☞ *Université de Nancy I* — *UER de Mathématiques* ☜

ETAPES
SUR LE CHEMIN
du
GENIE LOGICIEL

∞∞∞∞∞∞∞∞

Thèse présentée à l'Université de Nancy I

pour l'obtention du grade de

DOCTEUR ES SCIENCES MATHEMATIQUES

(*Mention : Informatique*)

par

Bertrand Meyer

∞∞∞∞∞∞∞∞

Thèse soutenue le 9 septembre 1985
devant la Commission d'Examen,
composée de MM. :

Claude **Pair** (*Président*),

Alain **Bossavit**,
Jean-Pierre **Finance**,
Jean-François **Perrot** (*Rapporteurs*)

Jean-Raymond **Abrial**,
Jean-Claude **Derniame**,
Michel **Galinier**,
Gilles **Kahn** (*Examinateurs*).

TOME 1

☞ *Université de Nancy I* — *UER de Mathématiques* ☜

ETAPES
SUR LE CHEMIN
du
GENIE LOGICIEL

∞∞∞∞∞∞∞∞

Thèse présentée à l'Université de Nancy I

pour l'obtention du grade de

DOCTEUR ES SCIENCES MATHEMATIQUES

(*Mention : Informatique*)

par

Bertrand Meyer

∞∞∞∞∞∞∞∞∞

Thèse soutenue le 9 septembre 1985
devant la Commission d'Examen,
composée de MM. :

Claude **Pair** (*Président*),

Alain **Bossavit**,
Jean-Pierre **Finance**,
Jean-François **Perrot** (*Rapporteurs*)

Jean-Raymond **Abrial**,
Jean-Claude **Derniame**,
Michel **Galinier**,
Gilles **Kahn** (*Examinateurs*).

TOME 1

Examinateurs :

M. Jean-Raymond Abrial, Ingénieur-Conseil à Paris

M. Alain Bossavit, Attaché au Chef du Service "Informatique et Mathématiques Appliquées" de la Direction des Etudes et Recherches d'EDF

M. Jean-Claude Derniame, Professeur à l'Université de Nancy-I

M. Jean-Pierre Finance, Professeur à l'Université de Nancy-I

M. Michel Galinier, Directeur de l'Institut de Génie Logiciel (IGL)

M. Gilles Kahn, Ingénieur de Recherche à l'INRIA

M. Jean-François Perrot, Professeur à l'Université de Paris-VI

M. Claude Pair (Président), Professeur à l'Institut National Polytechnique de Lorraine

RESUME

Les recherches présentées dans cette thèse sur travaux couvrent une période de dix ans (1975-1985). Elles portent sur divers aspects du génie logiciel et plus particulièrement de la méthodologie de la programmation ; on y trouvera des études sur la structure macroscopique des systèmes logiciels, la spécification formelle, la modularité, les structures de données, l'algorithmique, les langages de programmation, la conception par objets, les méthodes de construction systématique de programmes, les outils logiciels interactifs, l'édition structurelle et la programmation des super-ordinateurs.

Vint-et-un articles sur ces différents sujets sont regroupés dans la troisième partie de la thèse. La première partie est un document de synthèse sur les principaux problèmes du génie logiciel ; la seconde, une introduction thématique aux articles qui suivent.

TABLE DES MATIERES

LISTE DES TEXTES REPRODUITS DANS LA TROISIEME PARTIE

[76a] "Description des Structures de Données," *Bulletin de la Direction des Etudes et Recherches d'Electricité de France, Série C (Informatique)*, no. 2, Clamart, 1976.

Méthodes de Programmation, Eyrolles, Paris, 1978. 687 pages. (Avec Claude Baudoin). Edition corrigée, 1984. Trois extraits:

[78a] extrait du chapitre 5 (*Structures de données*)

[78b] extrait du chapitre 8 (*Vers une méthodologie de la programmation*): modularité.

