ou du PL/I.

Il n'est point interdit d'imaginer la création future de procédés complètement automatiques qui généreraient un programme à partir de la description de ses spécifications, toutefois il faut considérer plus humblement que pour les tâches que nous attribuons au programmeur, nous ne possédons pas vraiment de méthodes : il s'écoulera encore beaucoup de temps avant que des méthodes "mécanisables" soient élaborées pour effectuer la division d'un algorithme en sous-algorithmes ou pour

déterminer les conditions de terminaison d'un algorithme, sauf si - et malheureusement, c'est ce qui arrivera - on se limite artificiellement à des classes de problèmes suffisamment simples et probablement dénués de tout intérêt pratique.

#### 3. ELEMENTS FONDAMENTAUX DU HPL

Les éléments fondamentaux du langage HPL sont :

a) les variables,

et

b) les holons.

Par "variable", on désigne un objet dont la valeur peut varier au cours de l'exécution d'un programme. L'ensemble de toutes les variables qui existent à un moment précis de l'exécution d'un programme forme l'environnement de ce programme à cet instant précis.

Le chapitre IV est entièrement consacré à l'étude des variables et de l'environnement; pour ce qui suit, il importe seulement de savoir que, d'une part, un nom peut être associé à chaque variable et permet de l'identifier, et que, d'autre part, il est possible de déclarer une variable, c'est-à-dire d'ajouter un élément à l'ensemble des variables et de lui attribuer un nom.

Par "holon", on désigne une opération dont l'exécution produit un effet-net, c'est-à-dire une modification des valeurs des variables d'un programme, et dans certains cas, une modification de l'environnement de ce programme.

Un holon possède :

- a) une définition qui est la spécification de son effet-net,
- b) une description qui est une représentation de son mode d'exécution.

En d'autres termes, la définition spécifie "ce que le holon fait", la description explique "comment il le fait". La définition définit un et un seul holon, mais ce holon peut être décrit de diverses manières.

On dira conventionnellement qu'un holon est défini par son "nom de holon" et qu'il est décrit par son "corps de holon". On utilisera la représentation schématique suivante pour un holon H<sub>i</sub> de nom de holon nh, et de corps de holon bd. :

Le choix du terme "holon" sera justifié ultérieurement; étymologiquement, ce mot signifie "partie d'un tout"; dans le cas présent, il est utilisé pour désigner un nouveau concept de programmation qu'il ne faut surtout pas associer à celui de procédure, à cause de l'analogie possible avec "nom" et "corps" de procédure.

Les holons sont associés selon des règles précises pour former un programme-holon. Celui-ci est un ensemble (qui peut être vide) de holons; il est désigné par un "nom de programme"; on représentera comme suit le programme-holon de nom np et formé par un ensemble de n holons :

$$P[np, \{H_0, H_1, \dots, H_{n-2}, H_{n-1}\}].$$

#### 4. REGLES DE CONSTRUCTION D'UN PROGRAMME-HOLON

Afin d'expliquer les règles de construction d'un programme-holon, nous devons expliciter la structure d'un corps de holon.

Pour représenter la description d'un holon, un corps de holon contient une séquence de déclarations de variables et une séquence de noms de holons; la première séquence décrit les variables que ce holon ajoute à l'environnement du programme, la deuxième les holons qui sont utilisés pour produire l'effet-net attendu, cette dernière séquence spécifie également comment les holons dont elle contient les noms, collaborent pour produire l'effet-net. Toutefois, pour définir les règles

de construction d'un programme-holon, nous pouvons ignorer cette caractéristique de la deuxième séquence; nous représenterons désormais un holon par :

où  $\Sigma$  de et  $\Sigma$  nh symbolisent respectivement la séquence des déclarations et celle des noms de holon. La séquence des déclarations peut être vide.

Les règles de construction portent essentiellement sur les conditions que doit vérifier un holon qu'on désire ajouter à l'ensemble des holons qui forment un programme-holon.

Initialement, seul existe le nom du programme, l'ensemble des holons est vide, soit :

 $P[np, \{\phi\}]$ , où  $\phi$  désigne l'ensemble vide.

On notera par "+" l'adjonction d'un holon à un programme-holon; l'introduction du premier holon d'un programme-holon est symbolisée par :

$$P[np, \{\phi\}] + H_0 \Rightarrow P[np, \{H_0\}]$$

Cette opération est subordonnée à la vérification de la condition suivante :

le nom nh  $_{\rm O}$  du holon H  $_{\rm O}$  doit être identique au nom np du programme P, soit nh  $_{\rm O}$   $^{\rm E}$  np.

Un programme-holon contenant n holons est un programme légal si les conditions d'addition d'un i-ème holon à un programme de (i-1) holons ont été vérifiées pour  $o \le i \le n$ .

Soit un programme P légal contenant n holons :

$$P[np, \{H_0, H_1, H_2, \dots, H_{n-1}\}],$$

nous allons expliciter les conditions qui doivent être vérifiées pour que le même programme P auquel on adjoint un holon  $H_n$ , soit un programme légal.

Cette adjonction de H correspond à l'opération :

$$P[np, \{H_0, H_1, H_2, \dots, H_{n-1}\}] + H_n[nh_n, bd_n(\epsilon, \epsilon, \epsilon, h)]$$

$$\Rightarrow$$
M np,  $\{H_0, H_1, H_2, \dots, H_{n-1}, H_n\}$ 

Les conditions d'exécution de cette opération sont :

- 1)  $\forall$   $nh_i$  (o  $\leq$  i  $\leq$  n-1) :  $nh_i \neq nh_n$  , c'est-à-dire que, dans un programme légal, un nom de holon désigne un et un seul holon.
- 2) ∃nh dans les bd<sub>i</sub>(Σ dc,Σ nh) (o ≤ i ≤ n-1) : nh = ·nh<sub>n</sub> c'est-à-dire que le nom nh<sub>n</sub> du holon H<sub>n</sub> est identique à un nom de holon cité au moins une fois dans une (ou plusieurs) séquence Σnh appartenant à un holon H<sub>i</sub>, avec o ≤ i ≤ n-1.
- 3) le holon  $\mathbf{H}_n$  ne peut impliquer aucune incohérence de structure et/ou d'environnement lors de son adjonction au programme  $\mathbf{P}$ .

A ce stade de l'exposé, il n'est pas possible de développer ce qu'on entend par "incohérence" de structure et/ou d'environnement; le sens de ces expressions apparaîtra par la suite.

Quant aux "nh" qui appartiennent à  $\operatorname{bd}_n(\Sigma \operatorname{dc},\Sigma \operatorname{nh})$ , chacun de ceux-ci doit vérifier l'un des prédicats suivants :

- 1)  $nh = nh_i$ , avec  $0 \le i \le n-1$ ;
  - 2) nh = nh,

ce cas est à distinguer du précédent parce qu'il détermine les opérations de nature directement récursive; la première condition peut également introduire des opérations récursives, mais de nature indirecte. 3)  $\underline{\text{not}}$   $\exists \text{nh}_i \ (\text{o} \leq i \leq n) : \text{nh}_i \equiv \text{nh}$ 

dans ce cas, le nh correspond à la définition d'un nouveau holon dont la description sera ajoutée ultérieurement au programme.

Enfin, on dira qu'un programme-holon P contenant (n+1) holons est complet, si et seulement si :

- 1) P est un programme légal, et
- 2) pour tout nh appartenant à un bd<sub>i</sub> (avec o ≤ i ≤ n), on a deux possibilités, soit qu'il existe un H<sub>j</sub> tel que nh<sub>j</sub> ≡ nh, soit que nh est le nom d'un holon dit terminal, c'est-à-dire une opération dont la description en termes d'instructions d'une machine donnée est connue du système chargé d'exécuter le programme.

#### 5. STRUCTURE DU CORPS DE HOLON

#### 5.1. Relation d'enclenchement.

Entre un holon H et les holons  $\Sigma$  nh contenus dans son corps de holon, il existe une relation de nature hiérarchique : la relation d'enclenchement.

Le holon H; "enclenche" le holon H; si et seulement si :

le nom nh du holon  $H_j$  apparaı̂t au moins une fois parmi les noms de holon  $\Sigma$  nh du corps du holon  $H_i$ .

Soit H<sub>i</sub>[nh<sub>i</sub>,bd<sub>i</sub>(\(\sigma\) dc,\(\sigma\) nh)]

et  $H_{j}[nh_{j},bd_{j}(\dots)]$ ,

 $H_i$  enclenche  $H_i$ , relation que nous noterons  $(H_i + H_i)$ , si et seulement si :

gnh  $\epsilon$  bd<sub>i</sub>( $\Sigma$  dc, $\Sigma$  nh) tel que nh  $\equiv$  nh<sub>i</sub>.

La relation d'enclenchement est non transitive : si  $(H_i + H_j)$  et si  $(H_j + H_k)$ ,  $(H_i + H_k)$  seulement si  $nh_k$  appartient à  $bd_i(\Sigma dc, \Sigma nh)$ .

On appellera"ensemble des subordonnés directs" d'un holon  $H_i$ , l'ensemble des holons  $H_j$  tels que  $(H_i + H_j)$ . Cet ensemble contient donc les mêmes éléments que la séquence  $\Sigma$  nh de  $bd_i$ .

De même, on appellera"ensemble des supraordonnés directs" d'un holon  $H_j$ , l'ensemble des holons  $H_j$  tels que  $(H_i + H_j)$ . La deuxième condition d'exécution d'une opération d'addition d'un holon à un programme P garantit que, dans un programme légal, l'ensemble des supraordonnés directs de  $H_j$  n'est jamais vide pour i > o.

On forme l'ensemble des subordonnés directs et indirects d'un holon  $H_i$  en ajoutant à l'ensemble des subordonnés directs de  $H_i$  que nous notons  $\{H_j\}$ , les ensembles de subordonnés directs de chaque  $H_i$  appartenant à  $\{H_i\}$ , et ainsi de suite...

Par un processus identique, on forme l'ensemble des supraordonnés directs et indirects; le holon  $H_{\hat{0}}$  appartient toujours à cet ensemble, quel que soit le holon  $H_{\hat{1}}$  pour lequel on le construit.

Puisque le langage est construit sur la base de la notion de holon, la syntaxe qui sera choisie pour le représenter donnera une place primordiale à la représentation d'un holon, ce choix implique qu'il existera une forme de représentation immédiate de l'ensemble des subordonnés directs d'un holon  $\mathbf{H_i}$  puisque tous les éléments de cet ensemble seront groupés dans le corps du holon  $\mathbf{H_i}$ . Au contraire, il n'existera aucune correspondance immédiate entre la représentation d'un holon  $\mathbf{H_i}$  et l'ensemble de ses supraordonnés directs; cet ensemble constitue une information de toute première importance pour le programmeur pendant la conception du programme, car cet ensemble permet de déterminer les divers environnements qui peuvent exister lors d'un enclenchement de  $\mathbf{H_i}$ ; il est indispensable de connaître exactement cet ensemble soit quand on va entreprendre la description de  $\mathbf{H_i}$ , soit quand on envisage d'ajouter un nouvel élément à cet ensemble. En d'autres termes, on peut affirmer que l'ensemble des supraordonnés est une information aussi cruciale pour le programmeur que ne l'est l'ensemble des subordonnés pour le système d'exécution du programme; puisque la syntaxe ne pourra fournir directement au programmeur cette information, il est nécessaire qu'elle lui soit

fournie indirectement par un système d'analyse du programme.

Ce point est très important, car il explicite très exactement la différence de conception entre le HPL et un langage "classique". L'inventeur d'un langage classique définit la syntaxe et la sémantique de celui-ci; ensuite, l'auteur d'une "implémentation" du langage conçoit un "compilateur" qui transforme toute séquence de symboles syntaxiquement correcte en une séquence d'instructions-machine qui respecte la sémantique associée au texte du programme. Dans la conception du HPL, nous tentons d'analyser les relations qui peuvent exister entre un programmeur et le programme qu'il fait passer de l'état "d'idée" à celui de "chose concrète". Or, il apparaît fréquemment que la syntaxe d'un langage de programmation ne peut mettre en évidence avec la même intensité des caractéristiques importantes du programme.

A notre avis, c'est une erreur de vouloir tout exprimer par l'intermédiaire de la seule syntaxe; il nous semble que la syntaxe doit être le résultat d'un compromis : la syntaxe mettra en évidence un aspect du programme, mais permettra également qu'un système automatique puisse extraire aisément les informations nécessaires pour l'analyse des autres aspects du programme. Donc, outre la syntaxe et la sémantique du langage, nous définissons les opérations que le système de traitement doit effectuer pour transmettre ces informations au programmeur.

#### 5.2. Réalisation effective d'un effet-net.

On fera une distinction entre "réalisation d'un effet-net" et "réalisation effective d'un effet-net" afin de supprimer toute ambiguîté dans les descriptions du processus d'exécution d'un holon.

Supposons un holon  $H_i$  et l'ensemble  $S_R(H_i)$  de ses subordonnés directs,

$$S_B(H_i) = \{H_j : (H_i + H_j)\}.$$

On dira qu'il y a "réalisation de l'effet-net de  $H_i$ " dès que l'exécution de ce holon est entamée, ceci a lieu dès que le holon  $H_i$  est "enclenché" par un de ses supraordonnés Dès qu'il est enclenché, le holon  $H_i$  enclenche à son tour, selon un ordre que nous examinerons ultérieurement, ses subordonnés directs.

On dira qu'il y a "réalisation effective de l'effet-net du holon  $H_i$ ", dès que l'exécution de ce holon est terminée. Si on appelle  $t_o$  l'instant d' enclenchement de  $H_i$  par un se ses supraordonnés, il existe un instant  $t_f$ , postérieur à  $t_o$ , à partir duquel on n'observera plus aucun enclenchement d'un des subordonnés directs ou indirects de  $H_i$ ; à  $t_f$  succèdera un instant  $t_f$  à partir duquel on n'observera plus aucune opération dont l'exécution résulterait de l'enclenchement de  $H_i$ ,  $t_f$  est l'instant où l'exécution de  $H_i$  est terminée. En fait,  $t_f$  correspond au dernier enclenchement d'un holon terminal appartenant à l'ensemble des subordonnés directs ou indirects de  $H_i$ , et  $t_f$  est l'instant à partir duquel on constate qu'il n'y a plus aucun holon terminal, subordonné direct ou indirect de  $H_i$ , qui soit "actif".

En d'autres termes : un holon  $H_i$  a effectivement réalisé son effet-net, dès que tous les holons  $H_j$ , subordonnés directs de  $H_i$  et enclenchés par lui, auront réalisé effectivement leurs effets-nets. (Dans la suite, il arrivera qu'on utilise le verbe "produire" comme synonyme de "réaliser", mais l'interprétation dépendra toujours de la présence ou de l'absence de l'adverbe "effectivement").

On voit qu'un holon dont l'exécution ne se terminerait pas, ne produira effectivement jamais son effet-net.

#### 5.3. Enclenchement séquentiel.

Nous avons décidé que les enclenchements des subordonnés directs d'un holon se feront selon un ordre d'enclenchement séquentiel.

Pour cela, un nouveau type de relation est établi entre les subordonnés directs d'un holon : la relation "succède d".

Soit l'ensemble  $S_{R}(H_{1})$  que nous supposons contenir noccurrences de holons  $H_{1}$ . (Le terme occurrence est utilisé car le même holon peut être représenté plusieurs fois). Entre ces noccurrences, il existe (n-1) relations "succède à" qui établissent un ordre total entre les éléments de  $S_{R}(H_{1})$ .

En convenant de représenter la relation "H; (m+1) succède à H; (m)"

par

$$H_{j(m)} \rightarrow H_{j(m+1)}$$

on peut représenter les (n-1) relations comme suit :

1) 
$$H_{j(0)} \rightarrow H_{j(1)}$$

2) 
$$H_{j(1)} + H_{j(2)}$$

$$n-1)$$
  $H_{j(n-1)} \rightarrow H_{j(n)}$ 

que nous convenons de représenter comme suit :

$$H_{j(0)} \rightarrow H_{j(1)} \rightarrow H_{j(2)} \rightarrow \cdots \rightarrow H_{j(n-2)} \rightarrow H_{j(n-1)} \rightarrow H_{j(n)}$$

Deux occurrences de holons  $H_{j(k)}$  et  $H_{j(k+1)}$  vérifient la relation

$$H_{j(k)} \rightarrow H_{j(k+1)}$$

si et seulement si la réalisation de l'effet-net de  $H_{j(k+1)}$  ne commence qu'après la réalisation effective de l'effet-net de  $H_{j(k)}$  et avant la réalisation de l'effet-net de tout autre holon  $H_{j(k+1)}$  avec  $i \ge 2$ .

Nous décidons ensuite que la syntaxe d'un corps de holon doit représenter explicitement les relations "succède à" qui existent parmi ses éléments.

Le holon H; est représenté par :

les relations "succède à" existent entre les subordonnés directs de  $H_i$  dont les noms forment la séquence  $\Sigma$  nh; nous pouvons décider que la relation  $H_j(k) \stackrel{+}{\rightarrow} H_j(k+1)$  peut être appliquée aux noms des holons  $H_j(k)$  et  $H_i(k+1)$ , soit

$$nh_{j(k)} \rightarrow nh_{j(k+1)}$$
.

Ensuite, nous choisissons de représenter  $\Sigma$  nh par :

étant entendu que :

$$^{\mathrm{nh}}(k)$$
;  $^{\mathrm{nh}}(k+1)$ 

signifie que les holons  $H_{j(k)}$  et  $H_{j(k+1)}$  tels que

$$^{nh}_{j(k)} = ^{nh}_{(k)}$$
 et  $^{nh}_{j(k+1)} = ^{nh}_{(k+1)}$ 

vérifient la relation

$$H_{j(k)} \rightarrow H_{j(k+1)}$$

Finalement, on décide de délimiter le corps d'un holon par begin et end, et nous obtenons :

$$bd_i = \underline{begin} (\Sigma dc), nh_{(0)}; nh_{(1)}; \dots; nh_{(n-1)}; nh_{(n)} \underline{end}$$

#### 5.4. Enclenchement en parallèle et non-déterminisme.

La décision de choisir un enclenchement séquentiel pour les subordonnés directs d'un holon est, en fait, arbitraire. On pourrait tout aussi bien envisager un enclenchement "en parallèle", c'est-àdire un enclenchement tel que tous les subordonnés H<sub>j</sub> de H<sub>i</sub> entament la réalisation de leurs effetsnets sans tenir compte de la réalisation effective des effets-nets. Ou encore, admettre qu'au cours de la réalisation de son effet-net, un holon agisse en fonction de l'état d'avancement des réalisations des effets-nets des autres holons enclenchés en même temps que lui. Ces formes d'enclenchement correspondent à la programmation de processus simultanés ou coopératifs.

D'autre part, des travaux récents [Dij74] ont attiré l'attention sur une forme de programmation non-déterministe, c'est-à-dire que lors de l'exécution d'un programme une opération parmi d'autres est choisie d'une manière aléatoire. Cette manière simplifie la programmation de certains algorithmes où la spécification de l'ordre des opérations est une opération superflue. On pourrait également imaginer qu'un des subordonnés directs d'un holon est choisi pour être enclenché, ce qui permettrait d'appliquer la méthodologie non-déterministe de programmation.

Toutefois, cette étude se limite à l'enclenchement séquentiel sans rejeter la possibilité qu'une extension du langage tienne compte des points de vue coopératif ou non-déterministe.

#### 6. LES INSTRUCTIONS-HOLON

Un holon produit effectivement un effet-net qui résulte d'une succession de productions effectives d'effet -net par ses subordonnés directs. Ce procédé est trop rigide pour nos besoins de programmation, il faut que nous puissions sélectionner un holon parmi d'autres ou réitérer la production de l'effet-net d'un holon.

La structure du corps de holon, exposée dans la section précédente, est conservée, mais nous étendons le domaine d'application des relations "enclenche" et "succède à"; celles-ci s'appliqueront désormais à des instructions-holon et non plus uniquement à des holons.

Une instruction-holon est :

- soit un holon (ce qui correspond au cas envisagé, lors de l'étude de la structure du corps de holon);
- 2) soit une association d'un ensemble de holons à un ensemble de "holons booléens"; les différentes formes d'associations permettent de construire des instructions de sélection ou d'itération.

Un holon booléen possède une valeur; celle-ci est de type booléen (vrai- faux) ou est indéterminée. Avant l'enclenchement d'un holon booléen, sa valeur est indéterminée; la production d'une valeur booléenne fait partie de la réalisation effective de son effet-net. L'effet-net effectif d'un holon booléen est soit la production d'une valeur booléenne, soit cette production et la production effective d'un effet-net comme un simple holon. La valeur booléenne produite est utilisée à l'exécution par le système d'interprétation des instructions-holon afin de décider ou non de l'inhibition de l'enclenchement de certains holons.

#### 6.1. Méthode de description des instructions-holon.

Les quatre formes d'association choisies pour les instructions-holon, dont la description va suivre, correspondent extérieurement à des instructions équivalentes des langages de type-Algol; toutefois, nous décrirons leur sémantique en termes de relations d'enclenchement et de succession. Cette forme de description est essentielle, car l'idée fondamentale du HPL est la composition de programmes par association hiérarchisée de "machines", de "holons". L'exécution d'un programme est, en fait, une succession d'enclenchements de "machines" qui produisent un effet-net et rien de plus.

#### 6.2. Instructions-holon de sélection.

#### a) IF-THEN-ELSE.

L'instruction-holon if-then-else associe un holon booléen BH à deux ensembles de holons ordonnés par la relation "succède à".

L'enclenchement de l'instruction-holon implique les opérations :

- 1) le holon booléen BH est enclenché et il produit effectivement son effet-net;
- 2) l'un des deux ensembles de holons est enclenché, le choix étant basé sur la valeur booléenne produite par BH; la production effective de l'effet-net de l'instruction-holon coîncide avec la production effective de l'effet-net de l'ensemble enclenché.

Cette instruction-holon est schématisée par :

$$if$$
-then-else[BH,{H<sub>0</sub>,H<sub>1</sub>,...,H<sub>n</sub>}<sub>T</sub>,{H<sub>0</sub>,H<sub>1</sub>,...,H<sub>m</sub>}<sub>F</sub>],

où l'indice T (F) identifie l'ensemble de holons enclenché si la valeur de BH est "vrai" ("faux").

L'ensemble T, contrairement à l'ensemble F, ne peut jamais être vide.

La syntaxe concrète (1) choisie pour cette instruction-holon est :

<if-then-else>::=

if<nom de holon booléen>then{<nom de holon>;}...
[else{<nom de holon>;}...]fi

On remarquera que dans la syntaxe concrète, un holon est désigné par son nom uniquement; la correspondance entre les ensembles T et F et les séquences de noms de holons suit la règle classique de l'Algol.

Remarque: En pratique, une petite variation syntaxique est admise: les points-virgules qui précèdent le "else" et le "fi" peuvent être omis. Cette possibilité est introduite afin de résoudre le problème de la suppression ou du maintien obligatoire du point-virgule devant un délimiteur; cette question est résolue de diverses manières selon les langages, il en résulte de stupides et frustrantes erreurs syntaxiques.

b) CASE.

L'instruction-holon case associe n holons booléens ( $BH_1$ ,  $BH_2$ ,..., $BH_n$ ) à (n+1) ensembles de holons ordonnés par la relation "succède à"; les n holons booléens sont également ordonnés.

L'enclenchement de l'instruction-holon implique les opérations suivantes :

1) les holons booléens sont enclenchés l'un après l'autre, le (i+1)ème étant enclenché dès que le "i"-ème a produit effectivement son effet-net et si sa valeur booléenne produite est "faux"; si celle-ci est "vrai", le "i"-ème ensemble de holons est enclenché; si aucun holon booléen ne produit une valeur "vrai", le (n+1)-ème ensemble de holons est enclenché; la production effective de l'effet-net de l'instruction-holon coîncide avec la production effective de l'effet-net de l'ensemble enclenché.

Cette instruction-holon est schématisée par :

les indices des holons booléens et des ensembles de holons indiquent les possibilités de succession d'enclenchements :

 $BH_i \rightarrow \{...\}_i$  si valeur de  $BH_i \equiv "vrai"$ ,

еt

 $BH_{i} \rightarrow BH_{(i+1)}$  si valeur de  $BH_{i} \equiv "faux"$ ,

sauf

$$BH_n \rightarrow \{...\}_{n+1}$$
 si valeur de  $BH_n \equiv "faux"$ .

La syntaxe choisie pour l'instruction "case" est :

<case>::=

case<nom de holon booléen>then<séquence de noms de holon>
[{eor<nom de holon booléen>then<séquence de noms de holon>}...]
else<séquence de noms de holon>esac

où <séquence de noms de holon>::={<nom de holon>;}...

Les symboles réservés <u>case</u> et <u>esac</u> servent de délimiteurs à gauche et à droite de l'instructionholon; entre ces délimiteurs, chaque holon booléen est associé par le symbole réservé <u>then</u> à son ensemble de holons, le <u>then</u> exprime exactement le fait que l'enclenchement de l'ensemble de holons

<sup>(1)</sup> Cette syntaxe est décrite en BNF, nous rappelons que les mots soulignés sont des symboles réservés du langage, que la notation {} ... signifie que la suite d'éléments contenue entre les deux accolades doit apparaître une ou plusieurs fois, que la notation [] permet l'omission des éléments entre crochets.

est subordonné à la production d'une valeur "vrai"; chaque couple "holon booléen-ensemble de holons" est séparé par le symbole réservé <u>cor</u>, abréviation de "exclusive or", qui rappelle qu' au maximum, un seul des holons booléens produira une valeur "vrai"; <u>else</u> est utilisé pour délimiter à gauche l'ultime ensemble de holons puisque cet ensemble sera enclenché si le dernier holon booléen produit une valeur "faux".

Remarque : Comme dans l'instruction "if-then-else", le point-virgule suivi par un symbole réservé peut être omis.

Des considérations de méthodologie de programmation justifient l'obligation pour le (n+1)-ème ensemble de ne pas être vide. En effet, l'emploi d'une instruction-holon "case" s'impose si, dans un algorithme, on est arrivé à un stade tel qu'une (ou plusieurs) condition parmi un ensemble de (n+1) conditions est nécessairement vraie. Ces conditions sont des prédicats que les objets de l'environnement de l'instruction-holon peuvent ou non valider.

A ce stade de la conception d'un algorithme, deux cas peuvent se présenter :

1) le programmeur est à même d'établir une liste de "n" prédicats qui constitue une liste exhaustive de toutes les conditions "indépendantes" susceptibles d'être vérifiées. Par condition "indépendante", on entend une condition qui exprime une propriété "positive" des objets de l'environnement et non la négation d'une autre condition; par exemple, supposons que notre environnement est constitué par une variable "p" qui contient la valeur d'une lettre de l'alphabet, on peut établir une liste exhaustive des vingt-six prédicats que "p" peut valider : p = "a", p = "b",..., p = "y", p = "z"; chacun de ces prédicats exprime une caractéristique que "p" peut "objectivement" ("positivement") posséder; par contre, si nous remplaçons p = "z" par :

 $\underline{\text{non}}(p \equiv \text{"a"})\underline{\text{et}} \ \underline{\text{non}}(p \equiv \text{"b"})\underline{\text{et}} \dots \underline{\text{et}} \ \underline{\text{non}}(p \equiv \text{"y"}),$ 

on obtient un prédicat équivalent parce que "l'univers" des valeurs possibles de "p" est nettement délimité; mais, en pratique, il n'en est pas toujours ainsi.

2) le programmeur peut établir une liste de "n" prédicats qui constitue une série d'événements éventuels; à cette série, il faut adjoindre un dernier événement, conjonction de la négation de chacun des événements éventuels, ce dernier événement ne pouvant recevoir une interprétation "positive" immédiate; l'instruction classique "if-then-else" correspond à la version la plus simple de ce deuxième cas.

Ces deux cas diffèrent par le degré de connaissance que le programmeur possède des événements possibles, différence de degré de connaissance marquée par la manière d'affirmer qu'à ce stade de l'algorithme deux événements peuvent se produire :

- 1) soit A, soit B : connaissance objective des événements,
- 2) soit A, soit non A : événements dépendants.

Cette différence persiste même si, après examen, on conclut que  $\underline{\text{non}}$  A  $\equiv$  B, justement parce qu'il faut un examen de non A pour arriver à cette conclusion.

Supposons trois événements possibles :  $p \in A$ ,  $p \in B$ ,  $p \in C$ ; imaginons pour fixer les idées que "p" soit un caractère et que A,B,C soient respectivement l'ensemble des chiffres, des opérateurs et de l'espacement, et qu'en outre chaque réalisation d'un événement implique une opération. Les trois programmes qui suivent représentent diverses manières de programmer le même algorithme.

#### Programme I

esac

#### Programme II

else rapporter erreur et agir en fonction

esac

#### Programme III

esac

Le programme I a le défaut de ne pas documenter correctement l'algorithme; en effet, on peut conclure de la forme donnée à l'instruction que "action pour C" est enclenché si  $\underline{\text{non}}(p \in A)\underline{\text{et}} \underline{\text{non}}(p \in B)$ ; l'équivalence de cette condition à p  $\epsilon$  C n'est pas évidente, surtout pas pour l'exemple choisi. En fait, le programmeur devrait faire accompagner ce programme d'une justification, d'une "preuve", de l'équivalence, en ce point du programme, entre

$$p \in C$$
 et  $\underline{non}(p \in A)\underline{et} \underline{non}(p \in B)$ .

Les programmes II et III n'ont pas ce défaut, car les conditions sont complètement explicitées et une action est envisagée s'il advenait que

$$\underline{\text{non}}(p \in A)\underline{\text{et}} \underline{\text{non}}(p \in B)\underline{\text{et}} \underline{\text{non}}(p \in C)$$

est "vrai". Les programmes II et III diffèrent par la manière de traiter l'erreur.

L'inclusion, dans la syntaxe d'un langage d'instructions pour le traitement des erreurs est un problème qui n'a pas reçu l'attention qu'il mérite. En effet, dans de nombreux cas d'application, il est impossible de justifier un programme comme le programme I par une méthode de preuve "formelle" sans introduire des hypothèses invérifiables : dans le cas le plus favorable, le programme I est précédé par des opérations qui attribuent à "p" une valeur qui appartient certainement à l'un des ensembles A,B,C, le programme I est correct s'il n'y a aucune altération de "p" entre les deux opérations; mais à cause de cette condition, le programme n'est sans doute pas "fiable". Si les deux opérations communiquent uniquement par des transferts en mémoire centrale, on suppose généralement qu'aucune altération n'est à envisager; mais dans la majorité des applications, des transferts plus complexes sont chose courante, impliquant l'intervention de mémoires externes (disques, bandes), et supposer l'impossibilité d'une altération est plus que téméraire [Gi174].

Cette discussion met en évidence la différence entre un programme correct et un programme "fiable"; en pratique, l'exactitude du programme, déjà si difficile à réaliser, ne suffit pas.

Remarque : L'instruction case of a été inventée par C.A.R. Hoare, sous la forme :

case i of 
$$\{Q_1; Q_2; Q_3; \dots; Q_n\};$$

où i est une expression de type entier dont la valeur permet de sélectionner un des  $\mathbf{Q}_i$  pour l'exécution.

En Pascal, Wirth a introduit une généralisation de cette instruction:

où i est une expression, si cette expression a la valeur  $L_k$ , l'instruction  $S_k$  est choisie et exécutée,  $L_i \not= L_i$  pour i  $\not= j$ .

Le <u>case</u> que nous proposons est plus général que ces deux formes et beaucoup plus apparenté à la fonction COND du LISP qu'à une forme <u>case</u> of.

#### 6.3. Instructions-holon d'itération.

Le principe des deux instructions-holon d'itération est d'associer un ensemble de holons ordonnés par la relation "succède à" à un holon booléen; le holon booléen et l'ensemble de holons produisent effectivement leurs effets-nets à tour de rôle, et ce, jusqu'à ce qu'une valeur booléenne "vrai" ou "faux", selon que l'instruction-holon est "repeat-until" ou "do-while", soit produite par le holon booléen; les deux instructions-holons diffèrent également par la relation de succession entre holon booléen et ensemble de holons : dans le cas du "repeat-until", le holon booléen succède à l'ensemble de holons, dans le cas du "do-while", c'est l'inverse; la production ef-

fective de l'effet-net d'une instruction-holon d'itération coîncide avec la production effective de la valeur booléenne qui met fin à tout nouvel enclenchement de l'ensemble de holons.

a) DO-WHILE.

Cette instruction-holon est schématisée par :

où l'ensemble de holons ne peut être vide.

La syntaxe choisie est :

<do-while>::=

while < nom de holon booléen > do { < nom de holon > ; } ... od

b) REPEAT-UNTIL.

Cette instruction-holon est schématisée par :

repeat-until[
$$\{H_0, H_1, H_2, \dots, H_n\}$$
,BH],

où l'ensemble des holons ne peut être vide.

La syntaxe choisie est :

<repeat-until>::=

repeat(<nom de holon>;)...until<nom de holon booléen>;

Remarque : Le point-virgule suivi du symbole réservé "od" ou "until" peut être omis.

#### 6.4. Justification du choix des instructions-holon.

Les instructions-holon choisies et la structure imposée au corps de holon garantissent que la structure d'un programme-holon est équivalente à un "emboîtement" d'opérations, à une structure d'arbre. De cette manière, tout programme-holon représente réellement un algorithme par une division périodique de celui-ci en sous-algorithmes, ce qui est nécessaire pour appliquer la réciproque du théorème de König.

#### 7. SYNTAXE DU NOM DE HOLON

La syntaxe du nom de holon doit permettre d'exprimer toutes les formes possibles de définition d'un holon, c'est-à-dire de spécification d'un effet-net.

Un nom de holon est formé par une séquence d'un nombre fini de symboles quelconques, à l'exclusion des symboles réservés du langage; de plus, certains délimiteurs spéciaux utilisés dans la séquence permettent d'attribuer certaines propriétés aux symboles de la séquence qu'ils délimitent.

Les séquences suivantes sont des noms de holon syntaxiquement corrects :

inverser la matrice le philosophe pense le philosophe mange ses spaghetti X:=Y+Z composer une table des cinq cents nombres premiers.

Les délimiteurs spéciaux sont :

1) la paire de guillemets : " "; il est requis que la séquence des symboles qu'ils délimitent corresponde à :

[<blancs>]<identificateur de variable>[<blancs>]

Les guillemets servent à isoler dans le nom de holon l'occurrence d'un identificateur de variable; cet identificateur est interprété comme désignant effectivement l'objet portant ce nom dans l'envi-

ronnement et dont le type est fixé par la première occurrence du nom de holon dans le programme-holon ( $^{(1}\star)$ ) le symbole \$ ; la séquence des symboles qui le suit doit débuter par :

#### [<blancs>]<identificateur de variables><blancs>

Le symbole \$ indique dans le nom de holon un paramètre dont le type est fixé par la première ocurrence du nom de holon dans le programme-holon; lors des autres occurrences du même nom de holon, l'identificateur qui suit le \$ peut être différent de celui utilisé dans la première occurrence du nom de holon, mais il doit être de même type. (1)

Deux noms de holons sont  $\ell gaux$  si les symboles de leurs séquences réduites correspondent un à un; on forme la séquence réduite à partir d'un nom de holon comme suit :

- 1) supprimer les blancs qui débutent ou terminent le nom de holon,
- 2) remplacer toute séquence de un ou plusieurs blancs par un seul,
- 3) supprimer la première séquence de blancs (si elle existe) et l'identificateur de variable qui suivent tout symbole %,
- 4) supprimer les blancs entre guillemets;

l'ordre de ces quatre opérations est sans importance.

Les noms de holons qui suivent sont égaux :

```
inverser la matrice % quarante résultat dans "re" inverser la matrice % quarante résultat dans "re" inverser la matrice % complex résultat dans "re"
```

leur séquence réduite est identique à :

#### |inverser la matrice % résultat dans "re"|

La première occurrence d'un nom de holon est déterminée comme suit : Soit un programme-holon  $P[np,\{H_0,H_1,\ldots,H_n\}]$  qui contient un certain nombre d'occurrences du nom de holon "nhx" (il n'est pas nécessaire que ce holon soit déjà décrit, c'est-à-dire que le nom d'un des  $H_i$  soit égal à "nhx"); soit le holon  $H_i$  tel qu'il n'existe aucune occurrence de "nhx" dans les holons  $H_i$  avec  $0 \le i < j$ ; l'occurrence de "nhx" correspondant au premier enclenchement éventuel d'un holon de nom "nhx" dans  $H_i$  est la première occurrence du nom de holon "nhx" dans le programme-holon P, les autres occurrences de "nhx" dans P sont qualifiées d'occurrences ultérieures.

La première occurrence d'un nom de holon est extrêmement importante; en effet, à partir de celle-ci le système de traitement des programmes-holon peut déterminer :

- 1) la séquence réduite du nom de holon;
- 2) une liste ([ id) des identificateurs qui apparaissent dans le nom; il s'agit uniquement des identificateurs délimités par des guillemets;
- 3) une liste (Σ p) des paramètres, il s'agit des identificateurs précédés par le symbole %.

Ces deux listes permettent de vérifier la cohérence vis-à-vis de l'environnement des occurrences ultérieures d'un nom de holon; en effet, lors d'une occurence ultérieure, il est nécessaire que les objets identifiés par ([] id) soient toujours accessibles à l'endroit de l'occurrence et que tous les paramètres possèdent des types compatibles avec ceux des paramètres de la première occurrence.

La raison primordiale de cette distinction entre identificateurs d'un objet bien spécifié et paramètres lors de la définition de l'effet-net d'un holon est de faire indiquer, par la syntaxe, la dépendance d'une opération vis-à-vis de son environnement; si un nom de holon ne contient que des identificateurs entre guillemets, on peut en inférer a priori que la description de l'effet-net de ce holon tire probablement parti des caractéristiques des variables identifiées; par contre, la présence de paramètres attire l'attention sur la grande diversité d'environnements que ce holon peut rencontrer et sur les contraintes probables que cette diversité aura imposées à la description du

<sup>(1)</sup> Nous n'avons pas encore défini, à ce stade de l'exposé, le "type" d'une variable; cette notion est étudiée en détail dans le chapitre IV. En attendant, le lecteur peut sans inconvénient interpréter "variable" et "type" selon leurs significations habituelles en Algol ou en PL/I.

<sup>(1\*)</sup> Voir également VI. 3.4.2.

holon. En d'autres termes, le nom de holon avec identificateurs correspond à une opération "spécialisée", au contraire de son homologue à paramètres qui représente toute une "famille" d'opérations. Rien n'empêche un programmeur d'utiliser des noms de holon sans identificateur ni paramètre, mais en échange, il perd tout le bénéfice de la vérification du contexte des occurrences ultérieures.

#### Remarque :

- L'exposé précédent est incomplet, car ni le mode d'accès à un paramètre, ni le traitement des constantes ne sont abordés; ces questions sont résolues dans le chapitre IV.
- 2. Le nom du holon H ne peut être qu'un nom sans identificateur ni paramètre, puisqu'il est le premier d'un programme-holon, il ne peut exister d'objets auxquels il pourrait se référer.

  3. On verra au chapitre IV, qu'un identificateur peut désigner une partie d'un objet; dans ce cas, la partie indiquée peut être paramétrique; si "racine-équation" est un identificateur de tableau, le nom de holon qui suit est légal: imprimer "racine-équation[ZI]".

#### 8. STRUCTURE D'UN PROGRAMME-HOLON

#### 8.1. Conditions de production effective d'un effet-net.

Soit le programme-holon

où chaque H; est de forme

$$H_{i}[nh_{i}(\Sigma id, \Sigma p), bd_{i}(\Sigma dc, \Sigma I)],$$

avec

$$bd_i = \underline{begin} (\Sigma dc) I_0; I_1; I_2; ...; I_n \underline{end},$$

le symbole I signifiant instruction-holon; et le point-virgule, une relation de succession.

Un programme-holon P est exécutable s'il est complet; l'exécution d'un programme-holon P est synonyme de réalisation de son effet-net; celle-ci est provoquée par l'enclenchement du holon  $H_0$ , et la production effective de l'effet-net du programme-holon P coîncide avec la production effective de l'effet-net du holon H.

Un programme-holon P produit effectivement son effet-net si et seulement si chaque holon H, produit effectivement son effet-net pour tout enclenchement susceptible d'être effectué dans le contexte du programme-holon.

En effet, l'enclenchement de  ${\rm H}_{
m o}$  implique les enclenchements successifs des instructions-holon de Ho: Io, I1, ... Chacune de ces instructions-holon enclenchera à son tour un ensemble de holons et éventuellement un holon booléen; puisque le programme P est complet, chaque holon dont le nom apparaît dans une instruction-holon existe dans l'ensemble des holons formant le programme P, d'où ce holon peut être enclenché et peut réaliser son effet-net en enclenchant à son tour les holons dont le nom apparaît dans son corps de holon; si chaque instruction-holon produit effectivement son effet-net, chaque holon enclenché produira effectivement son effet-net et finalement, le holon Ho produira effectivement son effet-net.

D'autre part, puisque le programme P est complet, tous les holons qui le composent appartiennent à au moins une instruction-holon susceptible d'être enclenchée; l'enclenchement d'un holon lors d'une exécution de P dépend de la valeur prise par le holon booléen de l'instruction-holon à laquelle ce holon appartient.

Puisque le but primordial d'un programme est la production effective d'un effet-net, il est nécessaire de s'assurer que toutes les instructions-holon d'un programme P produisent effectivement leurs effets-nets si elles sont enclenchées.

Supposons provisoirement que tout holon, appartenant à une instruction-holon, produit effectivement son effet-net pour tout enclenchement de l'instruction-holon à laquelle il appartient. S'il s'agit d'une instruction-holon de sélection, celle-ci produira toujours effectivement son effet-net; en effet, la production effective de cet effet-net résulte de la production effective des effets-nets des holons qui composent l'instruction-holon. Au contraire, s'il s'agit d'une instruction-holon d'itération, la production effective de son effet-net dépend de la possibilité pour le holon booléen de produire effectivement une valeur booléenne déterminée par le type d'instruction-holon. Que cette valeur booléenne est effectivement produite pour tout enclenchement de l'instruction-holon doit être montré ("prouvé") par l'auteur du programme.

Il reste à établir les conditions pour que tout holon "x" appartenant à une instruction-holon produise effectivement son effet-net.

Premièrement, il est nécessaire que tout holon terminal subordonné direct ou indirect de "x" produise effectivement un effet-net; cette condition sera toujours vérifiée parce qu'elle sera utilisée comme condition initiale pour la définition des opérations des langages "terminaux" utilisés pour écrire les holons terminaux. (1)

Deuxièmement, il est nécessaire que les supraordonnés de ces holons terminaux appartenant au holon "x" produisent effectivement un effet-net. Pour cela, il est nécessaire que les supraordonnés directs des holons terminaux produisent effectivement un effet-net, ce qui implique que les instructions-holon de ces supraordonnés directs (instructions-holon qui ne contiennent que des holons terminaux) produisent effectivement leurs effets-nets; si cette condition est vérifiée, on peut assimiler les supraordonnés directs des holons "terminaux" à des holons "terminaux" d'un niveau supérieur aux premiers et montrer que les supraordonnés de ces nouveaux "terminaux" produisent effectivement leurs effets-nets; on remonte de cette manière dans les relations de subordination jusqu'à ce qu'on atteigne le holon "x".

#### 8.2. Cas du holon récursif.

Un holon est récursif s'il est membre de l'ensemble de ses subordonnés directs ou indirects.

Lorsqu'un holon récursif produit son effet-net, il advient qu'il enclenche un holon qui n'est autre que lui-même; pour que l'effet-net soit effectivement produit, il est nécessaire que ces auto-enclenchements soient en nombre fini. En d'autres termes, pour que le holon récursif "x" produise effectivement son effet-net, il est nécessaire qu'il existe au moins une séquence d'enclenchements des subordonnés de "x" qui ne contienne pas "x". Cette condition nécessaire est une condition sur la structure du programme, nous pouvons donc en confier la vérification au système de traitement d'un programme-holon. La condition suffisante est que, pour tout enclenchement de "x" lors de l'exécution du programme-holon P, une des séquences d'enclenchements des subordonnés de "x" qui ne contient pas "x" soit enclenchée; cette condition dépend principalement des divers environnements lors de l'enclenchement de "x", sa vérification est laissée au programmeur.

Les subordonnés directs et indirects d'un holon récursif "x" forment un ensemble S. Par examen de la structure du holon "x" et de ses subordonnés, on peut établir une liste de toutes les suites d'enclenchements éventuels des subordonnés de "x"; lors de l'établissement de cette liste, un holon "x" appartenant à S est assimilé à un holon terminal. Nous allons établir les conditions que doit vérifier la structure du holon "x" pour garantir l'existence d'au moins une séquence sans occurrence d'un "pseudo-terminal" "x".

Pour établir ces conditions, nous utiliserons la notion de niveau d'une instruction-holon par rapport à un holon qui la contient; un holon "x" possède conventionnellement le niveau "0", et les instructions-holon qui sont ses subordonnées directes le niveau "1"; les instructions-holon qui sont subordonnées directes d'un holon appartenant à l'ensemble des holons d'une instruction-holon de niveau "i" sont de niveau "i + 1"; les holons de l'ensemble de holons et les holons booléens qui forment une instruction-holon de niveau "i" sont dits être de même niveau "i" que l'instruction-holon.

Un holon  $H_{\chi}$  appartient de manière exhaustive à une instruction-holon de sélection si chaque ensemble de holons de cette instruction-holon contient directement ou indirectement le holon  $H_{\chi}$ . Par exemple, pour l'instruction-holon suivante :

 case
 un
 then
 x;y;z

 eor
 deux
 then
 x;y;w

 eor
 trois
 then
 y;w;h

 else
 y;x

esac

<sup>(1)</sup> Voir plus loin.

Le holon "y" est contenu de manière exhaustive.

Soit un holon récursif "x" dont les subordonnés directs et indirects forment S; de cet ensemble, nous extrayons le sous-ensemble R contenant toutes les occurrences de "x" dans S; à chaque élément de R, nous associons le niveau et le type de l'instruction-holon qui contient cette occurrence de "x" (si elle est contenue dans une instruction-holon).

Si une des conditions suivantes n'est pas vérifiée, les conditions nécessaires pour que le holon "x" produise effectivement un effet-net ne sont pas remplies :

- 1) tous les éléments de R sont contenus dans une instruction-holon de niveau i≥1, sinon il existe une occurrence de "x" qui appartiendra à toutes les suites d'enclenchements des subordonnés de "x", puisque ces suites peuvent différer uniquement par les ensembles de holons des instructionsholon dont on peut supposer le non-enclenchement.
- 2) si une occurrence de "x" appartient à une instruction-holon "repeat-until" de niveau "i" ou si des occurrences de "x" appartiennent de manière exhaustive à une instruction-holon de sélection de niveau "i", alors il faut que le niveau "i" de cette instruction-holon soit supérieur à "1" et que l'instruction-holon soit contenue dans une instruction-holon "do-while" ou de sélection qui soit de niveau j < i; de plus, si cette instruction-holon de niveau "j" est une instruction-holon de sélection, il faut qu'elle ne contienne pas de manière exhaustive des occurrences de "x". En effet, ces conditions garantissent qu'une occurrence de "x" dans un "repeat-until" ne conduira pas à une séquence infinie d'enclenchements due au fait que le holon booléen de cette instruction-holon succède à l'ensemble de holons; dans le cas de l'instruction-holon de sélection, la condition permet l'existence d'une "sortie" pour le processus de récursion.

#### Remarque :

L'instruction-holon  $\underline{if}$  BH  $\underline{then}\{H_0,H_1,\ldots,H_n\}$   $\underline{fi}$  associe au holon booléen BH  $\underline{deux}$  ensembles de holons dont l'un est vide; un holon peut donc  $\underline{jamais}$  appartenir de manière exhaustive à une instruction-holon de ce type.

#### Exemples :

1) <u>holon</u> x begin :

a;b;c; if h then e fi;x;

end

2) <u>holon</u> y begin

a;b;c; if h then w else y fi

end

3) <u>holon</u> m

begin

a;b;c; repeat m until v;

end

Les séquences d'enclenchements éventuels sont respectivement :

1) pour le holon "x"

{a,b,c,h,e,x}

ou {a,b,c,h,x}

2) pour le holon "y" :

{a,b,c,h,y}

{a,b,c,h,w}

3) pour le holon "m"

{a,b,c,m,v}

Les holons "x", "y" et "m" sont récursifs, mais seul le holon "y" remplit les conditions structurelles nécessaires pour produire effectivement un effet-net.

Le programmeur doit fournir une "preuve" que la condition suffisante de production effective d'un effet-net est vérifiée pour chaque holon récursif, c'est-à-dire que pour tous les environnements susceptibles d'exister lors d'un enclenchement d'un holon récursif "x", une des séquences d'enclenchements des subordonnés de "x", qui ne contient pas "x", est toujours susceptible d'être enclenchée. Les éléments qui servent de base à cette "preuve" sont la description des environnements possibles lors de l'enclenchement de "x" et les effets-nets de chaque holon subordonné de "x".

La description des environnements possibles est primordiale comme en témoigne l'exemple suivant :

Envisageons une procédure de parcours d'un arbre binaire représenté par trois tableaux L,A,R, (un noeud de l'arbre est représenté par L(i),A(i), R(i) où L(i) contient l'indice du sous-arbre à gauche, A(i) l'information du noeud et R(i) l'indice du sous-arbre à droite); en un langage de type-Algol, on peut écrire :

procedure treeprint(t); integer t; value t;
 if t \* 0
 then treeprint(L(t));
 print (A(t));
 treeprint(R(t));
 fi;

Il est généralement admis que cette procédure permet de visiter tous les noeuds d'un arbre binaire, elle correspond à la définition du parcours en ordre symétrique. Toutefois, cette procédure ne produit effectivement un effet-net que si elle est appliquée à un arbre fini.

La représentation de l'arbre binaire par trois tableaux de dimensions [1:n] n'implique pas nécessairement que l'arbre est fini, car on peut imaginer que parallèlement à la procédure "treeprint", une autre procédure construise un arbre binaire infini en réutilisant les mémoires des noeuds déjà traversés et "bloque" la procédure "treeprint" dans le parcours d'une branche infinie... L'exemple peut paraître académique, mais la conclusion ne l'est pas.

Cette discussion a montré qu'à partir de la syntaxe d'un programme-holon, on peut en déduire la structure et vérifier si les conditions nécessaires à la production effective d'un effet-net sont réalisées; le système de traitement peut également dresser la liste des holons booléens qui sont impliqués dans la génération d'une séquence d'enclenchements libre de tout appel récursif, et associer à chacun de ces holons booléens la valeur booléenne qu'il est nécessaire qu'il produise pour mettre fin à une expansion récursive; cette information est très utile pour l'établissement par le programmeur de la preuve de la vérification des conditions suffisantes.

#### Remarque :

La discussion précédente sur la structure d'un holon récursif ne présente d'intérêt que si l'on peut aisément détecter les holons récursifs d'un programme-holon; la méthode de détection sera abordée ultérieurement.

#### 9. STRUCTURE DU HOLON BOOLEEN

#### 9.1. Structure générale d'un holon booléen.

La discussion précédente sur la structure d'un programme-holon a fait ressortir le rôle important des holons booléens; la vérification des conditions de production effective d'un effet-net implique que le programmeur analyse comment la valeur booléenne est produite en fonction des divers environnements lors de l'enclenchement. Toutefois, si on considère un programme complet, tous les holons booléens ne doivent pas être étudiés avec la même attention, en effet :

1) si un holon "x" est non-récursif, la production effective de l'effet-net dépend des holons booléens appartenant aux instructions-holon d'itération; on peut écarter de l'analyse les holons booléens qui appartiennent aux instructions-holon de sélection; si le programme est logiquement bien construit, les deux valeurs booléennes sont, toutes les deux, susceptibles d'être produites, et même s'il n'en est pas ainsi, la production effective de l'instruction-holon de sélection n'est pas compromise;

2) si un holon "x" est récursif, la structure du programme indique les holons booléens dont l'analyse s'impose.

L'obligation pour un holon booléen de produire effectivement une valeur (en plus de l'éventuelle production effective d'un effet-net) implique une restriction des structures possibles pour le holon booléen.

Pour tout holon booléen "b", on peut établir une liste des séquences d'enclenchements éventuels des subordonnés de "b", soit T(b) cette liste.

Exemple :

"b" = b[b, begin a; if h then p fi; if z then q else v fi; end]

La liste T(b) contient :

(a,h,p,z,q), (a,h,p,z,v), (a,h,z,q), (a,h,z,v).

A partir des éléments de T(b), on peut dresser la liste L(b) des holons groupant les holons qui terminent les séquences d'enclenchements; dans l'exemple :  $L(b)\equiv (q,v)$ . Pour que le holon booléen "b" produise effectivement une valeur booléenne, il est nécessaire que les holons appartenant à L(b) produisent effectivement une valeur booléenne, c'est-à-dire que ces holons doivent être des holons booléens produisant effectivement une valeur booléenne. Pour chaque holon n appartenant à L(b), on peut à nouveau établir des listes T'(n) et L'(n); le processus peut être réitéré jusqu'à ce qu'on obtienne des listes L qui ne contiennent plus que des holons "terminaux"; si le programme-holon est complet, ces listes sont obtenues après un nombre fini d'étapes; l'examen syntaxique de ces holons "terminaux" permet de vérifier s'ils sont susceptibles ou non de produire effectivement une valeur booléenne, si oui, le holon "b" possède une structure de holon booléen.

La méthode de vérification de la condition peut sembler relativement complexe; mais, en pratique, la condition est utilisée pour vérifier a priori lors de la construction progressive de l'ensemble des holons qui forment le programme-holon, si certains holons sont booléens comme l'exige la structure du programme.

En effet, considérons un programme-holon P qui contient (n + 1) holons :

Si P est incomplet, il existe un certain nombre de "noms de holon" qui apparaissent dans les corps des holons  $H_i$  ( $0 \le i \le n$ ) et qui ne correspondent à aucun des noms de holons  $H_i$ . Nous pouvons donc représenter "l'état de développement" du programme-holon P par l'association à P d'une liste UH ("undescribed holon") contenant tous ces holons définis mais non-décrits :

(les indices n'impliquent aucune relation entre éléments de P et de UH : nh e UH n'est pas nh e H ).

Les éléments de UH appartiennent à des instructions-holon de P, l'examen syntaxique de ces instructions-holon permet de décider si  $\operatorname{nh}_i$   $\epsilon$  UH doit être un holon booléen ou non; le même  $\operatorname{nh}_i$   $\epsilon$  UH ne peut évidemment pas apparaître comme holon booléen et comme holon. Cette liste UH est utilisée pour contrôler toute adjonction d'un holon au programme-holon P. Pour simplifier les opérations, la liste UH est scindée en deux sous-listes : UH groupant les noms de holon et BUH groupent les noms de holon booléen.

Exemple :

Soit le programme P[np, {H\_}]

A ce stade, nous avons :

UH(k,j) et BUH(b).