[78c] extrait du chapitre 8 (*Vers une méthodologie de la programmation*): spécification en Z.

[79a] "Les Langages de Spécification," *Bulletin de la Direction des Etudes et Recherches d'Electricité de France, Série C (Informatique)*, no. 1, pp. 39-60, Clamart, 1979. (Avec Michel Demuynck).

[79b] "Quelques concepts importants des langages de programmation modernes, et leur expression en SIMULA 67," *Bulletin GROPLAN (AFCET), 9* ; également dans *Bulletin de la Direction des Etudes et Recherches d'Electricité de France, Série C (Informatique)*, no. 1, pp. 89-150, Clamart, 1979.

[80a] "A Basis for the Constructive Approach to Programming," in *Information Processing 80 (Actes du congrès IFIP 1980, Tokyo, 6-9 octobre)*, ed. S. H. Lavington, pp. 293-298, North-Holland Publishing Company, Amsterdam (Pays-Bas), 1980.

[80b] "Un Calculateur Vectoriel: Le Cray-1 et sa Programmation (Version 2)," Atelier Logiciel no. 24, HI-3452/01, Electricité de France, Clamart, 4 juin 1980. Extraits.

[80c] "Article *Langages de Programmation*," in *Encyclopédie "Techniques de l'Ingénieur"*, Paris, décembre 1980.

[81a] "The Design of Vector Programs," in *Algorithmic Languages*, ed. Jaco de Bakker et R.P. van Vliet, pp. 99-114, North-Holland Publishing Company, Amsterdam (Pays-Bas), 1981. (Avec Alain Bossavit).

[82a] "Towards a Two-Dimensional Programming Environment," in *Proceedings of the European Conference on Integrated Computing Systems (ECICS 82), Stresa (Italie), 1-3 September 1982*, ed. Pierpaolo Degano and Erik Sandewall, North-Holland, Amsterdam (Pays-Bas), 1983. (Traduction anglaise, avec quelques corrections, de l'article paru en français dans les Actes des Journées Bigre, Grenoble, janvier 1981).

[82b] "Principles of Package Design," *Communications of the ACM*, vol. 25, no. 7, pp. 419-428, juillet 1982.

[84a] "Cépage : Un Editeur structurel Pleine Page," in *Second Colloque de Génie Logiciel*, pp. 153-158, AFCET, Nice, juin 1984. (Avec Jean-Marc Nerson). **Note:** on a ajouté à la suite de cet article, au moment de la mise sous presse de la thèse, un document d'août 1985 décrivant la nouvelle version de Cépage.

(suite au verso)

[84b] "An Application of Program Transformation to Supercomputer Programming," in *VAPP Conference*, Oxford, Août 1984. Version révisée, à paraître dans *Computer Physics Communications*. (Avec Alain Bossavit).

[85a] "On Formalism in Specifications," *IEEE Software*, vol. 3, no. 1, pp. 6-25, janvier 1985.

[85b] "Showing Programs on a Screen," *Science of Computer Programming*, vol. 5, no. 2, 1985. (Avec Jean-Marc Nerson et Soon Hae Ko).

[85c] "Control Structures," Notes de cours, University of California, Santa Barbara, janvier 1985. (Chapitre 3 du livre "Applied Programming Methodology"). Extrait.

[85d] "Modularity," Notes de cours, University of California, Santa Barbara, janvier 1985. (Chapitre 6 du livre "Applied Programming Methodology"). Extrait.

[85e] "Incremental String Matching," *Information Processing Letters*, à paraître.

[85f] "The Software Knowledge Base," in *8th International Conference on Software Engineering (Londres, Août 1985)*.

[85g] *M: A System Description Method*, en cours, 1985.

REMERCIEMENTS

Je suis très vivement reconnaissant aux membres du Gotha francais de la programmation qui ont bien voulu accepter de participer au jury de cette thèse.

Je voudrais tout d'abord remercier M. Claude Pair qui me fait l'honneur de le présider ; il a été l'un des tous premiers, dans les milieux universitaires francais, à promouvoir la programmation comme sujet digne d'une étude scientifique, et tout chercheur de ce domaine sait quel rôle il a joué par ses propres travaux de recherche, par la mise en place du CRIN et par son action dans les organismes gouvernementaux, pour en faire une discipline à part entière.

Ma dette est considérable à l'égard de Jean-Pierre Finance, grâce à qui cette thèse a pu être présentée à Nancy, et qui a su m'obliger (avec la combinaison de gentillesse et de fermeté qui le caractérise) à faire en sorte que sa forme réponde à peu près à ce qu'on attend d'un travail de cette sorte. Je lui suis plus reconnaissant encore pour nos déjà nombreuses années de collaboration, avec le plus souvent l'AFCET comme prétexte (autour de GROPLAN, du congrès de Nancy en 1980, du bureau informatique de la grande époque et surtout de TSI), et pour nos longues discussions techniques, auxquelles la soutenance de cette thèse ne mettra certainement pas fin.

Quiconque a travaillé de près ou de loin avec moi sait ce que je dois à Jean-Raymond Abrial, à sa profonde culture scientifique, à sa rigueur intellectuelle, à la richesse de son expérience de concepteur. Aucune influence n'a été plus déterminante que la sienne sur mon travail depuis 1977, et je le remercie profondément d'avoir bien voulu accepter de participer à ce jury.

Alain Bossavit a lui aussi joué un grand rôle, d'abord en fournissant la preuve vivante qu'on pouvait être chercheur à EDF, mais surtout en m'apportant grâce son expérience de spécialiste en mathématiques appliquées de nombreux éléments de méthode et toute une source de problèmes nouveaux. Ayant pris d'emblée mes travaux au sérieux (qu'ils en fussent dignes ou non), tout en ne me laissant pas quitte de mes affirmations à l'emporte-pièce, il a été à l'origine d'une collaboration dont témoignent les deux publications communes reproduites ci-après, et qui continue malgré la distance.

Je dois aussi beaucoup à Michel Galinier pour m'avoir depuis longtemps soutenu et encouragé, pour avoir lui aussi été de l'aventure AFCET—génie logiciel—TSI, pour m'avoir fait bénéficier de sa connaissance approfondie des milieux industriels, et pour avoir démontré par l'exemple réussi de ses deux carrières successives que la coupure entre recherche et industrie n'est pas inéluctable en génie logiciel.

Avec Jean-Pierre Finance et Alain Bossavit, Jean-François Perrot a bien voulu être rapporteur de cette thèse et je le remercie de l'effort qu'il a accepté d'y consacrer, lui apportant l'expérience d'un spécialiste de la programmation passionné aussi bien par les fondements théoriques de la discipline que par les nouvelles techniques qui s'offrent aujourd'hui au praticien, programmation fonctionnelle et programmation par objets.

Gilles Kahn m'a donné l'exemple d'un chercheur en programmation qui est aussi un programmeur. Il m'a aussi apporté, comme à tous ceux qui ont compris (longtemps après lui) le potentiel considérable des éditeurs structurels, l'exemple d'un des premiers systèmes de ce type qui ait abordé l'ensemble des applications possibles, et qui reste sans doute le plus complet de ceux qui ont été développés jusqu'ici. Un bref examen de Mentor et de Cépage suffit pour se convaincre de ce que le second doit au premier.

Jean-Claude Derniame a étudié depuis longtemps les principes de la programmation modulaire et les a mis en pratique dans ses travaux de conception de langages ; nous nous sommes rencontrés il y a quelque temps déjà dans ces parages, que j'appellerais volontiers le Mont Parnas. Je voudrais aussi à travers Jean-Claude Derniame, Directeur du CRIN, ainsi qu'à travers Claude Pair et Jean-Pierre Finance, remercier ce laboratoire et l'Université de Nancy 1, qui ont accepté de m'accueillir pour cette thèse.