Ajoutons à P, le holon H1:

H,[b, begin a; if h then p fi; if z then q else v fi; end]

Les listes UH et BUH deviennent :

UH(k,j,a,p) et BUH(h,z,q,v)

Les holons "q" et "v" sont booléens, car ils sont les derniers holons à être enclenchés lors d'une exécution du holon booléen "b". Ajoutons  $H_2$  et  $H_3$ :

H2[q, begin a;c;d;w end]

H3[v, begin c;a;m;d;x end]

Nouvelles valeurs de UH et BUH :

UH{k,j,a,p,c,d,m},

BUH{h,z,w,x}.

En pratique, il est utile d'associer également au programme P les listes DH et BDH des holons et des holons booléens déjà décrits :

DH{np} et BDH{b,q,v}.

#### 9.2. Restrictions pour les instructions-holon d'itération.

Supposons provisoirement que la structure d'un holon booléen soit :

- 1) begin ... ;a;repeat w until z;end
- ou 2) begin ... ;a; while z do w od; end

Si on établit les listes T des séquences d'enclenchements éventuels, on obtient pour la structure (1) des séquences qui se terminent toutes par :

(...,a,[w,z]\*),

οù

[w,z]\*

signifie un ou plusieurs enclenchements successifs du couple (w,z); pour la structure (2), les éléments de la liste T se terminent soit par  $(\ldots,a,[z,w]^*,z)$ , soit par  $(\ldots,a,z)$ .

Le holon "z" est bien entendu booléen et doit produire une valeur booléenne propre à l'instructionholon d'itération; on ne peut concevoir qu'il produise en outre la valeur booléenne du holon booléen
qui est son supraordonné puisque cela signifierait que cette valeur est toujours "vrai" ou "faux" suivant qu'on a la structure (1) ou (2). Confier la production de cette valeur booléenne au holon "w"

n'est pas une solution : en effet, dans le cas de l'instruction-holon "repeat-until", "w" est toujours
enclenché au moins une fois, et la valeur booléenne effectivement produite lors de son ultime enclenchement pourrait être attribuée au supraordonné mais dans le cas de l'instruction-holon "do-while",
il peut advenir que "w" ne soit jamais enclenché, ce qui laisserait le holon booléen sans valeur booléenne. Proposer qu'en l'absence d'un enclenchement de "w", ce soit "a" qui produise la valeur booléenne, ne simplifie pas la structure du holon, car si, à la place de "a", nous avions une nouvelle
instruction-holon de type "do-while", il serait nécessaire de déterminer un nouvel holon pour la production de la valeur booléenne!

En conclusion, il a été décidé que la dernière instruction-holon d'un holon booléen ne peut être une instruction-holon d'itération.

#### Remarque :

Même s'il semble qu'on puisse utiliser sans problème le "repeat until", il nous a paru préférable de ne point admettre cette instruction-holon comme ultime instruction-holon d'un holon boo-léen, car une instruction-holon "repeat-until" peut être interprétée sous la forme d'un "do-while".

#### 9.3. Holon booléen récursif.

Un holon booléen peut être récursif; toutefois, les conditions de structure diffèrent quelque peu des conditions imposées aux holons récursifs, car l'occurrence du nom de holon qui entraîne la récursivité du holon n'est pas limitée aux ensembles de holons, mais elle peut apparaître comme holon booléen d'une instruction-holon.

#### Exemple:

Soit dans le corps d'un holon, l'instruction-holon :

if %n est impair positif then ... else ... fi;

dont le holon booléen est décrit d'une manière récursive :

[% n est impair positif. begin case %n ≤ o then resultis false eor %n = 1 then resultis true else % (n-2) est impair positif esac end]

(Dans ce holon booléen, resultis est un holon terminal utilisé conventionnellement pour produire une valeur booléenne).

Supposons qu'on donne une autre forme à la partie "else" de l'instruction-holon "case" de ce holon-booléen :

... else autre chose esac.

[autre chose, begin if %(n-2) est impair positif then resultis true else resultis false fi end]

Dans les deux cas, l'occurrence de "%(n-2) est impair positif" dans la partie "else" entraîne la récursivité du holon; mais, dans le premier cas, le holon est booléen et appartient à un ensemble de holons d'une instruction-holon "case", dans le deuxième cas, il est booléen et appartient à la partie booléenne d'une instruction de sélection.

Mis à part cette différence, les conditions de structure sont identiques à celles d'un holon récursif; ainsi, par exemple, si un holon booléen récursif appartient à la partie booléenne d'une instruction-holon de sélection, il est nécessaire que cette instruction-holon de sélection appartienne elle-même à une partie à sélectionner d'une instruction de sélection d'un niveau inférieur (ou à une instruction-holon "do-while") en respectant la condition d'appartenance non-exhaustive; cette condition est vérifiée par "autre chose" dans l'exemple précédent, elle serait violée si la description de "%n est impair positif" était (erronément) :

[% n est impair positif",

begin case  $n \le 1$  then %n est impair positif else autre chose esac

end]

où la description de "autre chose" n'est point modifiée.

#### 9.4. Justification de la structure du holon booléen.

L'exécution d'un holon booléen se présentera comme l'enclenchement d'une série de holons auquel

succède l'enclenchement d'un holon booléen; le caractère récursif de cette définition de l'exécution d'un holon booléen est résolu par l'existence de holons booléens qui sont des expressions booléennes exprimées dans un langage "terminal".

Le "préfixage" de l'expression booléenne par une série d'opérations est justifié par des considérations méthodologiques sur les instructions de répétition.

Knuth a proposé [Knu74] une forme généralisée des instructions de répétition :

A : S;
 if B then goto Z fi;
 T ; goto A;
Z:

- si S est vide, on retrouve l'instruction "do-while"; si T est vide, l'instruction "repeat-until".

  Comme exemples typiques d'application de cette forme généralisée, on a :
- 1) S représente les instructions pour acquérir ou générer une nouvelle "information", B est un test de fin de "données", T contient le traitement de l'information acquise;
- 2) le code qui précède l'étiquette A accomplit l'initialisation de la boucle qui lui succède, S calcule des éléments utilisés par le test B de convergence, T accomplit la mise à jour des variables pour l'itération suivante.

Des programmes de ce genre sont extrêmement fréquents, (Dijkstra les appelle les boucles "n + 1/2").

Dans les langages sans goto, on retrouve ce schéma de programme sous l'une des trois formes suivantes :

a) avec recopie du code S:

#### S; while non B do T; S od;

b) avec introduction d'un code  $T^{-1}$ , inverse du code T, c'est-à-dire que l'effet-net de la succession " $T^{-1}$ ; T" est équivalent à l'effet-net d'une instruction "vide" :

c) avec dédoublement du test B :

#### repeat S; if non B then T fi; until B;

Ces trois versions sont d'autant moins satisfaisantes que les codes S et T sont longs; pour résoudre ce problème, certains langages permettent d'utiliser des instructions "exit", par exemple :

#### repeat begin S; when B exit; T; end.

Il existe une quatrième possibilité, "à la holon booléen" :

d.1.) while(S; non B) do T od;

d.2.) repeat S until(if B then true else(T; false)fi);

Cette manière de procéder évite toute recopie de S ou de T et toute introduction de  $T^{-1}$ ; de plus, si on se réfère aux exemples typiques d'application, la structure de la partie booléenne est clairement justifiée :

Supposons que la spécification d'un programme soit d'effectuer un traitement T sur tous les enregistrements d'un fichier; cette application est de type "1", on peut lui faire correspondre un programme de structure (a) :

programme(a)
R:=read(file);
while R ≠ eof
do T;R:=read(file)od;

ou un programme de structure (d.1) :

programme(d.1)
 while(R:=read(file); R ≠ eof)
 do T od;

Le fonctionnement du programme (d.1) est décrit par :

"tant que l'élément suivant du fichier existe, le traiter"; en comprenant que l'élément suivant d'un fichier qu'on vient "d'ouvrir" est le premier élément de ce fichier ou "eof" s'il n'y en a pas. La structure de (d.1) suit exactement la description du fonctionnement. Dans le cas du programme (a), ce n'est pas aussi simple : la fonction du premier "r:=read(file);" ne s'explique que par l'examen de l'instruction qui suit, et il est nécessaire de combiner le deuxième "r:=read(file);" avec le test "r ≠ eof" pour comprendre le principe de l'itération...

La version avec goto de ce programme est :

A:r:=read(file);

if r=eof then goto Z fi;
T;goto A;
Z:

#### Remarques:

1. Les formes (d.1) et (d.2) peuvent être programmées en Algol ou en PL/I en utilisant l'effet-debord d'une fonction booléenne.

2. Le code compilé pour un programme de forme (d.2) peut facilement transformer le <u>"true"</u> en un "goto z" et le <u>"false"</u> en un <u>"goto</u> A", et faire "disparaître" le "until" produisant un code aussi efficace que celui de la forme généralisée.

3. La syntaxe des instructions-holon exposée dans la section (6) doit être complétée pour inclure les formes que peuvent prendre les instructions-holon lorsqu'elles appartiennent à un holon booléen.

#### 10. PROGRAMMES ET ARBRES FINIS

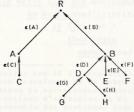
## 10.1. Rappel.

Un arbre orienté est un ensemble de noeuds et un ensemble d'arcs, chaque arc conduit d'un noeud V à un noeud V'. Si "e" est un arc de V à V', on dit que V est le noeud initial de "e", et que V' est le noeud final, on note :

En outre, il existe un noeud R tel que :

- a) chaque noeud V = R est le noeud initial d'un seul arc, noté e(V);
- b) R n'est le noeud initial d'aucun arc;
- c) R est une racine, c'est-à-dire que pour chaque noeud V = R, il y a un chemin orienté menant de V à R.

(si  $e_1, e_2, \dots, e_n$  sont des arcs (avec  $n \ge 1$ ), on dit que  $(e_1, e_2, \dots, e_n)$  forment un chemin orienté de longueur n allant de V à V', si  $V=init(e_1)$ ,  $V'=fin(e_n)$ , et  $fin(e_k)=init(e_{k+1})$  pour  $1 \le k < n$ ).



un arbre orienté

Un sous-arbre orienté est l'arbre orienté obtenu en prenant un noeud X quelconque d'un arbre orienté et en considérant qu'il possède les propriétés d'une racine, c'est-à-dire qu'on forme un sous-ensemble de l'arbre orienté qui contient tous les noeuds et tous les arcs qui forment des chemins tels que X est le noeud final de ces chemins.

On notera arbre orienté et sous-arbre orienté en indiquant simplement leur racine; par exemple, dans la figure :

arbre orienté:R; sous-arbres orientés:A,B,C,R,D,E,F,G,H.

Un arbre ordonné est un arbre orienté dont les sous-arbres sont classés selon une relation d'ordre. (Conventionnellement, on fait apparaître cette relation d'ordre dans la représentation graphique de l'arbre; si le sous-arbre A est d'un ordre plus élevé que le sous-arbre B, on dessine ce dernier à la droite de A). La relation d'ordre est entre noeuds initiaux d'un ensemble d'arcs dont le noeud final est identique.

Dans [Knu68, pages 362-383], on trouvera un exposé plus détaillé sur les propriétés des arbres orientés.

### 10.2. Arbres orientés et instructions de contrôle.

La contrapositive du théorème de König exprime une condition suffisante pour que l'exécution d'un algorithme se termine; cette contrapositive peut être reformulée comme suit : un algorithme dont toute exécution peut être représentée par un arbre orienté *fini* se termine toujours.

Si nous limitons la syntaxe d'un langage à la description de successions d'opérations primitives et d'instructions de sélection, les programmes écrits dans ce langage auraient une structure d'arbre orienté fini (si le texte du programme est fini). Toutefois, ce langage ne permettrait que la résolution d'une classe relativement restreinte de problèmes, puisque tout processus itératif est inexprimable dans ce langage.

Si on ajoute à ce langage la possibilité syntaxique d'exprimer des itérations, on peut écrire dans ce langage tout programme qui correspond à la solution d'un problème exprimée par un "flow-chart" <sup>(1)</sup>. Mais, l'introduction d'une expression syntaxique de l'itération (soit sous forme de récursion ou d'une instruction de répétition) a pour conséquence l'impossibilité de décider du caractère fini de toute exécution du programme, sur base de sa représentation syntaxique (sur base du "texte"). Il en résulte qu'un texte syntaxiquement correct ne représente plus nécessairement un "programme" mais bien une "classe de calculs" (en traduisant par "calcul", le terme anglais "computation", étant donné qu'on appellera "méthode de calcul" - computational method - un algorithme susceptible de ne point se terminer).

Le rôle du programmeur est de construire des "programmes" et non des "classes de calculs"; l'avantage de l'utilisation d'un langage à syntaxe "structurée" par rapport à l'utilisation classique, où l'instruction de "branchement" est explicite, réside dans la possibilité de détecter, sur des bases purement syntaxiques, les points cruciaux du texte, écrit dans le langage de programmation, qui permettront de déterminer les conditions qui transformeront le texte en "programme" et non en "méthode de calcul".

# Exemple :

Considérons le programme suivant, publié par N. Wirth [Wir73, page 75], et destiné à recopier le contenu d'un fichier "input" dans un fichier "output" en éliminant les espacements redondants : (nous avons préfixé les expressions booléennes par les "étiquettes" ①, ②, ③ afin de les désigner dans la discussion; l'opération "read(c)" lit dans le fichier "input" et "write(c)" écrit dans le fichier "output" par définition).

## 1) version\_"structurée" :

<sup>(1)</sup> Ce résultat est dû à Jacopini[Jac66] qui a montré que tout programme, sous forme de "flow-chart" peut être transformé en un autre programme qui calcule le même résultat et qui est formé par une combinaison de successions, sélections ou itérations des instructions du programme original et d'instructions d'affection et de tests supplémentaires. L'intérêt de ce résultat est de montrer que l'ensemble des problèmes pour lesquels on peut exprimer une solution dans un langage limité à la succession, la sélection et l'itération, est identique à l'ensemble des problèmes qui admettent une solution sous forme de "flow-chart". Le théorème ne signifie pas que le programme écrit dans la syntaxe réduite sera nécessairement "simple" et "élégant".

write(c);

end;

end;

end

2) version avec branchements explicites

end

Le problème est que le fonctionnement correct de cet algorithme exige que le fichier donné "input" possède une structure bien précise qu'on peut déterminer par une analyse de l'algorithme; nous allons faire cette analyse sur la version "structurée".

En représentant une relation d'enclenchement par une flèche pointant vers le bas,et une relation de succession par une flèche horizontale pointant vers la droite, on peut schématiser la division de l'algorithme en sous-algorithmes commé suit :

Cet algorithme est un "programme" si les expressions booléennes ① et ③ sont susceptibles de valoir respectivement "faux" et "vrai" pour toute exécution de l'algorithme; la valeur de ② est sans importance puisque cette expression booléenne appartient à une instruction de sélection.

La condition ① : non eof(input) est susceptible de produire une valeur "faux", en effet :

Lors de tout enclenchement de l'instruction-holon (a), le fichier "input" est soit vide, soit non-vide; s'il est vide, la valeur de ① est "faux"; s'il est non-vide, toute production effective d'un effet-net par la partie itérative de l'instruction (a) comprend la lecture d'au moins un caractère du fichier, d'où une progression dans le fichier dont la fin sera certainement atteinte après un nombre n d'itérations, le fichier étant fini.

Montrer que  $\textcircled{3}: c \neq "$  est susceptible de produire une valeur "vrai" implique une analyse plus complexe. L'examen de l'instruction (d) permet de conclure que son effet-net est de progresser dans le fichier de n caractères à partir du caractère "c\_0", dernier caractère lu avant l'enclenchement de l'instruction (d), la valeur de n étant égale au nombre d'espacements consécutifs qui suivent directement "c\_0" augmenté d'une unité. En d'autres termes, si on représente le contenu du fichier par :

$$c_{o-2}, c_{o-1}, c_{o}, c_{1}, c_{2}, c_{3}, \dots, c_{n-1}, c_{n}, c_{n+1}, c_{n+2}$$

où la flèche A indique la position de lecture avant enclenchement de l'instruction, tous les caractères à gauche de cette flèche ont déjà été lus; l'effet-net de l'instruction sera de déplacer la flèche de la position A à la position B, étant donné que les caractères  $c_i$ , avec  $1 \le i \le n-1$ , sont

des espacements et  $c_n$  est différent d'un espacement. Pour que (c  $\neq$  ' ') produise une valeur "vrai", il faut que la structure du fichier soit :

<fichier>::={<séquence>}...
<séquence>::=[{<blanc>}...]<caractère>,

c'est-à-dire que le fichier ne se termine pas par une séquence d'espacements !

D'autre part, pour qu'une instruction "repeat-until" puisse être enclenchée correctement, il faut qu'on puisse exécuter la première fois la partie répétitive; dans le cas présent, il faut être sûr qu'avant tout enclenchement de l'instruction (d) il reste au moins un caractère sur le fichier. L'examen de l'instruction (b) supra-ordonnée de (d) nous apprend que (d) n'est enclenchée que si on vient de repérer un espacement; d'où si la condition sur la structure du fichier est vérifiée, il n'y a aucun danger d'enclenchement incorrect de l'instruction (d) puisque un espacement ne terminera jamais le fichier.

On remarquera que notre analyse a été guidée par la forme syntaxique du programme; de celle-ci, on a pu déterminer immédiatement les conditions cruciales sur les possibilités de production de valeur booléenne et sur la nécessité de pouvoir accomplir une opération précise. Dans la version avec branchements explicites, ces points cruciaux sont plus difficiles à déterminer, de plus on voit aisément que, pour un programme de plus grande taille, la détermination de ces points cruciaux sera beaucoup plus difficile <sup>(1)</sup>; à notre avis, la représentation graphique sous forme de "flow-chart" est de peu de secours.

Dans la discussion précédente, nous avons utilisé une représentation schématique des relations d'enclenchements et de succession qui a déjà une allure d'arbre; nous allons voir que par un arbre orienté, il est possible de représenter en même temps la structure statique du texte du programme et la structure dynamique de son exécution. Cette forme de représentation utilise les conventions suivantes :

1) les instructions "while b do s od" et "repeat s until b" sont représentées respectivement par :

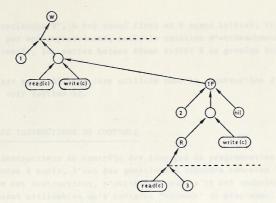


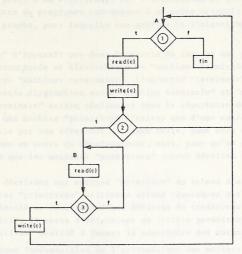
2) l'instruction "if b then a else z fi"par



- 3) un noeud vide correspond à une "collection" d'une série d'opérations;
- 4) une ligne horizontale pointillée indique la possibilité de répétition infinie d'un sousarbre;
- 5) les arcs sont orientés conformément à la définition des arbres orientés, il faut interpréter la pointe d'un arc comme identifiant le supraordonné du noeud d'où l'arc est issu.

<sup>(</sup>¹) En effet, une expression booléenne telle que ② n'est pas cruciale parce qu'aucun branchement ne renvoie à une instruction comprise entre le premier then du programme et l'instruction if qui contient ② ; imaginons maintenant qu'au lieu d'une dizaine de lignes, le programme en comporte deux ou trois mille ... débrouiller la structure du programme devient bien complexe...





# 10.3. Programmes séquentiels et arbres ordonnés.

Pour les programmes séquentiels, il est nécessaire de représenter l'ordre d'exécution des opérations; c'est pourquoi on utilise les arbres ordonnés.

En effet, les deux arbres suivants sont équivalents s'ils sont interprétés comme arbres orientés, et différents s'ils sont interprétés comme arbres ordonnés.

# 10.4. Programmes-holon et arbres binaires.

Un arbre binaire est un arbre ordonné tel que chaque noeud est un noeud final pour deux arcs au plus; ces deux arcs sont qualifiés respectivement d'arc à gauche et d'arc à droite.

Les holons d'un programme-holon sont associés par les relations "enclenche" et "succède à"; nous représenterons un programme-holon par un arbre binaire tel qu'un arc à droite représente une relation "succède à" et un arc à gauche une relation "enclenche". En gardant la convention d'orientation des arbres orientés, si "B succède à A", le noeud A sera le noeud final de l'arc et B le

noeud initial, si "A enclenche B", A est noeud final et B noeud initial; si toute une série de holons est enclenchée par son supraordonné, seule la relation d'enclenchement du premier holon de la série est représentée, les autres holons étant reliés à ce premier holon par les relations de succession.

La représentation par arbre binaire sera utilisée pour la construction d'un modèle d'exécution d'un programme-holon; voir section 12.

# 11. HIERARCHISATION DES INSTRUCTIONS DE CONTROLE

A notre avis, les instructions de contrôle des langages de programmation feront l'objet d'études sérieuses dans les années à venir; l'une des questions à résoudre concerne l'opportunité d'établir une hiérarchisation de ces instructions, c'est-à-dire voir s'il est souhaitable que certaines formes de contrôle ne soient utilisables qu'à certains "niveaux" du programme.

Dans le cas du langage par holons, nous n'avons pas introduit d'instruction de branchement explicite à n'importe quel point d'un algorithme, car l'utilisation de cette instruction ne permet plus de garantir que la structure du programme correspond à un arbre orienté; la structure d'arbre fait place à une structure de graphe, pour laquelle une analyse de l'algorithme ne peut être aussi simple et complète.

Toutefois, si le "goto" n'apparaît pas dans la partie du langage qui permet la création de programmes par association structurée et hiérarchique de "machines terminales", rien n'est affirmé quant à la structure de ces "machines terminales". L'adjectif "terminale" associé à "machine" implique qu'il existe une nette distinction entre "machine terminale" et "machine primitive" (1). En effet, une machine "terminale" existe réellement dans le répertoire des machines "de base" du programmeur; par contre, une machine "primitive" n'existe que d'une manière purement "virtuelle", elle est clairement définie par son effet-net et comme telle, peut être utilisée pour justifier l'exactitude d'un programme en cours de développement; mais, pour qu'un programme devienne "exécutable", il est nécessaire que les machines "primitives" soient décrites en termes de machines "terminales". (2)

Il est clair que nous décrivons une machine "primitive" en termes d'autres machines "primitives" jusqu'à ce que les machines "primitives" utilisées soient identiques aux machines "terminales" que nous possédons; ces dernières résultent d'une décision de transformer un ensemble de machines "primitives" en "terminales", il reste à déterminer un critère permettant de sélectionner cet ensemble de machines "primitives" destiné à former le répertoire des machines "terminales".

Mis à part les opérations fondamentales de l'arithmétique des entiers, la prédétermination d'un ensemble d'opérations "terminales" n'est possible que par un examen du domaine d'application de l'utilisateur éventuel du langage. Une opération est "terminale" pour un utilisateur donné si elle fait partie des opérations qu'il utilise couramment et qu'il désire utiliser sans devoir entrer dans tous les détails de sa réalisation (1\*). Un ensemble d'opérations "terminales" représente la transformation d'un équipement de base d'usage général en une version "attrayante" pour un utilisateur donné. L'ensemble des opérations "terminales" n'est donc pas déterminé par l'ensemble des opérations "physiquement" réalisables sur un équipement donné.

Lors de la description de ces opérations "terminales", il n'est pas exclu qu'une méthode de description (un autre ensemble d'instructions de contrôle) différente de celle utilisée pour la description des opérations "primitives" soit utilisée; le changement se justifie s'il permet un accroissement de l'efficacité de l'exécution-machine de l'opération sans porter atteinte à l'exactitude de l'opération; l'accroissement d'efficacité est directement mesurable puisque nous envisageons une exécution par un matériel donné.

<sup>(1)</sup> Au sens de "primitive" utilisé par Dijkstra, dans [Dij72]. (2) Le mathématicien peut s'arrêter au stade de la machine "primitive", mais le programmeur <u>doit</u> aller jusqu'au stade de la machine "terminale".

<sup>(1\*)</sup> Exemple: la majorité des algorithmes de dimensionnement des engrenages font appel au calcul de "arcinvolute(y)" où "involute(x)" est égale à "tangente(x)-(x)"; cette opération est sans aucun doute "terminale" pour un ingénieur mécanicien impliqué dans des calculs d'engrenage, et est sans intérêt pour tout autre utilisateur. On remarquera que l'algorithme de calcul de "arcinvolute" n'est pas une simple combinaison des fonctions trigonométriques qui accompagnent les langages classiques. Il existe donc un domaine d'application où "arcinvolute" est "terminale", et, vu la fréquence d'utilisation, exige une programmation efficace.

#### Remarque :

La "coroutine" n'existe pas en HPL; elle devrait être abordée dans le cadre d'une version du langage destinée à la programmation de problèmes dont la structure de la solution correspond à une coopération de processus.

## 12. LES LANGAGES TERMINAUX

### 12.1. Définition d'une opération terminale.

La structure du HPL présentée jusqu'ici apparaît comme une forme d'organisation de l'ordre de production d'effets-nets par des opérations dites "terminales". La fonction des langages terminaux est d'exprimer ces opérations terminales qui se distinguent des opérations "primitives" par leur caractère de réalité "matérielle", dans le sens que toute opération terminale est immédiatement disponible sur l'équipement utilisé.

La caractéristique d'une opération terminale est la production effective d'un effet-net qui est défini comme suit : si, avant l'enclenchement de l'opération terminale "t", il existe un prédicat A vérifié par des objets appartenant à un (ou des) type d'objet bien défini, alors l'enclenchement de l'opération terminale entraîne la production effective d'un effet-net tel qu'un prédicat P est vérifié soit par les objets (ou un sous-ensemble de ceux-ci) qui vérifiaient le prédicat A, soit par un nouvel objet qui n'intervenait pas dans la vérification du prédicat A, soit encore par une combinaison de ces deux éventualités.

· On note cet effet-net :

{A}t{P}

# Exemples :

- 1) A = (a est entier > o and b est entier > o and X est entier)
  - $t \equiv (X := PGCD(a,b))$
- P = X est le plus grand commun diviseur de a et de b
- $\equiv \underline{\text{non}}$   $\exists Y: (a=Y,q_1 \underline{\text{and}} b=Y,q_2 \underline{\text{and}} q_1 > o \underline{\text{and}} q_2 > o \underline{\text{and}} Y > X)$
- 2) A = (T est un tableau d'entiers de [m:n] éléments and m ≤ n)
- t = sort (T)
- $P \equiv (m < n \text{ and } VT[i] \text{ avec } m < i \le n : T[i-1] \le T[i]) \text{ or } (m=n \text{ and } T[m])$

Ces deux exemples montrent qu'une distinction doit être faite : en effet, le prédicat P du premier exemple doit être évalué en prenant les valeurs des objets "a" et "b" telles qu'elles étaient avant l'enclenchement de "t"; par contre, dans le prédicat P du deuxième, ce sont les valeurs de T après l'exécution qui sont utilisées; de plus, ce dernier prédicat est incomplet car il devrait préciser que toutes les valeurs existant dans T lorsque A est "vrai" existent encore après enclenchement de "t". Il semble utile d'introduire une spécification "historique" des valeurs qui interviennent dans les prédicats P en ajoutant à leurs noms des indices "ante" ou "post" selon qu'ils se réfèrent à la valeur "avant" ou "après" l'exécution de "t".

Selon cette convention, le premier exemple s'écrit :

- A = (a est entier > o and b est entier > o and X est un entier)
- t = (X := PGCD(a,b))
- $P = \underline{\text{non}} \quad \exists Y (a_{\text{ante}} = Y, q_1 \ \underline{\text{and}} \ q_1 > o \ \underline{\text{and}} \ b_{\text{ante}} = Y_n, q_2 \ \underline{\text{and}} \ q_2 > o \ \underline{\text{and}} \ X_{\text{post}} < Y)$

Les indices "ante" et "post" permettent de distinguer dans le prédicat les éléments qui sont effectivement impliqués par l'opération de ceux qui sont là à titre démonstratif; s'il n'existe aucun de ces derniers, on peut supprimer les indices "ante" pour alléger l'écriture; exemple, le prédicat P du deuxième exemple :

$$P = [(m < n \text{ and } VT_{post}[i] \text{ avec } m \le i \le n : T_{post}[i-1] \le T_{post}[i])$$

or (m=n and T[m] = T[n])

tous les éléments de T se retrouvent dans  $T_{post}$ .

#### Remarque :

Si le PGCD est calculé par l'algorithme d'Euclide sans utiliser de variables locales, on doit remplacer P par P' tel que :

$$P' \equiv P \text{ and } (a_{post} \equiv b_{post} \equiv X_{post}).$$

Cette manière de caractériser la production effective de l'effet-net est incomplète; en effet, elle précise exactement ce qui arrive si A est vérifié, mais elle n'indique point l'effet produit si c'est <u>non</u> A qui est "vrai". La pratique générale de déclarer que, dans ce cas, l'opération est "non-définie" ne se justifie pas, car il est nécessaire que chaque opération terminale produise un effet-net; c'est pourquoi, une opération terminale "t" sera totalement définie par :

ce qui signifie qu'en cas de non-vérification du prédicat A, tout enclenchement d'une opération est prohibée, et l'exécution du programme est définitivement interrompue.

# 12.2. Forme généralisée d'une opération terminale.

La forme généralisée d'une opération terminale est :

$$X \leftarrow \underline{\text{apply}}(OP_1, OP_2, OP_3, \dots, OP_n) \underline{\text{to}}(Y_1, Y_2, \dots, Y_m)$$

où X et les  $Y_i$  représentent des objets du programme et où chaque  $\mathrm{OP}_i$  représente une opération.

La signification de cette expression est :

"L'objet X prend une nouvelle valeur qui est calculée par l'application des opérations  $OP_{\hat{1}}$  aux objets  $Y_{\hat{i}}$ , selon un ordre d'application prescrit par la définition de  $\underline{apply}$ ".

Le terme "objet" désigne des "data structures" au sens large; une ligne de l'imprimante peut très bien correspondre à X; de même, par "opération", on entend aussi bien des fonctions (sinus, tangente,..) que des actions (rewind, backspace,...). Toute opération, dont résulte la transformation de la valeur d'un objet ou la génération d'une valeur pour un objet, peut appartenir à un langage terminal; toutefois, un langage terminal ne peut décrire une opération qui syntaxiquement se présenterait comme une opération de sélection ou d'itération; les particularités de la description de l'algorithme utilisé pour évaluer une opération terminale sont "inconnues" et inaccessibles au niveau du HPL.

Pour le programmeur et pour le langage HPL, les opérations terminales sont des "boites noires" qui produisent un effet-net si on les enclenche correctement et l'interruption de l'exécution du programme dans le cas contraire.

## Exemple :

A = a est entier <u>and</u> b est entier <u>and</u> X est entier

t = X:=max(a,b)

P = [(a<sub>ante</sub> ≥ b<sub>ante</sub> <u>and</u> X<sub>post</sub> = a<sub>ante</sub>)

<u>or</u> (a<sub>ante</sub> < b<sub>ante</sub> <u>and</u> X<sub>post</sub> = b<sub>ante</sub>)]

<u>and</u> (a<sub>post</sub> = a<sub>ante</sub> <u>and</u> b<sub>post</sub> = b<sub>ante</sub>)

définit l'opération "max"; que cette opération corresponde au programme :

n'est pas connu au niveau du HPL.

# 12.3. Syntaxe d'une opération terminale.

Nous distinguons deux classes d'opérations terminales :

a) celles qui produisent une nouvelle valeur qu'on peut attribuer à un objet;

# Exemples :

X:=MAX(a,b)
X:=ADD(a,b)
B:=EQUAL(z,x) (B est booleen)

b) celles qui modifient un objet (ou des objets)

Exemples :

sort(A)
incr(A,1) (équivalent à A:=A+1)

A partir de cette classification, on peut définir une syntaxe "théorique" des opérations terminales :

<opération terminale>::=
 <opération à valeur attribuable>|<opération de modification>

<opération de modification>::=
 <nom d'opération>[<liste d'objets>]

d'objets>::=
 ([{{<nom d'objet>|<opération terminale>},}...]
 {<nom d'objet>|<opération terminale>})

La syntaxe de "liste d'objets" permet d'écrire des opérations du genre :

Y:=add(X:=max(a,b),Z:=add(W,Y)),

expression équivalente à :

X:=max(a,b);
Z:=add(W,Y);
Y:=add(X,Z);

Cette équivalence est une conséquence de la définition de apply, à savoir :

Pour évaluer  $X:=OP(Y_1,Y_2)$ , on évalue d'abord les  $Y_i$  dans l'ordre croissant de leurs indices; quand tous les objets sont évalués, on attribue la valeur de l'opération à X.

On admettra que :

Y:=add(X:=max(a,b),Z:=add(W,Y))

s'écrive :

Y:=add(max(a,b),add(W,Y)),

si les valeurs prisent par "X" et "Z" ne nous intéressent qu'à titre purement temporaire, pour l'évaluation de Y.

Une autre syntaxe peut être substituée à la syntaxe "théorique" pour la représentation des mêmes opérations, à condition que la correspondance entre cette nouvelle syntaxe et la syntaxe "théorique" soit entièrement définie.

# Exemples :

 $A \equiv a$  est un entier and b est un entier and X est un entier

 $t \equiv X := add(a,b)$ 

P = (Xpost = ante + bante)and [("a" = "X" and bante = bpost)or("b" = "X" and ante = a post) or ("a" \neq "X" and "b" \neq "X" and apost = ante and bpost = bante) or ("a" = "b" = "X" and bpost \neq bante and apost \neq and apost \neq ante)] ("a" signifie le nom de l'objet(a), et a la valeur de l'objet).

L'effet-net de cette opération n'est pas modifié, si la syntaxe de "t" devient :

en postulant que cette écriture n'est qu'une variante de la syntaxe de X:=add(a,b).

A = X est un tableau[m:n]and 1 est entier and u est entier

and  $1 \ge m$  and  $u \le n$ 

t ≡ zéro(X,1,u)

 $P = (X_{post}[i] \text{ avec } 1 \le i \le u = o) \underline{\text{and}} (\text{les \'el\'ements restant } X_{post}[i] = X_{ante}[i])$ 

Cette opération peut être exprimée par :

for I:=1 step 1 until u do X[i]:=0;

# 12.4. Définition d'un langage terminal.

Un langage terminal est un ensemble d'opérations terminales auquel on attribue un nom.

Un holon terminal est un holon dont le coprs est formé par une séquence d'opérations terminales appartenant toutes au même langage terminal.

La définition d'un langage terminal inclut la définition des éléments suivants :

- 1) le nom du langage terminal et son désignateur, ce dernier est une abréviation du "nom" utilisée par chaque holon terminal pour signaler au système de traitement du programme-holon quel est le langage terminal qui permet d'interpréter les opérations du holon;
- 2) la liste des opérations terminales qui composent le langage; chaque opération est définie par
- la triplet A,t,P où "t" représente l'opération selon la syntaxe théorique;
- 3) éventuellement, les variations syntaxiques admises pour représenter les opérations terminales,
- et la correspondance entre ces variations et l'ordre d'évaluation tel qu'il est défini par apply.

L'ensemble des opérations terminales qui forment un langage terminal définit également le domaine des types d'objets accessibles aux opérations du langage, ces types sont définis pour chaque opération terminale dans le prédicat "ante" de l'opération. Normalement, un langage terminal contient une série d'opérations de relations entre ses objets, ces opérations produisent une valeur de type booléen qui peut être utilisée dans la programmation des holons booléens.

Une opération terminale construite selon la syntaxe théorique ne peut mêler des noms d'opérations qui appartiennent à des langages terminaux différents.

La réunion des opérations terminales en vue de former un langage terminal doit suivre un critère logique de regroupement des opérations selon des domaines d'application : arithmétique des entiers, arithmétique des réels, manipulation de tableaux, traitement de caractères, édition de textes, opérations entrée-sortie, manipulation de graphes, etc... Sans oublier des langages terminaux très spécialisés pour des applications particulières : calcul d'engrenages, traçage de courbes, opérations de commande de machine-outil, etc...

## 12.5. Syntaxes externe et interne.

Du point de vue du programmeur, le langage se présente comme suit :

- 1) une syntaxe externe qui permet de décrire des opérations primitives et les relations de successions d'exécution entre ces opérations;
- 2) une *syntaxe interne* qui permet d'invoquer des opérations terminales pour lesquelles il existe un système d'exécution.

Les termes syntaxe externe et syntaxe interne sont empruntés à M.V. Wilkes [Wil68].

Les règles de définition d'un langage terminal doivent servir "d'interface" entre un utilisateur

éventuel qui désirerait utiliser un nouveau langage terminal et la personne chargée d'en réaliser une implémentation. La manière de réaliser cette implémentation importe peu, seul doit être garanti que chaque apparition du nom d'une opération terminale dans un programme-holon impliquera l'enclenchement du système physique qui permet la réalisation de cette opération. Il n'est pas exclu qu'entre le "nom" et le système physique s'insère un intermédiaire : soit un langage de programmation conçu à cet effet, soit un langage d'assemblage.

On peut concevoir l'existence d'un langage d'implémentation de langages terminaux qui permettrait à certains utilisateurs de définir leurs propres langages terminaux. Ce langage pourrait être un langage sans-type dont l'objet de base est le "mot-machine", les instructions de "flow of control" devraient être élémentaires : une instruction conditionnelle, des étiquettes et le "goto", une possibilité de définition récursive des opérations; une version simplifiée du BCPL pourrait sans doute convenir.

Les syntaxes externe et interne n'ont qu'un élément en commun : la notion d'objet; les noms des mêmes objets peuvent apparaître dans l'une et l'autre syntaxes, le système de traitement des programmes HPL doit être à même de réaliser les corrélations entre les noms d'objets utilisés dans les textes des opérations "primitives" et "terminales". Dans le chapitre IV, on verra que la syntaxe externe peut avoir accès au prédicat A d'une opération terminale "t".

#### 13. MODELE D'EXECUTION D'UN PROGRAMME HPL.

Le modèle donne une représentation d'un programme-holon sous la forme d'un arbre binaire fini; en même temps, il permet de simuler l'exécution du programme-holon par une traversée de l'arbre. Son intérêt pratique résidedæns le caractère visuel de la représentation et dans la possibilité de définir, à partir du comportement du modèle lors de la traversée de l'arbre, les caractéristiques qui doivent être communes à toutes les implémentations d'un programme-holon.

#### 13.1. Eléments du modèle.

Les différentes catégories de noeuds utilisés dans une représentation sont :

- a) noeud correspondant à un nom de holon, la représentation inclut le nom de holon (ou une abréviation adéquate) à l'intérieur du noeud; exemple : si le nom du holon est  $A:\widehat{A}$  .
- b) noeuds correspondant à une instruction-holon,
  le premier symbole réservé (ou une abréviation de celui-ci) de l'instruction-holon apparaît à
  l'intérieur du noeud ;
  exemples :

  (IP) (W) (CA) (RE)
- c) noeud correspondant à un ensemble de holons, l'ensemble de holons est représenté par un noeud "vide", relié à gauche au premier holon de l'ensemble; ce noeud correspond au nom d'un holon implicite de la structure; si besoin est, un nom fictif (sous la forme d'un nombre) est attribué à ce holon implicite; exemples :
- d) nocud correspondant à un holon booléen, ce nocud, de format carré, contient le nom du holon booléen; exemples :

A B

e) noeud d'aiguillage,
ce noeud est utilisé pour représenter une sélection, sa représentation est d (d pour "dispatching");
on peut imaginer que de ce noeud, partent quatre arcs : deux arcs "structuraux" et deux arcs "opérationnels"; les arcs structuraux indiquent les ensembles de holons enclenchés selon la valeur booléenne
produite par le holon booléen voisin du noeud; les arcs opérationnels sont utilisés, lors de la tra-

versée de l'arbre, pour parcourir l'ensemble de holons qui résulte de l'opération de sélection et celui-là seulement.

f) noeud correspondant à un holon terminal, se représentation en forme de triangle contient le nom de l'opération terminale; exemple :

/T\ /B\

# Remarque :

Une distinction est à faire entre noeud (RE) et noeud (RE)

# 13.2. Sous-arbres binaires correspondant aux éléments du langage-holon.

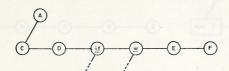
Les méthodes de construction de ces sous-arbres pourraient être explicitées par des règles très précises, mais nous nous contenterons de les exposer par une série d'exemples, vu la simplicité des constructions.

## a) Holon.

Soit le holon

[A,begin C;D;if...fi; while...od;E;F;end]

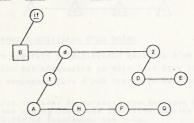
Représentation :



Cette forme de représentation se répétant pour les holons C,D,E,F.

#### b) if-then-else

# if B then A; H; F; Q; else D; E fi



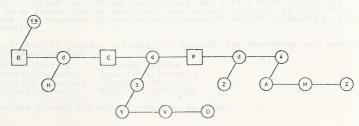
## c) case

 case B
 then H;

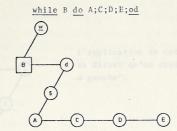
 eor C
 C then Y,V,D;

 eor F
 then Z;

 else A;M;Z



### d) while-do

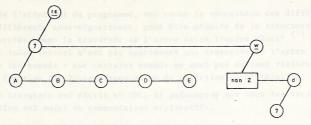


### e) repeat-until

# repeat A; B; C; D; E; until Z;

La représentation est basée sur l'équivalence entre une instruction-holon "repeat" et une instruction-holon "while", à savoir :

# repeat S until B; = S; while non B do S od;



# f) holon terminal

Soit un holon T composé d'une succession d'opérations terminales A,B,C,D.



# 13.3. Règle en cas d'occurrences multiples d'un holon.

Si un même nom "x" de holon apparaît en différents endroits d'un programme-holon, une seule représentation de sa description est nécessaire et elle doit être liée à la gauche du noeud "x" qui sera le premier de ce nom rencontré lors d'une traversée de l'arbre selon "l'ordre infixé". (1)

Remarque : Dans la deuxième édition de [Knu68], les adjectifs associés aux traversées sont modifiés par rapport à la première édition :

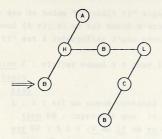
Première édition Deuxième édition Preorder traversal Preorder traversal Postorder traversal Inorder traversal Endorder traversal Postorder traversa

C'est la notation de la deuxième édition qui est utilisée dans ce texte.

<sup>(1)</sup> Traversée d'un arbre binaire :méthode de parcours systématique de tous les noeuds d'un arbre binaire, en accomplissant sur chaque noeud une opération appelée "visite du noeud"; lors d'une traversée, chaque noeud est visité exactement une fois.

La traversée selon l'ordre "infixé" est définie récursivement par : si l'arbre binaire est vide, il est "traversé" en ne faisant aucune opération; dans le cas contraire, la traversée procède selon trois étapes : I) traverser le sous-arbre à gauche; 2) visiter le noeud; 3) traverser le sous-arbre à droite.

Par exemple : dans l'arbre suivant, la description du holon B doit être liée à la gauche du noeud pointé par -.



L'application de cette règle est exprimée en disant qu'on choisit le noeud "le plus à gauche".

## 13.4. Exécution d'un programme-holon.

Soit un arbre binaire représentant un programme-holon complet, c'est-à-dire que tous les noeuds terminaux des branches de l'arbre correspondent soit à une opération terminale, soit au nom d'un sous-arbre existant dans le même arbre selon la règle du "plus à gauche". On suppose, d'autre part, que les informations sur l'environnement du programme et ses modifications en cours d'exécution sont également disponibles, (elles ne sont pas représentées dans l'arbre binaire).

La production de l'effet-net du programme, vue comme la succession des différents effets-nets produits par les différents sous-algorithmes, peut être déduite de la structure de l'arbre. L'exécution sera représentée comme la traversée de l'arbre selon l'ordre "dual" (1) par un index textuel. Le parcours de cet index textuel n'est pas exactement une traversée de l'arbre - qui impliquerait une visite de tous les noeuds - car certains noeuds ne sont pas du tout visités ou sont visités plusieurs fois en fonction des instructions-holon de sélection ou d'itération.

L'algorithme de traversée est décrit en HPL, il présuppose que tous les arcs "fonctionnels" sont établis. Chaque holon est suivi de commentaires explicatifs.

```
holon A : traverser arbre de racine % t
 begin declare G,D;
   if B : % t ≠ nil
     then C: visiter noeud % t pour la première fois ;
           D : "G":=AG(% t);
           A : traverser arbre de racine % G;
           E : visiter noeud % t pour la deuxième fois;
      F : "D":=AD(% t);
           A : traverser arbre de racine % D;
    fi
  end
noloh
```

# Commentaires :

Chaque nom de holon est précédé d'un préfixe formé d'une lettre suivie du symbole ":", la lettre sera utilisée pour représenter le holon dans le schéma sous forme d'arbre binaire qui va suivre; ce n'est point une obligation syntaxique, il faut remarquer que ce préfixe fait syntaxiquement partie du nom du holon.

Les déclarations ne précisent pas le type des variables déclarées, ce qui est contraire à la syntaxe; toutefois, dans le cas présent, il suffit de savoir que "T,G,D" représentent des valeurs de

<sup>(1)</sup> L'ordre "dual"est une combinaison des ordres "préfixé" et "infixé", il est définit comme suit : si l'arbre binaire est vide, ne rien faire; dans le cas contraire : visiter le nocud pour la première fois; traverser le sous-arbre à gauche selon l'ordre "dual"; visiter le nocud pour la deuxième fois;

d) traverser le sous-arbre à droite selon l'ordre "dual".

même type, à savoir la position précise d'un noeud dans l'arbre binaire.

Le nom de holon booléen " $\$  t  $\neq$  nil" signifie que l'existence du noeud  $\$  t est vérifiée; le noeud n'existe pas dans le cas des successeurs à droite et à gauche d'un noeud terminal.

Le nom de holon "G":=AG(% t)" signifie qu'on attribue à G la position du noeud lié "à gauche" du noeud (% t); si un tel noeud n'existe pas, la valeur attribuée à G est "nil"; le nom ""D":= AD(% t)" est à interpréter d'une manière identique mais par rapport à une "liaison à droite".

```
holon C : visiter noeud % t pour la première fois
 begin
   case
    L: % t est un noeud terminal
     then RB : rapporter que le noeud terminal % t est enclenché;
   eor OP: % t = (W ou if ou re ou ca)
     then HB : rapporter que l'instruction-holon % t est enclenchée;
   eor ES : % t = noeud d'ensemble de holons
     then EB : rapporter que l'ensemble de holons % t est enclenché;
   eor HY: % t = noeud correspondant à un holon ou à un holon booléen
     then HYB : rapporter que le holon % t est enclenché
    else Z : pas d'opération
   esac
  end
noloh
holon E : visiter noeud % t pour la deuxième fois
 begin declare G;
   case
   L: % t est un noeud terminal
     then T : exécuter l'instruction terminale qu'il contient;
     R : rapporter que l'effet-net du noeud terminal % t a été effectivement produit;
   eor M : % t = W avec arc opérationnel à gauche ≠ nil
     then D : "G":=AG(% t);
        A : traverser arbre de racine % G;
           E : visiter noeud % t pour la deuxième fois;
   eor N : % t = W avec arc opérationnel à gauche = nil
     then H : rapporter que l'effet-net de l'instruction-holon % t a été effectivement produit ;
           I : restaurer la valeur de l'arc opérationnel à gauche de % t;
   eor 0 : % t = (re ou if ou ca)
     then H : rapporter que l'effet-net de l'instruction-holon % t a été effectivement produit;
    eor P : % t = d
     then Z : pas d'opération;
     else Y : constater que l'effet-net du noeud % t a été effectivement produit;
    esac
noloh
```

#### Commentaires :

La variable t représente la position d'un noeud dans l'arbre, le nom de holon "\$ t  $\Xi$  d" (ou tout équivalent) signifie "le noeud dont la position est \$ t est de type d".

```
holon Y : constater que l'effet-net du noeud % t a été effectivement produit
begin declare D;
if Q : noeud % t est booléen
then F : "D":=AD(% t);
FA : établir valeurs des arcs opérationnels de "D" en fonction de la valeur booléenne
de % t;
```

W : modifier s'il y a lieu la valeur de l'arc opérationnel à gauche d'un noeud (w) supraordonné de % t:

J : rendre la valeur booléenne de % t indéfinie;

fi

YY : rapporter que l'effet-net du noeud % t a été effectivement produit;

end noloh

# Commentaires :

Ce holon explicite l'effet d'une valeur booléenne pouvant impliquer des sélections et des itérations; le noeud D à la droite d'un noeud booléen est toujours un noeud d'aiguillage.

holon fa : établir valeurs des arcs opérationnels de "D" en fonction de la valeur booléenne de % t

#### begin

F1 : copier valeurs des arcs structuraux de "D" dans ses arcs opérationnels;

#### case

FV : valeur holon booléen % t est (vrai)

then F2 : casser l'arc opérationnel à droite de "D";

eor FF : valeur holon booléen % t est (faux)

then F3 : casser l'arc opérationnel à gauche de "D";

else abort (holon booléen A produit une valeur indéfinie)

esac

end

noloh

#### Commentaires :

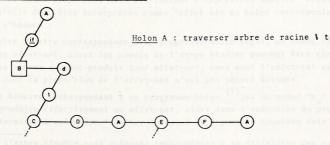
Le rôle du holon "abort"sera exposé au chapitre III. Son effet est évident:si l'arbre binaire correspond à un programme légal, le holon "abort" ne sera jamais enclenché puisqu'un holon booléen doit toujours produire une valeur booléenne définie; abort est une opération de vérification qui n'entraîne aucune perte d'efficacité de l'algorithme.

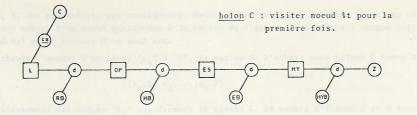
# Remarques :

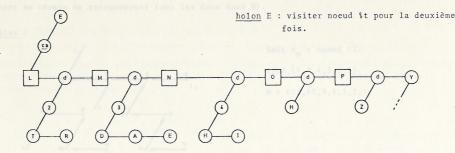
- 1. L'algorithme décrit une série d'opérations "primitives" qui doivent encore être décrites en termes d'opérations "terminales".

  2. Une opération importante de l'algorithme n'a pas été exprimée afin de limiter la description à l'essentiel : la recherche du sous-arbre de même nom "x" "le plus à gauche", quand l'index textuel atteint un noeud correspondant à un nom de holon "x" non-terminal et qui ne possède au cun successeur à gauche; on peut imaginer qu'on copie temporairement à gauche de ce noeud le sous-arbre de même nom, ou si le holon "x" n'est pas récursif, l'index textuel peut continuer sa traversée de l'arbre binaire en parcourant temporairement le sous-arbre "x" "le plus à gauche" et en revenant au nocud "x" abandonné des que le sous-arbre aura été entièrement traversé. 3. Le terme "index textuel" est utilisé car sa fonction est identique à celle de l'index textuel
  - utilisé par Dijkstra [Dij68] pour caractériser l'évolution d'un processus de calcul.

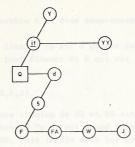
# 13.5. Arbre binaire de l'algorithme de traversée.







holon Y : constater quc l'effet-net
 du noeud % t a été
 effectivement produit.



#### 14. PROPRIETES D'UN PROGRAMME HPL.

La représentation sous forme d'arbre binaire permet de déterminer des propriétés intéressantes de l'exécution d'un programme-holon; ces propriétés ne doivent pas être attribuées à l'arbre binaire en tant que tel, mais à la structure de la syntaxe du langage dont l'arbre binaire n'est qu'une "transformation conforme" plus maniable pour l'établissement de ces propriétés.

### 14.1. Propriété d'exécution.

Dans l'énoncé de cette propriété, nous utiliserons l'expression abusive "effet-net d'un noeud" étant entendu qu'elle doit être interprétée comme "effet-net du holon correspondant au noeud".

La propriété s'énonce :

Si dans un arbre binaire correspondant à un programme-holon légal, un noeud vient de produire effectivement son effet-net, alors les noeuds de l'arbre binaire peuvent être classés en trois ensembles distincts : ceux qui ont produit leur effet-net, ceux dont l'effet-net est partiellement produit et ceux dont la production de l'effet-net n'est pas encore entamée.

Soit un arbre binaire correspondant à un programme-holon  $^{(1)}$ ; si un noeud "xo" de cet arbre vient juste de produire effectivement un effet-net, alors dans l'exécution du programme-holon, on vient juste de terminer l'opération "visiter le noeud (xo) pour la deuxième fois". Puisque l'arbre

<sup>(1)</sup> Les arcs de l'arbre binaire sont orientés conformément à la définition des arbres orientés.

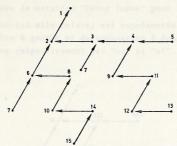
est binaire, il est ordonné et, par conséquence, orienté; puisqu'il est orienté, il existe un chemin orienté menant d'un noeud quelconque à la racine de l'arbre; ce chemin est unique puisque chaque noeud est noeud initial d'un seul arc.

Soit le chemin C menant d'un noeud "x\_" à "R", la racine de l'arbre; nous notons C comme suit :

$$C(x_0, x_1, x_2, ..., x_n, R)$$

Soit N, l'ensemble des noeuds " $x_i$ " qui forment le chemin C, le nombre d'éléments de N étant égal à n + 1, si n est le nombre d'arcs du chemin, (c'est-à-dire que deux noeuds de nom identique appartenant au chemin se retrouveront tous les deux dans N).

# Exemples :



Soit x<sub>o</sub> = noeud 12. C(12,11,9,4,3,2,1) N = {12,11,9,4,3,2,1}

Nous allons effectuer une partition de l'ensemble N en deux sous-ensembles disjoints ND et NG définis comme suit :

- 1) NG contient tout élément de N qui est noeud final d'un arc à gauche dans le chemin C;
- 2) ND contient le noeud intial du chemin C, et tout élément de N qui est noeud final d'un arc à doite dans le chemin C.

Dans l'exemple : NG =  $\{1,4,11\}$  et ND =  $\{12,9,3,2\}$ .

Cette partition est complète, c'est-à-dire que l'union de NG et ND est un ensemble équivalent à l'ensemble N. En effet, tout élément de N, excepté "x<sub>0</sub>", est noeud final d'un arc du chemin C; d'où il se retrouvera soit dans ND, soit dans NG, mais jamais dans les deux.

A partir de cette partition, on peut déterminer l'état de l'exécution du programme, car :

- l'ensemble NG contient tout noeud dont l'effet-net est en cours de production (partiellement produit);
- 2) l'ensemble ND contient tout noeud qui est racine d'un sous-arbre dont l'effet-net a été effectivement produit;
- 3) tout noeud de l'arbre qui est noeud initial d'un arc dont le noeud final appartient à NG ou est "xo", est racine d'un sous-arbre dont la production de l'effet-net n'a pas été entamée.