Remerciements

Plusieurs des textes reproduits plus loin ont été écrits en collaboration, et j'ai grandement bénéficié de l'aide et de l'expérience de mes co-auteurs : Alain Bossavit ; Claude Baudoin, qui à Stanford puis pendant les trois années qu'a duré la préparation des *Méthodes de Programmation* a presque toujours supporté mon mauvais caractère et mes imparfaits du subjonctif, et grâce à qui un livre véritable a pu résulter de ce projet étrange ; Michel Demuynck, qui a apporté au travail sur la spécification son expérience des systèmes d'information ; et Jean-Marc Nerson, qui a pris au sérieux l'idée de ce qui allait devenir Cépage et, de quelques indications au tableau noir, a fait un système d'aide à la programmation.

Je voudrais aussi remercier, trop brièvement, tous ceux qui m'ont aidé à apprendre mon métier : M. Jean-Claude Simon, dont l'enseignement de reconnaissance des formes à l'X et a l'Université Paris VI, en 1971-72, m'a conduit à devenir informaticien ; mes professeurs de Stanford, J. McCarthy, R.W. Floyd et D.E. Knuth ; les grands informaticiens vis-à-vis desquels ma dette transparaît à chacune des pages de cette thèse, tout d'abord C.A.R. Hoare et E.W. Dijkstra, mais aussi O.-J. Dahl et K. Nygaard, D. Parnas, M. Sintzoff, C.B. Jones, B. Liskov et (dans un domaine plus éloigné de mes propres travaux) B. Boehm ; les membres non encore cités de l'équipe de l'AFCET, de TSI et du groupe "génie logiciel", en particulier Jean-Claude Rault et son savoir encyclopédique, Emmanuel Girard et sa langue acérée, Jacques André (animateur d'un projet de manuel de génie logiciel qui n'a jamais vu le jour mais a suscité bien des discussions enrichissantes), Dominique Potier ; Claude Kaiser, qui m'a fait confiance pour enseigner à l'IIE ce que j'étais en train d'apprendre, et Didier Thibault qui a partagé l'expérience de Télécom ; mes collègues d'EDF, particulièrement D. Spohn (qui, chef du Département MMI en 1974, n'imaginait sans doute pas les conséquences de sa remarque "*vous devriez vous renseigner d'un peu plus près sur la programmation structurée...*"), Michel Dostatni qui a guidé mes premiers pas aux Etudes et Recherches, Michel Gondran, Eric de Drouas et les membres de la Division APCOL, tout particulièrement l'équipe de l'"Atelier Logiciel" : Eugène Audin, Gérard Brisson (dont la collaboration a été déterminante lors de la mise sur pied d'un programme moderne de formation à la programmation en 1975, et un peu plus tard dans la réalisation de nos premiers outils), Bernard Logez et Françoise Vapné.

Il serait injuste d'oublier, à côté des individus, les institutions qui m'ont aidé. Electricité de France, plus précisément le Service "Informatique et Mathématiques Appliquées" de la Direction des Etudes et Recherches, m'a fourni pendant plusieurs années un environnement propice à la recherche appliquée, dont sont sortis nombre des travaux présentés ici ; la Direction des Etudes et Recherches est aussi l'éditeur du livre *Méthodes de Programmation*, et je lui suis en outre reconnaissant d'avoir bien voulu assurer le tirage de cette thèse. J'ai consacré depuis 1978 beaucoup de temps à l'AFCET, et je souhaite que des chercheurs et des praticiens plus nombreux comprennent à leur tour combien, lorsqu'il s'agit de militer dans une société savante, donner est aussi recevoir. Plus récemment, l'Université de Californie m'a fait découvrir un monde parfois rude mais toujours stimulant.

Enfin, bien sûr, je remercie ma famille : épouse, ascendants, descendants et collatéraux.