En effet, l'appartenance d'un noeud "y", différent de "xo", à l'ensemble ND signifie que dans le chemin C, ce noeud est atteint par la droite, c'est-à-dire "xo" appartient au sous-arbre à droite de ce noeud "y"; dans ce cas, l'opération "visiter le noeud (y) pour la deuxième fois" a été effectuée, puisque la traversée du sous-arbre à droite de (y) est une opération qui succède à cette "visite du noeud" dans l'algorithme de traversée de l'arbre et que "xo" ne peut avoir été visité que si la traversée du sous-arbre à droite de "y" a vraiment été entamée.

D'une manière similaire, on montre que les sous-arbres à gauche des noeuds de NG ne sont pas entièrement traversés; de là, on déduit que les noeuds de NG ne peuvent avoir produit effectivement leurs effets-nets, mais que la production de ceux-ci est entamée (sinon on ne pourrait pas les atteindre par la gauche).

Quant aux noeuds du troisième ensemble, il est évident que la production de leurs effets-nets ne peut avoir été commencée puisqu'ils sont soit membre d'un sous-arbre à droite d'un noeud dont

l'opération "visiter le noeud pour la deuxième fois" n'a pas été effectuée, soit membre du sousarbre à droite du noeud pour lequel cette opération vient juste de se terminer (cas du noeud "13" dans l'exemple).

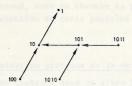
La propriété d'exécution justifie complètement l'idée de l'index textuel, dans le sens que, à partir de n'importe quel point d'un programme-holon, on peut déterminer exactement l'état de réalisation effective de l'effet-net du programme. Cette propriété est due essentiellement à l'absence de "circuit" dans le programme, c'est-à-dire à l'ensemble restreint d'instructions de contrôle utilisées.

# 14.2. Détermination pratique de la propriété d'exécution.

Le système de notation "Dewey Index" pour les arbres binaires est défini comme suit :

La racine (si elle existe) est représentée par la séquence "1"; les racines (si elles existent) du sous-arbre à gauche et du sous-arbre à droite du noeud représenté par la séquence " $\alpha$ " sont représentées respectivement par " $\alpha$ 0" et " $\alpha$ 1".

# Exemple :



Dans l'exemple utilisé lors de la démonstration de la première propriété, on a la correspondance suivante entre numéro de noeud et notation Dewey :

1 + 1 6 + 100 11 + 101101 2 + 10 7 + 1010 12 + 1011010 3 + 101 8 + 1001 13 + 10110101 4 + 1011 9 + 10110 14 + 1001015 + 101111 10 + 10010 15 + 1001010

### Le chemin C est :

# L'ensemble NG :

12 + 1011010  $\{1 \rightarrow 1$  11 + 101101  $4 \rightarrow 1011$   $9 \rightarrow 10110$   $11 \rightarrow 101101\}$   $4 \rightarrow 1011$   $3 \rightarrow 101$  $2 \rightarrow 10$ 

L'index Dewey permet de déterminer facilement les ensembles NG et ND correspondant à un noeud de l'arbre binaire.

Tous les noeuds du chemin C d'un holon " $x_0$ " se déduisent de l'index Dewey de " $x_0$ " en supprimant le dernier symbole de la séquence initiale et en recommençant cette opération sur la nouvelle séquence obtenue jusqu'à ce qu'elle se réduise à la séquence "1".

Les éléments de l'ensemble NG sont obtenus en extrayant de la notation Dewey du noeud " $x_0$ " toutes les sous-séquences qui commencent par le même symbole initial que le Dewey de " $x_0$ " et qui sont suivies dans la notation Dewey de " $x_0$ " par un symbole "o".



Les éléments de l'ensemble ND sont obtenus par simple soustraction de {C} - {NG}.

### 14.3. Propriété de "durée de vie" d'un holon.

La durée de vie d'un holon d'un nom donné est la différence entre l'instant où un holon de ce nom est enclenché pour la première fois et l'instant où un holon de ce nom produit effectivement et pour la dernière fois un effet-net.

Après chaque étape de la production de l'effet-net d'un programme-holon, il est possible d'établir la liste des holons nécessaires pour la poursuite de l'exécution et la liste de ceux dont on n'aura plus jamais besoin. En d'autres termes : la structure d'un programme-holon permet d'établir exactement l'évolution des besoins en "ressources" du programme-holon au cours de son exécution. Cette information est spécialement importante pour les programmes dont la taille excède celle de la mémoire centrale; elle permet, en effet, soit de concevoir un système "d'overlay", soit de "planifier" un système à segmentation (ou à pagination) du programme.

La "durée de vie" d'un holon est connue si, dans l'arbre binaire complet <sup>(1)</sup> la position "la plus à gauche" et la position "la plus à droite" du nocud correspondant à ce holon sont déterminées. Si le nocud, dont on cherche la position "la plus à droite", possède un supra-ordonné récursif, une approximation de cette position extrême pourra tout au plus être obtenue.

## 14.4. Détermination pratique de la durée de vie.

L'information contenue dans un arbre binaire est rassemblée sous la forme de deux dictionnaires :

- le premier, basé sur un ordre lexicographique des index Dewey ("o"<"blanc"<"1"), associe à chaque index le nom du noeud correspondant;
- 2) le deuxième, basé sur un ordre alphabétique des noms des noeuds (suivi par un ordre des noeuds spéciaux : "ensemble", "d", "if", etc...) associe à chaque nom de holon la liste des index Dewey correspondant aux apparitions de ce nom dans l'arbre binaire. (2)

Nous représenterons les contenus de ces dictionnaires, respectivement par :

- I)  $id(i) \rightarrow nh(i)$
- II)  $nh(j) + id(j)_1$ ,  $id(j)_2$ ,  $id(j)_3$ ,..., $id(j)_n$

La liste des id(j); est ordonnée selon l'ordre croissant (id(j); < id(j); 1)

Appelons "x" le nom du holon dont on veut déterminer les positions "la plus à gauche" et "la plus à droite" que nous noterons respectivement pg(x) et pd(x).

L'examen du deuxième dictionnaire permet de trouver  $nh(j) \equiv x$ , s'il existe dans le programme; s'il n'existe pas, la détermination de pg(x) et de pd(x) est évidemment sans objet.

Le pg(x) est donné immédiatement  $\binom{3}{1}$  par  $id(j)_1$ ; c'est une conséquence de l'application de la règle de construction de l'arbre binaire.

Pour déterminer pd(x) nous allons procéder par "approximations successives" : la meilleure approximation que nous connaissons, à priori, de pd(x) est  $id(j)_n$  avec j tel que  $nh(j) \equiv x$ ; on peut certainement écrire que :

$$id(j)_n \leq pd(x)$$
.

Cette valeur est effectivement égale à pd(x) si, et seulement si, il n'existe aucun noeud dont l'index Dewey est supérieur à  $id(j)_n$  et qui est en même temps racine d'un sous-arbre non-recopié

<sup>(1)</sup> Il ne suffit pas de connaître les deux positions extrêmes du holon dans l'arbre binaire construit selon la règle de non-duplication des sous-arbres dont les racines sont des noeuds identiques; cet arbre binaire doit être complété par tous les sous-arbres dont la traversée peut advenir lors d'une exécution quelconque du programme.

<sup>(2)</sup> L'arbre binaire en question dans cette méthode, n'est pas complet, mais est construit selon la règle de non-recopie.

<sup>(3)</sup> Voir commentaires en fin de cette section sur la récursivité.

qui pourrait contenir un noeud (x).

Construisons l'ensemble des supra-ordonnés directs ou indirects de (x): pour chaque  $\operatorname{id}(j)_i$  (avec 1  $\le$  i  $\le$  n), on construit l'ensemble NG qu'on note NG(i); à partir de ces n ensembles NG(i), on construit un ensemble S qui contient tous les éléments différents de l'ensemble résultant de l'union de tous les NG(i). Ce nouvel ensemble S contient tous les supra-ordonnés directs ou indirects de (x).

Soit

$$S = {ids(x)_1, ids(x)_2, ids(x)_3, \dots, ids(x)_m},$$

les éléments de S sont des index Dewey; par examen du dictionnaire I, on peut remplacer chaque élément de S par le nom du holon auquel il correspond, on obtient :

$$s_n = \{nh_1, nh_2, nh_3, \dots, nh_m\}$$

où  $nh_i = nh(i)$  tel que  $ids(x)_i = id(i)$ .

L'ensemble  $S_n$  contient une liste sans redondance  $\binom{1}{2}$  et exhaustive  $\binom{2}{2}$  des supra-ordonnés directs et indirects de (x).

A partir des l'ensemble  $\mathbf{S}_{\mathbf{n}}$ , on construit à l'aide du dictionnaire II, l'ensemble X :

à chaque  $\mathrm{nh}_i$  de  $\mathrm{S}_n$ , on fait correspondre l'index Dewey  $\mathrm{id}(\mathrm{j})_n$  avec  $\mathrm{j}$  tel que  $\mathrm{nh}_i \equiv \mathrm{nh}(\mathrm{j})$ ; c'est à-dire qu'on cherche pour chaque nom de holon de  $\mathrm{S}_n$  l'index correspondant à l'apparition "la plus à droite" de ce nom, l'ensemble X contient ces différents index Dewey.

On détermine l'élément de X qui est l'élément de valeur maximum si on ordonne les éléments de X selon l'ordre lexicographique des index Dewey ("o"<"blanc"<"1,"); soit max(X) cet élément. Si  $\max(X) > \mathrm{id}(j)_n$  avec j tel que  $\mathrm{nh}(j) \equiv x$ , alors  $\max(X)$  est une meilleure approximation de  $\mathrm{pd}(x)$ .

On peut affirmer qu'il n'y aura plus d'enclenchement d'un holon(x) après la production effective de l'effet-net du holon correspondant au noeud obtenu comme meilleure approximation de  $pd(x)^{\binom{3}{2}}$ . Supposons que l'index Dewey du noeud correspondant à la meilleure approximation de pd(x) soit la séquence de symboles "a", on peut affirmer que l'ensemble des noeuds h dont l'index Dewey est supérieur à "a" et n'appartient pas à NG de "a", ne contient pas de noeud (x)  $\binom{4}{3}$ .

Supposons que (x) ne soit pas impliqué dans une chaîne de récursivité (c'est-à-dire qu'aucun des supra-ordonnés de (x) n'est récursif); dans ce cas, si la meilleure approximation de pd(x) a été obtenue à partir de X, elle peut être améliorée jusqu'à valoir la valeur exacte de pd(x). En effet, max(X) est la racine d'un sous-arbre qu'on peut construire par recopie du sous-arbre dont l'index Dewey est  $id(j)_1$  avec j tel que  $nh(j) \equiv nh(i)$  avec id(i) = max(X). Il n'est pas nécessaire de reconstruire tout le sous-arbre mais uniquement la première branche à droite; choisir dans cette dernière le supra-ordonné de (x) "le plus à droite" et recommencer l'opération jusqu'à ce qu'on ne trouve plus de supra-ordonné de (x) à recopier.

<sup>(1)</sup> La liste est sans redondance à cause de l'application de la règle de non-recopie des sousarbres de même non; toute apparition d'un noeud "x" dans l'arbre binaire du programme appartient à un sous-arbre "original", toute redondance a donc été éliminée lors de la construction de S, à partir des NG.

à partir des NG.

(2) Le nombre d'éléments de S est fini et est borné par une valeur que l'on peut déterminer à partir de la liste des id(j) (pour nh(j) = x). En effet, vu la règle de construction de NG, le nombre maximum d'éléments de S que peut générer un id(j) est égal au nombre de symboles "o" utilisés pour exprimer id(j); si pour id(j), ce nombre est q, alors le nombre maximum d'éléments de S est égal à 1 + \( \Sigma(q-1), car on peut soustraire de là contribution de chaque id(j), la racine de l'arbre qui k est une supra-ordonnée commune à tous.

de l'arbre qui ^ est une supra-ordonnée commune à tous.

(3) En effect, s'il n'en est point ainsi, il existe, dans l'arbre à droite du noeud correspondant à la meilleure approximation ou dans un sous-arbre à droite d'un noeud appartenant à l'ensemble NG de ce noeud, une nouvelle apparition d'un noeud (x) que nous appelons "y". Si "y" existe, il doit appartenir soit à la liste des id(j), avec j tel que nh(j) = x (si "y" appartient explicitement à l'arbre binaire), soit à un sous-arbre dont seule la racine appartient explicitement à l'arbre binaire; selon notre méthode de construction de l'approximation de pd(x), ces deux cas sont impossibles.

<sup>(4)</sup> L'ordre lexicographique choisi entraîne des inégalités peu familières :

### Exemple :

Soit le sous-arbre binaire décrivant le holon A :

Supposons qu'on recherche le  $\operatorname{pd}(X)$  et que la meilleure approximation soit un noeud "A" d'index Dewey " $\alpha$ "; la recopie du sous-arbre "A" se fait comme suit :

La valeur de pd(X) est :  $\alpha 0111110010$ 

Si (x) est impliqué dans une chaîne de récursivité, il faut arrêter la recopie dès qu'on a recopié une branche à droite contenant un supra-ordonné récursif de (x). Dans ce cas, la valeur de pd(x) varie d'une exécution à l'autre.

La récursivité peut également modifier la valeur réelle de pg(x) comme en témoigne l'exemple suivant :

où il risque d'y avoir une occurrence de X dans le sous-arbre "10010".

### 14.5. Propriété de récursivité d'un holon.

Tout au long de cet exposé, il est apparu que la propriété pour un holon d'être ou de ne pas être récursif influence la structure du programme et la détermination de certaines propriétés, de plus, elle influencera la méthode d'implémentation du holon.

Puisqu'un holon "x" est récursif s'il existe un holon "x" parmi les subordonnés directs et indirects de "x", on peut en déduire une méthode de détection de la nature récursive d'un holon : un holon "x" sera récursif si un des noeuds du sous-arbre de racine "x" contient un noeud de même nom que "x".

En pratique, on procède comme suit ; la construction progressive du programme-holon entraîne une construction pas-à-pas de l'arbre et des modifications du contenu des deux dictionnaires; chaque fois qu'une description d'un holon est ajoutée au programme-holon, le dictionnaire II peut subir des modifications de deux types : soit l'introduction d'une nouvelle entrée (un nouveau nom de holon et son index Dewey), soit l'adjonction d'un nouvel index Dewey à la liste des index d'une entrée déjà existante; dans le deuxième cas, on peut vérifier si cette adjonction entraîne ou non la récursivité du holon.

En effet, supposons que cette adjonction doit être faite à la liste des index du holon nh(j) dont l'entrée dans le dictionnaire II se présente comme suit :

$$nh(j) \rightarrow id(j)_1, id(j)_2, \dots, id(j)_n$$

où les  $\operatorname{id}(j)_i$  sont classés par ordre croissant selon l'ordre lexicographique (1) ("blanc"<"o"<"1"); l'index Dewey à adjoindre à cette liste est : "a".

Nous supposons qu'avant l'adjonction de "a", on n'ait pas encore découvert de récursion pour  $\mathrm{nh}(j)$ , c'est-à-dire qu'aucun  $\mathrm{id}(j)_j$  avec  $i \geq 2$  ne corresponde à un noeud appartenant au sous-arbre  $\mathrm{nh}(j)$  d'indice  $\mathrm{id}(j)$ , ce qui implique que :

$$id(j)_{i} \equiv id(j)_{1}.1...$$

Tout indice  $\operatorname{id}(j)_i$  est équivalent à la concaténation (symbolisée par ".") de l'index  $\operatorname{id}(j)_1$  et d'une séquence de symboles dont le premier est un "1".

Trois cas sont possibles :

- 1) "a" < id(j),
- 2) id(j)<sub>1</sub> < "a" < id(j)<sub>2</sub>,
- 3) id(j)<sub>2</sub> < "a".

Le premier cas signifie que "a" correspond à une occurrence du holon dans un sous-arbre "plus à gauche" que  $\operatorname{id}(j)_1$ , ce qui ne peut pas impliquer directement la récursion de  $\operatorname{nh}(j)$ . Toutefois, vu la règle de construction de l'arbre binaire, si il existe déjà une description de  $\operatorname{nh}(j)$  liée à la gauche de  $\operatorname{id}(j)_1$ , elle doit être déplacée et liée à la gauche du noeud "a"; ce déplacement peut impliquer indirectement une récursion pour un holon qui appartiendrait aux supra-ordonnés de "a" et également aux subordonnés de  $\operatorname{id}(j)_1$ .

Le troisième cas est similaire au premier mais sans déplacement de la description de nh(j) puisque la nouvelle occurrence a lieu "plus à droite" de id(j)<sub>1</sub>.

Le deuxième cas implique la récursivité de  $\operatorname{nh}(j)$  et éventuellement d'autres holons. En effet, on introduit dans le sous-arbre décrivant  $\operatorname{nh}(j)$  un noeud de même nom. D'autre part, si pour "a", on détermine l'ensemble NG et si on extrait de cet ensemble les éléments "y" tels que  $\operatorname{id}(j)_1 < "y" < "a"$ , on obtient la liste de tous les holons qui sont enclenchés entre deux enclenchements successifs du holon  $\operatorname{nh}(j)$  tels que le deuxième enclenchement a lieu pendant la production de l'effetnet du premier enclenchement; les holons qui correspondent aux valeurs "y" sont eux aussi récursifs. (Les noeuds "y" qui ne correspondent pas à des holons ou à des holons booléens ne sont pas à considérer).

### Exemple :

Supposons qu'au cours de la construction d'un programme-holon, on décrive les holons suivants :

La structure de l'arbre binaire, où l'adjonction de  $H_{m+2}$  est représentée par un arc en pointillé, est donnée par la figure suivante :

<sup>(1)</sup> Cet ordre est différent de celui utilisé dans la section précédente.

Après l'introduction des holons  $H_m$  et  $H_{m+1}$  dans le programme-holon, le dictionnaire II contient, entre autres, les entrées suivantes :

$$A \rightarrow 101$$
  
  $B \rightarrow 10,1010$ 

L'introduction du holon  $H_{m+2}$  entraı̂ne une modification de l'entrée B, à savoir :

 $B \rightarrow 10,10010100,1010.$ 

La comparaison des index  $id(B)_1$  et  $id(B)_2$  permet de conclure que  $id(B)_2$  appartient au sous-arbre à gauche de  $id(B)_1$  et donc que "B" est récursif.

La construction du NG relatif à id(B), donne :

NG(10010100) = {1,10,1001,100101,1001010}

dont le sous-ensemble des "v" est :

"y" =  $\{1001, 100101, 1001010\}$ .

Aucun des "y" n'est noeud de holon ou de holon booléen, il n'y a donc aucune récursion indirectement induite; la présence dans l'ensemble des "y" d'un noeud d'instruction-holon est obligatoire pour garantir la structure légale du programme-holon; par manipulation des index Dewey et des deux dictionnaires, les conditions de légalité du programme-holon sont déduites :

- 1) "Z" doit produire une valeur booléenne "faux",
- 2) "Y" ne peut contenir "B" d'une manière exhaustive.

D'autre part, l'introduction de  $H_{m+2}$  entraîne le déplacement du sous-arbre de "A" depuis "la gauche" de "101" à "la gauche" de "100101001". Ce transfert produit une nouvelle modification de l'entrée "B" du dictionnaire II :

# $B \rightarrow 10,10010100,1001010010.$

Nous savons déjà que "B" est récursif, mais nous devons examiner si cette nouvelle apparition de "B" induit ou non d'autres récursions. Le calcul de NG et de "y" pour cette nouvelle valeur d'index révèle que "A" est indirectement récursif.

Les conditions sur la structure du programme-holon impliquées par la récursivité de "A" sont :

- 1) "Z" doit produire une valeur booléenne "faux",
- 2) "Y" ne peut contenir "A" d'une manière exhaustive.

Ces conditions sont à associer aux conditions introduites par la récursivité de "B", ce qui donne comme conditions communes :

- 1) "Z" doit produire une valeur booléenne "faux",
- 2) "Y" ne peut contenir ni "A" ni "B" d'une manière exhaustive.

Supposons, pour terminer cet exemple, que le holon  $H_{m+2}$  soit décrit d'une autre manière équivalente à la première version :

# H<sub>m+2</sub>[B, begin W, if Z then X; C else Y fi; end]

Si on modifie l'arbre en conséquence, on montre que :

- 1) le holon "X" est récursif,
- 2) la récursivité de "X" entraîne celle de "B";
- 3) les conditions de structure du programme-holon sont :
- a) "Z" doit parvenir à produire une valeur "faux",
- b) "Y" ne peut contenir ni "X" ni "B" d'une manière exhaustive.

Ces conditions ne sont point différentes des conditions obtenues précédemment; en effet, la récursivité de "X" a remplacé celle de "A", mais le résultat est équivalent car "A" est inclus dans "X"; d'autre part, la condition <u>ni</u> "X" <u>ni</u> "B" est équivalente à <u>ni</u> "B" <u>ni</u> "A", car dire <u>ni</u> "B" implique qu'on interdit également tout supra-ordonné direct ou indirect de "B" qui le contiendrait d'une manière exhaustive, or "A" contient "B" d'une manière exhaustive puisque tout enclenchement de "A" enclenche inévitablement le holon "B".

L'exemple précédent montre quelles complications peuvent résulter de l'emploi de processus récursifs, ce qui ne signifie pas qu'il ne faut pas les utiliser, mais bien qu'il est préférable de les incorporer dans un langage dont la syntaxe est susceptible d'aider à étudier la structure du programme. D'autre part, les difficultés de la détection de la récursivité nous portent à croire qu'il faut dans la définition d'un langage soit supposer à priori le caractère récursif éventuel de toutes les procédures (cas de l'Algol), soit envisager une détection automatique de la récursivité, mais qu'en aucun cas, la récursivité ne peut être assimilée à une "option" (cas du PL/I); détecter les procédures indirectement récursives est d'autant plus difficile que la structure du programme est "lâchement" exprimée par la syntaxe.

On peut objecter que dans le cas du HPL la détermination du caractère récursif d'un holon est largement facilitée par l'impossibilité de transmettre un nom de holon comme paramètre d'un holon; les raisons méthodologiques de cette limitation sont exposées dans la section suivante.

## 15. LIMITATION DE LA NATURE DES PARAMETRES

Les paramètres admis dans un nom de holon sont uniquement des noms d'objets, il n'est pas permis de transmettre un nom de holon; par rapport aux possibilités des langages classiques (Algol, PL/I,...), il s'agit d'une limitation. Toutefois, les possibilités de passage d'identificateurs de procédures (ou de fonctions) comme paramètres dans un langage classique nous semblent excessives par rapport aux applications.

En général, cette possibilité est justifiée par la nécessité de programmer des fonctions de fonction : "une procédure ou une fonction F est utilisée comme paramètre d'une procédure ou fonction G, si F doit être calculée durant l'exécution de G, et si F représente différentes procédures ou fonctions pour différents appels de G". L'exemple classique est le calcul de l'intégrale G d'une fonction F, cette intégrale étant calculée par une procédure qui admet F comme paramètre; de même, une procédure de traversée d'un arbre peut avoir comme paramètre une procédure "visiter le noeud"; finalement, dans les algorithmes de tri, on peut laisser en paramètre la fonction booléenne qui fixe la relation d'ordre entre les éléments à trier.

En fait, il existe une différence fondamentale entre une variable d'un type donné (entier, réel, booléen,...) et une procédure utilisée comme paramètre d'une procédure: la possibilité de remplacement du paramètre par le nom d'une variable de même type correspond à une forme "d'abstraction" du programme que nous pouvons concevoir aisément; que les noms des variables changent d'une exécution à l'autre nous importe peu, si nous pouvons comprendre l'effet de l'exécution de la procédure sur une collection de variables dont le type de chacune est bien défini; mais si, d'un appel à l'autre, une partie de la procédure devient modifiable, l'étude de la procédure elle-même échappe au programmeur, car l'effet-net que produira cette procédure ne peut plus être déterminé par l'étude du texte seul de la procédure, il devient fonction d'une inconnue : la procédure utilisée à chaque appel pour remplacer le paramètre formel.

Le programmeur qui utilise des procédures comme paramètres formels dans ses procédures, produit un texte qui n'est plus, à proprement parler, équivalent à un programme, mais bien à une espèce de "schéma" de programme. On peut rejeter cette manière de voir, en avançant que : lorsque la composition du programme est terminée, tous les soi-disant schémas de programme sont effectivement remplacés par de "vrais" segments de programme. Mais, le point important est qu'il est nécessaire d'avoir accompli entièrement la composition du programme pour que l'indétermination que peuvent introduire les procédures-paramètres soit levée; durant toute la phase de composition, le programmeur doit effectivement raisonner sur un "schéma" de programme.

Dans les trois exemples que nous avons présentés, cette indétermination ne semble pas faire de difficultés à première vue; toutefois en deuxième analyse, on conçoit que, pour chaque cas, la procédure (ou la fonction) qui sera substituée au paramètre formel doit appartenir à un ensemble bien déterminé de procédures (ou de fonctions): pour l'intégrale, la fonction doit être "intégrable" sur l'intervalle d'intégration; pour la traversée de l'arbre, la procédure doit correspondre à une opération exécutable sur les éléments de l'arbre sans modifier la structure de l'arbre d'une manière irréversible; pour tout algorithme de tri, on doit avoir une fonction définissant effectivement un ordre entre les éléments à trier. En d'autres termes, on peut dire que les paramètres admis pour les procédures sont des noms d'objets ou des noms de procédures (1), mais dans l'un ou l'autre cas, les substitutions de paramètres doivent respecter le type correspondant au paramètre formel. Dans la phrase précédente, nous admettons explicitement que l'on puisse classer les procédures selon certains "types", de la même manière que l'on classe les variables selon leurs types. Cette classification existe - d'une manière insuffisante pour le cas présent - pour les fonctions que l'on peut classer d'après le type de la valeur produite, mais rien d'équivalent n'existe pour les procédures.

A notre avis, l'origine de cette difficulté vient de la manière de définir l'effet d'une procédure: c'est-à-dire la règle de la "recopie". Cette forme de définition attribue un caractère purement syntaxique à une notion qui dès son introduction n'était point comprise de cette manière (2). En fait, la règle de "recopie" correspond plus à une manière de définir l'effet d'une instruction de modification d'un texte dans un programme d'édition de textes qu'à la notion intuitive que tout programmeur associe à la procédure : une possibilité de créer un répertoire d'instructions qui lui soit propre.

En conclusion, il semble que, si on accepte l'idée de laisser le programmeur créer à l'intérieur de son programme l'équivalent d'un schéma de programme, on doit limiter les possibilités de ces créations aux schémas pratiquement acceptables, c'est-à-dire aux schémas dont les possibilité d'interprétation sont confinées à un ensemble à priori bien délimité. Une manière d'imposer cette forme de limitation serait de permettre uniquement aux "opérations terminales" d'être paramètres formels d'un nom de holon, étant entendu que toute substitution est limitée aux opérations terminales appartenant au même langage terminal que le paramètre formel et produisant une valeur de même type. Par cette limitation à un ensemble bien défini de substitutions possibles, on peut déterminer les propriétés communes aux effets-nets résultant d'une substitution quelconque. Ces limitations des possibilités de substitution ne sont pas éloignées de l'utilisation que le programmeur prudent fait en pratique du mécanisme plus général de l'Algol et du PL/I.

Nous ignorons le cas des "étiquettes" qui est en fait une manière détournée de "passer" l'équivalent d'une procédure.
 En considérant qu'on peut l'interpréter comme équivalente au concept de sous-routine fermée.

# CHAPITRE III

#### LOGIQUE DU DATATEST

#### I. INTRODUCTION

Dans ce chapitre, nous justifions l'introduction dans le langage des éléments syntaxiques "datatest" et "postest". La syntaxe détaillée de ces éléments syntaxiques est décrite complètement; d'autre part, l'influence de ces éléments sur la méthodologie de programmation et sur la conception d'un programme en attribuant une égale importance à son exactitude et à son efficacité est abordée.

#### II. DOMAINE LR ET DATATEST

Soit un texte P écrit en respectant la syntaxe du HPL; ce texte correspond à un "programme" P si, au cours de toute exécution des opérations décrites dans le texte P, tout holon booléen appartenant à un ensemble C de holons booléens produit une valeur booléenne déterminée pour chaque holon; si P est un "programme", l'exécution des opérations décrites par le texte P peut effectivement être assimilée à la traversée d'un arbre orienté fini. Par l'examen de la structure de la syntaxe de P, on détermine - sans effectuer de tentative d'exécution - les holons booléens qui appartiennent à l'ensemble C et la valeur booléenne que chacun d'eux doit produire, (voir chapitre II).

Supposons que nous ayons un segment de texte qui inclut une instruction-holon de répétition :

while p(x) do M(x,y,z) od;

Pour que le texte soit un "programme", il est nécessaire que l'instruction-holon produise effectivement son effet-net pour tout enclenchement lors de l'exécution du programme; pour cela, il faut que le holon booléen p(x) produise effectivement une valeur booléenne "faux", succédant éventuellement à une séquence finie de valeurs "vrai". On peut exprimer cette condition par (1):

 $(\exists x) (p(x) \equiv "faux").$ 

Cette expression est sans intérêt, car elle exprime uniquement le but que l'on désire obtenir et elle n'apporte aucune information sur la possibilité de l'obtenir. La valeur de "x" appartient à une séquence de valeurs "x" qui résultent de la transformation successive d'une valeur initiale " $x_0$ " par le holon "M(x,y,z)" (et éventuellement par "p(x)"). D'où, le résultat qui présente un intérêt est la détermination d'un prédicat  $T(x_0)$  tel que si  $T(x_0)$  est "vrai" alors  $(\exists x)(p(x) \equiv \text{"faux"})$  est "vrai", soit :

$$T(x_0) = (\exists x) (p(x) \equiv "faux")$$

étant entendu que la variable "x" de  $(\exists x)$  doit appartenir à la séquence de valeurs "x" générées à partir de "x<sub>0</sub>" par les holons de l'instruction-holon.

On dira que  $T(x_0)$  définit le domaine LR de " $x_0$ " pour l'instruction-holon en cause; "LR" signifie "le moins restrictif". Dans le prédicat  $T(x_0)$ , la variable " $x_0$ " ne peut être quantifiée que par le quantificateur V; de cette manière, la valeur de  $T(x_0)$  est calculable pour tout enclenchement de l'instruction-holon à laquelle il est associé.

<sup>(1)</sup> Il faut lire (p(x) = "faux")comme suit : la valeur booléenne produite effectivement par p(x) est "faux". Sans oublier que p(x) peut produire également un effet-net sur les objets du programme.

```
Exemple:
```

Soit le texte P :

```
begin
  declare a,b,r,q : integer;
  read(a);
  read(b);
r:=a;
q:=o;
while r ≥ b
     do r:=r-b;
      q:=q+1;
     od;
  print(q);
end
```

Ce texte est un programme si, pour toute tentative d'exécution, l'instruction-holon "while" se termine; pour cela, il faut qu'il existe un couple de valeurs de "r" et de "b" tel que r ≥ b soit "faux"; en caractérisant par un indice "k" l'ordre de génération des éléments des ensembles des valeurs de "b" et de "r" pour un enclenchement quelconque de l'instruction-holon "while", on a la condition :

$$(\exists r_k) (\exists b_k) (r_k < b_k).$$

Par examen de l'instruction-holon, on voit que :

$$(\forall k) (b_k \equiv b_0)$$

où bo représente la valeur de b avant l'enclenchement de l'instruction-holon.

Déterminons  $r_k$  en fonction de  $r_0$ ; on voit que :

$$r_1 \equiv r_0 - b$$
 ,  $r_2 \equiv r_1 - b$   $c - a - d$   $r_2 \equiv r_0 - 2b$ .

Montrons que :

$$(\forall k) (r_k = r_0 - k.b).$$

En effet, par induction ; c'est évidemment "vrai" pour k = o; supposons que ce soit "vrai" pour toute valeur k et montrons que l'expression est "vrai" pour k = k'+1;

 $r_k \equiv r_{k'+1} \equiv r_{k'}-b$ or

r<sub>k'</sub> = r<sub>o</sub>-k'.b,

 $r_k = r_0 - k' \cdot b - b = r_0 - (k'+1) \cdot b$ 

$$r_k = r_0 - kb$$
.

Donc,

D'où

D'où,

$$(\forall k) (r_k = r_0 - k.b)$$
 avec  $k \ge 0$ .

La condition de terminaison de l'instruction-holon devient :

$$r_0$$
-kb < b,

et puisque  $r_0 = a$ , on obtient :

$$a-kb < b$$
 ou encore  $a < (k+1)b$ .

Ce qui signifie qu'avant tout enclenchement de l'instruction-holon "while", il faut que les valeurs de "a" et de "b" soient telles que :

$$\exists k > o(a < (k+1)b) or (k=0 and a < b)$$
 (a)

Les objets "a" et "b" sont du type "entier"; on doit donc déterminer à quels sous-ensembles des entiers "a" et "b" doivent appartenir pour vérifier la condition ( $\alpha$ ). Si on considère les valeurs entières positives ou nulles, on voit qu'il faut que :

$$a \ge 0$$
 and  $b > 0$  ( $\alpha.1$ )

Si on considère les valeurs entières négatives ou nulles, on obtient :

$$(a < o \text{ and } b = o) \text{ or } a < b$$

qui se réduit à :

$$a < o \text{ and } b \le o \text{ and } a < b \quad (\alpha.2)$$

Si on considère les valeurs entières positives pour l'un et négatives pour l'autre, on a :

$$a < o$$
 and  $b \ge o$   $(\alpha.3)$ 

Le prédicat T(a,b) qui définit le domaine LR pour les objets "a" et "b" est donc :

$$T(a,b) \equiv (a \ge 0 \text{ and } b > 0) \text{ or } (a < 0 \text{ and } a < b)$$

On dira que T(a,b) est le *datatest* de l'instruction-holon "while" du programme P. En d'autres termes, le datatest est la condition que doivent respecter les valeurs des objets pour que le texte P corresponde à un programme.

En fait, sous la condition que (a  $\geq$  o <u>and</u> b > o), le programme peut être interprété comme la division de "a" par "b" en n'utilisant que des opérations d'addition ou de soustraction; il est toujours nécessaire de faire une nette distinction entre un programme et l'interprétation qu'on associe à son exécution; le programme de l'exemple, s'il est exécuté sous la condition T(a,b) produira un certain effet-net, à savoir :

1) si (a ≥ o and b > o) est "vrai", alors

$$P = a_{post} = a_{ante} \ \underline{and} \ b_{post} = b_{ante}$$

$$\underline{and} \ a_{post} = q_{post} \times b_{post} + r_{post} \quad (\beta)$$

$$\underline{and} \ o \le r_{post} < b_{ante}$$

En effet, montrons que l'élément ( $\beta$ ) de cette conjonction est "vrai" quel que soit le nombre d'itérations effectuées avant que l'instruction-holon ne produise effectivement son effet-net ( $^1$ ). Si le nombre d'itérations est nul, la relation ( $\beta$ ) est "vrai"; supposons que ( $\beta$ ) soit "vrai" après k itérations, montrons qu'il est "vrai" après la (k+1)-ième exécution :

$$a_k \equiv q_k \times b_k + r_k$$
 (e.1),

il faut montrer que :

$$a_{k+1} \equiv q_{k+1} \times b_{k+1} + r_{k+1}$$
 (e.2);

on peut réécrire (e.1) comme suit :

<sup>(1)</sup> Pour les autres éléments de la conjonction, la démonstration est immédiate par inspection du texte du programme.

$$a_k \equiv (q_k + 1 - 1) \times b_k + r_k$$
  
 $a_k \equiv (q_k + 1)b_k + r_k - b_k$  (e.3),

or, par examen du programme, on voit que :

$$q_k + 1 \equiv q_{k+1}$$
,  $r_k - b_k \equiv r_{k+1}$  et  $a_k \equiv a_{k+1}$ ,

D'où (e.3) est identique à (e.2), et la relation (ß) est vérifiée quel que soit le nombre d'itérations effectuées avant la fin de l'instruction-holon "while".

2) si (a < o and a < b) est "vrai", alors

$$\frac{\text{and }}{\text{bpost}}$$
 =  $\frac{\text{bante}}{\text{ante}}$ 

$$\underline{\text{and}} \ q_{\text{post}} = \text{o} \ \underline{\text{and}} \ a_{\text{post}} = q_{\text{post}} \times b_{\text{post}} + r_{\text{post}}$$

En effet, dans ce cas, il n'y a aucune itération, ce qui implique l'absence de toute modification de la valeur de "q".

L'examen de ces deux possibilités d'effet-net permet de conclure que l'effet-net produit sous la première condition correspond bien aux propriétés du quotient et du reste de la division de deux entiers "a" et "b" tels que (a  $\geq$  o and b > o). Quant à l'effet-net produit sous la deuxième condition, on ne retrouve certainement pas les propriétés d'une opération de division (1).

En conclusion, le prédicat T(a,b) transforme le texte P en "programme", et en choisissant de restreindre le domaine "LR" défini par T(a,b), on obtient un "programme" à l'effet-net duquel nous pouvons associer une interprétation; le prédicat T'(a,b) qui définit cette restriction de "LR" est également dénommé le "datatest" du programme P.

#### Remarque :

Considérer le texte P de l'exemple sans faire initialement aucune référence à la fonction supposée du programme peut sembler artificiel. Il est évident que la composition du texte P résulte de la volonté de créer un programme qui calcule le quotient de deux entiers; toutefois, ce qui nous semble peu naturel, c'est la manière habituelle de présenter ce programme : "calculer le quotient de deux entiers "a" et "b" tels que a  $\geq$  o and b > o". En effet, il est plus que probable que cette spécification soit une conséquence d'une tentative de conception d'un programme pour la division de deux entiers "a" et "b" avec b  $\neq$  0, partant de l'idée d'un programme dont l'effet serait d'obtenir, par exemple, (-5) pour la division de (-10) par 2, vu la propriété de la division:

 $a = b \times q + r$  et |r| < |b|.

Il est absurde de cacher que, fréquemment, la construction du programme apprend au programmeur les véritables spécifications du programme en cours de développement. On s'empresse de cacher ce fait, une fois le programme terminé; toutefois, si l'on désire apporter une contribution aux méthodologies de développement de programmes, on ne peut point ignorer délibérément cet aspect de la réalité. Malheureusement, la littérature "informatique" est pleine d'inhibitions, et s'il a fallu attendre les années soixante pour que l'on discute ouvertement des problèmes de l'exactitude des programmes, il ne faut surtout pas en inférer qu'avant cette époque l'on n'écrivait que des programmes corrects, on refusait plus simplement d'admettre qu'on en composait surtout des erronés. Le problème des spécifications de programme pourra être examiné d'une manière féconde quand on acceptera de l'envisager sous les contraintes de la réalité.

## 2.1. Rôle du datatest.

Par "dégénérescence d'un programme", on entend deux phénomènes distincts : soit l'éventualité d'une itération qui ne converge pas, soit la production d'un effet-net auquel on ne désire associer aucune interprétation. La deuxième partie de cette alternative est très importante, car il est absolument nécessaire que nous puissions interpréter d'une manière identique tout enclenchement d'une opération qui produit un effet-net.

Chaque possibilité de dégénérescence d'un programme doit être protégée par un "datatest" qui peut être évalué avant toute exécution de l'opération susceptible d'engendrer cette dégénérescence.

Le "datatest" est une méthode de "programmation défensive"; nous verrons ultérieurement comment

<sup>(1)</sup> Si a < b < o, le quotient est zéro ! Si b = o, le quotient est aussi zéro ! Quels que soient a et b, le reste est toujours égal au dividende !

cette notion est compatible avec celle d'efficacité d'un programme.

#### 3. LE POSTEST

## 3.1. Efficacité du datatest.

La fonction du datatest est de remplacer le texte du programme P(x) par le texte :

# if $x \in LR$ then P(x) else abort fi.

Le datatest a été introduit pour garantir que P(x) soit effectivement un programme, ce qui explique que "x«LR" doit pouvoir être évalué et produire effectivement une valeur booléenne. Cette restriction sur la forme du datatest ne permet pas, par exemple, la promotion au rang de "programme" du texte:

f(x,y) = if 2 est le plus grand entier tel qu'il existe une solution à l'équation  $x^n + y^n = z^n$  en valeurs entières de x,y et z

then print(x,y,z) else f(x+1,y+1) fi

Toutefois, même en écartant des cas aussi extrêmes que cette intervention du théorème de Fermat, on doit constater que l'on ne pourra généralement concevoir des datatests aussi simples que celui du programme de division. En effet, dans de nombreux cas, le datatest vérifiera l'existence d'une caractéristique "structurelle" de l'environnement d'une opération P dont la vérification correspond pratiquement à l'exécution de l'opération P.

On définit *l'efficacité du datatest* comme le rapport du temps d'exécution de l'opération P qu'il protège au temps d'exécution du datatest suivi de l'opération P pour un environnement qui n'entraîne aucune dégénérescence de P. Comme le temps d'exécution de P peut varier considérablement en fonction de l'environnement choisi,il est nécessaire de baser la comparaison sur un environnement représentatif du "mode" (1) des environnements de l'opération.

Pratiquement, on peut évaluer l'efficacité du datatest en comparant le nombre d'accès à des objets de même type pour les deux programmes.

#### Exemple :

On a vu que le programme d'élimination des espacements redondants d'une bande (II.10.2) exige que la bande ne se termine pas par une séquence d'espacements; comme Weinberg [Wei74] l'a fait remarquer, cette condition ne sera que très rarement vérifiée en pratique, car il est très plausible que le transfert d'un dernier "buffer" incomplet ou que les opérations de clôture du fichier engendrent cette séquence d'espacements indésirables. On peut considérer que cette condition est le datatest du programme; mais il est peu probable que nous puissions le programmer d'une manière efficace. En effet, si le genre de matériel à notre disposition ne nous permet pas de repérer "immédiatement" la fin de la bande, nous serons amenés à programmer le datatest comme suit :

```
begin
  declare T : character;
T:=ni1;
while non eof(input) do T:=read(input) od;
if T z ' ' and T z ni1
  then rewind(input); yalue is 'true'
  else value is 'false'
fi
end
```

L'effet-net de cette opération est bien la production effective d'une valeur booléenne "yrai" ou "faux" selon que la condition du datatest est vérifiée ou non; mais, ce datatest ne peut convenir

<sup>(1) &</sup>quot;Mode" est pris dans le sens statistique du terme, c'est-à-dire le cas le plus fréquent d'une distribution.

car lors de son évaluation, on effectue exactement les mêmes opérations d'accès aux objets (les caractères de la bande) que le programme à protéger; le datatest protège effectivement le programme mais avec une efficacité proche de un demi.

# 3.2. Hypothèses sur les données d'un programme.

Le problème des hypothèses admissibles pour les données d'un programme est à rapprocher de l'exemple de la section précédente; le fait que la programmation soit une activité de nature mathématique n'implique pas que l'on puisse y utiliser la notion d'hypothèse de la même manière qu'en mathématiques; en effet, si l'on se limite à manipuler des entiers, des réels ou des tableaux de ceux-ci dans un langage de type Algol, les hypothèses sur le type et la valeur initiale d'une donnée ne présentent aucune difficulté car les types et les valeurs des objets sont fixés par "interprétation": la même séquence de "bits" devient "entier" ou "réel" selon la volonté du programmeur; mais dès que l'on modifie des objets qui doivent posséder une certaine "structure", le problème est tout différent car, seuls les éléments terminaux de la "structure" peuvent bénéficier de l'interprétation. Ce problème est bien mis en évidence par la comparaison d'un algorithme "arithmétique" et sa version programmée à un algorithme de la théorie des graphes et son programme correspondant: la version programmée de l'algorithme "arithmétique" est généralement un décalque fidèle de la description de l'algorithme, ce qui n'est généralement point vrai pour l'algorithme des graphes, toute la différence vient de l'impossibilité de transcrire en programme "soit un graphe connexe" aussi aisément que "soit un nombre réel".

On peut difficilement admettre qu'un programme fasse des suppositions sur la nature "structurelle" de ses données sans vérifier l'exactitude de ces suppositions (1). Par exemple, supposons que l'on se propose de composer un programme qui détermine le chemin le plus long entre deux noeuds terminaux d'un arbre; si on examine les algorithmes généralement avancés pour résoudre ce problème, on verra que la seule condition à la parfaite exécution du programme est que la description de l'arbre corresponde, effectivement à un arbre! Cette condition paraît triviale et il semble à première vue que l'on soit autorisé à produire un programme de ce genre, mais, en pratique, c'est moins sûr : en effet, la partie du programme destinée à vérifier la correspondance entre la description des données et un arbre risque de reprendre l'ensemble des opérations de l'algorithme.

Dans de tels cas, où un datatest se révèle "inefficace", le programme peut être reconstruit à partir de la notion de postest, c'est-à-dire de vérification a posteriori. On peut considérer que l'on inclut tout le programme dans le datatest et que l'évaluation du datatest coîncide avec l'exécution du programme dont les résultats ne sont acceptés qu'à condition de vérification ultérieure du datatest. Quand on procède de cette manière, la structure du datatest n'est généralement pas conservée, mais une espèce de "fractionnement" de celui-ci a lieu; ce "fractionnement" implique la création de l'un ou l'autre "postest".

# 3.3. Exemple de fractionnement d'un datatest.

Reprenons le problème de la détermination du plus long chemin entre deux noeuds terminaux d'un arbre que nous nous proposons de résoudre par application de l'algorithme suivant :

#### Phase A :

- 1) Prendre un noeud quelconque  $(n_{\hat{1}})$  de l'arbre T et considérer que ce noeud est la racine de T :  $R(T) \equiv n_{\hat{1}}$ .
- 2) Chercher dans l'arbre T(R), de racine égale à  $n_i$ , le plus long chemin entre R et un noeud terminal de T(R); soit  $n_i$  le noeud qui se trouve à l'extrêmité de ce chemin.

#### Phase B:

- 1) Prendre l'arbre T(R) avec R = n.
- 2) Chercher dans T(R) le plus long chemin entre R et un noeud terminal de T(R).

<sup>(1)</sup> Il en est de même pour des programmes traitant des applications en "temps-réel", l'exactitude de ce programmes ne peut reposer sur la supposition que certains "capteurs" enverront un certain type de signal sans prévoir qu'à la suite d'une défaillance le capteur n'expédie aucun signal.

3) Soit  $n_z$  le noeud qui se trouve à l'extrémité de ce chemin; le chemin  $(n_t, n_z)$  est le chemin cherché (1).

Quelle que soit la méthode de représentation des données, il peut advenir que celles-ci correspondent :

- a) à un arbre; dans ce cas, pas de problème car l'algorithme déterminera effectivement le plus long chemin;
- b) à une forêt d'arbres (cause : un ou des arcs manquants); le résultat de l'algorithme est fonction du noeud n; choisi;
- c) à un graphe (cause : un ou des arcs en trop); dans ce cas, il est possible que l'algorithme utilisé en A.2 et en B.2 ne se termine pas;
- d) à une combinaison des cas "b" et "c".

Il est donc nécessaire de déterminer si les données correspondent effectivement à un arbre, c'està-dire à un graphe connexe sans cycle. Un graphe est connexe s'il existe toujours un chemin entre deux noeuds quelconques du graphe. Un cycle est un chemin simple (²) passant par trois noeuds ou plus et menant d'un noeud à lui-même. Pour les cas utiles d'applications de l'algorithme (arbre d'une centaine ou plus de noeuds), la vérification des données est une opération très "coûteuse". En pratique, on peut concevoir l'algorithme en supposant que la structure des données correspond toujours à celle d'un arbre et en analysant les effets-nets que les opérations qui composent l'algorithme sont susceptibles de produire si les données ne sont pas celles d'un arbre; l'algorithme est complété par des "postests" qui détecteront toute production d'un effet-net inacceptable. Cette opération revient à remplacer un datatest inefficace par une combinaison de "postests" et de "datatests".

Les opérations A.2 et B.2 de l'algorithme peuvent être décrites par une forme de traversée de l'arbre avec enregistrement du meilleur chemin trouvé.

Supposons que l'opération A.2, conçue pour un arbre, soit appliquée à une forêt; l'effet-net consistera en la production du chemin le plus long d'un arbre de la forêt. En comparant, après l'exécution de l'opération A.2, l'ensemble des noeuds visités à l'ensemble des noeuds donnés, on peut déterminer si l'on a eu affaire à un arbre ou à une forêt, selon que le premier ensemble coîncide au deuxième ou non.

Supposons que l'on exécute l'opération A.2 sur un graphe; puisque cette opération consiste en la traversée d'un arbre, elle doit posséder une sous-opération destinée à passer d'un noeud à l'autre lors de la "descente" dans l'arbre; si on note au fur et à mesure les noeuds rencontrés, la sous-opération devra inévitablement retrouver un noeud déjà rencontré si ce dernier appartient à un cycle, un datatest associé à cette sous-opération peut donc détecter l'existence d'un cycle.

En conclusion, au lieu de vérifier par un datatest que les données forment un arbre, on suppose qu'il en est ainsi, et ou bien le postest de l'opération A.2 révélera que les données correspondent à une forêt, ou bien le datatest de la sous-opération de A.2 interrompra le programme si l'on opère sur un graphe, ou au contraire, si aucune interruption n'a lieu, on pourra être assuré que l'on a appliqué effectivement l'algorithme à un arbre.

Quant à l'opération B.2, elle est identique à l'opération A.2, mais si cette dernière est protégée par le datatest et le postest, nous sommes assurés que B.2 ne sera jamais enclenchée sur des données qui ne soient pas un arbre; l'opération B.2 sera donc équivalente à A.2 sans les protections. Cette observation justifie la forme syntaxique qui sera donnée au datatest et au postest.

Comme dernier exemple, nous reprendrons le problème d'élimination des espacements redondants sur une bande. On peut simplifier le problème en envisageant le cas d'une bande dont la structure est représentée par :

[<séquence de caractères>[{<blancs><séquence de caractères>}...]]

<sup>(1)</sup> Pour une preuve de cet algorithme, voir le chapitre VIII, consacré aux exemples.

<sup>(2)</sup> Si  $(v_0, v_1, \dots, v_n)$  est un chemin, il est simple si  $v_0, v_1, \dots, v_{n-1}$  sont distincts et si  $v_1, \dots, v_{n-1}, v_n$  sont distincts.

```
où <séquence de caractères>::={<caractère ≠ blanc>}...

<br/>
<br
```

On doit modifier la structure de la bande afin qu'elle devienne (si elle est différente d'une bande vide) :

<séquence de caractères>[{<un caractère blanc><séquence de caractères>}...]

Les holons suivant réalisent cet effet-net (les déclarations sont omises) :

```
holon changer la structure
 begin
  blancsequence:='false';
  while non eof(input)do
    "x":=read(input);
    traitement correspondant à la valeur de "x";
  od
 end
noloh
holon traitement correspondant à la valeur de "x"
 begin
  case "x" ≠ 'blanc'
     then clore une séquence de blancs s'il y a lieu;
          write("x");
     else blancsequence:='true'
  esac
noloh
holon clore une séquence de blancs s'il y a lieu
  begin
  if blancsequence = 'true'
 then blancsequence:='false';
           write('blanc');
fi
 end
noloh
```

Pour traiter le cas général, il faut faire précéder le holon "changer la structure" par un "datatest" qui vérifie l'absence d'une séquence de "blancs" en tête de la bande et qui imprime un "blanc" dans le cas contraire; d'autre part, le holon doit être suivi par un "postest" qui vérifie que "blancsequence" est 'faux' et qui imprime un blanc s'il n'en est point ainsi. La version "programmée" de ce datatest et de ce postest n'est pas donnée car nous n'avons pas encore décrit la syntaxe de ces deux éléments du langage.

## 4. SYNTAXE DU DATATEST ET DU POSTEST

4.1. Justification d'une forme syntaxique particulière.

Vu que la fonction du datatest peut être représentée par l'instruction :

if datatest then programme P else abort fi,

on pourrait en déduire que l'instruction conditionnelle et l'opération <u>abort</u> suffisent pour programmer les datatests et qu'il n'est pas nécessaire d'introduire une forme syntaxique particulière. En

fait, cette correspondance au point de vue modèle d'exécution n'est pas une raison suffisante pour refuser une syntaxe propre au datatest.

Le principe du HPL est : "toute opération ne peut être enclenchée que si les conditions préalables à une exécution correcte sont réunies". Ce qui ne signifie pas que les conditions sont toujours évaluées avant toute tentative d'enclenchement d'une opération même si l'on peut prouver qu'elles sont certainement vérifiées.

Pour mettre en application ce principe, il faut que notre syntaxe nous permette d'une part, de représenter d'une manière distincte du reste du programme les conditions préalables à toute exécution, et d'autre part, de signaler qu'à certains endroits précis du programme la totalité ou une partie des conditions d'une opération est inhibée.

On peut assimiler le datatest à une forme de "runtime-check", notion importante en programmation et qui n'a jamais reçu l'attention qu'elle méritait. L'exemple le plus fréquent de "runtime-check" est la vérification de la valeur des indices d'un élément de tableau en la comparant aux valeurs des bornes : généralement le choix de la vérification est assimilée à une "option" du code généré par le compilateur; pour justifier le procédé, on suggère d'utiliser l'option de vérification "pendant la phase de mise au point du programme" et de la supprimer pour la phase d'exploitation, vu la "coût" de l'opération de vérification (¹). Pourtant, il est primordial que l'opération d'extraction d'un élément de tableau soit toujours définie et, en conséquence, il semble que la vérification fasse nécessairement partie de l'opération. D'autre part, pour de nombreuses parties d'un programme, le programmeur est souvent à même de justifier la suppression de la vérification, ou plus exactement que si elle est vérifiée pour une opération donnée, alors elle est certainement vérifiée pour toute une série d'opérations (cas des boucles). Ce qui n'est qu'une variante du raisonnement appliqué aux opérations A.2 et B.2 de l'exemple du plus long chemin dans un arbre : si A.2 vérifie tous ses tests, B.2 peut être exécutée sans aucune vérification.

On oppose habituellement "vérifications à l'exécution" et "vérifications à la compilation", en donnant la préférence à ces dernières; cette préférence ne tient pas compte de la réalité : car, mis à part, la vérification des caractères généraux des objets (leurs types, par exemple), peu de vérifications peuvent être faites à la compilation. Par contre, il est exact qu'au fur et à mesure que le programmeur accroît sa compréhension du programme en cours de développement, il peut remplacer des vérifications "locales" par une seule vérification "globale".

Les vérifications présentent un problème particulier à la programmation parce que, contrairement aux autres sciences appliquées, toute vérification entraîne une dégradation des performances d'un équipement due aux ressources (²) qu'elle mobilise. Au contraire de l'introduction d'un fusible ou d'une soupape de sécurité dans un système physique, le coût d'une vérification n'est pas "marginal" pour le déroulement d'un processus de calcul. L'amélioration de l'efficacité d'un programme réside dans le transfert correct des vérifications globales; l'effort consenti par le programmeur pour parvenir à ce but doit apparaître concrètement dans le texte du programme.

Par l'utilisation d'une syntaxe spéciale, l'introduction ou la suppression d'un datatest apporte à l'utilisateur d'un programme un surplus d'information; en effet, la structure d'un programme HPL permet de déterminer automatiquement les points cruciaux du programme qui devraient être protégés par un datatest, et la définition des opérations terminales implique qu'elles soient également protégées par l'équivalent d'un datatest. Le programmeur devrait fournir une justification pour toute absence d'un datatest en un des points cruciaux ou en cas de suppression d'une vérification appartenant à une opération terminale; de cette manière, la fonction de l'équivalent HPL des "commentaires" est de documenter sur le raisonnement suivi par le programmeur pour outrepasser les opérations de vérification.

# 4.2. Description de la syntaxe du Datatest.

La syntaxe du datatest impose qu'il soit mis sous la forme d'une conjonction de conditions :

<sup>(1)</sup> Le "PL/I manual", par exemple, précise qu'elle est supprimée "par défaut" vu le "substantial overhead".

(2) Le temps d'exécution est une ressource.

Le nombre "n" de conditions n'est pas limité; la séparation du datatest en "n" composants est une décision propre au programmeur : chaque  $c_i$  peut être également une conjonction de conditions; la séparation en composants dépend de la précision de l'information que le programmeur désire obtenir en cas d'échec du datatest. Il est évident que le datatest n'est point vérifié dès que l'un des  $c_i$  a la valeur "faux"; toutefois, lors de l'évaluation du datatest tous les  $c_i$  sont enclenchés séquentiellement ( $^1$ ) et le message d'erreur transmis au programmeur en cas d'échec du datatest reprend la liste de tous les  $c_i$  qui ont produit une valeur "faux".

La syntaxe choisie est :

#### <datatest>::=

# datatest{<composant de datatest>}...end

où chaque "composant de datatest" correspond à un c;.

Pour la syntaxe de "composant de datatest", il existe deux possibilités :

<composant de datatest>::=

<nom de holon booléen>;

<nom de holon booléen>else{<nom de holon>;}...[<indicateur>]fi[;]

od <indicateur>::=resultis{false|true}[;]

# 4.3. Evaluation d'un datatest.

Pour cette description du fonctionnement de l'évaluation d'un datatest, celui-ci sera représenté par deux ensembles :

1) l'ensemble des composants de datatest :

$$C = \{c_1, c_2, \dots, c_n\}$$

où les composants sont ordonnés et distingués par un numéro d'ordre;

2) l'ensemble des valeurs des tests dont les éléments sont à associer aux éléments de C selon leurs numéros d'ordre :

$$V = \{v_1, v_2, ..., v_n\};$$

les éléments de V possèdent l'une des trois valeurs suivantes : "nil", "vrai", "faux"; avant toute évaluation d'un datatest, la valeur de chaque  $v_i$  est "nil".

Dans le cas le plus simple où chaque composant du datatest est un nom de holon, l'évaluation du datatest se fait comme suit :

begin

i:=o;

repeat

i:=i+1:

enclencher c; et v; :=valeur produite par c;;

until i=n;

end

La décision qui résulte de l'évaluation du datatest correspond à l'exécution du programme suivant :

<sup>(1)</sup> La nature séquentielle de cette série d'enclenchements impose que le programmeur tienne compte d'éventuels "effets de bord".

```
begin
  declare D : boolean;
D:='true'; i:=o;
  repeat
  i:=i+1;
D:=D and v<sub>i</sub>;
  ajouter nom du holon c<sub>i</sub> au fichier des messages d'erreur si v<sub>i</sub> est ('faux');
  until i=n;
  if D='false' then abort fi
end
```

L'opération <u>abort</u> implique, outre la terminaison du programme, la transmission du fichier des erreurs au programmeur; le message d'erreur pour un datatest non-vérifié comporte la position du datatest dans le programme et les valeurs de tous les paramètres des noms de holons booléens qui composent le datatest.

Si un composant  $c_i$  du datatest est de la deuxième forme, la plus élaborée, l'attribution de la valeur  $v_i$  résulte de l'algorithme suivant :

(on suppose que le composant c; est de forme :

test else opération;[indicateur]fi )

```
begin
```

enclencher (test) qui produit une valeur booléenne t;  $\frac{\underline{if}}{\underline{t}} \ \underline{t} = 'vrai' \ \underline{then} \ v_i := 'vrai' \ \underline{else} \ corriger \ valeur \ et \ l'attribuer \ a \ v_i \ \underline{fi} \ \underline{end}$  end

où "corriger valeur et l'attribuer à v;" est calculé par :

### begin

## 4.4. Description de la syntaxe du Postest.

La syntaxe est identique à celle du datatest, à l'exception du premier symbole :

### <postest>::=

postest{<composant de postest>}...end

La syntaxe d'un "composant de postest" est entièrement identique à celle d'un "composant de datatest"; il en est de même pour l'évaluation du "postest".

## 5. SYNTAXE COMPLETE D'UN HOLON

Tous les éléments syntaxiques d'un holon, à l'exception des déclarations d'objets qui sont examinées au chapitre IV, ont été décrits. La structure syntaxique d'un holon est donnée par :

```
<holon>::=
  holon(<nom de holon>|<nom de holon booléen>}
  [<text>]
  [<datatest>]
  [<corps de holon>]
  [<postest>]
  [<text>]
  noloh

<text>::=
  text<toute séquence de symboles à l'exclusion de "end">
end
```

Les <text> sont destinés à grouper les commentaires du programmeur sur le holon, et en particulier la justification de l'introduction ou de l'omission d'un datatest (ou d'un postest).

#### 6. MODIFICATIONS D'UN DATATEST (OU D'UN POSTEST)

où

Pour améliorer l'efficacité d'un programme, on doit pouvoir indiquer des inhibitions, des remplacements ou des insertions de datatests ou de postests. Pour cela, il est nécessaire que l'on puisse indiquer syntaxiquement dans quel holon on désire effectuer la modification et quels sont les composants de datatest impliqués par cette modification. La syntaxe définit un "modificateur de datatest (ou de postest)"; cette entité syntaxique peut suivre tout nom de holon (ou de holon booléen) utilisé légalement dans un programme-holon.

Les modifications sont groupées selon deux catégories : celles qui concernent les datatests et celles qui concernent les postests. Chaque catégorie est divisée en trois sous-catégories : inhibitions, insertions et remplacements.

Le choix des délimiteurs n'est pas primordial, tout autre couple de délimiteurs qui n'engendre aucun conflit avec les autres éléments syntaxiques peut convenir.

Dans l'une ou l'autre de ces deux dernières descriptions syntaxiques, il n'est pas nécessaire que les trois groupes soient présents ou que l'ordre soit respecté.

#### 6.1. Groupe des inhibitions.

<datatest modifications>::=

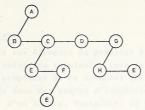
Le groupe des "inhibitions" consiste en une séquence d'ordres d'inhibitions; un ordre d'inhibition indique un ensemble de composants de datatest appartenant à des holons bien spécifiés; ces

holons sont désignés par une chaîne d'appartenance dont le dernier élément (non cité) est le holon auquel le "modificateur de test" est associé. Pour que l'ordre d'inhibition soit légal, il faut que les composants cités existent effectivement dans le holon indiqué et que la chaîne d'appartenance existe dans le holon auquel le "modificateur de test" est associé.

<inhibition groupe>::=
 inhibit<ordre d'inhibition>{next<ordre d'inhibition>}...endinhibit

<ordre d'inhibition>::=
<ensemble de composants de datatest>
 [of<nom de holon>[{in<nom de holon>}...]]

Le nom de holon qui suit le " $\underline{of}$ " est celui du holon "x" dont la description comprend un datatest contenant parmi ses propres composants l'ensemble des composants cités avant le " $\underline{of}$ ". La chaîne de noms de holon reliés par les " $\underline{in}$ " précise quelles sont les occurrences de ce holon "x" qui sont touchées par cet ordre d'inhibition. La chaîne d'appartenance permet une inhibition sélective des composants de datatest; par exemple, dans le cas du programme suivant :



Si le nom du holon "A" est accompagné de l'ordre d'inhibition : <u>inhibit</u><ensemble de composants de datatests> <u>of</u> E <u>in</u> C <u>endinhibit</u>, les composants cités du datatest de "E" ne seront inhibés que pour les deux occurrences appartenant à "C", le holon "E" appartenant à "G" ne subira aucune modification.

L'ensemble des composants de datatest est représenté par une succession de noms de holon booléen reprenant le holon booléen qui débute tout composant de datatest simple ou élaboré. Toutefois, comme les composants d'un datatest sont en nombre fini, il est fréquemment plus facile de désigner ceux que l'on désire inhiber en citant l'ensemble de ceux qui ne le seront pas, d'où l'utilisation éventuelle du préfixe "all but" (tous sauf).

<ensemble de composants de datatest>::=
 [all but] < nom de holon booléen>{;<nom de holon booléen>}...[;]

# 6.2. Groupe des insertions.

Le groupe des "insertions" suit une syntaxe semblable à celle des inhibitions :

<insertion groupe>::=

insert<ordre d'insertion>{next<ordre d'insertion>}...endinsert

<ordre d'insertion>::=

<ensemble de composants de datatest - complète description>
[into<nom de holon>[{in<nom de holon>}...]]

<ensemble de composants de datatest - complète description>::=
{<composant de datatest>}...

La seule différence consiste en l'évidente obligation de spécifier complètement le composant inséré. Les nouveaux composants sont insérés à la suite des composants déjà existants (s'il en existe, car "insert" peut être utilisé pour introduire un datatest dans un holon qui en est dépourvu);

l'ordre d'insertion est important car les composants de datatest sont évalués séquentiellement; toutefois, il est recommandé de ne pas abuser des possibilités d'effet-de-bord dans les datatests.

## 6.3. Groupe des remplacements.

Chaque opération du groupe des "remplacements" consiste en une opération d'inhibition suivie immédiatement par une opération d'insertion. Un ordre de "remplacement" comprend la description de deux ensembles de composants et la chaîne d'appartenance des holons dont le datatest est modifié.

<remplacement groupe>::=
 replace<ordre de remplacement>{next<ordre de remplacement>}...
endreplace

<ordre de remplacement>::=
 <ensemble de composants de datatest>
 [of<nom de holon>[{in<nom de holon>}...]]
 by<ensemble de composants de datatest - complète description>

## 6.4. Cas des opérations terminales.

La syntaxe externe du HPL a accès aux éléments du prédicat A d'une opération terminale. En fait, ce prédicat A est le datatest de l'opération terminale. Lors de la définition d'une opération terminale, on adjoint un "nom" à chaque composant de ce datatest qui n'est pas vérifiable à la compilation, ces noms peuvent être utilisés dans des ordres d'inhibition. Par exemple, on peut inhiber localement la vérification de la valeur d'un indice. Toutefois, le programmeur ne peut ni remplacer ni insérer un nouveau composant dans un prédicat A d'opération terminale, car la méthode de génération de code pour cette opération échappe à son contrôle. Il peut cependant introduire un nouveau test par la définition d'un holon composé d'un datatest contenant le nouveau test et d'un corps de holon formé par une occurrence de l'opération terminale dont on désire modifier le prédicat A.

## 6.5. Influence des modifications sur la structure d'un programme.

Les possibilités de modifications d'un datatest (ou d'un postest) réintroduisent le problème des "procédures utilisées comme paramètres" déjà examiné précédemment; en effet, les opérations "insert" et "replace" peuvent être interprétées comme une forme de passage de "nom de holon". Toutefois, il faut remarquer que cette possibilité est limitée aux datatests et aux postests ; les modifications correspondent soit à un renforcement des conditions d'exécution, soit à un relâchement de celles-ci et n'impliquent donc nullement que le programmeur manipule un schéma de programme.

Vu que les composants de datatest sont des holons booléens et que ceux-ci peuvent être de structure très complexe, ils pourraient impliquer des récursivités de holons appartenant aux supra-ordonnés directs ou indirects du holon auquel le datatest appartient, une telle structure n'est point admise. De même, pour les composants élaborés, les holons qui suivent le <u>else</u> ne peuvent entraîner de récursivité; par exemple, le holon suivant est malformé :

holon A

datatest Q; V else A fi; W else B fi end
begin B;C;D;end
noloh

à cause de la présence de "V else A" qui entraîne une récursivité pour "A".

Cette restriction sur l'induction de récursivités est imposée afin de limiter le datatest à la définition de l'environnement d'une opération.

#### 7. INFLUENCE DE LA METHODOLOGIE DE PROGRAMMATION

On divise schématiquement les méthodologies de développement de programmes en "top-down" et "bottom-up"; en fait, la réalisation d'un programme implique toujours une combinaison des deux, aucune réalisation n'est purement "top-down" ou purement "bottom-up" : en HPL, par exemple, la description d'un holon terminal correspond à un groupement "bottom-up" d'opérations pré-définies, la description d'un holon défini correspond à une construction "top-down".

Le choix de l'une ou l'autre méthodologie influence la composition du datatest d'un holon : en effet, si l'on procède selon un processus "top-down" lorsqu'on décide de créer une opération "primitive", cette création a lieu dans un contexte bien défini; si l'on détermine le "datatest" exigé par la "primitive" à créer, on peut immédiatement comparer les conditions exigées par ce datatest et les conditions vérifiables dans le contexte de création de l'opération, il est évident que le contexte doit toujours vérifier les conditions du datatest, et si l'on peut être assuré de ce fait, il est inutile de composer un datatest à évaluer à l'exécution. Toutefois, cette situation représente une vue simplifiée de la question, car l'opération "primitive" dont la création a été envisagée dans un certain contexte, peut se révéler utile dans un autre contexte du même programme, et pour ce dernier contexte il peut sembler nécessaire d'y implanter le datatest.

A première vue, il semble que le programmeur qui maîtrise complètement le programme qu'il développe, possède une connaissance complète de tous les contextes d'enclenchement de chaque opération du programme; dans ce cas, tout datatest est superflu. Mais, ce cas est "idéal", car il implique une confiance absolue du programmeur dans ses possibilités de démontrer les propriétés de son programme, quelle que soit la complexité de celui-ci. Dans la méthode du datatest, le programmeur peut analyser pour chaque holon, la composition théorique de chaque datatest; cette analyse n'implique qu'une étude <u>locale</u> du programme; par contre, déterminer si chaque apparition de la même opération a lieu dans un contexte qui confirme le datatest exige une étude <u>globale</u> du programme que le programmeur n'est peut-être pas à même de mener (soit à cause de la complexité de la tâche, soit par manque de temps ...); dans ce cas, il garantit la qualité du programme par l'inclusion du datatest.

Pour les mêmes raisons, il est parfaitement normal que les opérations terminales soient complètement protégées puisque l'on ignore a priori les différents contextes qu'elles rencontreront. Le même problème existe pour toute opération conçue dans une optique d'utilisation "bottom-up". D'autre part, pour des questions d'économie d'exécution, il apparaît que le programmeur doit étudier soigneusement les propriétés du programme; en effet, puisque les opérations terminales (¹) sont complètement protégées et qu'il existe une possibilité d'inhiber les tests de protection, le programmeur a intérêt, pour accroître les "performances" du programme, à déterminer dans chaque contexte quelles sont les opérations de vérification qui sont redondantes parce que confirmées par le contexte. Cependant, chaque décision d'inhibition d'une vérification doit être ostensiblement indiquée dans le texte du programme, et le programmeur doit être à même de la justifier.

### 8. INTERRUPTION DU PROGRAMME DANS UNE INSTRUCTION-HOLON "CASE"

En plus du datatest ou du postest, le programmeur a la possibilité d'interrompre un programme dans une instruction-holon "case". Le principe de cette interruption est le suivant : une instruction-holon "case" groupe des couples de condition-opération et une opération qui est liée à une condition déduite des conditions des couples; si tous les cas possibles sont représentés par l'ensemble des conditions des couples, l'opération dépendante ne sera, en principe, jamais enclenchée; toutefois, si au cours d'une exécution du programme, aucune des conditions des couples n'est vérifiée, il y a une erreur soit dans notre estimation des propriétés du programme, soit dans l'exécution de celui-ci; dans ce cas, il est normal de prévoir que l'opération dépendante soit l'arrêt du programme.

La syntaxe choisie pour exprimer cette opération est :

 $<sup>(^1)</sup>$  Et les opérations conçues selon une optique "bottom-up".

<abort dans un case>::=
 abort[(<séquence de caractères>)]

possibilité de choisir la forme des résultats d'une opération (2-16 ou A.3-1x), on pour le laisser dans l'ignorance totale de la forme utilisée pour celculer ces résultats.

La séquence de caractères est utilisée pour transmettre un message au fichier des erreurs; celui-ci et la position de l'instruction-holon "case" sont transmis au programmeur. L'ordre <u>abort</u> ne peut apparaître qu'entre le <u>else</u> et le <u>esac</u> d'une instruction-holon "case" et il doit être le seul composant de cet ensemble.

#### CHAPITRE IV

#### ENVIRONNEMENT ET OBJETS.

Dans les exposés des chapitres précédents, la notion d'environnement d'un holon ou d'une instruction-holon a été fréquemment utilisée sans plus de précision sur la signification exacte de cette notion. On sait toutefois que cet environnement se compose d'objets à la valeur desquels les holons peuvent avoir accès et qu'ils peuvent modifier. Dans ce chapitre, la nature de ces objets sera entièrement décrite, ainsi que les possibilités offertes au programmeur pour "créer" l'environnement de son programme.

## 1. TYPES, OBJETS ET OBJETS ABSTRAITS

#### 1.1. Types et Objets.

La notion de "type" est utilisée dans la définition des langages terminaux; il est spécifié que les opérations de ces langages sont applicables uniquement à des objets d'un "type" bien déterminé et précisé dans les pré-conditions "ante" de chaque opération. Associer un "type" à un objet correspond à lui attribuer un domaine d'interprétation sans préciser la structure de l'objet; le programmeur ignore les particularités de cette structure et il ne dispose d'aucun moyen pour accéder à ses sous-composants.

#### Exemples :

- 1) Les opérations applicables à un objet de type "entier" sont définies en fonction de la notion de l'"entier"; on ne peut définir une opération qui n'est pas applicable à la notion abstraite "entier", comme "imprimer le premier chiffre d'un objet entier", opération qui impliquerait la connaissance de la base utilisée.
- 2) Le type "complex" est un excellent exemple de l'ignorance que peut avoir le programmeur de la sousstructure d'un "type"; en effet, toutes les opérations sur les nombres complexes peuvent être définies indépendamment de la méthode de représentation utilisée; même si on laisse au programmeur la possibilité de choisir la forme des résultats d'une opération (a+ib ou A.e<sup>ix</sup>), on peut le laisser dans l'ignorance totale de la forme utilisée pour calculer ces résultats.

## 1.2. Création d'un type.

La "création" d'un type correspond à la définition d'un langage terminal, c'est-à-dire une collection d'opérations aux conditions d'enclenchement et aux résultats bien définis. En HPL, on considère qu'à toute tâche de programmation correspond un ensemble de langages terminaux nécessaires pour aborder efficacement la conception d'un programme destiné à accomplir cette tâche. Toutefois, la tâche envisagée peut être "marginale", c'est-à-dire qu'elle ne justifie pas l'effort de création des langages terminaux nécessaires pour la programmer; dans ce cas, le programmeur peut suppléer aux langages défaillants, c'est-à-dire aux "types" inexistants, par la création d'objets, composés d'éléments de "types" existants; ces objets lui permettent de "simuler" les "types" dont il a besoin; nous appellerons ces objets des "structures de données". On en déduit qu'il est nécessaire que le langage offre la possibilité de créer ces objets selon les besoins du programmeur. Ces objets, créés par le programmeur, ne sont pas des "types", car les opérations sur ces objets sont toujours définies par des manipulations de leurs composants.

### Exemple :

On peut concevoir le type "matrix" et un langage terminal d'opérations matricielles; d'autre part, on peut imaginer que le programmeur qui ne dispose point de ce langage puisse le simuler en utilisant la notion de "array" - qui est une possibilité offerte par la syntaxe externe - pour agréger des objets de même "type".

#### Remarque :

On considère généralement que le "niveau" d'un langage par rapport à un autre peut être déterminé par les différences de "types" accessibles, mais cette opinion n'est pas tout-à-fait exacte : en effet, qu'un langage soit de plus ou moins "haut niveau" ne peut être déterminé que par le programmeur en fonction de son domaine d'application. Pour un programmeur qui manipule des "spare matrix", un langage possédant le type "matrix" peut être de même niveau que celui qui ne le possède pas.

La relativité de l'intérêt d'un "type" en fonction des applications est un problème très important qui correspond à la relativité de l'utilisation d'une opération en fonction de son contexte; par exemple, l'opération "carré de (A)" dont l'effet-net est de produire une valeur B = A est sans intérêt si l'opération "primitive" à décrire est "créer une table des carrés des cents permiers nombres entiers", car la description de cette opération devrait exploiter le fait que le carré d'un nombre "n" est égal à la somme des "n" premiers nombres impairs pour former un programme efficace.

## 1.3. Objets "Abstraits".

Le choix d'une méthode de formation des objets composés de "types" prédéfinis peut être résolu de deux manières : soit obliger le programmeur à préciser, dès la création de l'objet, toutes les relations "structurelles" qui existent entre les types prédéfinis qui forment l'objet, soit permettre la création d'objets "abstraits". Par création d'un objet abstrait, on entend que le programmeur a la possibilité de postposer la description des relations "structurelles" entre les types prédéfinis et de décider la nature de celles-ci au fur et à mesure du développement du programme. En fait, le programmeur attribue un "qualificatif" à toute une catégorie d'objets aux propriétés bien définies, mais dont le programmeur ignore l'ensemble des opérations qui seront appliquées à ces objets au cours de l'algorithme.

#### Exemple :

Supposons que nous soyons en train de concevoir un programme pour jouer aux échecs et que nous désirions utiliser des objets répondant à la notion de "chess-board pattern", définie comme étant une configuration légale de pièces du jeu sur un échiquier; cette notion apparaîtra dès le début de la conception du programme, mais il ne nous sera possible de choisir une représentation pour "chess-bord pattern" que lorsque nous aurons un aperçu suffisamment précis des opérations qui seront appliquées à des objets de cette catégorie; notre connaissance de ces opérations ne se formera qu'au fur et à mesure du développement du programme; toutefois, nous ne pouvons renoncer à attribuer le "qualificatif" "chess-bord pattern" à diverses variables en attendant de posséder tous les éléments qui permettront de le définir en fonction des "types" définis dans les langages terminaux; en conclusion, il doit être permis d'utiliser des objets "abstraits" pendant le développement du programme et de les transformer progressivement en leurs composants "terminaux".

En général, les langages actuels de programmation ne permettent pas de postposer la définition des relations "structurelles"; à notre avis, seule la possibilité de postposer correspond vraiment aux besoins des programmeurs, et d'autant plus qu'on insiste sur une méthodologie de programmation "topdown" ou par "step-wise refinements".

# 2. CREATION ET RAFFINEMENT DES OBJETS ABSTRAITS

Nous avons besoin de deux types d'ordre pour les objets abstraits : un ordre destiné à créer une catégorie d'objets abstraits, et un autre destiné à expliciter les relations entre les composants d'un objet abstrait. Un troisième ordre sera également nécessaire pour associer un objet de l'environnement soit à un "type" prédéfini, soit à un objet abstrait. Ces trois ordres seront désignés en HPL respectivement par "create", "refine", et "declare".

# 2.1. Ensemble des Objets et ensemble des Types.

Les "types" prédéfinis des langages terminaux forment un ensemble To; les objets abstraits créés pendant le développement d'un programme forment un ensemble T<sub>p</sub>. L'ensemble To existe initialement dès qu'un système de programmation par holons contenant un ensemble de langages terminaux est défini.

On appelle "ensemble des types", l'ensemble T qui résulte de l'union des ensembles  $T_0$  et  $T_0$  :

La dénomination "ensemble des types" est un peu abusive car les objets abstraits ne sont pas à proprement parler des "types".

L'ensemble E, "ensemble des objets" correspond à l'environnement du programme; le contenu de cet ensemble dépend de l'exécution du programme : l'enclenchement d'un holon pouvant impliquer l'introduction d'une série de nouveaux objets dans l'ensemble E, la production effective de l'effet-net de ce holon signifiant la disparition des objets introduits lors de l'enclenchement.

L'ensemble T subit un accroissement cumulatif du nombre de ses éléments lors du développement du programme, tandis que l'ensemble E varie dynamiquement en fonction de l'exécution du programme.

Dans le corps de chaque holon  $H_i$  peut exister un ensemble d'ordres "create", "refine", "declare"; les deux premiers servent à modifier T, le troisième à modifier E.

# 2.2. Ordre "create".

Le résultat d'un ordre "<u>create</u>" est l'introduction d'un nouvel élément dans le sous-ensemble T<sub>p</sub> de T; l'ordre attribue également un nom à cet élément afin de l'identifier ultérieurement.

Les holons d'un programme sont ordonnés selon leur ordre d'introduction dans le programme, l'ordre des holons détermine celui des éléments de  $T_{\rm p}$ , c'est-à-dire que les éléments introduits par les ordres "create" d'un holon  ${\rm H_i}$  ont des numéros d'ordre inférieurs aux numéros attribués aux éléments introduits par les ordres "create" d'un holon  ${\rm H_j}$ , si j > i. De la même manière, les "create" d'un même holon sont classés selon l'ordre d'apparition dans le corps de holon.

Pour simplifier, on peut considérer qu'il n'y a qu'une opération "create" par corps de holon, mais que celle-ci entraîne l'introduction de plusieurs objets abstraits dans l'ensemble T<sub>p</sub>. Nous schématisons cette opération par :

$$[\underline{create}(H_i) : (a_0, a_1, a_2, \dots, a_n)]$$

qui signifie que dans le corps du holon  $H_i$ , on a requis la création des objets abstraits  $(a_0, a_1, \dots, a_n)$ . Il est clair que toute occurrence ultérieure du nom nh<sub>i</sub> du holon  $H_i$  n'implique pas de nouvelles "créations" d'objets abstraits.

La seule condition imposée à un "create" est l'originalité du nom attribué à l'élément créé, c'està-dire que l'ordre

## [create(H;) : b]

est correct si et seulement si aucun élément de nom "b" n'existe dans T au moment de l'occurrence de cet ordre "create". Si cette condition n'est pas vérifiée, l'ordre est illégal, et le système de traitement du programme en avertit le programmeur.

## 2.3. Ordre "refine".

L'ordre "refine" est utilisé pour préciser les relations d'appartenance entre les membres de T. La relation qui peut exister entre deux éléments "a" et "b" de T est :

"est un composant de".

Par exemple : a "est un composant de" b; relation que l'on note (b.a).

Une relation (b.a) est correctement construite si "a" et "b" appartiennent à T, et si "b" n'appartient pas à T $_{f a}$ .

Une collection de relations  $\{(b.a_0), (b.a_1), (b.a_2), (b.a_3), \ldots, (b.a_n)\}$  est correctement construite si tous les éléments  $(a_0, a_1, \ldots, a_n)$  appartiennent à  $T_p$ . Une collection de relations s'interprète comme suit :  $a_0$  "est un composant de" b, et également,  $a_1$  "est un composant de" b, et également,  $a_2$  "est un composant de" b, etc... Ce qu'on représentera par une "structure" :

$$\begin{array}{c}
a \\
a \\
a \\
a \\
2 \\
\vdots \\
a \\
n
\end{array}$$
(a)

Les éléments (a<sub>o</sub>,a<sub>1</sub>,a<sub>2</sub>,...,a<sub>n</sub>) peuvent à leur tour intervenir dans une relation "est composant de". Par exemple, si l'élément a<sub>o</sub> intervient dans (a<sub>o</sub>.c), la structure (α) devient :

b. 
$$\begin{pmatrix} a_0 \cdot c \\ a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix}$$
 (6)

Si au contraire, la relation est (c.a<sub>o</sub>), le schéma de représentation de (ɑ) est inchangé, mais on doit lui adjoindre une nouvelle structure :

b. 
$$\begin{pmatrix} a_0 & \text{et} & \text{c.a}_0 \\ a_1 & & & \\ \vdots & & & \\ a_n & & & \end{pmatrix}$$

Dès qu'une relation (a "est un composant de" b) est exprimée, elle s'applique à toutes les occurrences de l'élément "b" dans les "structures" déjà définies. Par exemple, si on a les structures suivantes :

b.h.d 
$$m \cdot \begin{cases} a \\ d \end{cases}$$
  $n \cdot \begin{cases} a \\ a_1 \\ a_2 \end{bmatrix}$ .d

a relation (e "est un composant de" d) les transforme en :

Une structure "a" est complète si tous les éléments "b<sub>i</sub>" qui vérifient la relation (a.b<sub>i</sub>) sont soit une structure complète.

Une structure "a" est *récursivement complète* si parmi les éléments "b $_i$ " qui vérifient la relation  $(a.b_i)$ , il en existe au moins un qui soit membre d'une chaîne de relations d'appartenance telle que :

$$p_1 \cdot p_2 \cdot \cdot \cdot \cdot p_i \cdot \cdot \cdot \cdot p_n$$

où  $p_1 \equiv a, p_2 \equiv b_i$  et

où  $p_n = p_j$  avec  $n \ge j+2$ , et où les éléments  $p_q$  avec j < q < n sont différents de  $p_j$ ; les autres  $b_j$  oivent être des structures complètes. La condition  $n \ge j+2$  est imposée parce que la relation (x.x) est .11 égale.

#### Exemple :

Supposons que T<sub>p</sub> comprenne, entre autres, les éléments "arbre","sous-arbre à gauche","sous-arbre l droite", et que nous ayons les relations suivantes :(arbre.sous-arbre à gauche), (arbre.sous-arbre l droite); si nous ajoutons les relations suivantes : (sous-arbre à gauche.arbre) et (sous-arbre à droite.arbre), la structure "arbre" est "récursivement complète" et est équivalente à :

arbre. sous-arbre à gauche.arbre sous-arbre à droite.arbre

Au fur et à mesure que les ordres "create" précisent les relations qui existent entre les éléments le T, des attributs peuvent être associés à chaque élément "a" de T<sub>p</sub> :

- 1) un élément "a" est :
- soit "composant principal", attribut qu'il reçoit lors de sa création et qu'il conserve tant qu'il n'intervient pas dans une relation (x.a) où "x" n'est pas un composant direct ou indirect de "a"; soit "composant" s'il a perdu l'attribut "composant principal";
  - 2) en <u>outre</u>, un élément "a" est :
- soit "raffiné", s'il existe au moins une relation qui précise qu'un autre élément "x" appartenant à T est un composant de "a";

- soit "brut", si une telle relation est inexistante;
  - 3) en outre, un élément "a" est :
- soit "défini", s'il appartient (en tant que composant ou composant principal) à une structure complète ou récursivement complète;
- soit "indéfini" dans les autres cas.
  - 4) en outre, un élément "a" est :
- soit "attribué", dès qu'il existe un ordre <u>declare</u> qui associe une variable à un objet abstrait de nom "a";
- soit "non-attribué", tant qu'une telle déclaration n'existe pas dans le texte du programme-holon.

Lors de sa création, les attributs d'un élément de  $T_{\rm p}$  sont :

"composant principal, brut, indéfini, non-attribué".

L'ordre "refine" est schématisé comme suit :

$$[\underline{\text{refine}}(H_i) : \{(a_0.b_0), (a_1.b_1), (a_2.b_2), \dots, (a_n.b_n)\}]$$

ce qui sifnifie que dans le coprs du holon  $H_i$ , on a requis que les relations  $(a_i.b_i)$  existent entre les éléments de l'ensemble T.

Pour que cet ordre "refine" soit correct, il faut que les éléments "a;" et "b;" (avec o  $\leq$  i  $\leq$  n) appartiennent à l'ensemble T, tel qu'il est après l'introduction de H; dans le programme, c'est-à-dire que tous les éléments doivent être apparus dans des ordres "create" appartenant à H; avec j  $\leq$  i. En outre, il est nécessaire qu'aucune relation (a; b;) ne viole les règles exposées dans cette section.

### 3. PROCEDES DE STRUCTURATION

Dans de nombreux cas de "raffinement" d'un objet abstrait, on peut désirer créer un composant qui représente une collection d'autres composants, tous de même nature. En HPL, il est possible de représenter cette collection d'objets identiques sous trois formes différentes :"array", "séquence" et "file". La différence entre les trois formes réside dans les possibilités d'accès à un élément de la collection, et également dans les contraintes que chaque catégorie impose aux processus de conservation en mémoire des éléments. Dans la définition de ces formes, nous imposons non seulement la méthode d'accès, mais également des considérations sur les "performances" de cette méthode d'accès; ces "performances" doivent être respectées par toute implémentation.

### 3.1. Array.

Supposons qu'on désire qu'une collection de n objets "a" devienne un composant de l'objet "b", on exprime cette construction par :

(array[...]of a "est un composant de" b),

que l'on note par :

Entre les deux crochets, on désignera les valeurs des bornes du domaine de variation de l'index du tableau : si le tableau est destiné à contenir n éléments, il faut que le domaine défini comprenne n valeurs entières distinctes pour l'index; ces valeurs peuvent être négatives, positives ou la valeur nulle. Pour accéder à un élément "a", on précise la valeur de son index; d'autre part, il est illégal de se référer à un élément par un index qui excède les valeurs des bornes.

Par exemple, si l'on a défini un tableau de 10 éléments :

le troisième élément est accessible par : b.a[-3].

La condition imposée à la méthode d'accès est qu'aucune valeur d'index n'implique un temps d'accès discriminatoire, c'est-à-dire que quelle que soit la valeur de l'index (dans les limites de son domaine de variation), les opérations d'accès impliquent le même "temps d'exécution".

#### Remarque :

Les valeurs des bornes peuvent être données sous forme de noms de variables de type "entier", (voir la définition du "declare").

## 3.2. Séquence.

La "séquence" permet, elle aussi, de grouper n objets ou plus de même type; mais, au contraire du "array", elle n'implique pas que tous les objets existent en même temps. On la représente par la relation :

(sequence[...]of a "est un composant de" b)

que l'on note :

## (b.sequence[...]of a).

La "séquence" indique une éventualité: le nombre de composants "a" dépendra de l'exécution du programme; initialement, lorsqu'un objet de cette forme est déclaré dans l'environnement E, il ne contient aucun composant; ceux-ci seront introduits (ou expulsés au fur et à mesure de l'exécution du programme et selon les besoins de l'algorithme; le nombre de composants peut même excéder la quantité définie par la différence des bornes. Par le domaine de variation de l'index, le programmeur précise ce qu'il considère être la dimension "moyenne" de la collection d'éléments; si cette dimension est dépassée, il admet implicitement que les performances se dégradent sans pour autant interrompre l'exécution de l'algorithme. En fait, chaque élément est accessible par un index de valeur entière; si la valeur de cet index n'appartient pas au domaine de variation indiqué par les bornes, l'accès à l'élément peut être plus "coûteux" et impliquer plus d'opérations.

Cette forme de structuration correspond à toutes les "data structures" dont la nombre d'éléments varie constamment en cours d'exécution d'un algorithme, cas d'une "pile", d'une "queue", d'une "doublequeue", etc...

## 3.3. File.

Dans le cas d'une "file", ni le nombre total de composants, ni une dimension "moyenne" ne sont spécifiés, mais l'accès aux éléments ne peut avoir lieu que d'une manière purement séquentielle à partir de l'origine de la "file" ou à partir de l'élément accédé précédemment. Il n'existe aucune section de la "file" qui bénéficierait d'un temps d'accès privilégié : le temps mis pour accéder au n-ième élément à partir du (n-1)-ième est indépendant de la valeur de n, et pour la même valeur de n peut même varier largement au cours de la même exécution.

On représente la relation

# ( $\underline{\text{file of}}$ a "est un composant de" b) par (b. $\underline{\text{file of}}$ a).

L'accès au n-ième élément à partir du dernier élément accédé dans la "file" est indiqué par : b.a[+ n], où  $n \ge 0$ ; une valeur négative pour n est illégale; n peut être remplacé par l'identificateur d'une variable de type "entier".

# 4. SYNTAXE DE "CREATE" ET DE "REFINE"

#### 4.1. Create.

La syntaxe suivante est choisie pour représenter un ordre "create" :

<create>::= create<liste de noms d'objet>;
<liste de noms d'objet>::=<nom d'objet>[{,<nom d'objet>}...]

La possibilité <nom d'objet abstrait>:<terminal> correspond à une re-définition d'un "terminal", par exemple :

## create état-du-programme : boolean;

ultérieurement dans le programme, des variables pourront être déclarées comme étant du "type" "état-du-programme".

#### 4.2. Refine.

La syntaxe de l'ordre "refine" est destinée à représenter une série de relations "est un composant de"; elle se présente comme suit :

<refine>::= refine<liste de structures de raffinement>; de structures de raffinement>::= <structure de raffinement>[{,<structure de raffinement>}...] <structure de raffinement>::= de noms d'objets abstrait> by <structure description> de noms d'objet abstrait>::= <nom d'objet abstrait>|(<nom d'objet abstrait>[{,<nom d'objet abstrait>}...]) <structure description>::={<élément de structure description>}... <fle><élément de structure description>::= <nombre><nom d'objet abstrait>[<structure description>|:<terminal>]| <nombre>array<index description>of(<structure description>|<terminal>}| <nombre>sequence<index description>of{<structure description> | <terminal>}| <nombre>file of{<structure description> | <terminal>} <index description>::=<domaine fixe>|<domaine variable> <domaine fixe>::= "["<limite inférieure>".."<limite supérieure>"]" <domaine variable>::= (voir section 5.5) imite inférieure>::=<nombre entier avec ou sans signe> dimite supérieure>::=<nombre entier avec ou sans signe>

### Remarque :

Les symboles "[","..","]" signifient que les caractères entre les guillemets ne doivent pas être pris dans leur interprétation méta-linguistique, mais dans leur signification graphique.

# 4.3. Règles de construction.

La description syntaxique ne permet pas d'exprimer toutes les règles de construction d'une «structure description»; il est plus aisé d'exposer ces règles en établissant une correspondance entre une «structure description» et la collection de relations "est un composant de qu'elle représente.

L'élément <nombre> sert à indiquer la <structure description> à laquelle appartient l'élément de description qu'il préfixe : selon la syntaxe, un élément de description peut à son tour contenir une <structure description>, si tel est le cas, il est requis que les éléments de description de cette <structure description> "contenue" soient préfixés par un nombre dont la valeur est égale à celle du nombre préfixant l'élément de description "contenant" augmentée d'une unité. Le préfixe de la <structure description> qui suit immédiatement le "by" dans une <structure de raffinement> est obligatoirement "2".

#### Exemple :

refine chess-board-pattern by 2 white 3 before-the-move 3 after-the-move

2 black

3 before-the-move

3 after-the-move;

Cette forme syntaxique correspond à l'ensemble de relations :

{(chess-board-pattern.white),(chess-board-pattern.black),
 (white.before-the-move),(white.after-the-move),
 (black.before-the-move),(black.after-the-move))

L'ordre suivant est identique à celui de l'exemple :

refine chess-board-pattern by
2 white 2 black,
(white,black) by
2 before-the-move 2 after-the-move;

La syntaxe pourrait suggérer que toute combinaison entre "array", "sequence" et "file" est possible, ce qui n'est pas le cas; en symbolisant les trois méthodes de structuration par leurs initiales (A,S,F), on obtient comme relations légales :

(S.A) (S.S) (S.F) (A.A) - (A.F) (F.A) - (F.F)

Les relations indiquées dans ce tableau peuvent être directes ou indirectes.

## 5. CONSTRUCTION DE L'ENVIRONNEMENT D'UN PROGRAMME

# 5.1. L'ordre "declare".

L'environnement d'un programme est constitué par E, "l'ensemble des objets". Tout élément de E est caractérisé par trois attributs :

- son nom qui permet, lorsqu'il est associé au troisième attribut (la portée), de distinguer sans ambiguîté tout élément de E;
- 2) son type qui est un élément de l'ensemble T;
- 3) sa portée qui précise exactement sa période d'existence pendant l'exécution d'un programme, la portée est associée à un "nom de holon".

Ces trois attributs sont associés à un élément de E, lors de l'exécution de l'ordre "declare"; celuici précise qu'un élément possédant tel "nom", associé à tel "type" et dont la "portée" est définie, est introduit dans l'ensemble E. On peut donc schématiser cette opération par :

# [declare (nom, type, portée)]

Pour préciser la valeur des deux premiers attributs, la syntaxe de l'ordre "declare" impliquera une association entre un ou plusieurs "noms" et un "type" :

### <declare>::=

declareliste de noms>:{<nom d'objet abstrait>|
[{array<index>of|sequence<index>of|file of}]
<terminal>};

de noms>::=<nom>[{ <nom>}...]

<nom>::= "séquence de caractères(lettres,chiffres et "-")
qui commence par une lettre et ne contient aucun espacement".

La condition de l'association d'un nom à un objet abstrait (ou à un terminal) est l'existence de ce dernier parmi les éléments de l'ensemble T.

#### 5.2. La portée.

Choisir une méthode de spécification de la portée est plus difficile et, à notre avis; des considérations de simplification de l'implémentation ont fréquemment guidé le choix des "inventeurs" de langage de programmation; nous prendrons en considération des problèmes de méthodologie de programmation.

Considérons le développement d'un programme selon une méthodologie "top-down" ou par "raffinement pas-à-pas"; si on examine les programmes développés selon ces méthodologies et publiés dans la littérature, on aperçoit que les déclarations, groupées en tête de bloc, rassemblent des variables dont l'introduction dans le programme est due à la conception d'un "bloc" interne au bloc préfixé par les déclarations. Par exemple, N. Wirth utilise, pour illustrer ses conférences sur la programmation par "step-wise refinement", un système de transparents : chaque raffinement est écrit sur un transparent dans une couleur qui lui est propre, le programme final est obtenu par empilement des transparents; il en résulte un texte composé de multiples "taches" de couleurs et non pas un ensemble "coloré" selon une certaine régularité (des emboîtements de couleurs, par exemple). La raison de cette structure des déclarations est facile à déceler : au fur et à mesure que le programmeur choisit une manière de réaliser une opération définie, il est engagé dans le choix de toute une série de variables qui seront utilisées par d'autres opérations (1) afin d'exploiter les résultats de l'opération définie; pour que les autres opérations puissent avoir accès à ces variables, il est nécessaire que la portée des variables comprenne ces opérations.

La conséquence de cette forme de programmation est "l'hermétisme" du programme pour ceux qui n'ont pas participé à sa réalisation (ou même pour ceux qui l'ont réalisé, s'ils y reviennent après quelques mois), car le texte du programme n'est plus un "témoin" fidèle de l'historique du développement du programme. Il en résulte que la lecture d'un programme devient le décryptement de la succession des décisions prises lors de sa conception, même si le texte est exprimé dans un langage dit de "hautniveau".

Il existe d'autre part, un autre type de transfert d'information entre une opération et les autres opérations du programme : en effet, pour qu'une opération produise un effet utile, il est nécessaire que les objets qu'elle a modifié survivent à son exécution, mais il n'est pas nécessaire qu'ils soient accessibles à toute autre opération; dans certain cas, il est souhaitable que les objets modifiés ne soient accessibles qu'à un nombre limité d'opérations ou qu'à l'opération elle-même. Cette dernière possibilité existe en Algol 60, où une variable peut être "rémanente" ("own"), c'est-à-dire qu'une variable "rémanente" est "locale" à un bloc, mais la dernière valeur attribuée à cette variable est conservée entre deux exécutions du bloc (2).

En conclusion, le principe suivi en HPL est :

- d'associer matériellement l'ordre de déclaration d'une variable à l'opération du programme qui a amené le programmeur à introduire cette variable;
- 2) de préciser dans la déclaration la portée de la variable.

La syntaxe du "declare" devient :

<declare>::=
 declare .....[<spécificateur de portée>];

où un "spécificateur de portée" peut être inséré avant le point-virgule.

 $<sup>(^1)</sup>$  et qui influencent certainement les descriptions choisies pour ces opérations.

<sup>(2)</sup> Le concept "own" est méthodologiquement très important, malheureusement l'Algol ne possède pas les moyens de l'utiliser efficacement, car la syntaxe est ambiguë pour les tableaux et l'initialisation de la variable "own" n'est pas prévue par la syntaxe!

## 5.3. Spécificateur de portée.

<spécificateur de portée>::=`
{[at<nom de holon>level[restricted]][<initialisation>]|
 same as<nom>in<nom de holon>}

#### Remarque :

<nom de holon> peut toujours être remplacé par <nom de holon booléen> dans la syntaxe de <spécificateur de portée>.

La syntaxe nous propose donc quatre possibilités, pour la spécification de la portée, dont les significations sont :

- s'il n'y a aucun spécificateur, la portée de la variable (ou des variables de la liste) est celle du holon qui contient la déclaration; cette variable est donc purement locale à un enclenchement de ce holon;
- 2) si le spécificateur est "at <nom de holon>level", la portée est définie par le holon dont le nom est cité entre "at" et "level"; ce qui signifie que la variable existe dès que ce holon est enclenché et qu'elle survit jusqu'à ce que cet enclenchement du holon ait produit effectivement son effet-net. Supposons que le holon qui contient la déclaration soit "x", et que le holon spécifié par "at...level" soit "y"; pour que l'ordre soit légal, il est nécessaire que toutes les occurrences du holon "x" dans le programme contiennent le holon "y" parmi les membres de leurs ensembles NG. Cette condition est vérifiée au cours du développement du programme : au moment où le holon "x" est décrit, c'est-à-dire où un holon H<sub>i</sub> dont le nom est équivalent à "x" et qui contient cette déclaration est ajouté au programme P, la déclaration est enregistrée et l'existence de "y" parmi l'ensemble NG(x) est vérifiée; ensuite, lorsqu'un holon H<sub>j</sub>, avec j > i, contenant dans sa description une occurrence de "x" est ajouté au programme P, l'existence d'un "y" pour le NG(x) de cet "x" est vérifiée; et il en sera de même pour l'introduction dans le programme d'un holon H<sub>k</sub>, avec k > j qui contiendrait une occurrence du nom du holon H<sub>i</sub> dans sa description.
- 3) si le spécificateur "at...level" est suivi de "restricted", la déclaration est presque équivalente à la déclaration d'une variable "rémanente" en Algol 60, car la variable déclarée ne peut être utilisée que par le holon qui contient la déclaration et par les holons qui spécifient expressément l'utilisation de cette variable par un "same as"; contrairement à l'Algol 60, il n'y a aucune ambiguîté pour une variable de type "array" à bornes variables, car un niveau bien défini du programme est spécifié pour l'introduction de la variable dans l'ensemble E.
- 4) le spécificateur "<u>same as</u>" est une extension de la troisième possibilité; supposons qu'un holon A contienne la déclaration :

## declare y:t2 at C level restricted;

et qu'un holon B contienne la déclaration :

## declare x:t1 same as y in A;

- si les conditions suivantes sont vérifiées :
  - 1) t<sub>1</sub> = t<sub>2</sub> et appartiennent à T;
  - 2)  $A = H_i$  et  $B = H_i$  avec j > i;
  - 3) C est supra-ordonné direct ou indirect de A et de B; alors les valeurs de "x" et de "y" sont confondues, même si les noms "x" et "y" ne sont pas identiques; un système de vérification identique à celui exposé pour le "at...level" est appliqué à toute occurrence explicite ou implicite d'un holon qui contient un ordre "declare...same as...".

## 5.4. Initialisation.

La syntaxe prévoit qu'un spécificateur de portée - sauf s'il s'agit de "same as" - peut être suivi d'un élément d'initialisation.

<initialisation>::=
 initial(<nom de holon>|begin<liste de noms de holon>end)
cliste de noms de holon>::=
<nom de holon>[{,<nom de holon>}...]

Le (ou les) holon cité "<u>initial</u>" est destiné à initialiser la variable déclarée; ce holon est enclenché pendant l'exécution du holon "x" qui contient la déclaration effective et avant tout enclenchement des holons qui composent le corps du holon "x".

#### Exemple:

Supposons que dans le corps du holon B, on ait :

declare x:t at A level initial D;

Lors de l'enclenchement du holon A, les opérations d'enclenchement de ses composants ont lieu dans l'ordre suivant :

- 1) le datatest:
- 2) les holons "E,F,D" qui suivent <u>initial</u> dans les déclarations; toutefois pour ces holons, <u>aucun</u> ordre d'enclenchement n'est précisé par la définition du langage, le programmeur doit donc veiller tout spécialement à éviter tout effet de bord dans la description de ces holons;
- 3) le corps du holon.

Les "initialisations" sont effectuées chaque fois que le holon est enclenché; d'autre part, dans le cas d'une déclaration "restricted", les holons d'initialisation ont bien entendu accès à la variable déclarée; mais, que la déclaration soit "restricted" ou non, son holon d'initialisation  $\underline{n}$ 'a  $\underline{pas}$  accès aux variables déclarées au même niveau effectif, ainsi dans l'exemple :

```
D a accès à x, mais pas à y ou z,
E a accès à y, mais pas à z ou x,
F a accès à z, mais pas à x ou y.
```

Le type d'un objet déclaré doit appartenir à l'ensemble T; mais il n'est pas requis qu'au moment de la déclaration, l'objet soit d'un "type défini"; on peut donc avoir la séquence d'ordres suivante:

create chess-board-pattern;
declare fischer : chess-board-pattern;

Tout élément de T utilisé dans, au moins, une déclaration est dit "attribué", et tout élément "attribué" doit être ultérieurement raffiné jusqu'à ce qu'il soit complètement défini.

Autoriser le programmeur à déclarer des objets qui, au moment de la déclaration, correspondent à des objets abstraits est très utile pour le développement d'un programme, car de cette manière, on peut appliquer aux "data structures" le même processus d'abstraction que celui qu'on applique aux opérations "primitives"; mais ce procédé introduit un problème de spécification pour les tableaux (ou séquences) à bornes variables.

#### 5.5. Variable-index.

Un tableau (1) est à bornes variables s'il est défini comme suit : "array[x..y]of a" où "x" et "y" représentent une collection de valeurs entières dont une valeur bien déterminée pourra être choisie lors d'une déclaration d'un objet de ce type.

Dans un langage conçu selon les principes de l'Algol, le "type" associé à toute déclaration est complètement connu au moment de la déclaration; au contraire, en HPL, il est fréquent que des variables soient déclarées comme étant un "objet abstrait" donné, et qu'ultérieurement, au cours du développement du programme, un ou des composants de cet "objet abstrait" soient raffinés par un tableau à bornes variables. Pour fixer les idées, prenons le cas où un holon H; contient les ordres suivants :

et un holon H; (avec j > i) contient le raffinement de "z" :

$$H_{i}[\dots \underline{refine} \ z \ \underline{by} \ 2 \ \underline{array[a..b]} \underline{of} \ \underline{real};\dots]$$

où "a" et "b" correspondent à des paramètres de type "entier". Supposons d'autre part que les variables que nous désirons associer à ces paramètres soient différentes pour "x" et pour "y"; par exemple, que "a" prenne la valeur d'une variable "n" pour "x" et la valeur d'une variable "m" pour "y". Pour que les dimensions du tableau soient connues à la déclaration de "x" et de "y", il faut qu'elles soient précisées par l'intermédiaire de variables qui existent dans l'environnement de la déclaration; on utilise une variable-index, c'est-à-dire une variable dont la déclaration reprend le nom donné au paramètre dans l'objet abstrait et lui associe le nom d'une variable dont le type déclaré est cet objet abstrait.

Exemple :

## declare q index of x:z(a);

signifie que la variable "q" doit être utilisée comme valeur de l'index "a", lors de l'introduction de l'objet "x" de "type" "z" dans l'environnement.

Une variable-index est de type "entier", et toute valeur entière ou toute variable de type "entier" peut lui être attribuée par une instruction d'affectation; elle ne peut être utilisée comme opérande dans aucune opération arithmétique. La déclaration d'une variable-index peut être suivie des options "at...level" et "initial", l'option "restricted" est sans utilité et n'est pas permise.

La déclaration peut associer plusieurs index de tableau à la même variable-index, par exemple :

declare q index of x:z(a),y:z(b);

ou encore :

### declare q index of x,y:z(a);

Si une variable-index n'est pas attribuée à un paramètre de bornes de tableau, le système de traitement du programme en fera rapport au programmeur.

### 6. SYNTAXE DES IDENTIFICATEURS DE VARIABLE

Nous appelons variable un élément de l'ensemble E; le terme "variable" doit être compris dans le sens "d'élément dont la valeur peut varier au cours de l'exécution du programme".

Le nom est utilisé pour repérer d'une manière non-ambigueun élément de E; toutefois, puisqu'un élément de E peut être formé par toute une collection d'élements dont le "type terminal" est précisé par les raffinements que le "type" abstrait subit, il ne suffit généralement pas d'avoir accès à un élément de E d'une manière globale, mais il est nécessaire de pouvoir accéder à chacun de ses composants d'une manière individuelle; en effet, les éléments dont la valeur est effectivement "variable" sont les éléments terminaux qui sont modifiables par des opérations des langages terminaux; c'est pourquoi

 $<sup>(^1)</sup>$  Tout le contenu de cette section est également applicable aux "séquences".

nous devons disposer d'un système d'identification plus élaboré que le simple "nom" afin d'identifier les sous-composants d'un élément de E.

On peut représenter tout élément de E par l'équivalent d'une structure d'arbre : le "nom" de l'élément est la racine de cet arbre, cette racine possède un successeur qui est le "type" de l'élement, celui-ci possède à son tour un ou plusieurs successeurs immédiats qui possèdent un ou plusieurs successeurs sauf s'ils sont de type "terminal".

On appellera identificateur la représentation syntaxique du chemin qui conduit dans l'arbre du "nom" au composant terminal que l'on désire utiliser; l'identificateur se présentera comme une séquence de "termes" qui permet de reconstituer le chemin menant au composant terminal.

Les règles suivantes doivent être appliquées pour déterminer la correspondance entre les arcs du chemin et les "termes" de l'identificateur :

- 1) le "nom" de la variable doit toujours être présent en tête de la séquence;
- 2) le "type" de la variable est toujours omis;
- 3) les autres "types" qui explicitent les raffinements du "type" de la variable et qui mènent au souscomposant à identifier doivent être tous présents et dans leur ordre d'appartenance; s'il s'agit d'un"tableau" ou d'une "séquence", le "type" de l'élément de tableau est suivi de la valeur de l'index précisée soit par une constante entière, soit par une variable de type "entier";
- 4) le "type terminal" est omis.

## Exemple :

Supposons qu'une variable "x" soit déclarée de type "y" raffiné comme suit :

4 characterstring

Les identificateurs suivants obéissent aux quatre règles et représentent la seule manière légale d'indiquer un des trois sous-composants de "x" :

Il est toujours nécessaire d'indiquer toute la séquence des "sous-types" même s'il semble que l'on puisse l'interrompre après un composant sans introduire une ambiguîté; dans l'exemple, on ne pourrait pas écrire :

car les raffinements des types sont cumulatifs, c'est-à-dire que si les formes (a.1) et (a.2) sont sans ambiguîté lorsqu'elles se trouvent dans le holon  $H_i$ , elles deviennent ambiguës si dans un holon  $H_i$ , avec (j > i), se trouve le raffinement suivant :

qui a pour effet d'ajouter un nouveau composant "q" aux composants "e" et "d".