Cette thèse est dédiée à la mémoire de ma tante Simone Kahn (Paris 1915-Auschwitz 1944).

PRESENTATION

On peut définir le *génie logiciel* comme la discipline qui a pour objet de favoriser la production de logiciel de qualité en s'appuyant sur des méthodes scientifiques. Il convient ici d'inclure dans la notion de *logiciel* non seulement les programmes et leurs données, mais aussi tout ce qui sert à leur construction (spécifications, documents de conception, etc.) et à leur utilisation (manuels, outils d'aide) ; on doit aussi se souvenir que la *qualité* d'un produit logiciel dépend non seulement de ses caractéristiques techniques comme sa fiabilité, sa facilité d'emploi ou sa facilité de modification, mais aussi bien des conditions économiques de sa réalisation (coûts et délais).

Les travaux présentés ci-après se rapportent à des aspects variés de cette discipline. Pour la forme, la thèse comprend trois parties qu'il est plus commode de citer ici en ordre inverse : la troisième, de loin la plus longue, reproduit vingt-et-un articles ou extraits de livres écrits entre 1975 et 1985 ; la seconde présente les principales idées développées dans ces textes ; et la première ouvre la voie aux deux autres par une description synthétique du domaine. Pour le fond, il s'agit d'aborder le génie logiciel non pas selon un plan d'attaque uniforme mais plutôt par une série de coups de boutoir latéraux et, du moins l'espère-t-on, convergents.

Les sujets immédiats des textes de la troisième partie sont éclectiques : ils vont de la spécification modulaire des systèmes informatiques à la programmation rationnelle des super-ordinateurs, en passant par l'édition structurelle et la programmation par objets. Mais la collection présentée tire son unité de la rémanence des thèmes sous-jacents : telle une balle de billard électrique, la discussion revient régulièrement aux mêmes ressorts, un petit nombre d'idées maîtresses qui définissent une conception générale de la construction de programmes de qualité.

On peut citer parmi les principaux de ces thèmes :

- la nécessaire complémentarité entre **méthodes, outils et langages** pour assurer des progrès réels en génie logiciel ;
- le rôle des **spécifications**, particulièrement des spécifications formelles ;
- la nécessité de produire des décompositions **modulaires** des systèmes, au niveau de la spécification comme à celui de la mise en oeuvre, et d'étudier les critères qui permettent d'affirmer qu'une décomposition est bien modulaire ;
- l'étude des **relations** entre composants d'un système logiciel ;
- le rôle des **outils logiciels interactifs**, composants indispensable d'un "environnement de développement" propice à la production de logiciel de qualité ;
- le rôle des **langages de programmation** dans le processus de construction des programmes ;
- l'importance des **structures de données** et la nécessité de les définir de façon rigoureuse (*types abstraits*) ;
- l'utilisation de méthodes de **construction systématique** de systèmes à partir de leur spécification ;
- l'influence des architectures de **super-ordinateurs** sur la construction de programmes.

Ces idées se sont précisées au fil des années et de l'expérience de l'auteur. Si l'on classe les textes eux-mêmes en fonction de leurs sujets directs, on trouvera les catégories suivantes :

- une série d'articles et de chapitres de livres sur la *méthodologie de la programmation*, depuis l'ouvrage *Méthodes de Programmation* de 1978 (écrit avec Claude Baudoin) jusqu'à des fragments d'un ouvrage en préparation ;
- des articles sur la *spécification formelle*, appliquée d'abord aux structures de données (1976) puis à des systèmes plus généraux ;
- des études sur la *construction rigoureuse* de programmes à partir de leur spécification, dans la lignée des travaux de Dijkstra ;
- des descriptions d'*outils logiciels*, en particulier de divers progiciels réalisés à EDF et, plus récemment, de l'éditeur structurel et visuel Cépage (en collaboration avec Jean-Marc Nerson) ;
- des présentations d'*algorithmes* dans divers domaines, avec un accent particulier mis sur leur présentation systématique ;
- des travaux sur la programmation des *ordinateurs vectoriels* du type du Cray-1, en collaboration avec Alain Bossavit.