Dans le cas d'une "file", l'index doit être précédé d'un signe "+", (voir section 3.3); si le signe

"+" et toute valeur d'index sont omis, soit "[]", on considère qu'il s'agit de l'élément actuellement atteint dans le parcours séquentiel de la "file".

Le cas des types à structure récursivement complète est examiné dans la section suivante.

### 7. STRUCTURE "RECURSIVEMENT COMPLETE" ET POINTEURS

Soit la structure récursivement complète :

### 1 arbre

2 information : character

2 sous-arbre-à-gauche

3 arbre

2 sous-arbre-à-droite

3 arbre

La structure est considérée comme étant "complète", car les composants "arbre" de niveau "3" sont assimilés à des types terminaux. En fait, le composant "arbre" de niveau "3" est un "pointeur" vers un élément de type "arbre", c'est-à-dire une variable qui contient des informations d'accès à un élément de type "arbre". La structure "arbre" peut être ré-écrite comme suit :

#### 1 arbre

2 information : character

2 sous-arbre-à-gauche : pointer(arbre)

2 sous-arbre-à-droite : pointer(arbre)

Cette forme de structure ne peut être utilisée que s'il existe, dans le système utilisé par le programmeur, un langage terminal qui permette la manipulation des "pointeurs"; dans ce cas, des variables de type "pointer" peuvent être déclarées et utilisées pour le traitement des structures récursivement complètes.

Un type existant dans l'ensemble T doit être associé à toute déclaration d'une variable de type "pointer"; ce type associé au pointeur restreint les possibilités d'utilisation de la variable, car elle ne peut être utilisée que pour "pointer" des éléments du type spécifié.

Lorsqu'une variable est déclarée de type "x" et que "x" est une structure récursivement complète, un élément correspondant à cette structure est ajouté à l'ensemble E et la valeur des pointeurs que cet élément contient est "nil", signifiant qu'il ne pointe aucun élément.

Supposons que dans le holon H;, on ait la déclaration :

### declare x:arbre;

où "arbre" est la structure définie antérieurement.

Dans le holon  $H_i$  et tous les subordonnés, nous pouvons utiliser des variables de type "pointeur" désignant un "arbre"; supposons que dans  $H_i$ , un des subordonnés de  $H_i$ , on ait la déclaration :

#### declare p:pointer(arbre);

et qu'une des opérations de ce holon soit :

### p:=new(arbre);

le résultat de cette opération (qui appartient au langage terminal de manipulation des pointeurs) est la création d'un nouvel élément adjoint à l'ensemble E; mais contrairement aux autres éléments de E, le "nom" de cet élément n'est point fixé, il peut varier au cours de l'exécution du programme, et peut même être "multiple".

Si "p" a été associé à un objet par l'opération "new" (si celle-ci n'a pas eu lieu, "p" contient "nil"), le contenu de cet objet peut être modifié, car "p" est effectivement devenu le "nom" d'un objet d'un type donné; dans le cas de notre exemple, on pourrait avoir l'instruction :

### p.information:="A";

cette dernière instruction appartient au langage terminal de manipulations de caractères et elle attribue une valeur à une variable de type "character" (ce que p.information est précisément). On pourrait, à ce stade, avoir une instruction telle que :

#### x.sous-arbre-à-droite:=p;

cette instruction appartient au langage des pointeurs et signifie l'attribution de la valeur du pointeur à droite du signe ":=" au pointeur à gauche de ce signe; les deux pointeurs doivent être de même type (dans le cas présent, pointer tous les deux vers des "arbres"). Il résulte de cette opération que l'élément créé lors de l'exécution du holon H<sub>j</sub> est pointé par un composant de "x". Si, à ce stade, l'exécution de H<sub>j</sub> se termine, la variable "p" est éliminée de l'environnement, mais l'élément qu'elle pointait existe toujours, car il est pointé par un composant d'une variable qui appartient toujours à l'environnement (parce que "x" appartient à H<sub>i</sub> dont l'exécution n'est point terminée).

Outre les opérations vues dans l'exemple précédent, le langage terminal de manipulation des pointeurs permet d'attribuer à tout pointeur la valeur "nil" et de vérifier l'égalité des valeurs de deux pointeurs, opération qui produit une valeur booléenne.

## Remarques :

- Il est intéressant de noter le parallélisme entre holon récursif et structure récursivement complète; l'un et l'autre possèdent des propriétés distinctes de celles des holons et des structures "simples".
- 2. Le langage permet de créer explicitement de nouveaux éléments de E, par l'opération "new"; toutefois, il ne possède aucune opération pour les supprimer explicitement : un élément créé au sein d'un holon existe ou n'existe plus, lorsque cet holon a terminé son exécution, selon que l'élément est pointé ou non par un composant d'une variable créée par un supra-ordonné de ce holon. Il est évident que le système de traitement des programmes-holon doit posséder un algorithme de "récupération" des éléments abandonnés, ("garbage collection").
- Plusieurs types peuvent être associés à un pointeur, à condition qu'ils soient tous précisés lors de la déclaration; par exemple : declare p : pointer(arbre or noeud).

## 8. PARAMETRES DANS LES NOMS DE HOLON

Nous rappelons brièvement qu'un nom de holon peut contenir des identificateurs (mis entre guillemets) et des paramètres (identificateur précédé du symbole "%" et suivi d'une séquence de blancs). Dans le cas d'un identificateur, il est clair que le holon se réfère directement aux éléments qu'il désigne; par contre, les paramètres désignent une collection d'éléments auxquels le holon peut se référer; chaque occurrence du même nom de holon dans le texte d'un programme peut différer des autres occurrences par ses paramètres. La première apparition d'un nom de holon dans un programme fixe le type des paramètres, car chaque symbole "%" doit être suivi par un identificateur qui existe dans l'environnement du nom de holon, d'où on peut déduire le type de l'identificateur.

# 8.1. Mode de transmission des paramètres.

Pour suivre la terminologie classique, les paramètres d'un nom de holon sont transmis par "référence", c'est-à-dire que, si în est paramètre d'un nom de holon, toutes les occurrences de în signifient que l'opération en cours concerne l'élément dont le "nom" est "n" dans l'environnement.

Il est également possible de transmettre des paramètres par "valeur", c'est-à-dire que la valeur du paramètre est évaluée lors de l'enclenchement du holon correspondant et que cette valeur est utilisée pour chaque apparition du nom du paramètre dans le corps du holon. Pour appeler par "valeur", il faut entourer le paramètre par les symboles "%(" et ")"; par exemple, %(%n) signifie que %n est utilisé par valeur.

L'appel par "valeur" implique les restrictions suivantes :

- il doit être utilisé pour toutes les occurrences du nom de holon; s'il en était autrement, il ne serait plus possible de définir clairement l'effet-net d'un holon; en effet, si un nom de holon contient n paramètres, il y aurait 2<sup>n</sup> effets-nets différents si on permettait de choisir le genre d'appel à chaque occurrence du nom de holon;
- 2) un paramètre appelé par "valeur" ne peut être l'objet auquel on attribue une valeur dans une opération terminale; toute violation de cette règle doit être détectée par le système de traitement du programme.

En contrepartie, le paramètre transmis par "valeur" peut être remplacé par une expression en langage terminal qui produit une valeur de même type que le paramètre et qui n'utilise aucune affectation.

#### Exemple

Si on a <code>\$(\$n)</code> avec <code>\$n</code> de type "entier", on peut remplacer le paramètre par : <code>\$(add(\$n,\$m))</code>, mais non par : <code>\$(add(\$n,b:=add(\$m,\$c)))</code> qui impliquerait un effet-net sur b, non prévu par la définition du holon qui contient ce paramètre; (on suppose que "add" appartient au langage terminal des entiers).

De la même manière, un paramètre appelé par "valeur" peut être remplacé par une constante.

## 8.2. Disjonction des paramètres.

Les paramètres d'un nom de holon doivent être "disjoints", c'est-à-dire que aucun des paramètres transmis par référence ne peut correspondre à la totalité ou à une partie de tout autre paramètre transmis par référence du même nom de holon.

L'exemple classique justifiant cette condition est la procédure de multiplication de deux matrices dont le résultat est assigné à une troisième matrice : en général, cette procédure est programmée en décalquant la définition mathématique du produit des matrices, mais si syntaxiquement les deux expressions sont identiques, leurs sémantiques ne le sont pas du tout ! En effet, le produit matriciel [A]x[B]=[C] est défini par : soit  $A=(a_{jk})$  une matrice (m x n) et  $B=(b_{jk})$  une matrice (r x p), alors le produit AB (dans cet ordre) est défini seulement quand r=n et est la matrice (m x p)  $C=c_{jk}$  dont les éléments sont :

$$c_{jk} = a_{j1} \cdot b_{1k} + a_{j2} \cdot b_{2k} + \ldots + a_{jn} \cdot b_{nk} = \sum_{i=1}^{n} a_{ji} \cdot b_{ik}.$$

La transcription dans un langage de programmation n'est pas équivalente, car le calcul de la valeur c<sub>jk</sub> résulte d'une succession d'opérations alors que dans la définition mathématique, il est "instantané"; si on utilise par exemple la procédure pour évaluer A=A x B, l'effet-net ne sera pas "le produit de deux matrices". L'origine de ce problème tient au fait que nous avons transformé une "réalité" mathématique en une opération d'affectation, ce qui n'est pas du tout la même chose.

Si les paramètres sont disjoints, la confusion nocive de l'égalité et de l'affectation disparaît; tout nom de holon qui ne vérifie pas cette propriété est incorrect et le système de traitement doit vérifier cette condition.

# 8.3. Identificateurs "généraux".

La syntaxe d'un identificateur représente en fait le chemin menant de la racine d'un arbre à une feuille (voir 6); il est requis que tout identificateur utilisé dans l'expression d'une opération terminale spécifie exactement ce chemin; mais dans un nom de holon qui représente en fait une opération "primitive", la description du chemin peut s'arrêter à un sous-arbre, car il n'est pas requis que dans la définition d'une opération "primitive" l'on détaille les structures jusqu'à leurs composants terminaux.

## 9. QUESTIONS DIVERSES

# 9.1. Commentaires dans les ordres "declare", "refine", "create".

Un commentaire, c'est-à-dire une séquence de symboles inclus entre les symboles "{" et "}", peut être inséré entre le dernier élément d'un ordre "refine", "create", ou "declare" et le point-virgule qui termine cet ordre; on remarquera que, contrairement à l'usage dans les autres langages de programmation, les commentaires ne peuvent apparaître qu'en certains endroits bien précisés du texte d'un programme.

# 9.2. L'option "external".

Si une variable est déclarée comme étant une "file" ou une "séquence" de composants d'un type quelconque, cette variable peut recevoir l'option "external" qui signifie que l'objet désigné par cette variable appartient à un environnement  ${\rm E_X}$  qui existe et survit à l'environnement  ${\rm E_A}$  du programme. Lors-

que la déclaration est rencontrée à l'exécution, l'objet en question est transféré de l'environnement  $E_{\rm x}$  à l'environnement  $E_{\rm r}$  à l'environnement  $E_{\rm r}$  dès que le holon contenant la déclaration a produit son effet-net, un transfert en sens inverse est effectué.

Cette option permet de soumettre périodiquement un objet à un traitement, elle implique qu'une mémoire est prévue pour assurer la correspondance entre le nom de la variable "external" et un fichier existant physiquement dans l'installation de traitement.

Exemple :

declare payroll file of record external;

du langage. Selon le point de vue de Sirechey, il est nettenant prétérable de classer les langages

Dans to chapitae, nous demerons une introduction & la notion de sémantique es par de programmation, telle qu'elle est exposée par Scott et C. Stracher (Stracher)

crirent la tégantique d'un langage "X" et, nous montrerens que ce langage correspind su Bolon Programming Language.

Veut établir des méthode de Affinition de la sémantique des languages de programmation en pénéral, dans notre cas, les restrictions appliquées dans la structure du MPL nous permettent d'unilaver de varaion ausplifiées de la méthode de Stracher. Mous utilisons todicées le mode de présentation et

les symboles de Streckey (Stefit).

A. SIMANTIQUE MATERIALIQUE

On peut introduire la gérantique mathématique en exposant la différence entre les "symboles numéraux" et les "nombres", les ans sont des "expressions" d'un langage tandis que les autres sont a

"abjets" qui correspondent à l'interprétation de ces expressions. Les différentes formes syntaxique utilisées pour représentes les "numéraux" possèdent la même sémentique, c'est-à-dire, la correspondence de les contractes de la contracte de la contracte

Example

Acta Oli(Ag)Ai

scentique des que reux, il fact que nous déterminions une fanction qui permette d'établir une co respondance entre chique expression produite par la syntaxe (c'est-à-dire chaque biénent de Noi)

un élément de Mi nous anions cette fémetion et la correspondance qu'elle établit pay :

four chique diément « de Em), le veleur de la fonction

T ( v )

Cette fonction pout être définie comme suit :

NI 40 I - I VI - I

#### CHAPITRE V

SEMANTIQUE MATHEMATIQUE DU HPL.

#### 1. INTRODUCTION

Selon la remarque de Strachey [Str73], nous ne possédons pas encore de méthode de "classement" des langages qui permettrait de les comparer entre eux; les expressions "langage de haut niveau", "langage de bas-niveau" signifient peu de chose, car en réalité dès qu'un langage possède quelques caractéristiques que ne peut posséder un "assembleur", il reçoit le qualificatif "haut-niveau". Classer les langages selon les propriétés des grammaires utilisées pour en définir la syntaxe est utile pour les réalisateurs d'analyseur syntaxique d'un langage, mais sans intérêt pour l'utilisateur du langage. Selon le point de vue de Strachey, il est nettement préférable de classer les langages selon les différents objets qu'ils peuvent manipuler et selon les possibilités de modification de ces objets; de cette manière, on "abstrait" le langage de sa syntaxe et, ses possibilités sémantiques, qui sont les caractéristiques fondamentales du langage, deviennent apparentes.

Dans ce chapitre, nous donnerons une introduction à la notion de sémantique mathématique d'un langage de programmation, telle qu'elle est exposée par Scott et C. Strachey [Str71]; ensuite nous décrirons la sémantique d'un langage "X" et, nous montrerons que ce langage correspond au Holon Programming Language.

Nous n'incorporerons les résultats de Strachey que d'une manière partielle; en effet, celui-ci veut établir une méthode de définition de la sémantique des langages de programmation en général; dans notre cas, les restrictions appliquées dans la structure du HPL nous permettent d'utiliser une version simplifiée de la méthode de Strachey. Nous utilisons toutefois le mode de présentation et les symboles de Strachey [Str71].

#### 2. SEMANTIQUE MATHEMATIQUE

### 2.1. Exemple introductif.

On peut introduire la sémantique mathématique en exposant la différence entre les "symboles numéraux" et les "nombres"; les uns sont des "expressions" d'un langage tandis que les autres sont des "objets" qui correspondent à l'interprétation de ces expressions. Les différentes formes syntaxiques utilisées pour représenter les "numéraux" possèdent la même sémantique, c'est-à-dire, la correspondance entre les expressions produites par la syntaxe et les nombres.

#### Exemple :

Considérons la syntaxe des "numéraux binaires" :

Appelons Nml, l'ensemble des "numéraux binaires" et N, l'ensemble des "nombres". Pour exprimer la sémantique des numéraux, il faut que nous déterminions une fonction qui permette d'établir une correspondance entre chaque expression produite par la syntaxe (c'est-à-dire chaque élément de Nml) et un élément de N; nous notons cette fonction et la correspondance qu'elle établit par :

Pour chaque élément v de Nml, la valeur de la fonction

79 1 V 1

est le nombre symbolisé par v .

Cette fonction peut être définie comme suit :

Ψ [ O ] = O Ψ [ 1 ] = 1 Ψ [ νΟ ] = 2 . Ψ [ ν ] + A la gauche du symbole "=", la fonction est appliquée à des expressions, tandis qu'à la droite on envisage les valeurs de la fonction pour un argument donné.

Le problème est de ne point confondre la représentation d'un nombre et la valeur de ce même nombre, de ne point confondre méthode de représentation et objet. Il est d'autant plus facile de faire la confusion qu'il existe des ensemble "réduits" de numéraux, c'est-à-dire des ensembles d'expressions qui correspondent d'une manière bi-univoque à l'ensemble des nombres; par exemple, si on définit la syntaxe des "numéraux binaires" par :

#### v::=0|1|1v

à toute expression correspond un et un seul numbre et réciproquement; la réciproque n'est pas vérifiée par la syntaxe précédente où, par exemple :

#### V[ 0001101 ]= V[ 1101 ]=13

Dans le cas d'un ensemble "réduit", le risque est grand de considérer que cet ensemble <u>est</u> l'ensemble des nombres, alors qu'il n'est qu'une forme d'expression parmi d'autres. Il est préférable de conserver la possibilité de passer d'un système de représentation à un autre, si ce dernier se révèle plus utile ou plus maniable. Par contre, ce qui n'est jamais modifié dans le passage d'un système de représentation à un autre est la sémantique de l'interprétation.

La sémantique des langages de programmation correspond au même problème que la relation "numéraux-nombres", dans le sens que, pour un même langage de programmation, il peut exister différentes manières de réaliser l'interprétation du langage, (par exemple, différents compilateurs sur une même machine ou sur des machines différentes); ces différentes interprétations doivent toutes exprimer la même sémantique. Il nous faut donc exprimer ce qui doit être commun à toutes ces "représentations" d'un langage, c'est-à-dire exprimer la "sémantique" du langage.

### 2.2. Sémantique et résolution de problèmes.

Le point de vue de Strachey, vis-à-vis de la nécessité de connaître la sémantique d'un ensemble d'implémentation d'un même langage de programmation, peut être étendu à la notion de résolution de problèmes, et à son corollaire, la conception de programmes. En effet, on peut voir la conception d'un programme comme la composition d'une représentation d'une solution à un problème donné, cette solution étant la sémantique du programme.

Les résultats les plus intéressants de la résolution de problèmes ("problem solving") sont ceux qui nous permettent d'apprécier l'utilisation de méthodes de représentation qui simplifient la découverte et l'établissement d'une solution correcte du problème. C'est de cette manière qu'il faut interpréter la "programmation structurée" de Dijkstra, chaque niveau introduit dans la composition du programme est une "interprétation" d'une opération d'un niveau supérieur, et il est requis que la sémantique de l'interprétation soit identique à celle de l'opération; les limitations des structures de "contrôle" existent afin de faciliter la démonstration de l'équivalence des sémantiques.

#### 3. SEMANTIQUE DU LANGAGE "X"

## 3.1. Ensemble des états.

Nous allons interpréter le langage en fonction d'un ensemble d'états; en effet, une expression d'un langage de programmation possède une valeur qui dépend de "l'état" existant lorsque son évaluation est initialisée. Les "états" forment un ensemble S; nous considérerons des transformations de S dans S; la sémantique du langage sera exprimée par les transformations que les commandes du langage peuvent produire sur l'ensemble S. Nous noterons l'ensemble de toutes les transformations possibles de S dans S par [S  $\rightarrow$  S]; une "transformation" est une fonction qui est définie sur l'ensemble S et qui produit une valeur appartenant à S; une fonction "f" de transformation établit une correspondance entre des éléments  $\sigma$  appartenants à S et des valeurs  $f(\sigma)$  qui appartiennent également à S.

Le but est de déterminer une méthode permettant d'associer une fonction de transformation à un programme; cette fonction exprimera la sémantique mathématique du programme, dans le sens que le passage d'un  $\sigma$  à un  $f(\sigma)$  représente le résultat de l'exécution du programme.

## 3.2. Syntaxe du langage "X".

Le langage "X" est une "abstraction" du langage HPL, dans le sens que la majorité des caractéristiques du HPL se retrouvent dans le langage "X", mais celui-ci est décrit selon une syntaxe "abstraite" qui est éloignée de la syntaxe du HPL. La différence entre les deux langages est donc purement syntaxique; le HPL a une syntaxe différente parce que les programmes écrits en langage X sont beaucoup moins "maniables"; par exemple, la syntaxe de X ne possède qu'un seul couple de symboles pour éliminer les ambiguîtés syntaxiques de certaines expressions, alors que en HPL les délimiteurs sont beaucoup plus nombreux.

Le composant de base de la syntaxe de "X" est le programme-holon dont la syntaxe est :

Programme-holon

$$[\S\xi_0, \xi_1, \xi_2, \dots, \xi_n; \mu_0, \mu_1, \mu_2, \dots, \mu_n \xi]$$

Les  $\xi_1$  sont appelés les *identificateurs* et sont associés par une correspondance un-à-un avec les éléments  $\mu_1$ , qui sont appelés *corps-de-holon*. La correspondance est symbolisée par ":", elle signifie que  $\xi_1$  est identificateur de  $\mu_1$ , que  $\xi_2$  est identificateur de  $\mu_2$ , etc... L'équivalent HPL de l'identificateur est le "nom de holon".

On distingue deux types de "corps-de-holon" : avec ou sans datatest; dans le premier cas, le corps-de-holon est symbolisé par  $\mu_{\lambda}$ ; la syntaxe est :

Corps-de-holon

$$\begin{array}{lll} \boldsymbol{\mu}_{\dot{\mathbf{1}}} :: = \boldsymbol{\mu}_{\Delta} \, \big| \, \boldsymbol{\mu} \\ \\ \boldsymbol{\mu}_{\Delta} :: = \boldsymbol{\Delta} \, \rightarrow \, \boldsymbol{\mu} \, , \, \, \text{abort} \\ \\ \boldsymbol{\mu} :: = \boldsymbol{\gamma} \, \big| \, \boldsymbol{\varepsilon} \\ \\ \Delta :: = \boldsymbol{\varepsilon} \, \big| \, \Delta \, ; \, \boldsymbol{\varepsilon} \end{array}$$

La règle syntaxique  $\mu::=\gamma|\epsilon$  signifie qu'un corps-de-holon est soit une *commande*  $\gamma$ , soit une *expression*  $\epsilon$ ; précisons tout de suite que "commande" et "expression" correspondent respectivement à la partie descriptive d'un "holon" et d'un "holon booléen".

Commandes

$$\gamma::=\phi \mid \text{dummy} \mid$$

$$\xi \mid (\gamma) \mid \varepsilon \rightarrow \gamma_0, \gamma_1 \mid \gamma_0; \gamma_1 \mid$$

$$\xi \in (\varepsilon \rightarrow \gamma; \xi_0, \xi_1) \nmid$$

Expressions

$$\begin{split} \epsilon &::= \pi \left| \left| \epsilon_T \right| \xi \right| \\ & \left( \epsilon \right) \left| \epsilon_0 \right. \Rightarrow \left. \epsilon_1, \epsilon_2 \right| \gamma \text{ resultis } \epsilon \right| \\ \epsilon_T &::= \text{true} \left| \text{false} \right. \end{split}$$

La grammaire de X est ambiguë, en effet elle ne précise pas quelle est l'analyse correcte de, par exemple :

le point-virgule sépare-t-il la commande  $\gamma_2$  de la commande  $\gamma_1$  ou de la commande  $\epsilon + \gamma_0, \gamma_1$ ? Ces ambiguîtés peuvent être résolues par l'usage de la forme ( $\gamma$ ) avec un nombre suffisant de paires de parenthèses; par exemple :

$$(\varepsilon + \gamma_0, \gamma_1); \gamma_2$$
 ou  $\varepsilon + \gamma_0, (\gamma_1; \gamma_2)$ 

#### Remarque :

Dans la description de la syntaxe des "commandes" et des "expressions", l'apparition d'un symbole  $\gamma_0$  ou  $\gamma_1$  (ou  $\xi_1$  ou  $\xi_2$ ) dans une partie droite d'une règle de production, doit être interprétée comme représentant une "commande" (ou une "expression") déjà définie. Si dans une même "commande", on trouve plusieurs symboles avec la même valeur d'indice, on doit considérer que chaque occurrence de ce symbole correspond à une "commande" identique (il en est de même pour les "expressions"); toutefois, des symboles dont les valeurs d'indices sont différents peuvent être remplacés par des "commandes" (ou "expressions") identiques. Le symbole  $\xi_0$  qui apparaît dans la syntaxe des commandes ne correspond pas nécessairement au  $\xi_0$  de la syntaxe du "programme-holon".

#### 3.3. Commandes et expressions.

Nous noterons Cmd l'ensemble des commandes  $\gamma$  . Nous pouvons définir la sémantique d'un élément de Cmd en lui associant une fonction représentant une transformation d'état; pour une commande  $\gamma$  , nous noterons cette fonction :

En d'autres termes, on peut dire que la sémantique d'une commande correspond à la transformation :

Nous noterons Exp l'ensemble des expressions  $\varepsilon$  . En plus de l'existence de l'ensemble S des "états", nous supposons l'existence de l'ensemble T des valeurs de vérité qui contient les valeurs " $\underline{\text{true}}$ " et "false".

$$\mathcal{E}: \text{Exp} \rightarrow [S \rightarrow [T \times S]]$$

(où T x S représente le produit cartésien des ensembles T et S).

Si  $\varepsilon$   $\varepsilon$ Exp et  $\sigma$   $\varepsilon$  S, alors :

où t  $\epsilon$  T et  $\sigma'$  est l'état qui résulte de l'évaluation de  $\epsilon$  . Le symbole "<.,.>" représente la fonction "paire".

### 3.4. Opérations mathématiques sur les fonctions.

Nous allons définir différentes opérations sur les fonctions, opérations que nous utiliserons dans la définition de la sémantique des "commandes" et "expressions".

#### a) La composition

Si 
$$f : B + C$$
 et  $g : A + B$ , alors

où "o" représente l'opérateur de composition; on a pour tout  $\alpha$   $\varepsilon$  A :

$$(f \circ g)(\alpha) = f(g(\alpha)).$$

## b) Le produit

Supposons :

$$P : A \rightarrow [B \rightarrow [A \times B]]$$

$$M_0$$
: A x B  $\rightarrow$  A M<sub>1</sub>: A x B  $\rightarrow$  B

tels que :

$$P(\alpha)(\beta) = \langle \alpha, \beta \rangle$$

$$M_0(\langle \alpha, \beta \rangle) = \alpha$$

$$M_1(\langle \alpha, \beta \rangle) = \beta$$

pour tout  $\alpha \in A$ , et tout  $\beta \in B$ .

Si p : A' + [B + C] et q : A + [A' x B], alors le produit "\*" sera défini par :

$$p * q : A \rightarrow C$$

tel que, pour tout  $\alpha \in A$ :

$$(p \star q)(\alpha) = p(M_0(q(\alpha)))(M_1(q(\alpha)))$$

## c) L'identité

Supposons I : A  $\rightarrow$  A tel que I( $\alpha$ ) =  $\alpha$ , pour tout  $\alpha \in A$ .

## d) La sélection

Supposons :

cond : 
$$[S \rightarrow T \times S]x[S \rightarrow T \times S] \rightarrow [T \rightarrow [S \rightarrow T \times S]]$$

tel que :

$$cond(e_1,e_2)(t) = t + e_1,e_2$$

c'est-à-dire que :

$$cond(e_1, e_2)(t)(\sigma) = \begin{cases} e_1(\sigma) & \text{, si } t = true \\ e_2(\sigma) & \text{, si } t = false \end{cases}$$

avec  $e_1, e_2 \in [S \rightarrow T \times S]$ ,  $t \in T$ ,  $\sigma \in S$ .

On utilisera également un opérateur de sélection opérant dans les domaines suivants :

cond : 
$$[S \rightarrow S] \times [S \rightarrow S] \rightarrow [T \rightarrow [S \rightarrow S]]$$
.

Le premier opérateur "cond" sera utilisé pour les "expressions", le deuxième pour les "commandes".

### 3.5. Sémantique des "expressions".

Il est nécessaire de définir la sémantique des "expressions" avant celle des "commandes", car les premières interviennent dans la syntaxe des deuxièmes.

### a) & [ (a) ] = & [ a ]

La fonction des parenthèses est de spécifier un ordre d'évaluation dans une expression; elles ne modifient pas la signification de  $\epsilon$  .

## b) & [ π ] = (pour un S fixé + [T x S ])

Les  $\pi$  sont des expressions "terminales", c'est-à-dire que pour un domaine donné, généralement un sous-ensembles de S, on connaît une transformation de  $S \to T$  x S. En fait, la signification est également définie en dehors du sous-ensemble fixé, nous reviendrons ultérieurement sur cette question.

La signification d'un identificateur (d'expression) est donnée par la signification du corps-de-holon-booléen correspondant; la signification dépend du caractère de  $\nu$ , c'est-à-dire s'il est ou non accompagné d'un "datatest", nous reviendrons sur cette distinction ultérieurement.

d) 
$$\mathcal{E}[\ \epsilon_T\ ] = \begin{cases} \text{si } \epsilon_T\ \equiv \ \text{true alors} \ P(\underline{\text{true}}) \\ \text{si } \epsilon_T\ \equiv \ \text{false alors} \ P(\underline{\text{false}}) \end{cases}$$

Il y a une nette distinction entre (true,false) qui sont des symboles syntaxiques et (<u>true,false</u>) qui sont des éléments de l'ensemble T.

Les symboles (true,false) sont des "constantes", dans le sens qu'ils expriment une valeur par eux-mêmes, valeur qui est déterminée sans aucune évaluation et qui n'implique aucune modification de l'état. Puisqu'une expression produit une transformation [S  $\rightarrow$  [T x S]], nous pouvons appliquer la définition de la sémantique d'un  $\varepsilon_{\rm T}$  à un élément de S, on obtient :

El true 
$$I(\sigma) = P(\underline{\text{true}})(\sigma) = \langle \underline{\text{true}}, \sigma \rangle$$
.

La paire <true, $\sigma>$  appartient bien à [T x S], et la signification de  $\epsilon_T$  n'implique effectivement aucune modification de la valeur  $\sigma$  au moment de l'évaluation de  $\epsilon_T$ . De la même manière on obtient :

$$\mathcal{E}_{\bullet}$$
 false  $\mathbb{I}(\sigma) = P(\underline{\text{false}})(\sigma) = \langle \underline{\text{false}}, \sigma \rangle$ .

En effet, l'expression " $\epsilon_0 \rightarrow \epsilon_1, \epsilon_2$ " correspond en fait à une évaluation conditionnelle, c'est-à-dire que selon la valeur " $\underline{\text{true}}$ " ou " $\underline{\text{false}}$ " de  $\epsilon_0$ , on évalue  $\epsilon_1$  ou  $\epsilon_2$ . On a spécifié dans la définition de "cond" que les arguments de l'opérateur devaient être de type [S  $\rightarrow$  T x S], ce qui est vérifié par  $\hat{\epsilon}$ [I  $\epsilon_1$  ] et  $\hat{\epsilon}$ [I  $\epsilon_2$  ].

Examinons l'effet de la signification proposée pour " $\epsilon_0 \rightarrow \epsilon_1, \epsilon_2$ " sur une valeur  $\sigma$  de S:

$$\mathcal{E}[\epsilon_0 + \epsilon_1, \epsilon_2](\sigma) = \operatorname{cond}(\mathcal{E}[\epsilon_1], \mathcal{E}[\epsilon_2]) \star \mathcal{E}[\epsilon_0](\sigma)$$

c'est-à-dire, en appliquant  $(p * q)(\alpha) = p(M_0(q(\alpha)))(M_1(q(\alpha)))$ ,

$$=\operatorname{cond}(\&[\ \epsilon_1\ ],\&[\ \epsilon_2\ ])(M_o(\&[\ \epsilon_0\ ]\ (\sigma)))(M_1(\&[\ \epsilon_0\ ]\ (\sigma)));$$

puisque & [  $\epsilon_0$  ] ( $\sigma$ ) doit produire une paire  $\langle t, \sigma' \rangle$  où t  $\epsilon$  T, on a :

= cond ( 
$$\epsilon_1$$
  $\epsilon_1$   $\epsilon_2$   $\epsilon_2$  ) ( $M_o(\langle t, \sigma' \rangle)$ ), ( $M_1(\langle t, \sigma' \rangle)$ )

c'est-à-dire, en vertu des définitions de  $M_0$  et  $M_1$ :

= cond(
$$\mathcal{E}$$
[  $\varepsilon_1$  ], $\mathcal{E}$ [  $\varepsilon_2$  ]) (t) ( $\sigma$ ')

tous les éléments appartiennent aux types prévus par la définition de "cond"; d'où l'expression est équivalente à :

si t = "true" alors 
$$\mathcal{E}$$
[  $\varepsilon_1$  ] ( $\sigma$ ')  
si t =  $\underline{\text{false}}$  alors  $\mathcal{E}$ [  $\varepsilon_2$  ] ( $\sigma$ ')

ce qui correspond exactement à l'interprétation attendue pour  $(\epsilon_0 \rightarrow \epsilon_1, \epsilon_2)$ .

On peut considérer que  $\hat{S}$  représente la "mémoire" de l'ordinateur et que  $\sigma$  correspond à un état de cette "mémoire";  $\sigma'$  correspond à l'état de cette "mémoire" après l'évaluation de  $\varepsilon_0$  effectuée dans un contexte de mémoire équivalent à  $\sigma$ . Toutefois, cette assimilation de  $\hat{S}$  à la "mémoire" n'est pas tout-à-fait exacte; en effet, dans une "implémentation" du langage une partie de la mémoire "physique" est utilisée pour représenter des valeurs appartenant à l'ensemble  $\hat{T}$  et pas uniquement des valeurs de  $\hat{S}$ ; la correspondance  $\hat{S}$  "mémoire" est néanmoins utile pour fixer les idées.

Cette expression signifie qu'il faut d'abord exécuter la commande  $\gamma$  dont  $\mathfrak{C}1$   $\gamma$  1 représente la sémantique, et qu'ensuite on évalue l'expression  $\epsilon$  .

### Remarque :

L'expression "γ resultis ε" ne correspond pas seulement à l'utilisation de "resultis" dans la syntaxe du HPL, mais aussi à tous les holons booléens composés d'une série de holons suivie par un holon booléen. 3.6. Sémantique des "commandes".

Pour les mêmes raisons qu'en 3.5(a).

Les  $\phi$  sont des "commandes terminales"; pour un domaine fixé, un sous-ensemble de S, chaque  $\phi$  implique une transformation S  $\to$  S connue.

La commande "dummy" correspond à l'absence d'opération; en effet :

$$\mathcal{E}$$
 dummy  $\mathbb{I}(\sigma) = \mathbb{I}(\sigma) = \sigma$ 

donc, il n'y a aucune modification de  $\sigma$   $\epsilon$  S.

Le développement est identique au développement fait en 3.5(c); toutefois, l'opérateur "cond" diffère, il s'agit du deuxième "cond" envisagé en 3.4(d) tel que :

cond : 
$$[S \rightarrow S] \times [S \rightarrow S] \rightarrow [T \rightarrow [S \rightarrow S]]$$

Mêmes considérations qu'en 3.5(c).

La commande " $\gamma_0$ ; $\gamma_1$ " correspond à la notion de "séquence de commandes", interprétée intuitivement comme signifiant que l'exécution de " $\gamma_0$ " est suivie de celle de " $\gamma_1$ ". Par la définition de la sémantique, on obtient :

si  $\sigma' = \mathcal{C} [ \gamma_0 ] (\sigma)$ , on obtient :

g) la sémantique de la dernière forme syntaxique est traitée dans la section 3.7.

#### Remarque :

La syntaxe des commandes est ambiguë, donc la sémantique basée sur cette syntaxe l'est également; toutefois, on peut considérer que l'on applique les définitions de la sémantique non pas aux expressions syntaxiques mais à une structure qui résulterait d'une analyse des expressions syntaxiques où un emboîtement suffisant des parenthèses a résolu toutes les ambiguités.

# 3.7. Itération et Récursion.

La dernière commande que nous devons analyser est :

Cette commande représente toutes les possibilités d'itération et de récursion. Sa signification intuitive est : exécuter la commande  $\xi_0$ ; cette commande implique l'évaluation d'une expression  $\epsilon$  qui implique l'exécution d'une commande  $\gamma$ , suivie de l'exécution de  $\xi_0$  si  $\epsilon$  produit une valeur "true", et dès que  $\epsilon$  produit une valeur "false", on exécute la commande  $\xi_1$ .

Nous allons définir la sémantique de cette commande; appelons la commande " $\lambda$ " :

$$\mathcal{C}$$
 I  $\lambda$  I =  $\mathcal{C}$  I  $\varepsilon \rightarrow \gamma; \xi_0, \xi_1$  I

or, 
$$\forall i \ \gamma; \xi_0 \ i = \forall i \ \xi_0 \ lo \ \forall i \ \gamma \ lo \ et \ \forall i \ \xi_0 \ l = \forall i \ \lambda \ l$$
,

D'où: 
$$\mathcal{E}[\lambda] = \operatorname{cond}(\mathcal{E}[\lambda] \log [\gamma], \mathcal{E}[\xi_1]) \star \mathcal{E}[\epsilon].$$

En posant : 1 = U λ I , e = U ξ I, f = U γ I et t = ε I ε I

avec 
$$1, f, e \in [S + S] \text{ et } t \in [S + T \times S]$$
,

on obtient :

$$1 = cond(1 \circ f, e) * t$$

Dans cette équation fonctionnelle, 1 est la fonction inconnue qui définit la sémantique de  $\lambda$  , à condition qu'une telle fonction existe. Cette fonction peut ne pas exister si les fonctions f et t sont telles que l'évaluation de 1 peut conduire à une itération infinie.

En fait, une commande  $\lambda$  doit correspondre à une transformation d'états  $S \to S$ ; dans le cas général, il existe des valeurs de  $\sigma$   $\epsilon$  S telles que l'évaluation de  $\lambda$  ne se termine pas; dans ce cas, on dira que  $\lambda$   $(\sigma)$  est indéfini. Mais, inversement, on peut dire qu'il existe un sous-ensemble  $S^O$  de S tel que, pour tout  $\sigma$   $\epsilon$  S, l'évaluation de  $\lambda$  produise une valeur définie.

Imaginons une expression  $\Delta\lambda$ , impliquant une transformation [S  $\rightarrow$  T<sub>D</sub> x S]; cette expression est définie pour tout  $\sigma$   $\epsilon$  S; l'ensemble T<sub>D</sub> contient deux valeurs : "true\*"et "false\*"qui sont distinctes des valeurs de l'ensemble T. L'expression  $\Delta\lambda$  produira une valeur "true\*" ou "false\*" selon que  $\sigma$   $\epsilon$  S° $\lambda$  ou non, S° $\lambda$  est le sous-ensemble de S tel que la commande  $\lambda(\sigma)$ , avec  $\sigma$   $\epsilon$  S° $\lambda$  est définie.

Nous pouvons composer une nouvelle commande sur le schéma  $\epsilon \to \gamma_0, \gamma_1$ :

$$\Delta\lambda \rightarrow \lambda$$
, abort.

Toutefois, nous allons modifier notre conception d'une commande et dire que toute commande correspond à une transformation de  $[S \rightarrow E \times S]$ , où E est l'ensemble des "caractéristiques de fin de commande"; cet ensemble E contient deux éléments : " $\underline{normal}$ " et " $\underline{normal}$ ".

Dans cette conception, la sémantique de  $\Delta\lambda$  +  $\lambda$  , abort est donnée par :

$$e$$
 Δλ → λ, abort  $e$  = cond( $e$  Continue  $e$  0  $e$  Λ  $e$  A abort  $e$  0  $e$  Δ λ  $e$ 

où "abort" est une transformation de [S  $\rightarrow$  E x S] telle que

$$\mathcal{C}$$
 abort  $I = P(\underline{anormal})$ ; et où  $\mathcal{C}$  continue  $I = P(\underline{normal})$ .

Il faut en outre considérer que toute commande  $\gamma$  , même celles qui ne sont pas de type  $\lambda$  , sont en fait de la forme :

$$\Delta \gamma + \gamma$$
, abort

avec  $\Delta\gamma$  = true si  $\gamma$  est définie pour toutes les valeurs  $\sigma\,\epsilon S;$  dans ce cas, on a :

ou encore si

$$(\sigma') = \mathcal{E} \Gamma \gamma \Gamma (\sigma),$$

= 
$$P(\underline{normal})(\sigma') = < \underline{normal}, \sigma' >$$

Après exécution de n'importe quelle commande, on peut par inspection de la paire qui en résulte voir s'il y a eu exécution d'une commande "abort".

## Remarque :

Il faut bien entendu distinguer l'opérateur "cond" qu'on vient d'utiliser des deux "cond" utilisés précédemment; celui-ci est défini par :

cond : 
$$[S \rightarrow E \times S] \times [S \rightarrow E \times S] \rightarrow [T_D \rightarrow [S \rightarrow E \times S]]$$

L'élément  $\Delta\lambda$  (ou  $\Delta\gamma$ ) est appelé le *datatest* de  $\lambda$  (ou de  $\gamma$ ). On voit que sa fonction est soit de garantir la terminaison d'une évaluation, soit de limiter les valeurs de  $\sigma$   $\epsilon$  S pour lesquelles on désire attribuer une interprétation au résultat d'une évaluation; ce dernier cas a lieu si l'on remplace un  $\Delta\lambda$  par un  $\Delta\lambda$  tel que :

ou encore en associant à une commande γ , un Δγ qui soit différent de la constante "true".

C'est ce qui est appliqué pour la sémantique des commandes et des expressions "terminales", cellesci ne sont définies que pour un sous-ensemble fixé de S.

#### 3.8. Sémantique du datatest.

La syntaxe du datatest spécifie deux formes différentes :

Nous représenterons la sémantique de ces deux formes par

où l'indice d signifie que l'on recherche la sémantique d'un datatest contenant l'expression  $\epsilon$  ou  $\Delta;\epsilon$ . Les  $\epsilon$  qui forment un datatest sont des transformations [S  $\rightarrow$  T x S], alors que le datatest est une transformation [S  $\rightarrow$  T  $_d$  x S].

Les valeurs "true\*" et "false\*" appartiennent à  $T_d$ ; quant à C I I, il est équivalent à un  $C_d$  I I ou un  $C_d$  I I selon la forme syntaxique de I I .

où "td" appartient à Td.

01)

Oi

$$= \begin{cases} \mathcal{C}_{d} \cdot c \cdot l \cdot (\sigma') & \text{si } t_d = \underline{\text{true}}^* \\ \text{cond} \left( P(\underline{\text{false}}^*), P(\underline{\text{false}}^*) \right) \cdot \mathcal{C}_{d} \cdot l \cdot c \cdot l \cdot (\sigma') & \text{si } t_d = \underline{\text{false}}^* \end{cases}$$

ce qui correspond au sens donné au datatest au chapitre III : en effet, la forme  $\Delta$ ;  $\epsilon$  correspond à un datatest sous forme d'une conjonction de conditions; la valeur de  $\Delta$ ;  $\epsilon$  sera  $\underline{false}^*$  quel que soit le résultat de l'évaluation de  $\epsilon$  si la valeur de  $\Delta$  est déjà  $\underline{false}^*$ .

# 3.9. Sémantique d'un programme-holon.

Le dernier élément du langage "X" dont nous devons définir la sémantique est le "programme-holon"; celle-ci est donnée par :

$$\mathcal{L}_{1}^{[5]} = \xi_{0}, \xi_{1}, \xi_{2}, \dots, \xi_{n} : \mu_{0}, \mu_{1}, \mu_{2}, \dots, \mu_{n}^{[5]} = \mathcal{L}_{5}$$

en appliquant les définitions de la sémantique, on a :

Dans le texte de  $\mu_0$ , on aura des occurrences des  $\xi_1$ , et de proche en proche, on pourra construire l'expression de la sémantique de tout le programme-holon.

## 4. CORRESPONDANCE ENTRE LE LANGAGE "X" ET LE HPL

Dans la section 3, nous avons indiqué régulièrement les correspondances entre le langage "X" et le HPL; dans cette section, nous exposerons les correspondances entre les éléments syntaxiques du HPL et ceux de "X". Nous désignerons par une lettre minuscule un nom de holon booléen et par une lettre majuscule un nom de holon. Pour les instructions qui n'impliquent aucune récursivité, la correspondance est immédiate :

- a) A;B;C;  $\Rightarrow \xi_A;\xi_B;\xi_C$
- b) if a then B else C fi  $\Rightarrow \xi_a + \xi_B \cdot \xi_C$
- c) if a then B fi  $\Longrightarrow \xi_a + \xi_B$ , dummy
- d) case a then Z
  - eor b then W
  - eor c then Y
  - else X

esac

$$\Rightarrow \xi_a + \xi_Z, (\xi_b + \xi_W, (\xi_c + \xi_Y, \xi_X))$$

Dans toutes les correspondances précédentes, il est bien entendu qu'un  $\xi_a$  appartient à Exp et qu'un  $\xi_A$  appartient à Cmd.

Dans les holons booléens, on peut avoir des modifications des formes syntaxiques (b,c,d), par exemple :

Cette possibilité ne change rien à la correspondance, si ce n'est pour l'appartenance de  $\xi_b$  et de  $\xi_c$  à Exp.

Pour les instructions d'itération, nous allons admettre que l'on puisse "attacher" un identificateur à chaque instruction "while-do"; nous l'indiquerons entre accolades dans la syntaxe de la forme HPL, car il n'est pas requis par la définition du langage; on se souviendra que les instructions-holons possèdent un "nom" dans le modèle de l'arbre binaire.

e) {loop:} while b do C od

$$\implies$$
 \$ loop :  $\xi_b + (\xi_C; loop), dummy $$ 

f) repeat C until b;

$$\implies$$
 C; while b' do C od avec b' "="non b

$$\implies \xi_{C}$$
; \$ loop :  $\xi_{b}$ ,  $\rightarrow (\xi_{C}; loop)$ , dummy \$

#### Remarque :

b' "≣" <u>non</u> b" signifie : un holon booléen de nom "b'" et tel que la valeur booléenne de b' égale la négation de celle de b. g) si un holon booléen se compose d'un holon suivi par un holon booléen, par exemple :

A; c on aura 
$$\xi_A$$
 resultis  $\xi_c$ 

h) envisageons un holon complet :

qui appartient à un programme-holon, on a la correspondance :

$$\hbox{\tt [§....,$\xi_A,....,$\mu_A,.....$]}$$

avec  $\mu_{\Lambda}$  qui correspond à :

$$\Delta_d$$
 + ( $\xi_B$ ; $\xi_C$ ; $\xi_D$ ;( $\xi_g$  +  $\xi_E$ ,dummy);( $\Delta_e$  + dummy,abort)),abort

i) si un holon implique une récursivité, par exemple :

 $\begin{array}{c} \underline{holon} \ A \\ \underline{begin} \ \underline{if} \ b \ \underline{then} \ A \ \underline{else} \ C \ \underline{fi} \ \underline{end} \\ \underline{noloh} \end{array}$ 

la correspondance est :

Autre exemple :

$$\begin{array}{c} \frac{\text{holon A}}{\text{begin }} & \frac{\text{if b then A else C fi}}{\text{if h then A else M fi}}; \\ & \frac{\text{end}}{\text{noloh}} \\ \\ \implies & \xi \xi_{\text{A}} : (\xi_{\text{b}} + \xi_{\text{A}}, \xi_{\text{C}}); (\xi_{\text{h}} + \xi_{\text{A}}, \xi_{\text{M}}) \xi_{\text{M}} \end{array}$$

j) un dernier cas à envisager : celui des noms de holon qui contiennent des paramètres appelés par "valeur"; en effet, dans ce cas, l'évaluation des paramètres correspond à l'exécution d'une séquence de "commandes"; on peut donc écrire que :

$$^{A}(p_{1},p_{2},p_{3},\ldots) \Rightarrow \xi_{A} \Rightarrow \gamma_{1};\gamma_{2};\gamma_{3};\ldots;\xi'_{A}$$

où  $p_1, p_2, \dots$  sont les paramètres transmis par valeur et où  $\gamma_1, \gamma_2, \dots$  sont les commandes destinées à l'évaluation de ces paramètres.

# 5. AVANTAGE DE LA SEMANTIQUE MATHEMATIQUE

La sémantique mathématique permet de faire abstraction de la syntaxe pour retrouver l'exacte signification des opérations; en effet, la même forme syntaxique peut impliquer des significations différentes. L'exemple le plus clair est donné par l'instruction "if-then-else" dont la signification dépend de la nature récursive ou non-récursive de l'opération.

On comprend qu'à significations différentes correspondront des méthodes d'implémentation différentes, puisque l'implémentation est une représentation de la signification. On peut comprendre pourquoi nous avions précédemment insisté sur la possibilité de détecter les structures récursives en HPL, car elles jouent un rôle important dans la détermination de la signification de l'instruction-holon qui les contient. Une autre possibilité, qui est adoptée par la grande mojorité des réalisations de compilateur ALGOL, est de choisir pour les formes syntaxiques à significations multiples, une signification commune qui comprend toutes les autres; ainsi, le compilateur peut considérer que toutes les procédures sont "potentiellement" récursives, et adjoindre à toutes ces procédures un mécanisme de récursivité, ce qui est manifestement superflu dans la majorité des cas.

thite du programme à compiler. For contra, confe vogle est en compradiçtion majeure avec une adthouc-

#### CHAPITRE VI

#### COMPILATION ET EXECUTION D'UN PROGRAMME-HOLON.

#### 1. INTRODUCTION

Dans ce chapitre, nous exposerons la procédure suivie par le programmeur pour composer un programme-holon à soumettre à un compilateur-holon pour analyse et transformation en code exécutable. Les principaux composants et le fonctionnement de ce compilateur seront également décrits.

#### 2. ROLE DU COMPILATEUR

Un programme-holon a été défini comme étant un ensemble ordonné de holons et est symbolisé par :

La construction d'un programme-holon consiste à composer une série de holons différents  $H_1$  et à les assembler selon un ordre adéquat pour former un ensemble ordonné; par ordre adéquat, on entend que les éléments de l'ensemble forment un programme légal et complet (voir chapitre II.4).

Quel que soit le langage de programmation, un programme est toujours un assemblage d'éléments simples : opérations et structures de données; toutefois, des différences considérables existent entre les langages, pour ce qui concerne les possibilités de créer de nouveaux éléments simples et l'ordre à respecter dans l'assemblage de ceux-ci. Il est généralement requis qu'un élément soit entièrement défini et décrit avant d'être utilisé comme composant d'un assemblage; l'avantage de cette règle est de favoriser la conception de compilateurs qui n'effectuent qu'un nombre minimum de "passages" sur le texte du programme à compiler. Par contre, cette règle est en contradiction majeure avec une méthodologie de développement du programme par "raffinements progressifs" ou avec une construction "top-down" du programme; dans ce cas, le texte final du programme résultera d'une pénible transposition du texte résultant du travail de développement du programme afin de rétablir, pour les éléments du programme, un ordre qui respecte le principe de "description complète avant l'usage". De plus, ce principe implique que le compilateur ne peut pratiquement fournir des informations que sur des programmes complets et est sans grande utilité pour la phase de conception et de développement du programme.

Le principe du compilateur-holon est différent : sa fonction principale est de surveiller la construction progressive de l'ensemble des holons qui formeront un programme-holon et de renseigner le programmeur sur les conséquences d'une modification éventuelle du programme; finalement, il peut comme tout compilateur transformer un programme complet en un programme exécutable sur une machine donnée.

On peut voir le système holon comme composé de trois éléments : le programmeur, une espèce de "banque de données" constituée par le programme en cours de développement et finalement le compilateur contrôlant les opérations requises par le programmeur pour compléter ou modifier le contenu de la "banque de donnée". Le contrôle exercé par le compilateur a pour but d'exclure toute transformation "anarchique" du programme.

## 3. REQUETES AU COMPILATEUR

Le programmeur peut soumettre une série de "requêtes" au compilateur afin de faire progresser le développement d'un programme; le terme "requête" - et non "ordre" - est utilisé car le compilateur peut refuser de satisfaire à la requête s'il considère que les conditions légales de son exécution ne sont pas réunies.

Les différentes requêtes possibles sont :

new program, purge, access, append, erase, change, synthetise, execute.

La syntaxe exacte proposée pour la présentation de ces requêtes est d'une importance secondaire; nous exposerons uniquement leurs significations, leurs conditions d'applications et les réponses éventuelles du compilateur.

# 3.1. New program.

Cette requête comporte deux éléments : un nom de programme et une identification du programmeur. Si le deuxième élément correspond à un programmeur autorisé à utiliser le système et si le nom de programme ne correspond à aucun des noms des programmes en cours de développement pour ce programmeur, alors le compilateur alloue dans la "banque de données" l'espace nécessaire pour commencer le développement d'un programme; en d'autres termes, le compilateur prend connaissance de l'élément :

# P[np, { | } ]

# 3.2. Purge.

Cette requête correspond à l'opération inverse de <u>new program</u>, elle implique le présence des mêmes éléments : un nom de programme et une identification de programmeur. Si ce couple correspond à une information contenue dans la "banque de données", la requête est exécutée et toutes les informations qui concernent le programme cité sont perdues.

# 3.3. Access.

Comme les deux requêtes précédentes, celle-ci implique un nom de programme et une identification de programmeur; son objet est d'obtenir la possibilité d'accéder aux informations du programme désigné. Elle doit être introduite avant toute série de requêtes <u>append</u>, <u>erase, change, synthetise</u> ou <u>execute</u>.

### 3.4. Append.

Cette requête est la plus importante de toutes; elle comporte un élément : un texte qui est supposé représenter une séquence ordonnée de holons. Le but de la requête est d'adjoindre la séquence de holons au programme auquel on a obtenu accès par l'exécution de la dernière requête <u>access</u>.

Le compilateur examine la syntaxe du texte fourni afin de vérifier si ce texte correspond effectivement à une séquence de holons syntaxiquement corrects. En cas d'erreur syntaxique dans un holon, toute la séquence est refusée, car vu que l'ordre de présentation des holons est capital pour la construction du programme, il est inutile de tenter l'introduction dans un programme des holons - même syntaxiquement corrects - qui succéderaient dans la séquence à un holon syntaxiquement incorrect. De toute manière, nême si tous les holons de la séquence sont syntaxiquement corects, il est utile qu'avant de tenter d'introduire la séquence dans le programme, le compilateur transmette au programmeur le texte analysé pour demander confirmation du résultat de cette analyse; en effet, par exemple, l'oubli d'un point-virgule entre deux noms de holons ne conduit pas à un texte syntaxiquement incorrect quoique sa sémantique ne soit certainement plus celle que le programmeur lui attribue. Cette forme de renvoi du texte syntaxiquement correct au programmeur pour confirmation n'implique nullement que le compilateur soit nécessairement implémenté sous une forme interactive, le programmeur et le compilateur peuvent "converser" sans problème dans un système par "fœurnée" (batch process), vu l'existence du fichier général des programmes.

Supposons que la séquence syntaxiquement correcte de holons ait été approuvée par le programmeur; le compilateur doit entreprendre l'opération d'adjonction de cette séquence au programme que nous noterons (voir chapitre II.4) :

$$\mathsf{P[np,\{H_{0},\ H_{1},\ldots,H_{i},\ldots,H_{n}\}]} + \mathsf{S[H_{\alpha},H_{\beta},\ldots,H_{\epsilon},\ldots,H_{u}]}.$$

Cette opération n'est possible que si le nouveau programme qui en résulte est un programme légal. Pour vérifier cette condition, le compilateur envisage tout d'abord l'adjonction de  $H_{\alpha}$  au programme P et si le programme qui en résulterait est légal, l'adjonction de  $H_{\beta}$  au programme  $\{H_{\alpha},\ldots,H_{n},H_{\alpha}\}$  et ainsi de suite pour tous les éléments de la séquence S. Si aucune illégalité ne résulte de ces adjonctions simulées, le programme P est augmenté par le contenu de la séquence et devient :

Au contraire, toute illégalité détectée entraîne le rejet immédiat de la séquence S qui est renvoyée au programmeur avec les raisons justifiant ce refus. L'acceptation ou le refus est toujours sur base de la séquence entière, en effet cette série de holons doit correspondre au développement d'une partie du programme formant un ensemble d'opérations plus ou moins dépendantes (la description d'un holon et celles des holons définis dans cette description, par exemple), il est donc logique que son introduction dans le programme se fasse d'une manière globale.

L'illégalité de l'adjonction d'un holon à un programme résulte de la violation d'un (ou de plusieurs) des conditions énoncées au chapitre II.4. Parmi celles-ci, les conditions de "cohérence de structure et/ou d'environnement" n'ont pas encore été exposées, elles le sont ci-dessous.

# 3.4.1. Cohérence de structure.

La cohérence de structure concerne le respect des règles de structure des holons booléens (voir chapitre II.9) et des holons récursifs (voir chapitre II.8.2). En général, le holon que l'on désire adjoindre à un programme doit, parce qu'il correspond à la description d'un holon défini dans certaines conditions (apparition de la définition du holon dans la partie booléenne d'une instruction, par exemple), remplir certaines conditions imposées par sa définition.

# 3.4.2. Cohérence d'environnement.

La vérification de la cohérence de l'environnement est probablement l'opération la plus complexe du compilateur, mais elle est la plus importante pour garantir la construction systématique du programme.

La description d'un holon consiste en une série d'opérations appliquées à des objets de l'environnement; certains de ces objets sont créés lors de l'exécution du holon, mais les autres sont supposés appartenir à l'environnement tel qu'il existe lors d'un enclenchement du holon. Puisqu'un holon peut être enclenché en divers endroits du programme exécuté, il est nécessaire que le compilateur s'assure que chaque holon accède toujours à l'environnement qu'il requiert. En d'autres termes, l'association d'un identificateur à un objet de l'environnement suit la règle de l'association à l'élément de même nom dont la création est la plus récente (1); pour cela il est nécessaire que cet élément soit également

Par exemple, dans le programme ALGOL :

```
begin integer a,b;
procedure su(...);
...
    a:=a + b;
...
    end;
...
    begin integer x,a,b;
call1:su(...);
...
end
end
end
```

les variables "a" et "b" utilisées pour l'évaluation de "a:=a + b" lors de l'exécution de l'appel de la procédure "su" dans le bloc interne sont les entiers déclarés dans le bloc externe qui contient la déclaration de procédure; en SNOBOL ou en APL, il s'agirait des variables déclarées dans le bloc interne (les déclarations les plus récentes). Cette dernière méthode implique généralement la vérification à l'exécution des types des variables de chaque expression, afin d'utiliser la signification correcte d'un opérateur polymorphique; par exemple, si la déclaration dans le bloc interne était "real x,a,b;", le "+" devrait être interprété comme l'addition de deux nombres réels et non plus de deux entiers. Le choix de l'association la plus récente pour les variables non-locales est fréquement rejeté lors de la conception d'un langage à cause des vérifications à l'exécution que la méthode impose, (en SNOBOL et en APL, ces vérifications existeraient de toute manière, indépendamment du choix de cette

<sup>(1)</sup> Cette manière de traiter le problème des références à des variables non-locales à une "procédure" rapproche le HPL des langages comme le SNOBOL et l'APL et l'éloigne considérablement des langages de type-ALGOL où les références aux variables non-locales se font sur base de la structure "statique" du programme.

d'un type déterminé par la définition du holon.

Afin de détailler le processus de construction de l'environnement d'un holon et de vérification de celui-ci, nous allons symboliser cet environnement pour un holon "x" par :

 $E(x) : \{L,NL\},$ 

où L représente l'ensemble des variables locales et NL celui des variables non-locales.

La première apparition d'un nouveau nom de holon correspond à la définition d'un nouveau holon, ce nom de holon comprend généralement une série d'identificateurs (zid) et une série de paramètres (zp). Pour chaque élément de (zid), le compilateur vérifie qu'un élément de même nom existe dans l'environnement de l'apparition du nom de holon, le "type" de l'élément trouvé est associé à l'identificateur; de même pour la liste des paramètres, le compilateur peut déterminer le "type" de chacun des paramètres. L'ensemble des variables non-locales est, à ce stade de la construction du holon, défini par une série de doublets (nom, type) et (paramètre, type). Vu la structure d'un programme-holon, le compilateur peut déterminer exactement l'état de l'environnement en un point quelconque du programme en simulant l'effet des déclarations de tous les holons enclenchés dont l'effet-net est en cours de production (ensemble NG, voir II.14).

A la définition d'un holon, succéderont une série d'apparitions de ce nouveau nom de holon (lors de la description de nouveaux holons) et une description de ce holon; la description existera nécessairement, par contre la série d'apparitions du nouveau nom de holon peut être vide. Pour chaque apparition du nouveau nom de holon, le compilateur détermine l'environnement au niveau de l'apparition et vérifie si la série des doublets établie lors de la définition du holon et l'état de l'environnement sont co-

Supposons qu'un holon "x" défini (et non décrit) possède une liste B(x) d'apparitions dans un programme-holon P; et supposons que le compilateur entreprend la vérification de la description de ce holon "x" en vue d'en autoriser l'adjonction au programme. Cette description de "x" peut contenir des occurrences de noms de holons déjà décrits, il en résulte que l'état de l'environnement pour chaque élément de la liste B(x) doit être compatible avec les exigences de ces holons subordonnés directs de "x" dans sa description. Pour vérifier cette condition, il est nécessaire de déterminer l'état exact de l'environnement en cette occurrence de "x", de le modifier "provisoirement" en fonction des déclarations contenues dans la description de "x" et de voir si ce nouvel environnement correspond aux exigences des subordonnés de "x". Cette opération de vérification est à entreprendre pour chaque élément de B(x) et pour toute future apparition du nom de "x" dans toute description ultérieure d'un holon; en même temps, on vérifie si les différents contextes de "x" peuvent accueillir les déclarations non-locales (at...level,same...as) de la déclaration de "x" (voir ch. IV.5.3.). (Les vérifications pour les opérations create et refine que pourrait contenir la description de "x" ne doivent être faites qu'une seule fois car l'ensemble des types forme une entité globale à tout le programme].

A première vue, la vérification de la cohérence de l'environnement peut paraître une opération longue et complexe; toutefois, il faut remarquer que sa complexité est proportionnelle au "risque" impliqué par la décision du programmeur; en effet, on peut dire que la décision la plus "risquée" est de décrire une opération qui apparaît en de multiples endroits du programme au moyen d'autres opérations déjà décrites; en fait, la vérification correspond à un travail mental que le programmeur devrait de toute manière accomplir.

D'autre part, il ne faut point perdre de vue que ces vérifications permettent de se passer de la multitude de tests à l'exécution qui caractérise les langages adoptant le principe de l'association la plus récente. On peut considérer que la vérification de la compatibilité de l'environnement est en fait une composante obligatoire du "datatest" de chaque holon, la "vérité" de cette composante est vérifiée à la compilation car la structure des programmes HPL le permet.

Un autre point de vue est de considérer que l'adoption du principe de l'association la plus récente est une erreur et qu'il aurait mieux valu adopter une méthode basée sur la structure "statique" du programme, comme en ALGOL; malheureusement, quels que soient les avantages que l'on peut attribuer à cette méthode pour l'efficacité du code généré, la méthode a un inconvénient majeur : l'obligation de devoir "déclarer" les procédures, c'est-à-dire de devoir décider d'une "portée" pour les opérations que l'on crée. En effet, le but d'une déclaration est de limiter le contexte d'utilisation d'un objet; dans le cas de création d'une opération primitive, décider à priori le contexte exact d'utilisation de l'opération n'est pas une décision simple; s'il apparaît qu'un contexte trop restreint a été attribué à une opération, on se voit dans la nécessité d'élargir le centexte en rapportant la déclaration à un niveau supérieur dans le programme, mais un tel transfert n'est pas aussi simple que celui d'une déclaration de variable, car l'opération est fréquemment définie en fonction de variables déclarées à un niveau intermédiaire entre le niveau actuel de l'opération et le nouveau niveau que l'on voudrait attribuer à l'opération; la méthode utilisée en HPL implique indirectement une portée à une opération, puisque le programmeur peut associer étroitement une opération à l'existence de certaines variables; la différence majeure avec le principe-ALGOL est que la portée d'une opération résulte du travail de construction du programme et non de l'obligation de prendre une décision qui risque de se révéler prématurée.

# 3.5. Erase.

La requête <u>erase</u> constitue l'opération inverse de <u>append</u>; elle comporte une liste de noms de holon qui sont supposés appartenir au programme auquel le programmeur a obtenu accès, le but de la requête est d'obtenir l'effacement des holons cités. On peut symboliser l'opération par :

P[np, 
$$H_0$$
,  $H_1$ ,..., $H_n$ ] -  $EL[nh_a,nh_b,...,nh_k]$ ,

où EL est la liste des holons que l'on désire effacer.

L'opération <u>erase</u> correspond à la décision souvent la plus lourde de conséquences pour l'histoire d'un programme : la modification du programme. Cpération dont le succès est fréquemment aléatoire. Dans le cas du HPL, cette opération ne peut avoir lieu que sous le contrôle du compilateur afin d'éviter tout résultat anarchique. L'origine de la difficulté et du danger d'une opération d'effacement provient, quel que soit le langage, de la difficulté de déterminer avec précision quelles sont les parties du programme dont la contribution à la production de l'effet-net du programme dépend de l'existence de la partie que l'on se propose d'effacer.

Supposons que dans le programme-holon P, on se propose d'effacer le holon  $H_i$ , à cause du principe de construction d'un programme-holon, nous sommes assurés que la suppression de  $H_i$  est sans effet notoire sur les holons  $H_0$  à  $H_{i-1}$ ; les seules conséquences pour ces holons concernent une modification possible de leur état, si par exemple la description de  $H_i$  entraîne la récursivité d'un holon précédent, mais cette modification ne compromettra jamais la cohérence de tout le programme. Par contre, il est très probable que des holons appartenant au sous-ensemble  $\{H_{i+1},\ldots,H_n\}$  ont été admis dans le programme-holon P parce que  $H_i$  existait. Le compilateur doit donc vérifier si cette éventualité existe et refuser l'effacement de  $H_i$  si d'autres holons dépendent de son existence. Dans ce cas, le programmeur devra d'abord requérir l'effacement de ces holons pour obtenir l'autorisation d'effacer  $H_i$ ; d'autres holons dépendent peut-être de l'existence de ceux-ci, et en procédant de cette manière, on peut déterminer exactement tous les holons qui dépendent de l'existence de  $H_i$ .

Le compilateur commence par trier par ordre décroissant les éléments de la liste EL, l'ordre étant fixé par les numéros d'ordre dans le programme-holon P des holons dont les noms apparaissent dans EL; tout élément de EL qui ne correspond à aucun holon de P est ignoré. Les éléments de EL sont triés selon l'ordre décroissant afin d'envisager d'abord l'effacement du holon le plus proche de  $H_n$ , c'està-dire celui dont dépend a priori le moins de holons. Supposons que  $\mathrm{EL}[\mathrm{nh}_a,\mathrm{nh}_b,\ldots,\mathrm{nh}_k]$  respecte cet ordre décroissant; s'il s'avère que  $H_a$  peut être effacé sans problème, le compilateur envisage l'effacement de  $H_b$  en considérant que  $H_a$  n'existe plus dans le programme-holon. Si des holons dépendent de l'existence d'un holon à effacer, le programmeur en est averti et la requête n'est pas exécutée. L'exécution de la requête n'a lieu que si tous les éléments de EL peuvent être effacés.

Supposons que l'on envisage l'effacement d'un holon  $\mathrm{H}_i$ , des holons appartenant à  $\{\mathrm{H}_{i+1},\ldots,\mathrm{H}_n\}$  peuvent dépendre de  $\mathrm{H}_i$  pour des raisons soit de "structure" du programme soit d'environnement. La vérification procède comme suit :

# 1) Structure du programme.

Pour chaque nom de holon  $\mathrm{nh}_{i}$  qui appartient à la description de  $\mathrm{H}_{i}$ , on teste si ce nom correspond à

un holon  $H_j$  tel que "j" soit supérieur à "i"; si le résultat du test est positif, on examine l'éventualité d'une occurrence du nom de holon  $nh_j$  dans l'un (ou plusieurs) des holons  $\{H_0, H_1, \ldots, H_{j-1}, H_{j+1}, \ldots, H_j\}$ ; il est nécessaire qu'une telle occurrence existe pour qu'il existe une définition de  $H_j$  après l'effacement de  $H_j$ ; si le résultat du test est négatif,  $nh_j$  correspond soit à un holon "défini" mais non "décrit", soit à une occurrence d'un holon décrit avant  $H_j$ ; dans ce dernier cas, la seule conséquence éventuelle de l'effacement de  $H_j$  pourrait être la suppression du caractère récursif d'un holon  $H_c$ , avec c < i.

# 2) Modifications de l'environnement.

Les commandes <u>create</u>, <u>refine</u> et <u>declare</u> qui appartiennent éventuellement à  $H_i$ , impliquent des modifications de la structure de l'environnement; le holon  $H_i$  peut être effacé si aucune description de holon ultérieure à celle de  $H_i$  n'utilise les éléments de l'environnement introduits par  $H_i$ . La méthode de vérification dépend de la commande:

# A. Create.

Il faut vérifier que les "types abstraits", créés dans  $H_i$ , n'ont pas été utilisés dans l'un des holons  $\{H_{i+1},\ldots,H_n\}$ . Le type abstrait créé peut apparaître soit dans un <u>refine</u>, soit dans un <u>declare</u>, soit (s'il a déjà été utilisé dans l'un des deux cas précédents) dans un identificateur. Tout holon contenant ces éléments dépend de l'existence de  $H_i$ , et empêche son effacement.

# B. Refine.

S'il existe une commande " $\underline{\text{refine}}$  x  $\underline{\text{by}}$  y;", les holons  $\{\text{H}_{\underline{\text{i+1}}},\ldots,\text{H}_{n}\}$  doivent être examinés afin de repérer tout identificateur qui serait de forme "...x.y...".

# C. Declare.

La procédure de vérification dépend de la forme de la déclaration; le but est de déceler si les variables déclarées sont utilisées en dehors du holon.

### 1) Declare local.

Pour les variables déclarées locales au holon  $H_1$ , aucune vérification n'est effectuée, car de toute manière on ne peut déceler aucun holon - qui dépendrait de ces déclarations - qui n'ait pas été repéré pendant la vérification de la structure du programme.

# 2) Declare at level.

Supposons que le holon H; contienne la déclaration :

# "declare x at H level;", où c < i.

La vérification procède par l'examen des subordonnés directs ou indirects de  ${\rm H_c}$  afin de vérifier qu'il n'y a pas d'occurrence de "x" parmi ces subordonnés. Toutefois, tous les subordonnés ne doivent pas être examinés; en effet, le subordonné  ${\rm H_i}$  de  ${\rm H_c}$  ne doit pas être examiné, de même que tout subordonné qui contiendrait une nouvelle déclaration de "x", ainsi que tous les subordonnés dont la description est antérieure à celle de  ${\rm H_i}$ .

# 3) Declare restricted.

Dans ce cas, il est nécessaire de vérifier qu'il n'existe aucune déclaration  $\underline{same}$   $\underline{as}$  pour cette variable dans les holons  $H_i$ , avec j > i.

# 4) Declare same as.

L'effet d'une déclaration de ce genre est purement local, et pour les mêmes raisons que celles invequées pour une déclaration locale, aucune vérification ne doit être entreprise.

Une possibilité supplémentaire est offerte au programmeur, à savoir <u>erase virtual</u> : dans ce cas, le compilateur envisage la possibilité d'effacement d'un holon, mais il n'exécute pas la requête, même si aucun holon ne dépend du holon à effacer. Cette requête est très utile pour déterminer l'ef-

fet d'une décision antérieure comme une déclaration <u>at...level</u> ou un <u>refine</u>; elle permet de déterminer jusqu'à quel point le choix fait a influencé la composition du programme.

On remarquera que la complexité de la vérification d'un effacement est proportionnelle à l'antériorité de la décision de description du holon et aux interactions de la décision sur l'environnement; effacer  $H_n$  ne pose jamais de problème, par contre  $H_o$  présente le maximum de dépendances. Les opérations de vérification menées par le compilateur ne font que simuler les opérations que doit faire un programmeur lors d'une suppression d'une partie de programme. On remarquera spécialement le rôle de "declare at level" : de la même manière qu'il permet de compléter une décision antérieure en fonction du développement du programme, il peut sans dommage pour la décision antérieure être supprimé par l'effacement du holon qui contient la décision d'inclure cette déclaration.

L'organisation du programme sous la forme d'un ensemble ordonné de holons, la déclaration <u>at level</u> et les possibilités offertes par la requête <u>erase</u> font que le Holon Programming Language est, à notre connaissance, le seul language de programmation à respecter la "polychromie des programmes" : tout programme est formé par un arrangement d'instructions qui résultent de différents stades de décision lors du développement du programme; si une couleur est attribuée à chaque stade de décision et si toutes les instructions qui résultent de ce stade de décision sont composées dans cette couleur, le programme final sera composé d'une multitude de taches de couleur; en cas de modification ou de correction, il est plus sûr de travailler à partir du programme "polychrome" plutôt qu'à partir d'une version "monochrome". (1) En HPL, toutes les taches de même couleur peuvent être extraites du programme par une requête <u>erase</u>.

# 3.6. Change.

La requête <u>change</u> est une composition d'une opération <u>erase</u> immédiatement suivie d'une opération <u>append</u>. Il advient fréquemment que la modification que nous désirons appliquer à un programme ne soit pas l'effacement pur et simple d'un holon, mais le remplacement d'un holon par une version quelque peu modifiée; dans ce cas, <u>erase</u> ne convient pas, car la version modifiée peut "récupérer" tous les holons qui dépendent de l'existence de l'ancienne version : par exemple, si nous voulons remplacer un holon par une version semblable à l'ancienne mais qui contiendrait quelques instructions-holon en plus. Dans ce cas, on effectue une requête <u>change</u>; celle-ci contient un élément : la nouvelle description d'un holon H<sub>i</sub> supposé appartenir au programme auquel on a obtenu l'accès.

Le compilateur vérifie tout d'abord si un holon de même nom appartient effectivement au programmeholon et si la nouvelle version proposée est syntaxiquement correcte; si ces conditions sont vérifiées, il entame l'analyse du changement proposé.

Supposons que le holon que l'on désire changer soit  $H_i$  et que la nouvelle description soit  $N_i$ ; le compilateur construit la "différence" entre  $H_i$  et  $N_i$ , c'est-à-dire qu'il répertorie les éléments de  $H_i$  qui ne se retrouvent pas dans  $N_i$  et les éléments de  $N_i$  qui n'existent pas dans  $H_i$ ; pour les premiers éléments, il vérifie les possibilités d'exécution d'un <u>erase</u> et pour les deuxièmes, celles d'un <u>append</u>. Si ces vérifications sont positives, le compilateur envisage les changements structuraux qui pourraient résulter des éléments communs aux deux versions mais qui n'occuperaient pas des positions identiques dans la structure du holon; par exemple, si  $H_i$  est un holon booléen contenant l'instruction-holon :

# if A then C ; D else B ; M fi,

et que N; contienne à la place de cette instruction :

# if A then C ; D else M ; B fi,

il en résulte que M a perdu son caractère booléen au profit de B; ce changement est inapplicable si B et M ont été décrits, car la vérification de la description d'un holon tient compte de son caractère.

Si toutes ces vérifications sont positives, la requête change est exécutée, sinon les raisons du

<sup>(1)</sup> Cette métaphore est due à Seegmüller [See74], inspiré d'un texte de Woodger [Woo71]. Voir également (chapitre IV.5.2.).

refus de l'exécuter sont transmises au programmeur.

Comme pour la requête <u>erase</u>, le programmeur peut envisager un <u>change virtual</u> afin de vérifier les possibilités de modification d'un programme.

# 3.7. Synthetise.

Cette requête ne peut être appliquée qu'à un programme-holon complet (c'est-à-dire que tous les holons définis ont été décrits). Le résultat de la requête est la production par le compilateur d'une version exécutable du programme-holon; le principe général de cette opération est exposé dans une section ultérieure.

### 3.8. Execute.

Cette requête n'est applicable qu'à un programme-holon pour lequel une version "machine" a été produite par un <u>synthetise</u>; le résultat de la requête est l'exécution du programme.

# 3.9. Analyseur et synthetiseur.

On considère que la partie du compilateur qui s'occupe du traitement des six premières requêtes décrites ci-dessus forme l'analyseur; par contre, les requêtes <u>synthetise</u> et <u>execute</u> sont traitées par le synthétiseur.

L'analyseur rassemble toutes les informations sur la structure du programme et les dispose selon un format accepté par le synthétiseur; celui-ci transforme la représentation du programme composée par l'analyseur en un code exécutable.

### 4. LE SYNTHETISEUR

# 4.1. Choix d'une méthode d'implémentation pour un holon.

La première tâche du synthétiseur est le choix d'une forme d'implémentation pour chaque holon. Celle-ci dépend du caractère récursif ou non du holon, et si le holon est non-récursif de la fréquence d'apparition de ce holon dans la description du programme et de la densité de code.

La distinction entre holon récursif et non-récursif est inévitable puisque l'implémentation d'un tel holon doit pourvoir à la simulation du processus de récursion pour lequel le "hardware" n'est généralement pas conçu. Par contre, dans le cas du holon non-récursif, il n'y a aucune nécessité de simuler l'équivalent d'une expansion de texte qui n'aura jamais lieu; cependant, on peut choisir de mettre en oeuvre un holon non-récursif soit sous forme d'une procédure, soit sous forme de l'équivalent d'une macro-génération. Dans l'un et l'autre cas, un holon est représenté par une séquence de code qui correspond aux opérations accomplies par ce holon, cette séquence étant accompagnée par une description des modifications à opérer pour que l'environnement corresponde à l'environnement attendu par le holon. Dans le cas de la procédure, il n'existe qu'une copie de la séquence de code à laquelle le programme se réfère, chaque fois que cela est nécessaire, en effectuant les modifications d'environnement. Dans le cas de la macro, la séquence de code et les opérations de modifications d'environnement sont "recopiés" pour chaque utilisation et appartiennent sans solution de continuité au programme "principal". La différence essentielle entre les deux techniques tient au fait que la procédure permet de repousser jusqu'au moment de l'exécution la décision de "lier" certains objets de l'environnement et une opération, alors que dans le cas de la macro, cette "liaison" a lieu lors de la génération du code; différer l'instant de cette "liaison" permet d'accroître les possibilités d'adaptation de la même séquence de code à différents environnements mais au prix d'une certaine diminution de l'efficacité.

La discussion précédente a exposé les avantages et les inconvénients du choix entre "procédure"

et "macro"; le synthétiseur doit opérer ce choix automatiquement, ses critères de décision sont basés sur l'opportunité d'utiliser ou non la souplesse d'utilisation que permet la procédure. Les critères de décision peuvent être groupés dans une table, en fonction de la fréquence d'apparition du holon dans le texte du programme <sup>(1)</sup> et en fonction de la longueur du code généré pour l'opération par le holon :

longueur du c⊙de généré	fréquence d'apparition		
	très fréquent	peu fréquent	une occurrence
long	procédure	procédure	macro
court	macro	macro	macro

La valeur pivot entre "très fréquent" et "peu fréquent" et entre "long" et "court" doit être fixée selon la machine pour laquelle le code est généré en fonction du "coût" en temps et en espace-mémoire d'une opération d'accès (et de retour) à un sous-programme. Le choix peut être guidé par d'autres considérations qui viennent se superposer à la fréquence et à la longueur du code; par exemple, l'occurrence à l'intérieur d'une instruction d'itération peut impliquer une préférence pour une implémentation sous forme de macro, pour cette occurrence particulière.

Les opérations "terminales" appartiennent généralement au "créneau" : "court-très fréquent"; les holons conçus par le programmeur ont tendance (du moins d'après notre expérience) à appartenir aux "créneaux" "court-très fréquent" et "long-peu fréquent"; dans le premier cas, on peut considérer que le programmeur a fait émerger les opérations qui forment le langage terminal approprié à l'application traitée.

Le choix de la méthode d'implémentation dépend de la longueur du code généré pour le holon, il est donc nécessaire de posséder cette information ou tout au moins une estimation de cette longueur. La fréquence d'apparition est connue d'office par examen des informations réunies par l'analyseur. Pour les opérations "terminales", le choix d'implémentation a été fixé d'une manière définitive lors de la conception des opérations terminales.

# 4.2. Détermination de la longueur du code d'un holon.

Le synthétiseur rassemble tous les holons récursifs dans un ensemble R; ensuite il rassemble dans deux sous-ensembles M et I les holons dont le nombre d'occurrences est supérieur à un et ceux qui n'ont qu'une seule occurrence respectivement; par occurrence, on entend occurrence du nom du holon dans les descriptions de holons qui forment le programme. Les ensembles I et M sont ensuite triés afin de former deux séries de sous-ensembles  $\{I_0,I_1,I_2,\ldots\}$  de I et  $\{A_0,A_1,A_2,\ldots\}$  de M. Ce tri est effectué de la manière suivante : le sous-ensemble  $I_0$  contient tout élément de I dont la description ne possède comme subordonnés que des instructions terminales ou des éléments appartenant à R; dès que  $I_0$  est construit, on peut construire  $A_0$  comme étant formé par les éléments appartenant à R ou à I; la construction des sous-ensembles progresse par une succession de construction de  $I_1$  suivi par celle d'un  $I_1$ ,  $I_2$  contient tout élément de I dont la description ne possède que des instructions terminales et/ou des éléments appartenant à R ou à un  $I_2$  ou à un  $I_3$  (avec o  $I_4$   $I_4$ ). A contient tout élément de M dont la description ne possède que des instructions terminales et/ou des éléments appartenant à R ou à un  $I_3$  (avec o  $I_4$   $I_4$ ); la construction est achevée dès que tout élément de M a été attribué à l'un des sous-ensembles  $I_4$ .

Dès que cette séparation des holons du programme est terminée, la génération du code peut commencer

<sup>(1)</sup> Il ne s'agit pas de la fréquence d'exécution, mais du nombre d'occurrences dans le texte du programme qui correspond à un arbre "minimal" où il n'existe aucune recopie d'un sous-arbre. Cette distinction a pour résultat de réduire le nombre de procédures générées pour un programme, en effet, le holon qui appartient à un holon "très fréquent" est en fait considéré comme "peu fréquent" et implémenté sous la forme d'une "macro" à l'intérieur de ce holon.

par le code des holons appartenant à  $A_0$ ; en effet, cette génération ne présente aucune difficulté puisqu'elle n'implique que l'expansion du code d'instructions terminales ou de holons qui appartiennent à  $I_0$  (ceux-ci sont d'office implémentés sous forme de macros) et d'éventuels appels à des holons appartenant à  $R_i$  dès que la génération du code pour un holon appartenant à  $A_0$  est terminé, la longueur du code généré est connue et le type d'implémentation pour le holon peut être déterminé. A ce stade, la génération du code pour les holons appartenant à  $A_1$  se fait sans difficulté puisque ces holons ne contiennent que des instructions terminales, des expansions de macros pour des holons appartenant à  $I_0$  et à  $I_1$ , des appels éventuels à des holons récursifs et des expansions (ou appels selon le cas) de holons appartenant à  $A_0$ ; la génération du code pour les holons appartenant à  $A_2$ ,  $A_3$ ,... est effectuée de la même manière.

Dès que la génération est terminée pour tous les holons appartenant à M, la synthétiseur génère le code pour les holons récursifs (dont la description ne contient que des instructions terminales, des holons appartenant à I ou à un  $I_i$  ou à M et des appels à un ou des holons récursifs). Après cela, le synthétiseur peut parcourir directement la description du holon  $H_0$  en générant systématiquement le code car la méthode d'implémentation de tout holon rencontré pendant ce parcours a été déterminée.

# 4.3. Implémentation des modifications d'environnement.

A l'exécution, les modifications d'environnement sont dues à l'introduction dans l'environnement de variables déclarées dans un holon lors de l'enclenchement de ce holon, et à l'expulsion de ces mêmes variables lorsque ce holon a produit son effet-net. Il est également nécessaire de pourvoir à l'existence simultanée dans l'environnement de variables de même nom et de garantir que l'accès est toujours accordé à la variable dont la création est la plus récente.

Toute référence à une variable se fait à l'exécution par l'intermédiaire d'une "table des noms"; cette table comporte une entrée par nom différent appartenant au programme, chaque entrée contient un "indicateur d'activité" et un pointeur vers la mémoire contenant la "valeur" de la variable ou bien directement la "valeur" de la variable. Ce dernier cas correspond à une variable dont le type est "simple": <a href="character">character</a>, <a href="integer">integer</a>, <a href="pointer">pointer</a>,... c'est-à-dire une variable telle que sa valeur occupe autant (ou moins) de mémoire qu'un pointeur. Le premier cas correspond à une variable de type "complexe" dont le nom sert de base à tout identificateur d'un élément terminal de la variable (voir chapitre IV.6); le pointeur de la table des noms correspond en fait à l'adresse de base de la représentation en mémoire de la "valeur" de la variable.

Dans le texte d'un holon, tout identificateur est transformé en un pointeur vers l'entrée dans la "table des noms" qui correspond à son nom, et en une expression permettant de calculer le déplacement dans la représentation de la variable correspondant à l'élément terminal indiqué par l'identificateur. Si plusieurs holons différents contiennent une variable de même nom, tous les identificateurs de ces deux holons pour cette variable pointent vers la même entrée dans la "table des noms"; l'expression du déplacement pouvant être différente, selon le type de la variable dans chaque holon.

On peut considérer que tout holon contenant des déclarations de variables possède un datatest libellé comme suit :

### datatest

il n'y a pas de conflits entre les noms des variables à déclarer et les noms des variables actives de l'environnement

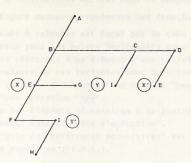
else résoudre ces conflits; <u>resultis</u> <u>true</u> <u>fi</u>

end

Pour chaque nom de variable à déclarer, on vérifie si "l'indicateur d'activité" est "on" ou "off"; s'il est "on", la valeur correspondante du nom est "sauvée" dans une "pile des références non-actives" et la nouvelle valeur prend sa place dans la table; s'il est "off", la nouvelle valeur est directement introduite dans la table et l'indicateur est placé sur "on". Lorsque le holon a produit son effet-net, les valeurs introduites dans la "pile des références non-actives" sont transférées de cette pile à la "table des noms".

Cette opération de modification est présentée sous la forme d'un "datatest" car elle peut être simplifiée dans le cas des holons dont le code généré ou la séquence d'appel est directement intégrée à H<sub>0</sub>; en effet dans ce cas, le synthétiseur peut simuler, lors de la génération du code pour H<sub>0</sub>, les activations ou désactivations des noms de variable; il en résulte que le test de "l'indicateur d'activité" peut être effectué au moment de la génération du code, et non plus à l'exécution. L'effet de cette simplification est très important, car le nombre de <u>procédures</u> emboîtées est généralement faible (par contre le nombre de <u>holons</u> emboîtés est généralement très grand) vu l'intégration des holons à la génération; la grande majorité des appels de procédure appartiennent donc à H<sub>0</sub> et cette simplification leur est applicable. Cette simplification n'est évidemment pas applicable aux macros et aux procédures qui appartiennent à d'autres procédures.

# Exemple :



Les holons E et I ont deux occurrences dans le programme. Les lettres cerclées désignent les environnements à l'enclenchement. Lors de la génération du code pour  $H_0 \equiv A$ , il est possible d'appliquer la simplification pour les environnements (X) (Y) (X); mais il est impossible de faire de même en (Y) puisque la valeur de (Y) varie en fonction de l'occurrence de E qui est enclenchée.

# 5. POSSIBILITES D'OPTIMISATION DU CODE

L'efficacité du code généré peut être accrue par l'optimisation du code; nous citerons sans les détailler trois possibilités d'optimisation : résolution des chaînes de "resultis", déplacement des déclarations et transformation des holons récursifs.

### 5.1. Résolution des chaînes de "resultis".

Lorsqu'une partie d'un programme est consacrée à l'analyse de données (voir le deuxième exemple du chapitre VIII), il est fréquent de voir apparaître des "cascades" de "resultis", c'est-à-dire que l'opération enclenchée après le "resultis" qui termine un holon booléen est également un "resultis"; cette situation résulte de la structure du holon booléen. Le code généré pour un "resultis" est un "goto"; dans le cas d'une chaîne de "resultis", on peut remplacer le premier "resultis" de la chaîne par l'équivalent en code généré du dernier "resultis" de la même chaîne.

### 5.2. Déplacement des déclarations.

Le but de cette opération d'optimisation est de faire "remonter" vers le premier niveau d'un holon "procédure" toutes les déclarations de variables dues aux holons "macros" qu'il contient. Cette opération est particulièrement utile si on utilise des opérations terminales très élaborées qui doivent déclarer de nombreuses variables. La conséquence de cette forme d'optimisation est d'améliorer considérablement la possibilité d'éviter le test sur l'état "actif" ou non d'un nom lors de la génération du cœle de H<sub>o</sub>.

# 5.3. Transformation des holons récursifs.

Certaines techniques de transformations de procédures récursives sont connues, elles sont en général basées sur une correspondance entre une opération récursive et une instruction "while" :

est équivalent à :

# while E do C od.

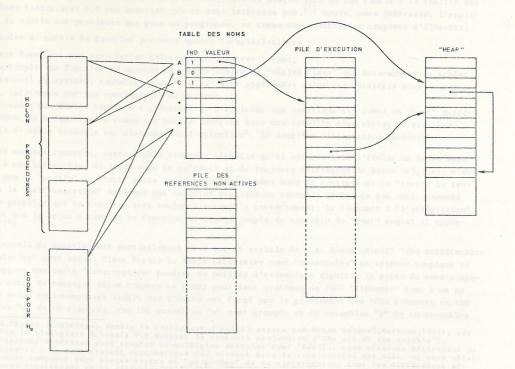
L'application de ces transformations exige de distinguer les procédures récursives des autres et de découvrir, parmi celles-ci, celles qui respectent certaines règles de structure. (1) La structure régulière d'un programme-holon permet de respecter ces deux conditions.

### 6. MACHINE VIRTUELLE POUR LE HPL

On peut considérer que les différentes structures de données et algorithmes utilisés pour l'exécution d'un programme sur une machine donnée correspondent à une machine "virtuelle" destinée à l'exécution d'instructions écrites dans le langage du programme.

La figure suivante représente les structures de données d'une telle machine pour le HPL:

- le code à exécuter est formé par le code généré pour H<sub>o</sub> et par les différentes sections de code générées pour les holons implémentés sous la forme de procédure.
- 2) toute référence à un élément d'une variable se fait par l'intermédiaire de la "table des noms"; les valeurs de ces variables sont soit inscrites directement dans la table des noms, soit contenues dans la "pile d'exécution" ou dans la "heap" selon que leur création résulte d'une déclaration ou d'une opération "new".
- 3) tous les éléments nécessaires à la gestion des holons "procédure" lors de l'exécution sont insérés également dans la "pile d'exécution".
- 4) la "pile des références non-actives" est utilisée pour la gestion des modifications d'environnement comme exposé en (VI.4.3.).



<sup>(1)</sup> Le lecteur intéressé par l'élimination de la récursivité peut consulter [Knu74, p. 280-282] ct [Bur73].

### CHAPITRE VII

LE HOLON

# 1. INTRODUCTION

Ce chapitre est destiné à l'explication du terme "holon" et des raisons qui nous ont amené à l'utiliser pour signifier l'élément de base du langage. La correspondance de la notion de "holon" avec la pratique de "l'intégration" sera explicitée.

# 2. LE MODELE "HOLON" DE KOESTLER

Le "holon" est un modèle d'études des systèmes complexes biologiques et sociaux proposé par A. Koestler [Koe67, Koe69]; selon ce modèle, les systèmes complexes susceptibles d'évoluer d'une manière stable sont composés d'éléments dont la fonction est bien définie, mais qui possèdent la faculté de réaliser correctement et efficacement cette fonction pour toute une large gamme de variations de leur environnement, en s'adaptant selon les circonstances. Ces éléments sont des "holons".

Le terme "holon" est forgé à partir du mot grec "holos", signifiant "tout" (dans le sens "un tout", "une totalité"), mot auquel est adjoint le suffixe "-on" pour suggérer une partie; la signification est donc "partie d'un tout", en comprenant que le terme "partie" possède simultanément les significations de "participant à" et "élément de".

Le modèle est destiné - dans l'esprit de son auteur - à proposer une nouvelle manière d'envisager les phénomènes biologiques (notamment l'évolution); que ce modèle soit ou non fidèle à la réalité des phénomènes biologiques est une question qui ne nous intéresse pas. (1) Seule, nous intéresse, l'application du modèle aux problèmes que pose un programme, vu comme une association complexe d'éléments.

Le holon du modèle de Koestler présente quatre caractéristiques :

- 1) il est formé par une association hiérarchique d'autres holons;
- 2) il accomplit sa fonction en suivant un nombre limite de "règles fixes" qui déterminent le schéma fonctionnel du système; toutefois, les "règles" sont appliquées selon une stratégie plus ou moins fixe, influencée par les contingences du milieu;
- 3) il possède une double tendance à préserver et à affirmer son individualité comme un "tout" quasiautonome, et à fonctionner comme une partie intégrée dans une totalité plus vaste; la première partie de cette tendance est dénommée "self-assertion", la deuxième "integrative tendency".

D'une manière concrète, cette double tendance signifie qu'il est possible d'isoler un holon appartenant à une hiérarchie, de l'extraire de celle-ci et de toujours distinguer le holon original, c'està-dire que la "partie" du "tout" reste toujours définie, même hors du contexte de ce "tout". La tendance à la "self-assertion" implique que le holon pratique un contrôle sévère de son environnement afin de garantir que sa fonction sera toujours accomplie correctement; la tendance à "l'intégration" implique que le holon accomplit sa fonction en tenant compte du contexte du "tout" auquel il appartient. (2)

Le travail de Koestler est partiellement basé sur un article de H.A. Simon [Sim62] "The architecture of complexity" dans lequel Simon évalue le temps nécessaire pour "construire" un système complexe si l'on suppose que toute "interruption" pendant la période d'assemblage signifie la perte du sous-composant en cours de montage: Simon compare ce temps pour deux systèmes de 1000 "éléments" dont l'un ne possède aucun sous-composant tandis que l'autre est formé par le groupement des 1000 éléments en 100 ensembles "a" de 10 éléments, ces 100 ensembles "a" sont groupés en 10 ensembles "b" de 10 ensembles

<sup>(1)</sup> W.H.Thorpe (Cambridge) semble le cautionner ("Animal nature and human nature", Methuen, 1974), par contre le biologiste écossais P.B.Medawar le conteste violemment ("The art of the soluble").

(2) Le lecteur intéressé peut trouver de multiples exemples dans [Koe67]; nous citerons brièvement un de ceux-ci : le comportement systématique des oiseaux dans la construction des nids; on peut effecquences constituent en la sélection en "programme" de la construction, dont les différentes séquences constituent en la sélection et la mise en oeuvre de matériaux possédant des propriétés bien définies pour chaque séquence, mais le type de ratériau ne semble pas fixé comme en témoigne l'inclusion de matériaux artificiels (plastique, ficelle,..); en fait, ces schémas de description de construction ressemble à de véritables algorithmes dont les opérations terminales seraient du genre: "ramener un matériau solide et flexible"...

"a", les 10 ensembles "b" sont finalement regroupés pour former le système final. Le deuxième système (hiérarchisé) requiert cent et une fois plus d'opérations d'assemblage que le premier système; Simon montre que, pour une probabilité p = 0.01 qu'une interruption ait lieu pendant qu'on ajoute un élément au système (ou à un sous-composant), on peut construire quatre mille systèmes hiérarchisés dans le temps mis pour assembler un système non-hiérarchisé. Il en conclut qu'un système complexe ne peut être conçu (cu compris) que par l'introduction d'une hiérarchie. Les systèmes complexes sont formés par une hiérarchie d'éléments stables, ou par des systèmes "presque décomposables". Ce dernier type de systèmes est défini comme étant un système hiérarchisé dont les interactions entre les soussystèmes sont faibles quoique non-négligeables. Dans un système de ce type, le comportement à courtterme d'un sous-système est presque indépendant du comportement à court-terme des autres composants, par contre le comportement à long terme de chaque sous-système dépend d'une manière globale du comportement des autres sous-systèmes.

Si on supprime, dans le modèle de Koestler, la propriété des holons d'avoir une "tendance à l'intégration", on retrouve le modèle de Simon. Mais, c'est précisément l'existence de la "tendance à l'intégration" qui fait l'attrait du modèle de Koestler comme guide pour la construction d'un système complexe.

# 3. LE PRINCIPE D'INTEGRATION

En pratique, le temps de conception (de "construction") d'un système complexe n'est pas le seul élément qui entre en jeu dans les facteurs d'appréciation des qualités d'un système; en fait, dès que le système existe, peu importe le mode de conception, seul importe le comportement du système lors de son utilisation; un facteur important de ce comportement est "l'efficacité" de fonctionnement c'est-à-dire que pour la réalisation de la même fonction, on donnera la préférence à une version plus rapide ou meins prodigue de l'espace-mémoire. L'existence d'une structure impose fréquemment un cloisonnement excessif dans l'accès à certaines fonctions et oblige parfois à réaliser ces fonctions d'une manière détournée et manifestement peu "efficace". En fait, d'une part, nous ne pouvons nous passer de la "structure" pour la conception du système complexe et pour l'organisation de modification ultérieures de celui-ci, mais d'autre part, l'existence de cette "structure" est fréquemment un obstacle à l'utilisation pratique du système produit.

Ce qui précède peut paraître évident, toutefois la majorité de nos manuels de programmation s'efforce de répandre l'idée que la "procédure" est à la disposition du programmeur pour structurer son programme, ou selon une idée plus récente, qu'il faut procéder par la conception d'une hiérarchie de "classes" dont chacune définit un "niveau d'abstraction " formé par des structures de données abstraites et une collection d'opérateurs définis sur ces structures de données  $\binom{1}{}$ . Après quelques essais pratiques, le programmeur comprend vite que "l'overhead" de la procédure impose une limite à la "finesse" de la structure, ou dans le deuxième cas qu'il est pratiquement impossible dans les applications un peu complexes de délimiter des "niveaux d'abstractions". (2)

A notre avis, la condition primordiale que doit vérifier un procédé de structuration d'un programme est l'indépendance vis-à-vis de toute méthode systématique d'implémentation. La structure apparente à la conception doit céder le pas à des objectifs d'efficacité et "d'économie" dans l'implémentation. Le problème dans le cas des systèmes complexes est de pouvoir laisser les opérations de choix d'implémentation à un système automatique qui n'ignorera aucune des contingences de l'opération d'intégration des éléments.

Le rejet de programmes "structurés", basé sur des considérations de manque d'efficacité n'est pas une erreur : même si le programme est parfaitement correct, il ressemble plus à un prototype de laboratoire où on peut discerner tous les composants qu'à un objet utilisable quotidiennement; la production d'objets "intégrés" est un principe d'enginérie inséparable de la structuration du processus de conception. En génie civil, ce principe est connu sous le nom de "critère de Shanley" : rassembler plusieurs fonctions dans le même élément de construction. On peut voir ce principe à l'oeuvre dans le

<sup>)</sup> Voir annexe A.

<sup>(2)</sup> Voir [Par75] où il est exposé que l'utilisation de machines "virtuelles" implique fréquemment une perte de "transparence" vis-à-vis de la machine de base, c-à-d que des fonctions utiles de celle-ci ne sont plus accessibles au niveau de la machine virtuelle. L'article contient également des remarques intéressantes sur les systèmes construits selon le point de vue de P. Brinch Hansen qui en se référant explicitement à l'article de Simon [PBH74] désire construire des operating systems presque décomposables.

passage des complexes structures métalliques des ponts du début de ce siècle à l'arche en béton précontraint qui forme une seule entité avec la route qu'elle supporte. Un autre exemple plus frappant nous est donné par l'évolution de la technologie des fusées : une coupe dans une fusée allemande V-2 fait apparaître une structure métallique supportant extérieurement le recouvrement externe de la fusée et intérieurement les parois des réservoirs; dans le cas de la fusée Saturn-B utilisée pour les expéditions lunaires, la paroi externe est en même temps un composant structural et la paroi du réservoir, de plus la rigidité de l'ensemble est assurée par la pression du carburant à l'intérieur du réservoir.

Le langage holon doit être vu comme une tentative de compromis entre la structuration de la conception d'un programme et la production automatique d'une version "intégrée" du résultat de la conception. Le terme "holon" nous a semblé le plus approprié pour désigner cette nouvelle unité de structuration indépendante d'une méthode d'implémentation.

Rous empruntous l'énoncé du premier exemple A Fijen, collaborateur de R.W. Hijkstra I Findhoven

Nous expliciterous nuive solution de co problème directement sous la forme d'un programme-belon

Contrairament à sa qui sur raquia par la syntaxe, mous municoterane les bolens qui forment la pa grames efin de famillare les références lurs des commentaires; en pretique, motre nunéretation a re-

Holes H

N at & Stant des entiers adturels donais.

La denviène contratate est b = s - le 11 n'existe aucune valeur poer b si k > a) il est denc ofconsulte que l s M poèr qu'il existe se acons une valeur de b Muntaptible d'appartenir l'un tripli

D'autre part, soit deux triplets solutions (a, b, c,) et (a, b, c,), si (a, a,) alors (b, a b, et (c, a c,); s'est-leuxe par tous les triplets solutions sont distingués par des valours distinctes pour a. Co peut décader de généres tous les triplets solutions en suivant l'ordre crois-

des valours de M. La presidere contraînte apécific que a s.M. d'où tous les e<sub>1</sub> solutions apparties ment o l'ensemble E de velours (0,1,2,...,M - 1,M); de la douziène contraînte, en éédais que l'entre de la la contraînte (0,1,2,...,M - 2,M - 1) out conflatrait à des valours négative

pour h.

le valour de M ext un entier naturef:

le valour de k est un entier naturef e M

size imprimer qu'il n's a pas de solution;

pour chaque a appoitement & F; calculer B et vérifier si c est un carré parfait, en cas de

#### CHAPITRE VIII

### EXEMPLES DE PROGRAMMES

### 1. INTRODUCTION

Il n'est pas simple de choisir des exemples de programmes pour un nouveau langage; en effet, il est nécessaire que les exemples choisis mettent en évidence les caractéristiques du langage tout en étant suffisamment intéressants pour le lecteur quel que soit le domaine d'intérêt de celui-ci. D'autre part, dans notre cas, les exemples doivent également représenter l'application d'une méthodologie de programmation.

### 2. PREMIER EXEMPLE

Nous empruntons l'énoncé du premier exemple à Fijen, collaborateur de E.W. Dijkstra à Eindhoven : Générer tous les triplets d'entiers naturels (a,b,c) tels que :  $a \le M$ , a - b = k et  $a^2 - b^2 = c^2$ ; M et k étant des entiers naturels donnés.

Nous expliciterons notre solution de ce problème directement sous la forme d'un programme-holon écrit selon la syntaxe du langage.

### Remarque :

Contrairement à ce qui est requis par la syntaxe, nous numéroterons les holons qui forment le programme afin de faciliter les références lors des commentaires; en pratique, cette numérotation n'est utilisée que par le système de traitement des programmes-holon.

# Holon H

Générer tous les triplets d'entiers naturels (a,b,c) tels que a  $\leq$  M, a - b = k et a<sup>2</sup> - b<sup>2</sup> = c<sup>2</sup>, M et k étant des entiers naturels donnés.

### Text

La deuxième contrainte est b = a - k; il n'existe aucune valeur pour b si k > a; il est donc nécessaire que  $k \le M$  pour qu'il existe au moins une valeur de b susceptible d'appartenir à un triplet D'autre part, soit deux triplets solutions  $(a_1,b_1,c_1)$  et  $(a_2,b_2,c_2)$ , si  $(a_1 = a_2)$  alors  $(b_1 = b_2)$  et  $(c_1 = c_2)$ ; c'est-à-dire que tous les triplets solutions sont distingués par des valeurs distinctes pour a. On peut décider de générer tous les triplets solutions en suivant l'ordre croissant des valeurs de a. La première contrainte spécifie que  $a \le M$ , d'où tous les  $a_1$  solutions appartiennent à l'ensemble E de valeurs  $\{0,1,2,\ldots,M-1,M\}$ ; de la deuxième contrainte, on déduit que l'on peut rejeter de E le sous-ensemble  $\{0,1,2,\ldots,k-2,k-1\}$  qui conduirait à des valeurs négatives pour b.

# end

### datatest

la valeur de M est un entier naturel; la valeur de k est un entier naturel ≤ M else imprimer qu'il n'y a pas de solution; resultis false fi

### end

# begin

pour chaque a appartenant à E, calculer b et vérifier si  $c^2$  est un carré parfait, en cas de succès imprimer le triplet calculé;

end noloh H

Le programme H<sub>0</sub> est correct et calcule certainement tous les triplets qui vérifient les contraintes le datatest permet de bien spécifier les conditions d'application de notre algorithme solution, il est sous-entendu que les valeurs de M et de k sont "lues" par les opérations du "datatest".

# holon H1

pour chaque a appartenant à E, calculer b et vérifier si c<sup>2</sup> est un carré parfait, en cas de succès, imprimer le triplet calculé

```
begin
```

```
declare M,k,a,b,y : integer at générer tous les triplets etc level;
 a:=k;
 repeat
   b:=a - k;
   calculer y:=k * (a + b) = a^2 - b^2 = c^2;
   si "y" est un carré parfait alors imprimer le triplet
 until
   (a := a + 1) > M;
end
noloh H,
```

Le corps du holon H<sub>1</sub> est une description immédiate de sa définition; les descriptions en langage terminal des opérations qui le composent ne présentent aucune difficulté sauf pour le holon "si "y" est un carré parfait..." pour lequel il nous faut trouver une description tenant compte de l'inclusion de ce holon dans une itération. En fait, nous rejetons l'idée d'une description "naîve" de ce holon, comme par exemple : "if sqrt(y) \* sqrt(y) = y then imprimer;" où "sqrt(y)" génère la valeur entière "x" telle que x² ≤ y.

# Holon H2

Si "y" est un carré parfait alors imprimer le triplet

### Text

```
Si on considère les changements de valeurs de a et de b entre deux itérations, on voit que
  a_{i+1} > a_i et que b_{i+1} > b_i, d'où y qui est équivalent à k * (a + b) est tel que y_{i+1} > y_i.
  D'autre part, le carré d'un nombre n est égal à la somme des n premiers nombres impairs (par
  induction sur (n + 1)^2 = n^2 + 2n + 1). D'où à partir d'un carré donné, nous pouvons générer les
  suivants par simple addition.
Après la i-ième itération, on a trouvé soit que y_i = j^2 soit que (j-1)^2 < y_i < j^2.
  Si y_i = j^2, alors y_{i+1} > j^2, ce qui signifie que la génération des carrés supérieurs à j^2 doit
  reprendre pour tester si éventuellement y i+1 est un carré parfait.
  Si y_i < j^2, alors y_{i+1} est soit < j^2, soit = j^2, soit > j^2; si y_{i+1} est < j^2, y_{i+1} n'est pas un
  D'où quel que soit le cas, la génération des carrés ne doit reprendre que si y_{i+1} > j^2
```

```
declare carré : integer at pour chaque a etc level
    restricted initial carré := 0;
while y > carré
  do reprendre la génération des carrés od
  if y = carré then imprimer le triplet fi
  end
noloh H2
```

Il faut remarquer que "carré" a été déclaré <u>restricted</u>; en fait, cette variable correspond effectivement à une <u>own</u> variable pour ce holon. La taille totale du programme ne justifie peut être pas cette déclaration, mais conceptuellement elle est justifiée.

```
Holon H<sub>3</sub>
```

Reprendre la génération des carrés

```
begin
    declare n, impair : integer at pour chaque a etc level
    restricted initial begin n:=0; impair:=-1 end
    {1'initialisation de impair est justifiée par l'algorithme qui suit};
    impair:=impair + 2;
    n:=n + 1;
    carré:=carré + impair;
end
noloh H,
```

Les holons définis mais non-décrits correspondent à de simples opérations disponibles dans un langage d'arithmétique entière; si ces descriptions sont fournies, le compilateur génèrerait le code-machine équivalent à celui du programme de type-ALGOL :

```
begin
    declare integer : M, k, a, b, y, carré, impair, n;
   read (M) ; read (k);
if not (k ≤ M) then abort fi
    a:=k; carré:=0; impair:=-1; n:=0;
    repeat
      b:=a - k;
   y := k * (a + b);
    while y > carré
           do impair:=impair + 2;
              n:=n + 1;
              carré:=carré + impair;
          od
      if y = carré then print(a,b,n)fi
      a := a + 1:
  until a > M;
end
```

Cette solution pourrait être considérée comme "acceptable", mais elle ne correspondrait pas à notre méthodologie; en effet, décrire le holon "calculer y:=k \* (a + b)..." par l'opération "y:=k \* (a + b);" ne tient pas compte de la position particulière de ce holon dans le programme, à savoir qu'il n'est enclenché qu'en un seul point appartenant à une itération. Etudions cette opération de la même manière que lors de la description du holon H<sub>2</sub>.

```
Holon H4
```

```
Calculer y:=k * (a + b) = a^2 - b^2 = c^2
```

### Text

```
Entre deux y successifs, y_i et y_{i+1}, on a les relations suivantes : y_i = (a + b)(a - b) et y_{i+1} = (a' + b')(a' - b'), où a' = a + 1 et b' = a' - k = a + 1 - k = b + 1.

D'où : y_{i+1} = k.(a' + b') = k.(a + b + 2)
= k.(a + b) + 2k
= y_i + 2k
```

Le tout premier Y; étant :

$$y_0 = k.(a_0 + b_0) = k.(k + 0) = k.k$$

end

begin

```
declare y : integer at pour chaque a etc level
  initial calculer y := k² - 2k;
y:=y + 2k;
end
```

noloh H

Il résulte de cette description que le holon de nom "b:=a - k" est superflu puisqu'il n'est plus nécessaire de calculer explicitement b pour déterminer la valeur de y; de cette constatation, on déduit la description du holon "imprimer le triplet".