Il est difficile de présenter convenablement un travail tel que celui-ci, qui appartient au genre un peu particulier de la *thèse sur travaux*, sans donner quelques éléments biographiques ; l'expérience professionnelle de l'auteur transparaît souvent dans les pages qui suivent, où l'on trouvera aussi quelques échos des actions menées en France depuis une dizaine d'années pour faire progresser les idées modernes sur le génie logiciel. Revenant de Stanford en octobre 1974, l'auteur s'est trouvé confronté à la pratique de la programmation scientifique dans un grand centre de calcul (l'un des plus importants d'Europe, celui de la Direction des Etudes et Recherches d'EDF) et, des années durant, s'est efforcé d'y introduire les principes du génie logiciel à travers l'organisation de cours et de séminaires, la réalisation d'outils logiciels, la mise en place d'une équipe spécialisée (l'*"Atelier Logiciel"*), et la rédaction de nombreuses notes internes. En tant qu'ingénieur-chercheur, puis chef d'une Division responsable, entre autres activités, de l'assistance aux utilisateurs, de la formation, des bibliothèques de programmes et d'autres activités directement en prise sur la pratique quotidienne dans l'industrie, il a pu mesurer l'effort nécessaire pour faire passer les concepts de la recherche vers la production.

Il était nécessaire, pour mener à bien cette activité de conseil et de réalisation, de suivre de près les développements de la recherche. L'ouvrage *Méthodes de Programmation*, cité plus haut, tentait une synthèse de nombreuses idées contemporaines sur la programmation selon trois points de vue complémentaires : méthodologie (modularité, construction systématique) ; techniques (algorithmes, structures de données) ; langages (trois langages étaient décrits en détail et appliqués à de nombreux exemples : Fortran, Algol W, PL/I). Au cours des années suivantes, diverses publications ont abordé des domaines variés du génie logiciel : travail sur la spécification en liaison avec le développement du langage Z par Jean-Raymond Abrial ; étude de la programmation vectorielle, suscitée par l'acquisition d'un Cray-1 par EDF, et continuant une réflexion engagée avec Alain Bossavit sur la programmation des applications numériques ; conception de divers progiciels, en particulier Gescran (gestion d'écrans avec multi-fenêtrage) et l'outil associé de "CAO d'écrans" Conscran, avec l'équipe de l'Atelier Logiciel[1] ; édition structurelle (Cépage) et algorithmes associés ; conception et programmation par objets, en particulier avec le langage Simula 67, utilisé dans l'équipe de l'Atelier Logiciel pour construire plusieurs systèmes (dont Conscran et un petit système de courrier électronique).

[1] La direction des Etudes et Recherches a depuis lors déposé la marque Conscran, qui couvre désormais Gescran et tous les produits associés. Nous nous en tenons ici aux dénominations d'origine, auxquelles se rapportent les mentions que l'on trouvera dans les articles de la troisième partie.

En sus de son travail à EDF, l'auteur a joué à partir de 1978 un rôle actif dans les diverses instances de l'AFCET (membre puis vice-président du bureau du Collège AFCET-INFORMATIQUE, membre du conseil de l'Association, président du comité de programme du congrès informatique de 1980). Avec Michel Galinier, alors Professeur à l'Université de Toulouse, il a créé à l'AFCET un groupe de travail "génie logiciel" qui est à l'origine de la série des colloques AFCET de même nom. Il a enfin joué un rôle important dans le lancement par l'AFCET, en collaboration avec les Editions Dunod et avec le soutien de l'Agence de l'Informatique, de la revue *Techniques et Science Informatiques* (TSI), dont il est, depuis sa création en 1982 et jusqu'à la fin de 1985, le rédacteur en chef.

Depuis la fin de 1983, B. Meyer est professeur associé à l'Université de Californie (Santa Barbara). Cette immersion dans un milieu cette fois-ci purement universitaire lui a permis d'approfondir ses activités de recherche, de consacrer plus d'efforts à des domaines théoriques comme la sémantique des langages de programmation, de faire progresser la rédaction de plusieurs travaux en cours, et d'aborder une nouvelle classe d'outils de génie logiciel avec le développement d'un système général pour la gestion de projets et la gestion de configurations, le système SKB. Il n'a toutefois pas abandonné tout lien avec le milieu industriel ; en juin 1985 il a créé une petite société pour développer un produit commercial fondé sur les mêmes idées que le prototype d'éditeur structurel Cépage.

Cette thèse reflète le double métier de son auteur : ingénieur et chercheur (triple, si l'on ajoute l'enseignement). Elle a pour ambition de marquer quelques jalons sur la route encore incertaine du génie logiciel, et de fournir quelques éléments d'une *méthodologie de la programmation appliquée* qui pourrait servir à améliorer (dès aujourd'hui peut-être) la façon dont on développe du logiciel dans l'industrie, et la qualité des produits qui en résultent.

SUR LE CHEMIN DE RONDE

Cette thèse est un tour des remparts. D'un côté, la Ville, avec son enchevêtrement de bâtisses aux styles si mélangés, aux formes si contournées, aux couleurs si repassées que le touriste se demande comment le peuple bourdonnant qui s'y affaire et dont on distingue ici et là quelques-uns des colifichets caractéristiques (une calculatrice hexadécimale, un crayon maintenu entre mèche et oreille, ou l'une de ces petites valises appelées attaché-case, *qui ressort à peine sur le fond uniforme des costumes trois-pièces et des quelques jupes faussement fantaisistes) peut trouver la tranquillité d'esprit pourtant indispensable à l'exécution de sa mission, qui est de construire le logiciel d'aujourd'hui.*

Une sorte de brouillard continuel flotte au-dessus de la cité, empêchant le visiteur de bien distinguer les quartiers. Voici pourtant la place Fortran, qui malgré les lubies récentes d'un conseil d'arrondissement soudain porté vers le baroque a réussi à conserver son allure simple et austère. Ce lacis de ruelles, au centre, où la foule épaisse se fraye à grand peine un chemin, c'est le faubourg Cobol ; du côté du passage PL/I, celui qui mène à l'immense quartier IBM, le fouillis devient si inextricable que l'on a peine a comprendre où le guide trouve à gloser sur l'"architecture" de l'endroit.

Le voyageur ne peut manquer d'être frappé par l'âge de la plupart des bâtiments. Seuls deux ou trois quartiers plus récents, serrés contre les remparts, ont conservé un peu de leur fraîcheur. Propre comme un chalet suisse, le faubourg Pascal, naguère encore à la mode, semble aujourd'hui bien désert : c'est peut-être que les constructions y sont réputées peu résistantes. On dit que ses habitants émigrent maintenant les uns après les autres vers l'une de ces nouvelles zones dont les promoteurs n'en tiennent que pour une architecture purement utilitaire, au mépris de toute considération d'esthétique, et dont le nom, de façon si caractéristique, est formé d'une simple lettre : le Quartier "C".

L'esprit d'entreprise, au demeurant, ne fait pas défaut chez les bâtisseurs, mais il est rarement couronné de succès ; c'est peut-être que les habitants, après tout, tiennent à leurs coursives nauséabondes. On peut cependant voir, ici et là, les traces de quelques projets qui chacun en leur temps soulevèrent les passions en même temps que la poussière. Sur le plus récent et le plus grandiose, les discussions continuent d'aller bon train : c'est le compleze Ada, autour de la caserne, et si de luxueux prospectus aux couleurs vertes de ce projet continuent d'être ponctuellement distribués dans les boîtes aux lettres, force est bien de noter que les fondations commencent seulement à sortir de terre, et que plus d'une famille un instant alléchée s'est aujourd'hui, lasse d'attendre, installée ailleurs. Les plus blasés parmi les résidents de longue date prédisent au compleze Ada le sort de la résidence Algol 68 qui devait peu à peu, selon ses architectes, s'étendre à toute la ville ; son plan seul, et jusqu'au mode d'emploi de ce plan, étaient au dire des rares savants qui pouvaient y entendre quelque chose de véritables monuments de science architecturale. De tous ces efforts ne subsistent que quelques affiches publicitaires jaunies au milieu d'un terrain vague.