Holon H<sub>5</sub>

Imprimer le triplet

begin

 $\frac{declare}{b := a - k;} \ \ \underline{n} : \underline{integer} \ \underline{same} \ \underline{as} \ \underline{n} \ \underline{in} \ reprendre \ la \ \underline{génération} \ des \ carrés; \\ \underline{imprimer} \ (a,b,n);$ 

noloh H

La déclaration "same as" est due à notre utilisation de la variable n  $\underline{\text{restricted}}$  par un holon n'appartenant pas aux subordonnés directs ou indirects du holon  $h_3$  qui contient la déclaration de n.

En relisant le texte des holons  $\mathrm{H}_1$  à  $\mathrm{H}_5$ , on voit que notre algorithme consiste à générer, élément par élément, deux séquences de valeurs pour "a" et pour "y" :

a: 
$$\{k, k + 1, k + 2, ..., M\}$$
  
y:  $\{k^2, k^2 + 2k, k^2 + 4k, ..., k^2 + 2Mk\}$ ,

une solution existe chaque fois que y est un carré parfait.

Il en résulte :

- 1) si les valeurs de k et de M sont telles que  $k \le M$ , alors il existe toujours au moins une solution : (k,0,k);
- 2) le premier élément de la séquence y est un carré parfait qui vaut la somme des k premiers nombres impairs; le "coeur" de l'algorithme est de déterminer si une expression de forme  $k^2$  + 2ik, avec  $1 \le i \le M$  est un carré, c'est-à-dire déterminer s'il existe un entier q tel que la valeur, pour i fixé, de  $k^2$  + 2ik est égale à celle de la somme des q premiers nombres impairs.

Soit: 
$$k^2 + 2ik = \sum_{n=0,q} (2n + 1)$$
  
or  $k^2 = \sum_{n=0,k} (2n + 1)$   
d'où  $2ik = \sum_{n=k+1,q} (2n + 1)$ 

Ces considérations nous amènent à réécrire certains holons :

# Holon H

Pour chaque a appartenant à E, calculer b et vérifier si  $c^2$  est un carré parfait, en cas de succès, imprimer le triplet calculé

```
begin
     declare M, k,a: integer at générer tous les etc level
      initial a:=k;
     imprimer le triplet (k,o,k);
     while (a:=a + 1) < M
      do
        calculer y appartenant à la séquence;
        si y est un carré parfait alors imprimer le triplet;
   end
noloh H
Holon H4
  Calculer y appartenant à la séquence;
begin
    declare y : integer at pour chaque a etc level
      initial y:=o;
   y:=y + 2k;
noloh H
```

Le holon  $\rm H_2$  ne doit pas être modifié, par contre les nouvelles initialisations des variables impliquent une re-composition du holon  $\rm H_3$ .

```
Reprendre la génération des carrés
```

```
begin
  declare n, impair : integer at pour chaque a etc
  level restricted initial begin n:=k;
  impair := 2k - 1; end;
  impair:=impair + 2;
  n:=n + 1;
  carré:=carré + impair;
  end
noloh H<sub>3</sub>
```

A partir de ces textes des holons  ${\rm H_0}$  à  ${\rm H_5}$  et des textes des opérations terminales, le compilateur génèrera un programme équivalent au programme ci-après; dans ce programme, la présence de "branchements" correspond à une représentation du programme tel qu'il est généré par le compilateur; le programme résulte d'une "intégration" des divers holons (voir chapitre VII).

```
11 : a:=a + 1;
if a < M
       then y:=y + 2k;
           12:if y > carré
               then impair:=impair + 2;
                    n:=n + 1;
                    carré:=carré + impair;
                    goto 12;
              fi
              if y = carré
               then b:=a - k;
                    imprimer (a,b,n);
              fi
   goto 11;
    fi
end
```

On aurait pu se contenter de donner la dernière version de la description des holons et laisser sous-entendre que nous sommes arrivés immédiatement à cette version finale; en rapportant toutes les étapes de la création, nous pensons représenter exactement un exemple de processus de développement d'un programme tel qu'il se présente ordinairement pour un programmeur : la détermination progressive des caractéristiques de la solution et la description des opérations en fonction de leurs environnements (dans le cas d'un programme plus complexe, la même opération peut exister dans de multiples environnements).

Un point important est la relativité du caractère terminal d'une opération; en effet, le fait de considérer "y:=k \* (a + b)" comme une opération primitive (et non-terminale) nous a amené à faire une amélioration importante de l'algorithme. Les langages terminaux devraient être conçus de manière à assurer le programmeur qu'il bénéficiera d'un code d'autant plus efficace que les parties variables d'une expression sont réduites, ainsi le code généré pour "y:=y + 2k" doit tenir compte du surplus d'information que le générateur de code peut posséder (la présence de y dans les deux membres, la multiplication par une constante de valeur "2"...); si les langages terminaux respectent ce principe, une grande partie de l'optimisation d'un programme est le résultat du choix d'une description correcte d'une opération en fonction de son environnement.

# 3. DEUXIEME EXEMPLE

Nous expliciterons complètement la réalisation du programme de détermination du plus long chemin entre deux noeuds terminaux, problème partiellement discuté dans le chapitre III (3.2 et 3.3).

Les problèmes relatifs à la manipulation de graphes ont la particularité d'avoir des solutions dont l'algorithme s'exprime aisément dans le langage des mathématiques, mais qui - au contraire des problèmes "numériques" - ne peut s'exprimer facilement dans un langage de programmation habituel, car les éléments de base de l'algorithme (structures de données et opérations) doivent généralement être simulés au moyen des objets élémentaires du langage.

Prenons comme point de départ pour la construction de l'algorithme un arbre quelconque donné, c'est-à-dire un ensemble de noeuds et d'arcs qui correspondent à une structure d'arbre. L'ensemble des noeuds possède un sous-ensemble formé par les noeuds terminaux, c'est-à-dire les noeuds qui sont le point d'aboutissement d'un seul arc. Nous devons déterminer les deux noeuds de sous-ensemble tels que le chemin qui les relie soit plus long que tout autre chemin reliant deux noeuds terminaux quelconques du même arbre. Considérons comme connu, qu'entre deux noeuds terminaux d'un arbre, il n'existe qu'un seul chemin. Un algorithme plausible serait de dresser la liste de tous les couples de noeuds terminaux, de calculer pour chaque couple la longueur du chemin joignant les deux noeuds et de choisir le couple dont le chemin est le plus long. (En toute généralité, il est possible qu'il

existe plusieurs couples de noeuds terminaux dont la longueur du chemin soit égale à la valeur maximale trouvée). L'algorithme proposé a deux défauts :

- 1) le nombre de chemins à envisager croît rapidement avec la taille de l'arbre, (s'il y a n noeuds terminaux, le nombre de chemins correspond au nombre de combinaisons de n objets pris 2 à 1a fois);
- 2) il est nécessaire de former le sous-ensemble des noeuds terminaux.

Le deuxième défaut peut être considéré comme mineur, mais certainement pas le premier.

Prenons un noeud terminal quelconque  $T_0$  d'un arbre qui comporte (n + 1) noeuds terminaux ( $T_0$ ,  $T_1$ ,  $T_2$ , plus long chemin que nous recherchons; soit  $(T_0, T_1)$  le plus long chemin résultant de la construction précédente, montrons que  $T_1$  est identique à  $T_1$  ou à  $T_2$ :

en effet, on peut considérer que  $T_0$  est la racine de l'arbre; supposons qu'il existe trois chemins distincts  $(T_0,T_1)$ ,  $(T_0,T_1)$  et  $(T_0,T_1)$ ; il existe un noeud non-terminal M qui appartient aux trois  $\text{chemins } (\textbf{T}_0,\textbf{T}_1), (\textbf{T}_1,\textbf{T}_1) \text{ et } (\textbf{T}_1,\textbf{T}_1), \text{ M est déterminé comme étant le noeud final du chemin } (\textbf{T}_0,\textbf{M}) \text{ qui final du chemin } (\textbf{T$ est commun aux chemins  $(T_0, T_i)$  et  $(T_0, T_i)$ ; notons les chemins de la manière suivante ; a  $= (T_i, T_j)$ ,  $m = (T_0, M), a_1 = (M, T_i), a_2 = (M, T_i)$  et  $1 = (T_0, T_1);$  d'autre part, nous notons la longueur d'un chemin h par L(h); on a:

$$L(1) > L(m + a_1)$$

et

$$L(1) > L(m + a_2),$$

puisque  $T_1$  est distinct de  $T_1$ , on peut construire un chemin joignant  $T_1$  à  $T_1$ ; les trois chemins  $(T_0, T_i)$ ,  $(T_0, T_1)$  et  $(T_i, T_1)$  possèdent un noeud commun non-terminal Q; par rapport au noeud M, le noeud Q peut occuper diverses positions :



(a) Cas (a) :



Le noeud Q est situé entre  $T_0$  et M,c'est-à-dire que  $L(T_0,Q) \le L(T_0,M)$ ; le chemin menant de  $T_i$  à  $T_1$  est  $(T_i, M, Q, T_1)$ , donc la longueur est égale à

$$L(T_{1},M) + L(M,Q) + L(Q,T_{1}),$$

$$\texttt{L}(\texttt{T}_{\texttt{o}}, \texttt{T}_{\texttt{1}}) \; \Rightarrow \; \texttt{L}(\texttt{T}_{\texttt{o}}, \texttt{T}_{\texttt{j}})$$

c'est-à-dire que

$$L(T_o,Q) + L(Q,T_1) > L(T_o,Q) + L(Q,T_j)$$

d'oil

$$L(Q,T_1) > L(Q,M) + L(M,T_j)$$

d'où

$$L(M,Q) + L(Q,T_1) > L(M,T_i)$$

c'est-à-dire que

$$L(T_i, T_1) > L(T_i, T_i)$$

et (T<sub>i</sub>,T<sub>i</sub>) n'est pas le plus long chemin contrairement à notre hypothèse.

Cas\_(b) :

Le noeud Q est situé entre M et  $T_i$ ; le plus long chemin par hypothèse est  $(T_i, Q, M, T_j)$  dont la longueur est égale à :

$$L(T_{i},Q) + L(Q,M) + L(M,T_{i});$$

comparons la longueur du chemin  $(T_i, T_1)$  à celle du chemin le plus long :

$$L(T_{i}, T_{1}) = L(T_{i}, M) + L(M, Q) + L(Q, T_{1}),$$

or

$$L(T_0,M) + L(M,Q) + L(Q,T_1) > L(T_0,M) + L(M,Q) + L(Q,T_1)$$

d'où

$$L(Q,T_1) > L(Q,T_i)$$

c'est-à-dire que contrairement à l'hypothèse, (T;,T;) n'est pas le plus long chemin.

Cas\_(c) :

Identique au cas (b), en inversant le rôle de  $\mathbf{T_i}$  et  $\mathbf{T_j}$  .

En conclusion, notre hypothèse sur le caractère distinct des trois noeuds terminaux  $T_i$ ,  $T_j$  et  $T_1$  nous conduit à une contradiction; d'où le noeud  $T_1$  doit être confondu avec l'un des noeuds  $T_i$  ou  $T_j$ .

Grâce à ce résultat, nous pouvons formuler l'algorithme suivant :

prendre un noeud terminal (To);

chercher le plus long chemin  $(T_0,T_1)$  où  $T_1$  appartient à l'ensemble des noeuds terminaux; chercher le plus long chemin  $(T_1,T_1)$  où  $T_1$  a été déterminé lors de l'opération précédente et où  $T_1$  appartient à l'ensemble des terminaux;

Par rapport au premier algorithme proposé, cette nouvelle version ne requière plus que l'équivalent de deux traversées de l'arbre et a perdu complètement le caractère combinatoire du premier algorithme. Toutefois, le deuxième défaut du premier algorithme n'est pas éliminé, il est toujours nécessaire de déterminer un nocud terminal. Ré-examinons la démonstration de l'algorithme afin de vérifier si le fait que  $T_0$  soit un nocud terminal est important pour l'exactitude de la démonstration. En fait, on voit que la démonstration se base sur la comparaison de la longueur du chemin supposé le plus long  $(T_i,T_j)$  à celle d'un chemin  $(T_i,T_1)$  ou  $(T_j,T_1)$  où  $T_1$  est le nocud terminal le plus éloigné de la racine  $T_0$  de l'arbre; le point important est que  $L(T_0,T_1) > L(T_0,T_k)$  pour tout  $k \neq 1$ , le fait que  $T_0$  soit un terminal n'intervient pas; nous pouvons choisir l'algorithme suivant :

prendre un noeud quelconque de l'arbre ( $R_o$ ); chercher le plus long chemin ( $R_o$ , $T_i$ ) où  $T_i$  appartient à l'ensemble des noeuds terminaux; prendre  $T_i$  comme racine de l'arbre et chercher le plus long chemin ( $T_i$ , $T_j$ ) où  $T_j$  appartient à l'ensemble des terminaux;

L'étude précédente est évidemment indépendante du langage de programmation choisi, mais maintenant, nous abordons la phase de construction du programme proprement dit. A ce stade, il est utile de se rappeler la remarque faite au chapitre III(3.2 et 3.3), à savoir que l'algorithme ne donnera le résultat escompté que si on l'applique à un arbre, il est donc vital de s'assurer que les données possèdent effectivement une structure d'arbre. Dans le développement qui suit, nous appliquerons le principe avancé au chapitre III, c'est-à-dire que la version programmée de l'algorithme ne repose pas sur la supposition que les données possèdent d'elles-mêmes la structure d'arbre.

Nous avons signalé au début du chapitre qu'un programme doit produire un effet-net; en fait, cette

notion peut être précisée en affirmant qu'un programme a pour but la détermination d'un objet bien spécifié appartenant à un univers bien délimité. Le rôle du programmeur est non seulement de décrire le processus envisagé pour déterminer l'objet spécifié, mais aussi de garantir que ce processus se déroulera tœjœrs dans l'univers supposé. La méthodologie appliquée tente simplement de parvenir à ces deux objectifs en s'appuyant sur un langage qui permet de représenter des éléments essentiels de cette démarche.

L'algorithme général peut être décrit par un premier holon :

# Holon H

Chercher le plus long chemin dans un arbre

### Text

Un résumé de la preuve de l'algorithme :

# end

```
begin
```

create arbre, noeud;
declare a : pointer(arbre);
declare ro,t1,t2 : pointer(noeud);
acquérir les données et construire la représentation interne "a" de celles-ci;
prendre un noeud quelconque "ro" et rechercher le noeud "t1" le plus éloigné de "ro";
rechercher le noeud "t2" le plus éloigné de la racine "t1" de l'arbre et le chemin
qui y mène;
imprimer le chemin reliant "t1" à "t2";
end
noloh H

Envisageons le problème des données, c'est-à-dire du "format" de présentation de celles-ci. Il semble que la méthode la plus simple pour l'utilisateur est celle qui permette une description de l'arbre formée par un ensemble de descriptions locales des noeuds; par exemple :

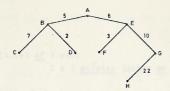
```
<arbre>::=<description d'un noeud>...
<description d'un noeud>::=
     <nom d'un noeud>{<nom d'un noeud><valeur arc>}...
```

où les noms de noeud suivis par une valeur d'arc représentent tous les noeuds adjacents du noeud décrit. Il est évident que cette forme de présentation des données ne garantit pas qu'il s'agit effectivement d'un arbre, mais cette correspondance doit être vérifiée par l'algorithme. Complétons le format ci-dessus par des délimiteurs qui faciliterent la lecture des données :

Par exemple, le texte de données :

```
# a : b(5),e(6);b:c(7),d(2),a(5);e:a(6),f(3),g(10);
c : b(7);d:b(2);f:e(3);g:h(22);h:g(22); #
```

correspond à l'arbre :



On peut simplifier la représentation en décidant qu'il n'est pas nécessaire de donner la description d'un noeud terminal, ce qui implique que tout noeud cité dans la description d'un noeud, et pour lequel aucune description n'est donnée, est un noeud terminal. Dans ce cas, le texte de données de l'exemple devient :

Une condition primordiale est l'absence de redondances parmi les descriptions de noeud. Cette forme de représentation externe de l'arbre nous paraît très simple pour l'utilisateur, surtout si les relations entre les noeuds sont susceptibles d'être modifiées.

Il reste maintenant à détailler et à composer systématiquement les holons définis dans la description du holon  $H_0$ . Cette décomposition fait suite à ce texte; il faut particulièrement remarquer la part importante de la décomposition destinée à la programmation du holon d'acquisition des données et de construction de la représentation interne.

Le présentation des holons suit un format standard : le texte du holon, suivi par une description schématique des holons définis qui fait correspondre à chaque définition de holon le numéro de sa description ou le symbole "#" s'il s'agit d'un holon considéré comme terminal; "resultis true" et "resultis false" sont symbolisés par T et F; une série de notes commentent les modifications subies par un holon au cours de la composition du programme. Un tableau général des déclarations, des "create" et "refine" permet de suivre les détails de la construction du programme.

A titre d'exemple, la description schématique de H est :

# Holon H

Acquérir les données et construire la représentation interne "a" de celles-ci

### Text

Le but de ce holon est de lire une série de descripteurs de noeuds et de construire la structure correspondante. Les éléments d'un noeud sont : son nom, une série d'arcs indiquant chacun un autre noeud qui est soit la racine d'un sous-arbre, soit un noeud terminal; à chacun de ces arcs est associé un "ccût" correspondant au "coût" du passage du noeud à un de ses successeurs. Pour chaque descripteur de noeud, nous devons créer une forme de représentation interne de ce noeud. Le nombre d'arcs issus de chaque noeud est variable, mais peut être connu à la lecture de la description d'un noeud. La représentation interne sera construite en deux temps :

- 1) une "passe" pour détecter le nombre d'arcs issus de chaque noeud,
- 2) une "passe" pour établir effectivement la structure.
- Il faut remarquer qu'à la fin de l'exécution de ce holon, il n'y a aucune garantie que la structure soit effectivement celle d'un arbre.

```
begin
```

create nom, coût, repère;

refine arbre by 2 noeuds;

refine noeud by 2 nom

2 array n[1..\*] of 3 coût

3 s : pointer (arbre or nom);

refine repère by 2 nom

2 p : pointer (arbre);

declare node-info : array i[1..2000] of repère

at chercher le plus long chemin dans un arbre level

initial mettre tous les pointeurs à (nil);

declare input : file of character external;

lire le fichier "input" et construire la table "node-info" en allouant la mémoire pour chaque noeud et en mémorisant dans chaque noeud les noms des sous-arbres du noeud;

implanter des mémoires pour chaque noeud terminal;

remplacer dans chaque noeud le nom par un arc vers le sous-arbre correspondant;

end

### Text

La déclaration de (node-info) à une dimension fixée arbitrairement à 2000; il serait évidemment possible d'utiliser une "séquence" ou un "array" à dimension variable. end

Noloh H<sub>1</sub>

H<sub>1</sub>[H<sub>2</sub>;H<sub>51</sub>;H<sub>41</sub>]

# Holon H2

Lire le fichier "input" et construire la table "node-info" en allouant la mémoire pour chaque noeud et en mémorisant dans chaque noeud les noms des sous-arbres des noeuds

On a toujours supposé que le fichier des données est écrit selon la syntaxe proposée; s'il s'avérait qu'un élément ne vérifie pas cette syntaxe, il serait nécessaire d'arrêter le processus de construction de la représentation interne.

end

# begin

create descripteur-de-noeud, nom-de-noeud;

refine descripteur-de-noeud

by 2 nom-de-noeud

2 sequence n[1..5] of 3 nom-de-noeud 3 coût;

declare new : descripteur-de-noeud;

accéder le fichier "input" et rechercher le premier caractère (#);

while on a trouvé les éléments d'un nouveau descripteur de noeud "new" correct dans "input" do s'il n'y a pas de redondance dans la description du noeud, remplir la table "node-info",

sinon interrompre la construction de cette table

od

### Postest

Aucune erreur syntaxique décelée dans "input" avant la fin des données end

### Text

La séquence "n" a été dimensionnée en supposant qu'il n'y aura que rarement des noeuds dont seraient issus plus de cinq arcs

end

Noloh H2

H<sub>2</sub>[H<sub>3</sub>; while H<sub>4</sub> do H<sub>23</sub> od | postest H<sub>65</sub> end]

Holon H<sub>3</sub>

Accéder le fichier "input" et rechercher le premier caractère (#)

begin

accès au début du fichier "input";
while "input [+1]" = (# ou eof)
do void od;

end

Postest

"input [+o]" z eof

<u>else</u> rapporter fichier vide;

<u>resultis false fi</u>

end

Noloh H<sub>3</sub>

H<sub>3</sub>[ # ; while # do void od | postest # else # ;F fi]

Note: "void" est un holon terminal correspondant à l'absence d'opération.

.-.-.-.-.-.-.-.-.-

Holon H

On a trouvé les éléments d'un nouveau descripteur de noeud "new" correct dans "input"

Text

Ce holon doit détecter un ruban de caractères qui correspond à la syntaxe d'un descripteur de noeud, c'est-à-dire :

<nom de noeud>:<nom de noeud><valeur arc>
[{,<nom de noeud><valeur arc>}...];

Tous les blancs qui pourraient s'intercaler entre les éléments sont à supprimer, les caractères de "fin de ligne" sont à assimiler à un blanc et la présence d'un caractère "eof" est une erreur fatale

```
begin
     "new" est mis à (zéro) pour tous les composants;
     sauter les blancs s'il en existe dans "input", (eof=erreur);
    if extraction d'une séquence de caractères légaux pour un nom de noeud est un succès
    then assigner la séquence trouvée à "new.nom-de-noeud";
         tenter d'extraire une séquence de couples (nom de noeuds, valeur arc)
    else signaler erreur syntaxique dans "input";
        resultis false
    fi
end
Noloh H
                            H4[#;H5; if H6 then H9;H10 else H20;F fi]
                                        -.-.-.-.-.-.-.-
Holon Hc
    Sauter les blancs s'il en existe dans "input", (eof=erreur)
    begin
     while "input [+1]" = blanc
      do void od
end
postest
 "input [+o]" ≠ eof
     else rapporter qu'une fin de fichier inattendue a été rencontrée;
          resultis false
     fi
end
Noloh Hs
                        H<sub>s</sub>[while # do void od | postest # else # ;F fi]
                                        -,-,-,-,-,-,-,-
Holon H<sub>6</sub>
    Extraction des caractères légaux pour un nom de noeud est un succès
Text
    Les caractères illégaux sont :\{\#|\;;|\;,|\;([))\;; la fin du nom de noeud est indiquée par ":" précédé
    ou non par des blancs; aucun "eof" ne peut être rencontré.
end
begin
declare c : character;
declare fin-de-données : boolean at lire fichier "input" et construire la table "node-info" etc level
        initial "fin-de-données" := (false);
declare pnom : vstring at on a trouvé les éléments d'un descripteur de noeud etc level
        initial "pnom" est vide;
while ("c":="input [+o]") est un caractère légal
do ajouter "c" à "pnom";
   "c":="input [+1]"
```

```
od
case "c" est un blanc
  then sauter les blancs s'il en existe dans "input", (eof)=erreur;
      on doit trouver un caractère (:)
eor "c" est un (:)
  then resultis true
eor "c" est un des caractères {,|;|(|)}
  then resultis false
eor "c" est un (#)
 then "fin-de-données":=(true);
 resultis false
else abort (eof rencontré d'une manière inattendue)
esac
end
Noloh H6
                      H_6[while H_7 do # od; case # then H_5; H_8 eor # then T
                      eor # then F eor # then # ;F else abort esac]
<u>Note</u> : vstri<u>ng</u> est un type terminal correspondant à un ruban de caractères de longueur variable.
Holon H,
("c":="input [+1]") est un caractère légal
begin
"c":="input [+1]";
<u>if</u> "c" ≠ {blanc|:|#|,|(|)|eof}
then resultis true
else resultis false
fi
end
Noloh H7
                                 H7[#; if # then T else F fi]
                                      -,-,-,-,-,-,-,-,-
Holon H<sub>8</sub>
   On doit trouver un caractère (:)
begin
case "input [+o]"=(:)
  then resultis true
eor "input [+o]" ≠ eof
then resultis false
   else abort (eof rencontré d'une manière inattendue)
```

Noloh H<sub>8</sub>

H<sub>8</sub>[case # then T eor # then F else abort esac]

-.-.-.-.-.-

Holon Ho

Assigner la séquence trouvée à "new.nom-de-noeud"

begin

refine nom-de-noeud by vstring;

"new.nom-de-noeud":="pnom";

end

Noloh Ho

H<sub>9</sub>[#]

Holon H<sub>10</sub>

Tenter d'extraire une séquence de couples (nom de noeud, valeur d'arc)

Text

Dans l'examen de "input", on vient de détecter un ":"; on doit détecter une série de un ou plusieurs couples <nom de noeud><valeur arc> jusqu'à ce qu'on trouve un couple délimité par un point-virgule. Nous allons rechercher ces couples un à un et attribuer les valeurs trouvées à "new.n[i]" où "i" représente le numéro d'ordre du couple trouvé.

end

begin

declare i : integer initial i:=o;

repeat chercher un nouveau couple délimité par une virgule ou un point-virgule et attribuer les valeurs trouvées à "new.n[i]" s'il n'y a pas d'erreur de syntaxe

until le couple délimité par un point-virgule est trouvé ou s'il y a une erreur de syntaxe;

if il y a eu erreur de syntaxe

then resultis false

else resultis true

end

Noloh H<sub>10</sub>

 $H_{10}[repeat H_{11} until H_{22}; if H_{21} then F else T fi]$ 

------

Holon H

Chercher un nouveau couple délimité par une virgule ou un point-virgule et attribuer les valeurs trouvées à "new.n[i]" s'il n'y a pas d'erreur de syntaxe

begin

"i":="i" + 1;

sauter les blancs s'il en existe dans "input", (eof=erreur);

if un nom de noeud appartenant à un couple est trouvé

```
then assigner ce nom de noeud à "new.n[i].nom-de-noeud";
   tenter d'extraire une valeur d'arc suivie par une virgule ou un point-virgule;
else signaler erreur de syntaxe dans "input";
fi
end
Noloh H
                                                                            H<sub>11</sub>[#;H<sub>5</sub>; if H<sub>12</sub> then H<sub>14</sub>;H<sub>15</sub>; else H<sub>20</sub> fi]
                                                                                                           -.-.-.-.-.-.-
Holon H<sub>12</sub>
       Un nom de holon appartenant à un couple est trouvé
Text
        On doit extraire une séquence de caractères délimitée par {blanc | () ; cette séquence ne peut
        contenir un des caractères {#|,|;|)} . Aucun (eof) ne peut être rencontré.
end
begin
declare c:character;
"pnom" est vide;
 while ("c":=input [+o]") \neq {#|,|;|)|(|blanc}
 do ajouter "c" à "pnom";
        "c":="input[+1]
 od
 case "c" est un blanc
         then sauter les blancs s'il en existe dans "input", (eof=erreur);
                   on doit trouver un caractère (();
 eor "c" est une (()
       then resultis true
 eor "c" est = {, |; |)}
         then resultis false
  eor "c" est un (#)
       then "fin-de-données":=(true);
   resultis false
    else abort (eof rencontré d'une manière inattendue)
  esac
  end
  Noloh H<sub>12</sub>
                                                                     H12[#; while # do # od; case # then H5; H13 eor #
                                                                              then T eor # then F eor # then # ;F else
                                                                              abort esac]
                                                                                                             The state of the s
```

Holon H<sub>13</sub>

On doit trouver un caractère (()

<u>begin</u>
<u>case</u> "input[+o]" = (()

<u>then resultis</u> <u>true</u>
<u>eor</u> "input[+o]" ≠ eof

then resultis false

else abort (eof rencontré d'une manière inattendue)

esac

<u>Ond</u>

Noloh H<sub>13</sub>

 $H_{13}[\underline{case} \# \underline{then} T \underline{eor} \# \underline{then} F \underline{else} \underline{abort} \underline{esac}]$ 

-.-.-.-.-.-

Holon H<sub>14</sub>

Assigner ce nom de noeud à "new.N[i].nom-de-noeud"

begin

"new.n[i].nom-de-noeud":="pnom";

end

Noloh H14

H<sub>14</sub>[#]

-.-.-.

Holon H<sub>15</sub>

Tenter d'extraire une valeur d'arc suivie par une virgule ou un point-virgule.

Text

La parenthèse à gauche a été détectée, il faut trouver un nombre suivi d'une parenthèse et d'une virgule ou d'un point-virgule.

end

begin

sauter les blancs s'il en existe dans "input",(eof = erreur);

if un nombre entier est trauvé dans "input"

then assigner ce nombre à "new.n[i].coût";

tenter d'extraire une parenthèse gauche suivie d'une virgule œu d'un point-virgule;

else signaler erreur de syntaxe dans "input"

fi

end

Noloh H<sub>15</sub>

H<sub>15</sub>[H<sub>5</sub>; <u>if</u> H<sub>16</sub> <u>then</u> H<sub>17</sub>; H<sub>18</sub>; <u>else</u> H<sub>20</sub> <u>fi</u>]

-,-,-,-,-,-,-,-

Holon H<sub>16</sub>

Un nombre entier est trouvé dans "input"

```
begin
declare n : integer at tenter d'extraire une valeur d'arc suivie par une virgule ou un point-
           virgule level;
if "input" contient une séquence de chiffres correspondant à un entier
then assigner cette valeur à "n";
 resultis true
else resultis false
fi
end
Noloh H<sub>16</sub>
                  H_{16}[\underline{if} # \underline{then} # ; T \underline{else} F \underline{fi}]
                                            -,-,-,-,-,-,-,-
Holon H<sub>17</sub>
   Assigner ce nombre à "new.n[i].coût"
begin
refine coût by integer;
"new.n[i].coût":="n";
end
Noloh H<sub>17</sub>
                                                  H<sub>17</sub>[#]
                                            -.-.-.-.-.-.-
Holon H<sub>18</sub>
   Tenter d'extraire une parenthèse gauche suivie d'une virgule ou d'un point-virgule
sauter les blancs s'il en existe dans "input", (eof=erreur);
<u>if</u> "input [+o]" = ')'
then extraire une virgule ou un point-virgule
else signaler erreur de syntaxe dans "input"
fi
end
Noloh H<sub>18</sub>
                                   H<sub>18</sub>[H<sub>5</sub>; if # then H<sub>19</sub> else H<sub>20</sub> fi]
                                             -.-.-.-.-.-.-
 Holon H<sub>19</sub>
   Extraire une virgule ou un point-virgule
 sauter les blancs s'il en existe dans "input", (eof=erreur);
 <u>if</u> "input[+o]" ≠ {,|;}
 then signaler erreur de syntaxe dans "input";
 fi
 end
```

Noloh H<sub>19</sub>

 $H_{19}[H_5; \underline{if} # \underline{then} T \underline{else} H_{20}; \underline{fi}]$ 

Holon H<sub>20</sub>

Signaler une erreur de syntaxe dans "input"

begin

 $\frac{\text{declare}}{\text{construire}} \ \frac{\text{boolean}}{\text{cH}_2} \ \frac{\text{at}}{\text{live fichier "input" et}} \ \text{et}$ 

initial "syntax-error":=false;

"syntax-error":=true;

end

Noloh H<sub>20</sub>

H<sub>20</sub>[ # ]

Manddatategr.fir.fif.fif.fir.

Holon H<sub>21</sub>

Il y a une erreur de syntaxe

begin

if "syntax-error"=true

then resultis true

else resultis false

fi end

Noloh H<sub>21</sub>

 $H_{21}[\underline{if} # \underline{then} T \underline{else} F \underline{fi}]$ 

Holon H<sub>22</sub>

Le couple délimité par un point-virgule est trouvé ou s'il y a une erreur de syntaxe

begin

if ("input[+o]"=';' ou "syntax-error")

then resultis true

else resultis false

fi

end

Noloh H<sub>22</sub>

H22[if # then T else F fi]

22

Holon H<sub>23</sub>

S'il n'y a pas de redondance dans la description du noeud, remplir la table "node-info", sinon interrompre la construction de cette table

### Text

Lorsque ce holon est enclenché, la variable "new" contient la description d'un noeud; il est absolument vital que cette description ne soit pas une re-description; si cette condition est respectée, la description peut être utilisée

end

## datatest

le "new.nom-de-noeud" ne correspond pas à un noeud déjà décrit, (l'introduire au besoin);

end

begin

allouer un (arbre) pour y inscrire le contenu de "new"; le pointeur de l'entrée de "node-info" correspondant à "new.nom-de-noeud" pointe l'élément alloué;

rattacher la mémoire allouée à l'entrée correspondante de "node-info";

les doublets de la description de "new" sont recopiés dans la mémoire allouée, les (nom-de-noeud) étant introduits dans la table "node-info" si besoin est;

· end

Noloh H<sub>23</sub>

H<sub>29</sub>[<u>datatest</u> H<sub>24</sub>; | H<sub>34</sub>; H<sub>35</sub>; H<sub>51bis</sub>; H<sub>36</sub>;]

Note : le holon H<sub>51bis</sub> a été ajouté après la conception du H<sub>49</sub>.

Holon H<sub>24</sub>

Le "new.nom-de-noeud" ne correspond pas à un noeud déjà décrit, (l'introduire au besoin)

### Text

"node-info" contient les noms de noeud déjà rencontrés; nous allons organiser cette table sous la forme d'une hash-table; la table sera accompagnée d'une table de "node-status" informant si une entrée est occupée et si elle est décrite; un "node-status"peut prendre trois valeurs : vide, occupée, décrite.

end

begin

create node-status;

declare st : array[1.. 2000] of node-status at chercher le plus long chemin dans un arbre level
initial tous les éléments de "st":=(vide);

declare entry : integer at s'il n'y a pas de redondance dans la description du noeud etc level;

déterminer à quelle entrée "entry" correspond % new.nom-de-noeud;

 $\frac{\underline{\mathtt{case}}}{\mathtt{mew.nom-de-noeud''}} \ \mathtt{correspond} \ \mathtt{\hat{a}} \ \mathtt{une} \ \mathtt{entree} \ \mathtt{"entry"} \ \mathtt{dans} \ \mathtt{"node-info"} \ \mathtt{qui} \ \mathtt{est} \ \mathtt{de} \ \mathtt{statut} \ \mathtt{(décrite)}$ 

eor "new.nom-de-noeud" correspond à une entrée "entry" de "node-info" qui est de statut (occupée)
then changer le statut de cette entrée "entry" de "node-info" en (décrite);

resultis true

else % new.nom-de-noeud correspond à une nouvelle entrée dans la table "node-info", introduire cette valeur;

attribuer à cette entrée dans "node-info" le statut (décrite);

resultis true

esac

Noloh H24

H<sub>24</sub>[H<sub>25</sub>; <u>case</u> H<sub>28</sub> <u>then</u> F <u>eor</u> H<sub>29</sub> <u>then</u> H<sub>32</sub>; T <u>else</u>
H<sub>33</sub>; H<sub>38</sub>; T <u>esac</u>]

Holon H25

Déterminer à quelle entrée "entry" correspond "new.nom-de-noeud"

begin

declare origine : integer;

calculer une hashvalue pour "new.nom-de-noeud" et l'attribuer à "entry" et à "origine";

repeat rapporter succès si l'entrée correspondant à "entry" est vide ou équivalente à "new.nom-de-noeuc until recherche donne succès sinon ("entry":="entry" + 1) = "origine" implique fin du programme;

end

Text

Si ce holon termine normalement, la valeur de "entry" indique l'élément dans la table "node-info" qui correspond à l'élément cherché

end

Noloh H25

H<sub>25</sub>[#; repeat H<sub>26</sub> until H<sub>27</sub>;]

Holon H<sub>26</sub>

Rapporter succès si l'entrée correspondant à "entry" est vide ou équivalente à "new.nom-de-noeud"

begin

<u>declare</u> succès : <u>boolean</u> <u>at</u> déterminer à quelle entrée "entry" correspond "new.nom-de-noeud" <u>level</u>
<u>initial</u> "succès":=(false);

if ("st[entry]"=vide)or "node-info.n[entry].nom" = "new.nom-de-noeud"

then "succès":=(true)

fi

end

Noloh H26

# H<sub>26</sub>[if H<sub>66</sub> then # fi]

<u>Note</u>: dans l'expression "node.info.n[entry].nom"="new.nom-de-noeud" la partie gauche possède un type terminal bien défini (<u>vstring</u>),par contre celle de gauche doit encore être raffinée. Ce holon ne pourra être décrit qu'après le choix d'un type terminal pour (nom).

-.-.-.-.-.-.-

Holon H27

Recherche donne succès sinon ("entry":="entry" + 1)="origine" implique fin du programme

begir

<u>declare</u> incr : <u>boolean</u> <u>at</u> déterminer à quelle entrée "entry" correspond "new.nom-de-noeud" <u>level</u> initial "incr":=false;

if "succès" = (true)

then resultis true

else "entry":=("entry" + 1)modulo 2000;

"incr":=true

resultis false

fi

postest

"entry" # "origine" or "incr" = false

end

Text

le postest a pour but d'éviter que la recherche ne boucle si la table est pleine et que l'on tente d'introduire une nouvelle valeur.

end

Noloh H<sub>27</sub>

$$H_{27}[\underline{if} # \underline{then} T \underline{else} #; #; F \underline{fi} | \underline{postest} #;]$$

Holon H<sub>28</sub>

"new.nom-de-noeud" correspond à une entrée "entry" dans "node-info" qui est de statut (décrite)

begin

if "st[entry]" = (décrite)

then resultis true

else resultis false

fi

end

Noloh H<sub>28</sub>

Holon H29

"new.nom-de-noeud" correspond à une entrée "entry" de "node-info" qui est de statut (occupée)

begin

if "st[entry]" = (occupée)

then resultis true

else resultis false

fi

end

Noloh H<sub>29</sub>

Holon H<sub>30</sub>

"st[entry]" = (décrite)

begin

refine node-status by integer {vide=0, occupée=1, décrite=2};

"st[entry]"=2;

end

Noloh H<sub>30</sub>

Holon H31

"st[entry]" = (occupée)

begin

"st[entry]" = 1

end

Noloh H31

H<sub>31</sub>[ # ]

s a pas de re-

Holon H<sub>32</sub>

Changer le statut de cette entrée "entry" de "node-info" en (décrite)

begin

"st[entry]":=2

end

Noloh H<sub>32</sub>

H32[#]

-.-.-.-.-.-.-

Holon H<sub>33</sub>

new.nom-de-noeud correspond à une nouvelle entrée dans la table "node-info", introduire cette valeur

#### Text

Un nouveau nom-de-noeud est déterminé, il faut que nous conservions cette information d'une manière ou d'une autre afin de la réutiliser pour d'autres descripteurs de noeuds; le problème de cette conservation de l'information tient au caractère variable du nombre de symboles qui composent un nom-de-noeud; toutefois, pour un noeud donné, il ne varie pas dès que le nom est déterminé. C'est pourquoi nous raffinons "nom" de "node-info" par un pointeur vers un (fstring) qui appartiendra à une (file); chaque fois qu'un nouveau nom est trouvé, il est ajouté à la (file) et un pointeur vers ce composant de la (file) est implémenté dans "node-info". Un (fstring) est une séquence de caractères dont le nombre est fixé, un (vstring) est une séquence de caractères dont le nombre varie entre o et 256; on peut obtenir le nombre de caractères d'un (string) par l'opération "length".

end

begin

refine nom by 2 pointer (fstring);

declare node-list file of fstring at chercher le plus long chemin dans un arbre level;

introduire "new.nom-de-noeud" dans "node-list[+1]";

"node-info[entry].nom":=address("node-list[+1]");

end

Noloh H<sub>33</sub>

H<sub>33</sub>[#;#;#;]

Holon H<sub>34</sub>

Allouer un (arbre) pour y inscrire le contenu de "new"

#### Text

Une donnée de type (arbre) comporte un tableau de dimension variable; la valeur de cette dimension est déterminée par le nombre d'éléments de la séquence "n" du descripteur-de-noeud "new"

end

begin

declare ib : upb of noeud.n;

declare pa : pointer (arbre) at s'il n'y a pas de redondance dans la description du noeud etc level;
"ib":=aupb("new.n");

"pa":=<u>new</u>(arbre);

end

#### Text

L'opération (aupb) correspond à "active upper bound" et permet de connaître la valeur de l'index supérieur d'une séquence; il existe de même une fonction (alob), et des fonctions (upb) et (lob) pour les tableaux.

end

Noloh H34

H<sub>34</sub>[#;#;] .....

Holon H<sub>35</sub>

Le pointeur de l'entrée de "node-info" correspondant à "new.nom-de-noeud" pointe l'élément alloué

begin

"node-info[entry].p" := "pa";

end

Noloh H<sub>35</sub>

H<sub>35</sub>[ # ]

Holon H<sub>36</sub>

Les doublets de la description de "new" sont recopiés dans la mémoire allouée, les (nom-de-noeud) étant introduits dans la table "node-info" si besoin est

### Text

En parcourant le contenu de la séquence "new.n", on obtient la liste de tous les noeuds adjacents au noeud en cours de description et également le coût du trajet jusqu'à ce noeud. Comme cette liste peut contenir des noeuds qui n'ont pas encore été décrits, nous implanterons dans "l'arbre" non pas des liens vers d'autres éléments de "l'arbre" mais uniquement des liens vers les noms de ces éléments introduits dans "node-info"; le "coût" sera implanté directement dans l'arbre.

end

begin

declare item : integer initial "item":=o;

declare max : integer initial "max":=aupb("new.n");

while ("item":="item" + 1) < "max"

do

lier "pa.noeud.n[item].s" au pointeur donné par "new.n[item].nom-de-noeud" après avoir introduit ce nom dans le fichier "node-list" si besoin est;

"pa.noeud.n[item].coût":="new.n[item].coût";

od end

Noloh H<sub>36</sub>

$$H_{36}[\underline{\text{while}} \# \underline{\text{do}} H_{37}; \#; \underline{\text{od}}]$$

--ipipi\_lpipipipi\_incom60)

# Holon H<sub>37</sub>

Lier "pa.noeud.n[item].s" au pointeur donné par "new.n[item].nom-de-noeud" après avoir introduit ce nom dans le fichier "node-list" si besoin est

# begin

déterminer à quelle entrée "entry" correspond % new.n[item].nom-de-noeud;

if "entry" dans "node-info" est de statut (vide)

then % new.n[item].nom-de-noeud correspond à une nouvelle entrée dans la table "node-info", introduire cette valeur;

attribuer à cette entrée dans "node-info" le statut (occupée);

fi

"pa.noeud.n[item].s":=<u>address</u>("node-info.i[entry].p");

end

# Text

La dernière opération est admissible parce que "s" est un pointeur vers un arbre ou vers un pointeur d'arbre; il faut noter qu'à ce stade, le contenu de "p" n'est pas significatif

end

Noloh H<sub>37</sub>

$$H_{37}[H_{25}; if H_{39}] + H_{33}; H_{40}; fi #;]$$

Holon Ha

Attribuer à cette entrée dans "node-info" le statut (décrite)

begin

"st[entry]":=2;

end

Noloh H<sub>38</sub>

Holon H<sub>39</sub>

"entry" dans "node-info" est de statut (vide)

begin

if "st entry "=o

 $\begin{array}{c|c} \underline{then} & \underline{resultis} & \underline{true} \\ \underline{else} & \underline{resultis} & \underline{false} \\ \underline{fi} & \\ \underline{end} & \end{array}$ 

Noloh H<sub>39</sub>

H<sub>39</sub>[<u>if</u> # <u>then</u> T <u>else</u> F <u>fi</u>]

Holon H<sub>40</sub>

Attribuer à cette entrée dans "node-info" le statut (occupée)

begin

"st[entry]":=1;

end

Noloh H<sub>40</sub>

H<sub>40</sub>[ # ]

Holon H41

Remplacer dans chaque noeud le nom par un arc vers le sous-arbre correspondant

Text

A ce stade, nous sommes dans la situation suivante : des éléments représentant les noeuds de l'arbre existent en mémoire; nous pouvons les atteindre grâce aux entrées effectives de "node-info"; les entrées effectives sont désignées par le tableau "st" : si st[i] = o, node[i] est sans intérêt, si st[i] = 1, node[i] est un noeud terminal, si st[i] = 2, il s'agit d'un noeud non-terminal; il s'agit de lier chaque noeud non-terminal à ses noeuds adjacents.

end

begin

declare max i : integer

initial begin "i" := o;

"max" := upb (node-info.i);

end;

while ("i" := "i" + 1) ≤ "max"

do lier le noeud "i" à ses noeuds adjacents s'il s'agit d'un noeud non-terminal

od

end

Noloh H41

 $H_{41}[\underline{\text{while}} * \underline{\text{do}} H_{42} \underline{\text{od}}]$ 

Holon H<sub>42</sub>

Lier le noeud "i" à ses noeuds adjacents s'il s'agit d'un noeud non-terminal

begin

if "st[i]" indique que le noeud est non-terminal

then déterminer le nombre de noeuds adjacents au noeud(i); pour chaque noeud adjacent, implanter l'arc;

fi end

Noloh H42

 $H_{42}[\underline{if} H_{43} \underline{then} H_{44}; H_{45}; \underline{fi}]$ 

Holon H<sub>43</sub>

"st[i]" indique que le noeud est non-terminal

begin

if "st[i]" = 2

then resultis true

else resultis false

fi end

afin .

H<sub>43</sub>[<u>if</u> # <u>then</u> T <u>else</u> F <u>fi</u>]

Holon H44

Déterminer le nombre de noeuds adjacents au noeud (i)

begin

 $\frac{\text{declare}}{\text{non-terminal}} \ \underline{\text{number-node}} : \underline{\underline{\text{integer}}} \ \underline{\text{at}} \ \text{lier le noeud "i" à ses noeuds adjacents s'il s'agit d'un noeud}$ 

level;

declare pn : pointer (arbre) at lier le noeud "i" à ses noeuds adjacents etc level;

"pn" := "node-info.i[i].p";

"number-node" := <u>aupb</u>(pn.noeud.n);

end

Noloh H<sub>44</sub>

H<sub>44</sub>[#;#]

Holon H<sub>45</sub>

Pour chaque noeud adjacent, implanter l'arc

Text

Les noeuds adjacents sont décrits dans un tableau "n" qui contient pour chaque noeud adjacent : le coût de passage et - à ce stade - un pointeur vers l'entrée de "node-info" qui contient un pointeur vers la mémoire allouée pour la description de ce noeud adjacent. Le but de ce holon est de remplacer le pointeur vers "node-info" par le contenu de l'élément pointé.

end

declare j : integer

initial j := 1;

declare pi : pointer(arbre);

repeat "pi" := valp(pn.noeud.n(j).s);

"pn.noeud.n[j].s":= "pi";

until ("j" := "j" + 1) > "number-node";

end

Text

L'opération  $\underline{valp}$  accède la valeur pointée par l'argument  $\underline{end}$ 

Noloh H<sub>45</sub>

H<sub>45</sub>[repeat # ; # ;until # ;]

Holon H<sub>46</sub>

Prendre un noeud quelconque "ro" et rechercher le noeud "ti" le plus éloigné de "ro"

# Text

Nous allons traverser ce qui est supposé être un arbre en caractérisant les noeuds rencontrés afin de vérifier si nous avons bien un arbre. Un noeud sera "actif" dès qu'il appartiendra à un chemin issu de "ro" : la condition de prolongation du chemin est que l'on ne lui ajoute aucun noeud "actif" sinon on construit un cycle. Intialement tous les noeuds sont "passif".

Pour que l'arbre ne soit pas une forêt, il faut que tous les noeuds initialement "passif" soient "actifs" à la fin de ce holon.

Les caractéristiques d'un noeud seront représentés par une valeur entière : passif = o, actif > o. end

begin

 $\frac{\text{declare}}{\text{initial array "node-index"}} \stackrel{\text{node-index}}{=} \frac{\text{array[1., 2000] of integer}}{\text{initial array "node-index"}} \stackrel{\text{def}}{=} \frac{\text{at}}{\text{chercher le plus long chemin dans un arbre }} \frac{\text{level}}{\text{long chemin dans un arbre$ 

déterminer la racine de départ "ro";

déterminer le noeud "t1" le plus éloigné de "ro";

end

postest

tous les noeuds dont le "node-status" est (décrit) ou (occupé) doivent avoir un "node-index" différent de (passif)

else on a une forêt et non un arbre, rapporter ce fait;

resultis false

fi end

Noloh H<sub>46</sub>

Holon H<sub>47</sub>

Déterminer la racine de départ "ro"

begin

declare départ : integer at chercher le plus long chemin dans un arbre <u>level</u>
initial "départ": = (upb(st) + 2) + 1;

<u>declare</u> arbre-ro : <u>pointer(arbre or noeud)</u> <u>at chercher le plus long chemin dans un arbre <u>level;</u> <u>while</u> "st[départ]" ≠ (décrite)</u>

<u>do</u> "départ" := ("départ" + 1) modulo 2000;

od

imprimer (noeud de départ : valp(node-info.n[départ].nom));
"arbre-ro" := "node-info.i[départ].p";
end

Text

A la fin de ce holon, "départ" représente l'index de la racine choisie, et "arbre-ro" la mémoire qui contient la description de cette racine. end

Noloh H<sub>47</sub>

 $H_{47}[\underline{\text{while}}\ H_{48}\ \underline{\text{do}}\ \#\ \underline{\text{od}};\ \#\ ;\ \#\ ;]$ 

-.-.-.-.-.-.-

Holon H<sub>48</sub>

"st[départ]" ≠ (décrite)

 $\begin{array}{l} \underline{\text{begin}} \\ \underline{\text{if}} \text{ "st[départ]" } \neq 2 \\ \underline{\text{then resultis true}} \\ \underline{\text{else resultis false}} \\ \underline{\text{fi}} \\ \underline{\text{end}} \end{array}$ 

Noloh H48

H<sub>48</sub>[<u>if</u> # <u>then</u> T <u>else</u> F <u>fi</u>]

Holon H<sub>49</sub>

Déterminer le noeud "t1" le plus éloigné de "ro"

# Text

On peut imaginer cette opération de la manière suivante :

on suppose un sous-arbre fictif de "ro" de nom  $(n_0)$  et tel que le coût de passage de "ro" à  $(n_0)$  soit négatif; d'autre part, à (ro) correspond un ensemble de sous-arbre  $\{n_i\}$ ; pour chaque  $(n_i)$ , on détermine son noeud le plus éloigné  $(n_{ti})$  et le coût  $c_{ti}$  du passage de  $(n_i)$  à  $(n_{ti})$ ; on détermine le coût le plus élevé en calculant le coût  $[c_{ti} + coût \ (ro \rightarrow n_i)]$ .

L'information que nous recherchons dans l'arbre, nous l'appelerons "noeud-ultime" et elle se compose de :

- 1) l'adresse d'un noeud terminal,
- 2) le père de ce noeud terminal (une entrée dans "node-info"),
- 3) le coût du passage du père au noeud ultime,
- 4) le coût total du passage de la racine de l'arbre à ce noeud ultime.

Chaque sous-arbre qui n'est pas un terminal possède son "nocud-ultime"; d'autre part, on peut conserver un "nocud-ultime" global qui contiendra à tout moment le nocud-ultime dont le quatrième élément correspond à la valeur maximale trouvée. On va procéder comme suit : pour chaque sous-arbre on recherchera le meilleur nocud-ultime, ensuite on le comparera au nocud-ultime global, modifiant celui-ci s'il y a lieu.

end

begin

create entry, noeud-ultime;
refine noeud by 2 entry;

refine entry by integer; refine noeud-ultime by 2 adresse-noeud : pointer(noeud) 2 entrée-dans-node-info : integer 2 coût-père-noeud : integer 2 coût-racine-noeud : integer {l'entrée-dans-node-info correspond au père du noeud ultime}; declare node-global : noeud-ultime at chercher le plus long chemin dans un arbre level; implanter dans % node-global des valeurs bidon; rechercher dans arbre pointé par % arbre-ro le meilleur noeud ultime et l'attribuer à "node-global" s'il est supérieur à celui-ci, conserver les valeurs % (% arbre-ro.noeud.entry) et %(% node-global. coût-père-noeud); déterminer les valeurs de "t1" à partir de "node-global", imprimer "t1"; end Noloh H49 H49[H50;H53;H58;] -.-.-.-Holon H<sub>50</sub> Implanter dans % node-global des valeurs bidon begin % node-global.cntrée-dans-node-info := o; % node-global.coût-père-noeud := -1; % node-global.coût-racine-noeud := -1; end Noloh H<sub>50</sub> H<sub>50</sub>[#;#;#;] -.-.-.-.-.-.-.-Holon H<sub>51</sub> Implanter des mémoires pour chaque noeud terminal Text En parcourant "node-info" en association avec "st", on peut déterminer les noeuds terminaux et allouer pour ceux-ci un élément de mémoire de type (noeud) end begin declare i, max : integer initial begin "i":=o; "max":=upb(node-info) end; while ("i":="i" + 1) ≤ "max" do si "st[i]" correspond à une entrée (occupée), c'est-à-dire un noeud terminal alors allouer une

mémoire pour ce noeud terminal

od end Noloh H<sub>51</sub>

H<sub>51</sub>[<u>while</u> # <u>do</u> H<sub>52</sub> <u>od</u>]

Holon H<sub>51bis</sub>

Rattacher la mémoire allouée à l'entrée correspondante de "node-info"

begin

"pa.noeud.entry":="entry";

"pa.noeud.nom":="node-info.n[entry].nom";

end

Noloh Hsthis

H<sub>51bis</sub>[#;#]

Holon H<sub>52</sub>

Si "st[i]" correspond à une entrée (occupée), c'est-à-dire un noeud terminal alors allouer une mémoire pour ce noeud terminal

begin

declare pa : pointer(arbre)

declare ib : upb of noeud.n

initial "ib":=1;

if "st[i]"=1

then "pa":=new(arbre);

"pa.noeud.nom":="node-info.n[i].nom";

"pa.noeud.entry":="i";

"pa.noeud.n[1].coût":=o;

"pa.noeud.n[1].s":=nil;

"node-info.i[i].p":="pa";

fi end

Noloh H<sub>52</sub>

H<sub>52</sub>[<u>if</u> \* <u>then</u> \* ; \* ; \* ; \* ; <u>fi</u>]

Holon H<sub>53</sub>

Rechercher dans arbre pointé par % arbre-ro le meilleur noeud-ultime et l'attribuer à "node-global" s'il est supérieur à celui-ci, conserver valeurs %(% arbre-ro.noeud.entry) et %(% node-global.coût-père-noeud)

Text

Afin de détecter un cycle éventuel, on vérifiera le caractère passif du noeud analysé  $\underline{\mathrm{end}}$ 

datatest

le nocud pointé par 1 arbre-ro ne peut être (actif), l'activer si correct;  $\underline{end}$ 

```
begin
declare coût-total : integer at chercher le plus long chemin dans un arbre level
        initial "coût-total":=o;
declare nombre-sousarbre : integer;
declare delta-cost : integer;
declare index : integer
   initial "index":=o;
"nombre-sousarbre":=upb(% arbre-ro.noeud-n);
while ("index":="index" + 1) ≤ "nombre-sousarbre"
do "delta-cost":=% arbre-ro.noeud.n[index].coût;
  "coût-total":="coût-total" + "delta-cost";
ajouter le nom du noeud au chemin si nécessaire;
   si le sous-arbre est un noeud terminal alors examiner si en tant que noeud ultime il peut remplacer
"node-global", s'il n'est pas terminal continuer à descendre dans l'arbre;
 retrancher le nom du noeud du chemin si nécessaire;
"coût-total":="coût-total" - "delta-cost";
od
end
Noloh H53
                      H<sub>53</sub> [datatest | while # do # ; # ; H<sub>60</sub>; H<sub>54</sub>; H<sub>61</sub>; # od]
	ilde{	ext{Note}} : les définitions des holons 	ext{H}_{60} et 	ext{H}_{61} ont été introduites lors de la description de 	ext{H}_{59}
Holon H54
 Si le sous-arbre est un noeud terminal alors examiner si en tant que noeud ultime il peut rem-
placer "node-global", s'il n'est pas terminal continuer à descendre dans l'arbre
Text
   On est parti d'un noeud "x" et on examine ses sous-arbres "y;", chaque "y;" peut posséder d'autres
   sous-arbres, mais parmi ces derniers existe "x" qu'il faut éviter de retraverser
end
begin
declare psa : pointer(arbre)
        initial "psa":=% arbre-ro.noeud.n[index].s;
declare it: integer;
"it":=% (% arbre-ro.noeud.entry);
case "psa" = nil
     then le comparer en tant que noeud-ultime à "node-global"
eor "psa.noeud.nom" = "node-info[it].nom"
     then ne rien faire car c'est le père du noeud
     else rechercher dans arbre pointé par &psa le meilleur noeud-ultime et l'attribuer à "node-
         s'il est supérieur à celui-ci, conserver valeurs %(% arbre-ro.noeud.entry) et %(% delta-
end
```

Noloh H<sub>54</sub>

Holon H<sub>55</sub>

Le comparer en tant que noeud-ultime à "node-global"

Text

La comparaison se fait sur base du "coût-total", le meilleur noeud-ultime étant celui dont le coût-total est le plus important

end

begin

 $\frac{\text{if}}{\text{coût-total"}} > \text{"node-global.coût-racine-noeud"} \\ \frac{\text{then}}{\text{fi}} \text{ attribuer ce noeud-ultime comme nouvelle valeur de "node-global"} \\ \frac{\text{fi}}{\text{end}}$ 

Noloh H<sub>55</sub>

Holon H<sub>56</sub>

Attribuer ce noeud-ultime comme nouvelle valeur de "node-global"

begin

"node-global.coût-racine-noeud":="coût-total";

"node-global.adresse-noeud":=% arbre-ro;

déterminer les valeurs de "node-global.entrée-dans-node-info"

et "node-global.coût-père-noeud";

copier le chemin trouvé dans le first chemin si nécessaire;

Noloh H<sub>56</sub>

 $\underline{\textit{Note}}$  : la définition de  $H_{62}$  a été introduite après la description de  $H_{59}$ .

-.-.-.

Holon H<sub>57</sub>

Déterminer les valeurs de "node-global.entrée-dans-node-info" et "node-global.coût-père-noeud"

begin

"node-global.entrée-dans-node-info":=%(% arbre-ro.noeud.entry);

"node-global.coût-père-noeud":=%(% node-global.coût-père-noeud);

Text

Les membres de droite désignent en fait des paramètres par valeur du holon  ${\rm H}_{53}$ 

end

Noloh H<sub>57</sub>

H<sub>57</sub>[#;#]

-,-.-,-.-.-.-

Holon H<sub>58</sub>

Déterminer les valeurs de "t1" à partir de "node-global", imprimer "t1";

begin

"t1":="node-global.adresse-noeud"

"t1.n[1].coût":="node-global.coût-père-noeud";

"t1.n[1].s":="node-info[node-global-entrée-dans-node-info].p";

imprimer "t1.nom" comme noeud trouvé;

end

Text

t1 pointe le noeud terminal trouvé et le contenu'de celui-ci est transformé afin d'en faire la racine d'un arbre.

end

Noloh Hcg

H<sub>58</sub>[#; #; #;]

-.-.-.-.-.-.-

Holon H<sub>59</sub>

Rechercher le noeud "t2", le plus éloigné de la racine "t1" de l'arbre, et le chemin qui y mène

Text

Cette opération est fort semblable à celle du holon précédant  $(H_{46})$ ; toutefois, dans ce cas, il n'est plus nécessaire de vérifier que les données forment un arbre, mais il faut s'organiser pour conserver le meilleur chemin

end

create chemin, nombre-de-noeuds;

refine nombre-de-noeuds by integer,

refine chemin by 2 nombre-de-noeuds

coût-du-chemin :

2 coût-du-chemin : integer
2 sequence ch[1..1000] of nom;
declare first, challenger : chemin at chercher le plus long chemin dans un arbre level initial "challenger.nombre-de-noeuds":=o;

declare mémoriser-chemin boolean at chercher le plus long chemin dans un arbre level initial "mémoriser-chemin":=(false);

"mémoriser-chemin":=(true);

implanter dans "node-global des valeurs-bidon;

"coût-total":=o;

rechercher dans arbre pointé par %t1 le meilleur noeud-ultime et l'attribuer à "node-global" s'il est supérieur à celui-ci,

conserver les valeurs %(% t1.noeud.entry) et %(% node-global.coût-père-noeud) datatest modification inhibit le noeud pointé par %arbre-ro ne peut être (actif), l'activer si correct endinhibit end; end Text La description de ce holon a impliqué la modification de  $H_{53}$  et de  $H_{56}$ end Noloh H<sub>59</sub> H<sub>59</sub>[#; #;H<sub>53</sub> datmod] Holon H<sub>60</sub> Ajouter le nom du noeud au chemin si nécessaire begin "mémoriser-chemin"=true then "challenger.nombre-de-noeuds":="challenger.nombre-de-noeuds" + 1; "challenger.ch[challenger.nombre-de-noeuds]:=% arbre-ro.noeud.nom; fi end Noloh H<sub>60</sub>  $H_{60}[if # then # ; # fi]$ Holon H<sub>61</sub> Retrancher le nom du noeud du chemin si nécessaire begin if "mémoriser-chemin" = true "challenger.nombre-de-noeuds":="challenger.nombre-de-noeuds" - 1; fi end Noloh H<sub>61</sub> H<sub>61</sub>[ # ] Holon H<sub>62</sub> Copier le chemin trouvé dans le first chemin si nécessaire begin if "mémoriser-chemin":=(true) then transférer le contenu de "challenger" dans "first" et conserver la valeur du coût-total fi end

Noloh H<sub>62</sub>

H<sub>62</sub>[<u>if</u> # <u>then</u> H<sub>63</sub>]

Holon H<sub>63</sub>

Transférer le contenu de "challenger" dans "first" et conserver la valeur du coût total

begin

declare i:integer

initial"i":=o;

"first.nombre-de-noeuds":="challenger.nombre-de-noeuds";

while ("i":="i" + 1) ≤ "first.nombre-de-noeuds";

do

"first.ch[i]":="challenger.ch[i]";

od

"first.coût-du-chemin":="node-global.coût-racine-noeud"

end

Noloh H<sub>63</sub>

H<sub>63</sub>[#; <u>while</u> # <u>do</u> # <u>od</u> #]

Holon H<sub>64</sub>

Imprimer le chemin reliant "t1" à "t2"

begin

imprimer (le plus long chemin est :);

imprimer tous les composants utiles de "first";

imprimer (coût du passage :) ;

imprimer la valeur de "coût-total";

end

Noloh H<sub>64</sub>

H<sub>64</sub>[#;#;#;]

-.-.-.-.-.-.-

Holon H<sub>65</sub>

Aucune erreur de syntaxe décelée dans "input" avant la fin des données

begin

if not("syntax-error") or "fin-des-données"

then resultis true

else resultis false

fi

end

Noloh H<sub>65</sub>

H<sub>65</sub>[if # then T else F fi]

-,-,-,-,-,-,-,-

Holon H<sub>66</sub>

("st[entry]"=vide) or "node-info.n[entry].nom"="new.nom-de-noeud"

begin

if("st[entry]"=vide)or valp(node.info.n[entry].nom)="new.nom-de-noeud"

then resultis true

else resultis false

t1 end

Noloh H<sub>66</sub>

 $H_{66}[if * then T else F fi]$ 

Holon H<sub>67</sub>

Le noeud pointé par %arbre-ro ne peut être (actif), l'activer si correct

begin

if "node-index[%arbre-ro.entry]"=actif

then resultis false

else "node-index[%arbre-ro-entry]":=actif;

resultis true

fi end

Noloh H<sub>67</sub>

H<sub>67</sub>[<u>if</u> # <u>then</u> F <u>else</u> # ;T <u>fi</u>]

# Ho: A : pointer(arbre); ro,t1,t2 : pointer(noeud); H<sub>1</sub> : node-info : array n [1..2000] of repère at H<sub>0</sub> level, initial; input : file of character external; H<sub>2</sub> : new : descripteur-de-noeud; H<sub>6</sub> : c : <u>character</u>; fin-de-données : bolean at H2 level, initial; pnom : vstring at H4 level, initial; H<sub>12</sub> : c : character; $H_{24}$ : st : $\underline{array}[1..2000]$ of node-status $\underline{at}$ $H_0$ $\underline{level}$ , $\underline{initial}$ ; entry : integer at H23 level; H<sub>25</sub> : origine : <u>integer</u> ; H<sub>26</sub> : succès : boolean at H<sub>25</sub> level, initial; H<sub>27</sub> : incr : boolean at H<sub>25</sub> level, initial; $H_{33}$ : node-list : <u>file of fstring at</u> $H_0$ <u>level</u>; H<sub>34</sub>: ib: <u>upb of noeud.n</u>; pa : pointer(arbre) at H23 level; H<sub>36</sub> : item : <u>integer</u>, <u>initial</u>; max : <u>integer</u>, <u>initial</u>; H41 : max : integer, initial; i : integer, initial; H44 : number-node : integer at H42 level; pn : pointer(arbre) at H42 level; H45 : j : integer, initial; pi : pointer(arbre); $H_{46}$ : node-index : $\underline{\text{array}}$ [1..2000] $\underline{\text{of}}$ $\underline{\text{integer}}$ $\underline{\text{at}}$ $H_0$ $\underline{\text{level}}$ , $\underline{\text{initial}}$ ; H<sub>47</sub> : depart : <u>integer</u> at H<sub>o</sub> <u>level</u>, <u>initial</u>; arbre-ro : pointer(arbre or noeud) at Ho level; H<sub>49</sub> : nodeglobal : noeud-ultime at H<sub>0</sub> level; H<sub>51</sub>: i : <u>integer</u>, <u>initial</u>; max : integer, initial; H<sub>52</sub>: pa : pointer(arbre); ib : upb of noeud.n, initial;

A. DECLARE

```
H<sub>53</sub> : coût-total : <u>integer</u> at H<sub>49</sub> <u>level</u>, <u>initial</u>;
       nombre-sous-arbre : integer;
       delta-cost : integer;
      index : <u>integer</u>, <u>initial</u>;
H<sub>54</sub>: psa: pointer(arbre), initial;
       it : integer;
H_{59}: first, challenger: chemin at H_{0} level, initial;
       mémoriser-chemin : boolean at Ho level, initial;
H<sub>63</sub> : i : <u>integer</u>, <u>initial</u>;
B. CREATE
Ho : arbre, noeud;
H<sub>1</sub> : nom, coût, repère;
H<sub>2</sub> : descripteur-de-noeud, nom-de-noeud;
H<sub>24</sub> : node-status;
H<sub>49</sub> : noeud-ultime, entry;
H<sub>59</sub> : chemin, nombre-de-noeud;
C. REFINE
H<sub>1</sub> : arbre by 2 noeud;
       noeud by 2 nom
                  2 <u>array</u> n[1..*] <u>of</u> 3 coût
                                        3 s : pointer(arbre or pointer(arbre));
       repère by 2 nom
                   2 p : pointer(arbre);
H<sub>2</sub> : descripteur-de-noeud
                by 2 nom-de-noeud
                    2 sequence n[1..5] of 3 nom-de-noeud
                                           3 coût;
H<sub>9</sub> : nom-de-noeud by vstring;
H<sub>17</sub> : coût by integer;
H<sub>30</sub> : node-status by integer;
H<sub>33</sub> : nom by pointer(fstring);
H_{49} : noeud-ultime by 2 adresse-noeud : pointer(noeud)
                           2 entrée-dans-node-info : integer
                           2 coût-père-noeud : integer
                           2 coût-racine-noeud : integer;
```

noeud by 2 entry; entry by integer;

 $H_{59}$ : nombre-de-noeuds by integer; chemin by 2 nombre-de-noeuds

2 coût-du-chemin : integer
2 sequence ch[1..1000] of nom;

Dans un article (Sivéé) destiné à explication de l'ende d'un problème à l'aide d'un ordinateur "pressit prosque toujours plus de temps que ce que l'en excendair", Christopher Strachen enviseges it le procédé habituel de division du travail en deux phèses :

1) le "systems enslysis" dons l'objet est d'unalyser le problème afin de décider exectement ce qui

) le "programming" qui consiste à composer les différentes opérations requises par le plan de résolution sous une forme adaptés à un ordinateur.

e la pure contine. Malheurenvenent, les faits démentant co point de vue, car moner à bien le deuiène phase demande blen souvent plus de temps que prévu et l'organisation de tous les éléments opérations et informations) qui forseront le programme requiert une précision et une constance unes aportantes que la première phase. La causé de cette difficulté véside probablement dans le fait que a deuxiène phase n'est pes une simple transformation d'un algorithme expriné d'une certaire manière als est que transposition très complete cer l'écert entre le manière d'expriser l'algorithme pendar a première phase et la forme d'expression accoptable par l'ordinateur est énorme; cette opération

Toutofeis, pour Strachey, il semblait que la distinction entre les deux étapes était sons utilité mais qu'erla se fainait que refléter le curatudre inadéquat de mos irangages de programmenten; il devrais Etra possible d'expriser toute la partie du "oyatem analysis" dans un langage fet que le reste la la composition du programme soit du ressort de l'ordinatour lui-séar.

conduir dans un langage propre, la description du programme qui en résulte est sirectement exploitée par le système de génération de code.

A matre evis, il est préférable de vensidérer qu'un langage de programmation est un artifices, t'ext-à-dire un objet artificiel conce dans un but déterminé, plutôt que de l'éthdier comme un objet marhématique décrit per certaines règles bien précises. La dernière approche a poir conséquence du négliger partiellement les problèmes du "system analysis" et totalement cou du "programmalig". De toute acaibre, toute conception d'un langage de programmation résulte du comprants saire ce, que con inventeur suppose souhaitable et ce qu'il suppose réalisable; il en est minsi parce su'ettue-liesent nous susmes incapables de préciser quelles atroctures de données et quels types d'instructions sent fondamentaux. Ve cet état de choses, le présentation généralement dognatique et pérenteure de la définition d'un langage est une véritable fortaiture : l'inventeur delt expliciter les raisons pro-

En pratique, l'intérêt du programmeur commence quand celui du mathématicien s'est évaneui; pour ce dernier, le "system analysis" peut dans certains cas présenter un quelconque intérêt, mais le "programmeur deit être à même de memer successivement l'une et l'autre phase, le "programming" correspond à la sommission d'un algorithme aux contingraces d'une réalisation pratique. Même pour le phase de "ayatem mualysis", les points de vue du untératicie et du programmeur penyent être totalement d'une granteur, l'intérêt princréial du mathématicien réside dans la détermination de l'existence d'un algorithme, colui du programmeur dans la conception d'un

### CHAPITRE IX

#### CONCLUSIONS

# 1. "SYSTEM ANALYSIS" ET "PROGRAMMING"

Dans un article [Str66] destiné à expliquer pourquoi la résolution d'un problème à l'aide d'un ordinateur "prenait presque toujours plus de temps que ce que l'on attendait", Christopher Strachey envisageait le procédé habituel de division du travail en deux phases :

- le "systems analysis" dont l'objet est d'analyser le problème afin de décider exactement ce qui doit être fait, et ensuite d'adopter un plan général de résolution;
- 2) le "programming" qui consiste à composer les différentes opérations requises par le plan de résolution sous une forme adaptée à un ordinateur.

La première phase semble contenir toute la partie intellectuelle du travail et la deuxième être de la pure routine. Malheureusement, les faits démentent ce point de vue, car mener à bien la deuxième phase demande bien souvent plus de temps que prévu et l'organisation de tous les éléments (opérations et informations) qui formeront le programme requiert une précision et une constance aussi importantes que la première phase. La cause de cette difficulté réside probablement dans le fait que la deuxième phase n'est pas une simple transformation d'un algorithme exprimé d'une certaine manière, mais est une transposition très complexe car l'écart entre la manière d'exprimer l'algorithme pendant la première phase et la forme d'expression acceptable par l'ordinateur est énorme; cette opération n'est pas une simple routine.

Toutefois, pour Strachey, il semblait que la distinction entre les deux étapes était sans utilité, mais qu'elle ne faisait que refléter le caractère inadéquat de nos langages de programmation; il devrait être possible d'exprimer toute la partie du "system analysis" dans un langage tel que le reste de la composition du programme soit du ressort de l'ordinateur lui-même.

On peut considérer que le HPL est une tentative dans ce sens : tout le "system analysis" peut être conduit dans un langage propre, la description du programme qui en résulte est directement exploitée par le système de génération de code.

# 2. LE LANGAGE DE PROGRAMMATION VU COMME UN ARTEFACT

A notre avis, il est préférable de considérer qu'un langage de programmation est un artefact, c'est-à-dire un objet artificiel conçu dans un but déterminé, plutôt que de l'étudier comme un objet mathématique décrit par certaines règles bien précises. La dernière approche a pour conséquence de négliger partiellement les problèmes du "system analysis" et totalement ceux du "programming". De toute manière, toute conception d'un langage de programmation résulte du compromis entre ce que son inventeur suppose souhaitable et ce qu'il suppose réalisable; il en est ainsi parce qu'actuellement nous sommes incapables de préciser quelles structures de données et quels types d'instructions sont fondamentaux. Vu cet état de choses, la présentation généralement dogmatique et péremptoire de la définition d'un langage est une véritable forfaiture : l'inventeur doit expliciter les raisons profondes des différentes décisions qui l'ont amené à la structure proposée du langage.

En pratique, l'intérêt du programmeur commence quand celui du mathématicien s'est évanoui; pour ce dernier, le "system analysis" peut dans certains cas présenter un quelconque intérêt, mais le "programming" certainement pas. Par contre, le programmeur doit être à même de mener successivement l'une et l'autre phase, le "programming" correspond à la soumission d'un algorithme aux contingences d'une réalisation pratique. Même pour la phase de "system analysis", les points de vue du mathématicien et du programmeur peuvent être totalement divergents; l'intérêt primordial du mathématicien réside dans la détermination de l'existence d'un algorithme, celui du programmeur dans la conception d'un algorithme efficace, c'est-à-dire qu'il ne suffit pas de savoir qu'un algorithme convergera mais bien

de déterminer à quelle vitesse il convergera.

Ce dernier point est très important, car il explique partiellement les raisons de la faiblesse du développement des principes fondamentaux de l'informatique : les tentatives d'application à la programmation d'outils théoriques développés antérieurement pour autre chose. Il est inévitable qu'en progressant de cette manière, on bute continuellement sur une succession de pierres d'achoppement.

# 3. UNE (TENTATIVE DE) DEFINITION DE LA PROGRAMMATION

Nous nous permettrons une définition de la programmation qui n'est ni plus ni moins mauvaise que toutes celles qui ont déjà été proposées, mais qui possède un triple avantage :

- 1) définir un cadre plus large pour ce travail,
- 2) expliquer indirectement pourquoi les outils théoriques que nous utilisons sont inadéquats,
- 3) décrire indirectement les éléments qui forment cette discipline :

la programmation est la science de la sélection effective, d'une manière efficace, d'un objet bien spécifié appartenant à un ensemble bien déterminé.

On ne peut nier que cette définition explicite exactement le but des programmes réalisés pratiquement, étant donné que nous ne faisons aucune supposition sur les caractéristiques de l'objet à déterminer, celui-ci pouvant être de nature très complexe. En outre, la définition donnée ne souffre pas d'un excès de généralité. (1)

Son principal mérite est - à notre avis - de délimiter les limites d'application des outils mathématique à l'étude de la programmation; en effet, ils sont soit trop généraux (par exemple, ce qu'on appelle la "théorie des algorithmes" traitent principalement de l'existence ou de l'inexistence d'algorithmes pour déterminer certaines valeurs sans insistance sur l'efficacité de l'algorithme), soit tout-à-fait étrangers à la notion de sélection d'un objet. Le problème soulevé par l'instruction d'affectation permet de préciser ce dernier point. Les mathématiques ne traitent généralement pas de problèmes impliquant des "variables", au sens donné à ce terme en programmation, c'est-à-dire des objets dont la valeur se modifie en fonction du temps. L'existence des instructions d'affectation rend difficile tout traitement rigoureusement mathématique de l'étude des programmes (au point de vue fiabilité, par exemple), c'est pourquoi la tentation est grande de ne voir de solution aux problèmes de la programmation que dans des langages de types "fonctionnels" où les variables sont elles-mêmes des fonctions, comme le LISP dont la structure récursive se prête à l'analyse formelle; la tendance extrême est de voir dans l'affectation une instruction dont le bannissement doit suivre celui du "goto". Toutefois, le fait que nous ne disposons pas d'une théorie adéquate pour rendre compte de l'instruction d'affectation ne signifie pas que celle-ci doit nécessairement disparaître de notre répertoire <sup>(2)</sup>; pour des motifs mal éclaircis, le plus fonctionnel des langages à réintroduit l'affectation (voir en LISP, la forme PROG). D'autre part, si on examine l'instruction d'affectation selon l'optique de la définition proposée pour la programmation, on remarquera qu'elle correspond à une opération normale lors de la sélection d'un objet : la réinitialisation d'un élément utilisé comme point de repère dans la recherche de l'objet à sélectionner.

La définition proposée enlève tout intérêt mathématique à la programmation puisqu'elle ramène tous les programmes au choix d'un élément appartenant à un ensemble, ce dernier étant nécessairement fini; d'un point de vue mathématique, ce problème est trivial. L'essence de la programmation est de résoudre ce problème en limitant au maximum le nombre d'éléments de l'ensemble examinés afin de déterminer

<sup>(1)</sup> On peut la comparer par exemple à la définition de Niklaus Wirth : "a methodology of constructive

reasoning applicable to any problem capable of algorithm solution".

(2) L'argument contre le "goto" est de nature toute différente [Dij68]; le fait que Hoare ait proposé une définition axiomatique du "goto" relativement simple [Knu74, page 289] n'annule pas l'argumentation contre le "goto".

l'élément spécifié; ceci implique l'invention de formes de représentations adéquates et d'algorithmes efficaces. Les deux exemples donnés dans le chapitre précédent sont une application concrète de ce principe.

Notre définition indique précisément les principes qui doit guider le programmeur dans la phase de "systems analysis" et précise que le "programming" ne peut être négligé puisque la sélection doit être effective.

Les diverses instructions utilisées en programmation peuvent être interprétées du point de vue de la sélection d'un objet : une instruction d'affectation est une sélection directe d'un élément comme par exemple : "x:=x + 1;" correspond à la sélection du successeur d'un élément dans l'ensemble des entiers. Les instructions "if-then-else" et "case" correspondent au choix d'un sous-ensemble parmi d'autres; les instructions d'itération sont l'équivalent d'une sélection d'un élément par parcours d'un ensemble, à condition que l'itération soit finie, c'est-à-dire qu'un élément soit effectivement sélectionné. Le "datatest" et le "postest" sont également impliqués par notre définition; en effet, leur fonction est de garantir que le processus de sélection est toujours conduit dans l'ensemble déterminé. Il est possible qu'en envisageant la programmation sous l'angle de la "sélection d'un objet", on puisse déterminer de nouvelles formes d'instructions de sélection plus précises quant au processus et au critère de sélection et à l'ensemble examiné; des instructions comme le "do-while" sont en fait des structures générales de sélection, mais l'exactitude de son utilisation repose sur la responsabilité du programmeur et son habileté à y adjoindre un processus et un critère de sélection cohérents.

Les "structures de données" peuvent être analysées selon leurs influences sur les processus de sélection, et selon l'aisance avec laquelle elles permettent (ou ne permettent pas) de passer d'un élément à un autre. Des considérations de ce genre nous ont guidé dans le choix des "array", "sequence" et "file" en langage holon.

# 4. LA PROGRAMMATION COMME UNE DISCIPLINE

C'est notre ferme conviction que la programmation ne pourra devenir une discipline par elle-même que si elle parvient à se construire un support théorique pour ses principes fondamentaux (qui sont encore à découvrir). Ce support théorique ne pourra certainement pas faire abstraction des buts de la programmation, ni des moyens à mettre en oeuvre pour les réaliser. Le HPL doit être vu comme une tentative de définition d'un langage à partir du but de l'utilisation des programmes : la production d'un effet-net; en outre, le langage tente de tenir compte au maximum du "programming" au sens de Strachey; une innovation importante est la volonté de faire apparaître dans le programme l'historique des décisions prises pour lui donner sa forme finale; une autre caractéristique originale est la dissociation opérée entre la définition d'une opération et la représentation "machine" de celle-ci, le choix de cette représentation est le résultat des contingences de l'environnement d'exécution du programme et il est accompli en vertu d'un principe d'intégration, qui est une idée maîtresse de "l'engineering" contemporain.

Par lui-même, le HPL ne forme pas une discipline de programmation, toutefois il tente de donner au programmeur la possibilité de développer des programmes complexes, en explicitant le pourquoi des décisions prises. La justification des décisions est le caractère d'une véritable discipline. La recherche en informatique doit s'efforcer de développer une telle discipline et elle ne peut pas décider d'ignorer perpétuellement les difficultés de développement et de mise en oeuvre des programmes, sinon la boutade de F.L. Bauer risque de devenir une vraie définition :

"software engineering is the part of computer science, which is too difficult for the computer scientist". [Bau71]

#### ANNEXE A

# LA "PROGRAMMATION STRUCTUREE" DE E.W. DIJKSTRA

Nous allons exposer systématiquement la méthodologie proposée par Dijkstra afin de faire ressortir les hypothèses qui servent de base à cette méthode de construction de programmes. Cet exposé est basé sur les références [Dij69, Dij72].

### A. TAILLE D'UN PROGRAMME

Si on envisage des programmes de "grande taille" (dont la taille dépend de la complexité de la tâche que le programme est censé entreprendre), on peut tenter d'évaluer la probabilité d'exactitude d'un programme de ce type en fonction de l'exactitude de ses "composants" (par "composant", il faut comprendre : module, routine, séquence d'instructions; la distinction entre ces éléments n'intervient pas ici). Si le programme comprend "N" composants et que chaque composant a une probabilité "p" d'être correct, la probabilité "P" que le programme complet soit correct ne sera pas supérieure à :

$$P = p^N$$
.

Si N est grand, "p" doit être très proche de l'unité si l'on désire que "P" soit plus proche de l'unité que de zéro. Combiner les composants en d'autres de plus grande dimension ne résoud pas le problème puisque :

$$p^{N/2} \times p^{N/2} = p^{N}$$
.

L'exactitude d'un programme est vitale et il est nécessaire de convaincre l'utilisateur d'un programme que celui-ci est correct. On peut envisager de démontrer cette exactitude soit expérimentalement, soit "intellectuellement"; mais il est raisonnable de trouver une méthode dont l'application implique un effort au maximum proportionnel à la taille du programme; on rejette toute méthode qui exigerait, par exemple, un effort croissant avec N.

# B. EXACTITUDE D'UN PROGRAMME

Affirmer l'exactitude d'un programme est affirmer que le résultat d'une exécution quelconque du programme vérifie certaines propriétés. Vu la quantité exhorbitante d'exécutions différentes d'un même programme, une vérification exhaustive de son exactitude par des séries de tests est pratiquement impossible, même pour les programmes les plus simples; cette constatation nous amène à rejeter toute démonstration "expérimentale" de l'exactitude d'un programme, seules restent les méthodes "intellectuelles".

Dijkstra conclut de l'examen des méthodes'mathématiques formelles' de preuve de programme qu'elles montrent qu'une preuve formelle d'un programme peut être construite, mais qu'elles imposent un effort qui est généralement plus que proportionnel à la longueur du texte. Il en déduit que le vrai problème n'est pas de prouver l'exactitude d'un programme, mais de déterminer des structures de programme pour lesquelles il existe une preuve d'exactitude qui ne requiert pas un effort exagéré, même pour un grand programme. Si on réussit à trouver ces structures, on peut tenter de déterminer une méthode de construction d'un programme formé de ces "structures" pour accomplir une tâche donnée. Cette manière de voir suppose que l'ensemble des programmes "structurés" (qui est un sous-ensemble de tous les programmes possibles) contienne suffisamment de programmes pour répondre à nos besoins (nos tâches à programmer). Dijkstra fait cette supposition.

Il en arrive à "une méthode constructive pour l'exactitude des programmes", à savoir : construire le programme en analysant comment une preuve peut être établie pour que l'ensemble des exécutions du programme satisfasse les spécifications; les exigences de la preuve "induisent" le programme.

#### C. ABSTRACTION ET INSTRUCTIONS "STRUCTUREES"

Considérant les programmes séquentiels, Dijkstra conclut que les instructions à utiliser dans leur composition peuvent être limitées à : la succession immédiate, l'instruction conditionnelle, les instructions de répétition "while...do" et "repeat...until", une instruction "case" de choix unique parmi une série de possibilités et les appels de procédure; l'utilisation d'instructions de branchement vers des instructions "étiquetées" est déconseillée, voire bannie.

Cette sélection n'est point arbitraire, mais est basée sur les possibilités de déduire l'effet de l'exécution d'une instruction sur une (ou des) assertion(s) supposée(s) vérifiée(s) avant l'exécution de cette instruction; le choix est guidé par la simplicité de la déduction, celle-ci étant exprimée en termes "d'axiomatisation de Hoare" [Hoa69].

Par ces restrictions, Dijkstra entend se limiter à la composition de "programmes intellectuellement maniables"; ce qui implique une nouvelle supposition :

"...I think that it is fair to say that hierarchy is a key concept for all systems embodying a nicely factored solution. I could even go one step further and make an article of faith out of it, viz. that the only problems we can really solve in a satisfactory manner are those that finally admit a nicely factored solution..."[Dij72 a]

Cette hiérarchisation de la solution (et du programme) est nécessaire pour empêcher tout accroissement excessif de l'effort de preuve d'un programme en fonction de l'accroissement de sa taille.

Soit un programme :

$$S_1; S_2; \dots S_n.$$

Supposons que pour établir l'exactitude du programme (1) à partir de la description de l'effet-net de chaque instruction S<sub>i</sub>, l'on ait besoin d'un raisonnement composé de N étapes; envisageons une instruction conditionnelle:

dont les effets des instructions  $S_1$  et  $S_2$  sont connus; nous supposons que pour établir l'effet de l'instruction (2) un raisonnement en deux étapes soit nécessaire, l'une pour le cas où B est "vrai", l'autre pour B "faux".

Soit un programme :

(3)

Pour établir que l'effet d'une instruction conditionnelle de (3) est équivalente à l'effet d'une instruction abstraite  $S_i$ , il faut un raisonnement en deux étapes. Puisque le programme (3) se compose de N instructions conditionnelles, il faut un raisonnement en 2N étapes pour ramener le programme (3) à un programme de forme identique à (1); pour comprendre (1), il faut un raisonnement en N étapes; au total, le programme (3) peut être compris par un raisonnement en 3N étapes.

Par contre, si on tente de comprendre l'effet du programme (3) "directement" sans faire intervenir les instructions abstraites  $S_i$ , nous devons essayer de comprendre l'effet de la succession de N instructions  $S_{ij}$ , ce qui exige un raisonnement en N étapes; or il y a  $2^N$  successions différentes possibles, d'où le raisonnement complet comprendra :

En fait, la méthode ne consiste pas à transformer (3) en (1), mais à construire un programme (3) à partir d'un programme (1).

### D. "DATA STRUCTURES" ABSTRAITES

En hiérarchisant notre programme, nous avons ramené l'expression de la fonction du programme à une série d'instructions "abstraites", ce qui suppose que des instructions de ce genre existent, c'est-à-dire est-il vraiment possible de définir exactement un programme complexe par une série d'instructions abstraites ?

Dijkstra affirme qu'il est possible d'écrire des instructions de ce genre si son effet est exprimé en termes de structures de données "abstraites" - elles aussi - qui permettent d'exprimer exactement l'effet de l'instruction.

Le passage du programme de son expression la plus abstraite à une forme (une série d'instructions) acceptable par une machine donnée implique une succession de "raffinements combinés" ("joint refinement" dans la terminologie de Dijkstra), c'est-à-dire qu'une instruction "abstraite" de niveau "n" opérant sur des structures de données "abstraites" est re-décrite en termes d'autres instructions "abstraites" de niveau "n+1" opérant, à un niveau de détails plus étoffé que la première instruction, sur de nouvelles data structures "abstraites" (ou concrètes suivant le degré de détails atteint) qui ont été choisies pour représenter les data structures "abstraites" utilisées dans l'instruction de niveau "n". Un "raffinement combiné" inclut les conséquences d'une décision prise en cours de conception du programme.

# COMMENTAIRES SUR LA PROGRAMMATION STRUCTUREE

On peut énumérer les points critiquables comme suit :

- 1) primauté accordée à la facilité de trouver l'exactitude d'un choix de conception sur l'efficacité de la solution choisie;
- considérer que le sous-ensemble des programmes "structurés" est suffisant pour programmer les problèmes courants;
- 3) la possibilité d'utiliser des data structures "abstraites" dans la définition des opérations d'un programme.

La majorité des critiques publiées concerne les deux premiers points: le troisième, malgré son importance est rarement abordé.

# 1. EXACTITUDE ET EFFICACITE

Le problème du programmeur est de choisir un "raffinement combiné" parmi tous ceux qui sont possibles dans un cas donné; cette décision est en fait très complexe car elle implique généralement une limitation des choix possibles pour les "raffinements combinés" ultérieurs. Avant chaque décision, le programmeur peut effectuer une forme de "sondage" pour évaluer les "raffinements" futurs que chaque décision implique; par exemple :

- [Wir71] rejeter dès le début une structure de programme à la suite d'une estimation "globale" du nombre d'essais tentés par l'algorithme pour déterminer une solution;
- 2) [Dij72, pages 65-67] rejet d'un schéma d'un programme sur base de la difficulté de démontrer l'exactitude d'un critère d'arrêt d'une boucle;
- 3) préférer une instruction de répétition de type "while" à une instruction "repeat" parce que les règles de vérification pour le "while" sont plus simples.

Comme contre-exemple voir [Led73] : il s'agit d'un programme de traitement de télégrammes, il est requis de signaler une "surtaxe" si un télégramme contient un ou plusieurs mots de plus de 12 caractères. Il résulte des raffinements successifs que dans le programme final, on continue à examiner si, parmi les mots qui suivent le premier mot qui provoque la surtaxe, il existe d'autres mots

de plus de douze caractères, opération évidemment inutile; comme le note l'auteur du programme [Led73, page 55] :

"Commitments affecting this point were at a very high level. One pays a high price for initial decisions. This I consider the most serious danger of this top-down method. That is, I think that the danger of not being aware of the full consequences of a decision is really more significant than that of making error".

Lorsqu'on utilise une méthode "top-down", il faut souvent beaucoup de courage de la part du programmeur pour remonter du niveau qui se révèle inefficace jusqu'au niveau qui est responsable de cette inefficacité. En aucun cas, la programmation structurée ne garantit, même pratiquée avec la plus grande attention, que les programmes composés sont les plus efficaces possibles.

Mais baser les décisions sur des critères d'efficacité du programme est très difficile à appliquer dès que la dimension du programme intervient :

- 1) l'efficacité ne peut jamais intervenir qu'après l'exactitude du programme;
- 2) l'efficacité ne peut pratiquement se comparer qu'en termes de temps d'exécution, il faut donc que le programme soit complètement élaboré ou que l'on puisse déterminer à priori les endroits cruciaux du programme au point de vue fréquence d'exécution. (Voir [Knu74] pour une discussion détaillée de ces différents points.)

En conclusion, les programmes les plus élégants ne sont pas nécessairement les plus efficaces; mais, le critère d'exactitude de Dijkstra n'exclut pas l'application postérieure d'un critère basé sur l'efficacité; d'autre part, une "optimisation prématurée" d'un algorithme est considérée comme très "dangereuse" même par les plus chauds partisans de l'efficacité [Knu74].

# 2. PROGRAMMES STRUCTURES ET PROBLEMES A RESOUDRE

"I now suggest that we confine ourselves to the design and implementation of intellectually manageable programs. If someone fears that this restriction is so severe that we cannot live with it, I can reassure him: the class of intellectually manageable programs is still sufficiently rich to contain many very realistic programs for any problem capable of algorithmic solution" [Dij72 a, page 864].

Dijkstra ne donne aucune preuve de cette affirmation, on peut toutefois tenir le raisonnement suivant : il a été prouvé que tout programme exprimé sous forme d'un "flow-chart" peut être transformé en un autre programme qui calcule les mêmes résultats en n'utilisant que des opérations de composition, de test ou d'itération; le programme transformé reprend les instructions (affectations, fonction, procédure) mais doit les compléter (dans certains cas) par l'introduction de nouvelles variables auxiliaires, de nouvelles instructions d'affectation et de nouveaux tests portant sur des variables auxiliaires; d'autre part, certaines opérations doivent être "recopiées". La théorie justifiant cette transformation est donnée par [Jac66] , et l'application pratique au "flow-charts" réguliers par [Mi172]. D'où 1'on peut déduire que l'ensemble des programmes sans "goto" contient suffisamment de programmes pour traiter tous les problèmes dont on peut exprimer par un flow-chart l'algorithme qui les résout. Malheureusement, l'ensemble des programmes "intellectuellement maniables" est un sous-ensemble des programmes sans "goto" ! L'absence de "goto" dans un programme ne signifie pas qu'il est "intellectuellement maniable"; par exemple, s'il fait un emploi abusif des "effet-debord". D'où l'on ne peut certainement pas conclure que l'ensemble des programmes "intellectuellement maniables" coîncide avec l'ensemble des solutions des problèmes à résoudre. Reste à déterminer "l'écart" entre les frontières des deux ensembles; à notre connaissance, personne ne s'est jamais attaqué à ce problème. En fait, jusqu'à présent, personne n'a pu présenter un exemple convainquant d'un problème qui n'aurait pas de solution (ou même pas de solution efficace) en programmation structurée.

#### Remarque :

On doit cependant faire une réserve (1) pour les problèmes dont la solution exige la coopération de deux "processus", comme par exemple, pour déterminer si deux arbres binaires sont "identiques" par rapport à une méthode de traversée (ils sont identiques si les contenus des noeuds visités forment deux séquences identiques, propriété que peuvent vérifier des arbres qui topologiquement seraient différents). Les programmes, apparemment les plus élégants, pour résoudre ce problème utilisent la notion de "coroutine". En fait, la difficulté provient du caractère parallèle et synchrone du problème proposé; le ramener à un algorithme séquentiel ne peut se faire sans complications et il en résulte une solution "artificielle".

Il est important que fondamentalement les programmes "intellectuellement maniables" ne soient pas limités aux programmes formés sur l'ensemble réduit d'instructions proposés par Dijkstra; en effet, les instructions choisies correspondent aux possibilités syntaxiques existant à l'époque de la composition de [Dij69], et il n'est pas exclu que par l'étude des raisonnements de programmation un autre ensemble d'instructions ne soit élaboré; par exemple, dans [Dij74], Dijkstra propose une méthode de programmation basée sur le non-déterminisme de la séquence d'opérations; en HPL, il n'y a pas de procédure au sens habituel de cette notion en programmation. La mise au point de nouvelles instructions est un problème complexe car il ne suffit pas de montrer qu'elles sont utiles dans certains cas particuliers, il faut réussir à convaincre de leur utilité globale.

En conclusion, on peut admettre qu'il existe une forte présomption que l'ensemble des programmes "intellectuellement maniables" contienne les programmes-solutions des problèmes qui peuvent se présenter. La présomption est fortement étayée par les résultats mis en évidence par Bohm et Jacopini [Jac66]; une précaution théorique de ce genre devrait exister pour toute suggestion de limitation; elle serait nécessaire pour pouvoir prêter plus d'attention aux propositions de bannissement des variables globales, des pointeurs et des instructions d'affectation.

# 3. DATA STRUCTURES "ABSTRAITES"

Le terme "data structure abstraite" tel qu'il est utilisé par Dijkstra a été généralement interprété dans le sens de "type abstrait" (abstract data type); cette interprétation est, à notre avis, abusive.

Elle est par exemple utilisée par B. Liskov pour son langage CLU.5 [Lis74] :

"An <u>abstract data type</u> defines a class of abstract objets which is completely characterized by the operations available in those objects. This means that an abstract data type can be defined by defining the characterizing operations for that type".

# et [Lis74 a] :

"We believe that the user of an abstract data type is interested in how the type's objects behave, and that the behaviour is best described in terms of a set of operations. We developed a set of criteria about the way abstract data types should be handled:

- A data type definition must include definitions of all operations applicable to objects of that type.
- A user of an abstract type need not know how objects of the type are represented in storage.
- A user of an abstract type may manipulate the objects <u>only</u> through the type's operations, and not through direct manipulation of the storage representation."

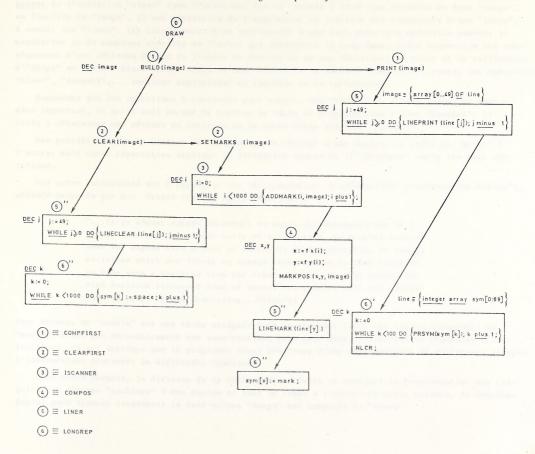
D'autre part, certains considèrent que le concept de "class" en SIMULA 67 permet d'associer à un

<sup>(1)</sup> La mise en évidence de cette difficulté est due à Strachey.

"objet" une collection d'opérations et est donc une possibilité d'implémenter la notion de "type abstrait".

A notre avis, le "type abstrait" est une "radicalisation" de la "data structure abstraite" de Dijkstra; de plus, il n'est pas prouvé qu'elle n'impose pas une contrainte trop sévère pour la conception. Il nous semble nécessaire que le programmeur puisse manipuler des "objets abstraits" sans être obligé de les définir sous forme d'un "type" avec ses opérations associées; au contraire, on peut considérer le "type" comme une collection de valeurs distinguables et qu'une variable d'un "type" donné possède un des membres de cette collection comme valeur; il n'est donc pas nécessaire de spécifier toutes les opérations dès la création du "type". Ce principe est mis en application en HPL (voir chapitre IV). En fait, les "abstract data structures" de Dijkstra sont un étalement dans le temps du choix de la représentation d'une variable; le choix est retardé le plus longtemps possible, car chaque décision de représentation implique un engagement qui a pour conséquence de limiter les différentes solutions possibles pour exprimer les détails d'une opération.

Le programme où Dijkstra présente de la manière la plus détaillée cette succession de choix, est "A second example of Step-Wise Program Composition" [Dij72, pages 50-63]. Le but du programme est de permettre d'utiliser une imprimante comme tracteur de courbes; la description du programme est relativement longue, nous nous sommes contentés de la résumer par un schéma qui représente sous forme d'un arbre la hiérarchie de "machines" imaginées par Dijkstra.



Dans le schéma, une flèche verticale signifie que l'opération située à l'extrémité supérieure de la flèche commence par l'exécution de l'opération pointée par la flèche; une flèche horizontale signifie qu'après exécution de l'opération située à gauche de la flèche, on exécute l'opération située à droite de celle-ci; ainsi l'opération "draw" implique l'exécution de "build", suivie de celle de "print", et ainsi de suite...

Dijkstra compose le programme en une série de six décisions : "compfirst", "clearfirst",...
(voir le schéma) qu'il interprète comme étant équivalentes à la spécification de "machines abstraites".
Les composants de ces machines sont indiqués sur le schéma par des numéros cerclés; si un même numéro cerclé est indicé par des apostrophes, ces apostrophes indiquent un ordre dans la composition des composants de cette "machine abstraite", par exemple (6)" a été composée avant (6)" et après (6)'.

Lors de la composition des opérations des "machines" ① à ⑤ , les effets-nets des composants de ces "machines" sont exprimés en fonction d'une data structure abstraite "image"; par contre, les trois composants de la cinquième "machine" (⑤', ⑥' et ⑥'') sont exprimés en fonction d'un "raffinement" de "image" en un ensemble ordonné de 50 "lines"; de même, les composants de la sixième "machine" résulte d'un raffinement de "line" en un tableau de 100 "symboles".

Le point important est que chaque décision de raffinement d'une data structure abstraite est prise dès qu'il est impossible de détailler - en termes de la data structure abstraite - comment une opération (dont on a défini l'effet-net) est réalisée; ainsi, on ne peut expliquer le fonctionnement de l'opération "clear" dont l'effet-net est de "mettre à zéro" une variable de type "image", en fonction de "image", il est nécessaire de l'expliciter en fonction des composants d'une "image", à savoir les "lines". Les conséquences d'un raffinement d'une data structure abstraite peuvent se manifester en de nombreux endroits de l'arbre qui schématise le programme; cette dispersion des conséquences d'une décision dépend de l'ordre de succession de ces décisions; en effet si le raffinement d'"image" en "lines" était décidé après la composition de la machine "compfirst", toutes les opérations "clear", "setmark",... seraient explicitées en fonction de ce raffinement.

Supposons que les opérations à concevoir pour remplir les "bulles" du schéma soient en nombre plus important, et qu'il soit décidé de confier la tâche de programmation à deux "équipes"; il reste à déterminer une méthode de division de la tâche entre les deux "équipes".

Une possibilité est de confier la réalisation de "build" à une équipe, et celle de "print" à l'autre; mais cette répartition implique la définition exacte de "l'interface" entre les deux opérations.

Une autre possibilité est d'appliquer une "dissimulation d'information" ("information hiding"), méthode proposée par D.L. Parnas [Par71, Par72, Par72 a] :

"...it is almost always incorrect to begin the decomposition of a system into modules on the basis of a flowchart. We propose instead that one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from the others. Since, in most cases, design decision transcend time of execution, modules will not correspond to steps in the processing..."[Par72]

Pour Parnas, un "module" est une tâche assignée à un programmeur (ou à une équipe homogène); un "module" n'est pas nécessairement une sous-routine, mais peut être un ensemble de séquences d'instructions, ce qui implique que le programme final résultera d'une agglomération de différents groupes d'instructions provenant de différents "modules".

Dans notre exemple, la division de la tâche peut se faire en confiant la programmation des cinquième et sixième "machines" à une équipe et tout le reste à l'autre; de cette manière, la deuxième équipe peut ignorer totalement le fait qu'une "image" est composée de "lines".

#### 4. CONCLUSIONS

La méthodologie proposée par Dijkstra possède comme qualité primordiale de rassembler d'une manière cohérente une collection de "bonnes pratiques", d'heuristiques destinées à renforcer la maîtrise du programmeur sur le programme qu'il développe.

Cependant, il ne faut pas se fier à son apparente simplicité, sa mise en pratique implique une sévère discipline. D'autre part, de nombreux points dont notamment les data structures "abstraites" sont très difficiles à utiliser pratiquement, vu le manque d'outils adéquats dans les langages actuels de programmation.

Maker P. T. . "System Custfortorotorotorotorod Programatical in APIPS

#### REFERENCES CITEES DANS LE TEXTE

- Abr 75 Abrahams, P., "Structured Programming Considered Harmful", SIGPLAN Notices, (April 1975), 13-24.
- Baker, F.T., "Chief Programmer Team Management of Production Programming",

  I B M Systems Journal, Vol. 11, n° 1 (1972), 56-73.
- Baker, F.T., "System Quality Through Structured Programming", in AFIPS

  Conference Proceedings, Vol. 41, part I, (1972), 339-343.
- Bau 71 Bauer, F.L., "Software Engineering," in *IFIP 71 Congress Proceedings*, invited papers, (1971), 267-274.
- Bur 73 Burstall, R.M. et Darlington, J., "A system which automatically improves programs," in *Proc. 3rd Intern. Conf. on Artificial Intelligence*, Stanford Univ., Stanford, Calif., (1973), 479-485.
- Dij 68 Dijkstra,E.W., "Go to statement considered harmful", Comm. ACM, Vol.11,  $n^{\circ}$  3 (mars 1968), 147-148.
- Dij 69 Dijkstra,E.W.,"Structured Programming", in Software engineering techniques",
  J.N. Buxton et B. Randell [Eds] NATO Scientific Affairs Division, Brussels,
  Belgium, (1970), 84-88.
- Dij 72 Dijkstra,E.W., Dahl,O.-J., and Hoare, C.A.R., Structured Programming, Academic Press, London, England (1972), 220 pp.
- Dij 72 a Dijkstra,E.W., "The humble programmer", *Comm. ACM*, Vol. 15, n° 10 (octobre 1972), 859-866.
- Dij 74 Dijkstra,E.W., "Guarded commands, non-determinacy and a calculus for the derivation of programs", EWD 418, à paraître dans *Comm. ACM* (manuscrit: juin 1974), 14 pp.
- Gil 74 Gilb, T., "Reliable EDP Application Design", Studentlitteratur, Lund, Suède, (1974), 200 pp.
- Hoa 69 Hoare, C.A.R., "An axiomatic approach to computer programming", Comm. ACM, Vol. 12,  $n^\circ$  10 (octobre 69), 576-580, 583.
- Jac 66 Jacopini, Guiseppe et Böhm, Corrado., "Flow-diagram, Turing machines, and languages with only two formation rules", Comm. ACM, Vol. 9, n° 5 (mai 1966), 366-371.
- Knu 68 Knuth, Donald E., "Fundamental algorithms, The art of computer programming", Vol. 1., Addison-Wesley, Reading, Mass. (1968), 634 pp.
- Knu 74 Knuth, Donald E., "Structured Programming with GO TO Statements", Computing Surveys, Vol. 6, n° 4 (décembre 1974), 261-301.

- Koe 67 Koestler, A., The Ghost in the Engine, Hutchinson, London, 1967.
- Koe 69 Koestler, A., et Smythies, J.R., The Alpbach Symposium: Beyond reductionism, Hutchinson, London, (1969).
- Led 73 Ledgard, Henry F., "The case for structured programming", BIT, Vol. 13, (1973), 45-57.
- Lis 74 Liskov, B., et Zillis, S., "Programming with abstract data types", SIGPLAN Notices, Vol. 9,  $n^{\circ}$  4 (avril 1974), 50-59.
- Lis 74 a Liskov, B., " A note on CLU", M.I.T. memo, (1974), 27 pp.
- Mills,H.D., "Mathematical foundations for structured programming", report FSC 72-6012, IBM Federal Systems Division, Gaithersburg, Md. (février 1972), 62 pp.
- Mills, H.D., "The New Math of Computer Programming", Comm. ACM, Vol. 18, n° 1 (janvier 1975), 43-48.
- Par 71 Parnas, D.L., "Information distribution aspects of design methodology", in IFIP Congress Proc., (1971), TA-3, 26-31.
- Par 72 Parnas,D.L., "On the criteria to be used in decomposing systems into modules", Comm. ACM, Vol. 15, n° 12, (1972), 1053-1058.
- Par 72 a Parnas, D.L., "Some conclusions from an experiment in software engineering techniques", in *Proc. AFIPS 1972 Fall Joint Computer Conf.*, (1972), 325-329.
- Par 75 Parnas, D.L., et Siewiorek, D.P., "Use of the Concept of Transparency in the Design of Hierarchically Structured Systems," *Comm. ACM* Vol. 18, n° 7 (juillet 1975), 401-408.
- PBH 74 Brinch Hansen P., "A programming methodology for operating system design," *IFIP*74 Congress Proceedings (1974), 394-397.
- See 74 Seegmüller, G., "Systems programming as an emerging discipline," IFIP 74 Congress Proceedings, (1974), 419-426.
- Sim 62 Simon, H.A., "The architecture of complexity," Proc. American Philosophical Society, Vol. 106, n° 6, (1962), 468-482.
- Sim 69 Simon, H.A., The sciences of the artificial, M.I.T. Press, (1969), 123 pp.
- Str 66 Strachey, Christopher, "System Analysis and Programming," Scientific American, Vol. 215, n° 3, (septembre 1966), 112-124.
- Str 71 Strachey, Christopher et Scott, Dana., "Toward a mathematical semantics for computer languages", in *Proceedings of the Symposium on Computers and Automata*, Polytechnic Institute of Brooklyn, (1971), 19-46.

- Str 73 Strachey, Christopher., "The varieties of programming language",

  \*\*Technical Monograph PRG-10, Oxford University Computing Laboratory,

  England (mars 1973), 20 pp.
- vWi 63 van Wijngaarden, A., "Generalised ALGOL", Ann. Review in Automatic Programming, Vol. 3 (1963), 17-26.
- Weinberg, G.M., "Review of Systematic Programming: an introduction",

  Datamation, (août 1974), 31 et 34.
- Wil 68 Wilkes, M.V., "The outer and inner syntax of a programming language", Computer J., Vol. 11, n° 3 (1968), 260-263.
- Wir 66 Wirth, N., et Weber, H., "EULER, a generalization of ALGOL, and its formal description", Comm. ACM, Vol. 9, n° 1 (janvier 1966), 13-25; n° 2 (février 1966), 89-99 et n° 12 (décembre 1966), 878.
- Wir 71 Wirth, N., "Program development by step-wise refinement", Comm. ACM, Vol. 14,  $n^{\circ}$  4 (avril 1971), 221-227.
- Wir 73 Wirth, N., "Systematic Programming: An Introduction", Prentice-Hall, (1973), 167 pp.
- Wir 74 Wirth, N., "On the design of programming languages", in *Proc. IFIP Congress 1974*, North-Holland, (1974), 386-393.
- Woo 71 Woodger, M., "On semantic levels in programming," *IFIP 71 Congress Proceedings*, (1971), TA-3, 79-83.



UNIVERSITE DE CIEGE FACULTE DES SCIENCES APPLICUEES

A TWO LONG AND CONTROL OF THE CONTRO

STET BENCTOD