Que l'on porte d'un seul coup son regard vers la campagne, et le contraste sera saisissant. Peu de choses remarquables, à vrai dire, aux abords mêmes de la cité, zone incertaine, obscure et marécageuse où seuls s'aventurent quelques voyageurs novices ou particulièrement audacieux ; c'est au loin, vers les riants paysages de la recherche, que sera tout de suite attirée l'attention de l'observateur.

Voici la plaine de la Programmation Modulaire avec sa marquetterie de champs protégés par des haies épaisses, que seules d'étroites ouvertures permettent de pénétrer. Si pourtant d'aventure on observe cette région à la jumelle, le paysage apparaîtra plus valonné qu'on ne l'avait cru tout d'abord : chaque champ est en réalité disposé au pied d'une petite colline, son gardien tutélaire en quelque sorte, que les habitants de cette contrée appellent un Type Abstrait. Un peu plus loin, vers le nord, le paysage ne change guère, mais les champs semblent plus autonomes, comme si chacun vivait de sa vie propre : c'est la province de la Programmation par Objets, dont la région occidentale est peuplée d'oiseaux infatigables qui volètent de pré en pré, comme s'ils devaient éternellement transmettre on ne sait quels messages.

Cette chaîne de sommets et de glaciers, là-bas, est le massif de la Programmation Rigoureuse, ainsi nommé pour le climat qui y règne. Son promontoire central, le pic des Spécifications Formelles, a eu raison du piolet et des crampons de plus d'un intrépide. Les schistes du versant oriental, par un curieux accident de métamorphisme, forment trois stries parallèles et parfaitement verticales : c'est la combe de la Méthode Déductive.

Cette autre montagne que dévalent des torrents impétueux est le mont des Super-Ordinateurs ; beaucoup de ceux qui y sont allés décrivent avec ravissement l'intarissable débit de ses rivières ; mais d'autres se plaignent que ce spectacle soit, après tout, bien monotone.

Peu de circulation entre ou sort de la ville malgré les portes percées de place en place dans la muraille. Une rare charrette marquée d'un sigle incompréhensible ("AFCET", "IEEE", "TSI", croit-on distinguer) franchit parfois les fortifications mais, le plus souvent, rebrousse chemin bien vite. Voici pourtant qu'un individu sort par une porte dérobée ; il semble hésiter, reste sur place et rentre finalement. Mais non, le voilà qui sort encore, porte son regard vers le lointain comme si l'appel des espaces illimités était tout à coup devenu irrésistible, puis à nouveau vers la porte qui lui permettrait de retrouver la ville, son confort et ses certitudes. Il ne sait décidément pas ce qu'il veut.

La visite des fortifications comprend deux parties. Pour suivre la première, il importe d'avoir de bonnes jambes, car le guide ne s'attarde pas : quelques minutes suffisent pour faire le tour complet, et chaque particularité du paysage ne suscite qu'un bref commentaire. C'est ensuite, pendant la seconde partie de la promenade, que l'on peut s'attacher aux détails ; un télescope disposé çà et là sur le chemin de ronde permet au touriste d'observer de plus près telle ou telle région qui a momentanément retenu son attention. C'est bien ce que nous ferons au cours de notre promenade nonchalante, sans du reste chercher à voir en détail tout ce qu'indique le guide : nous négligerons ainsi et la Vallée des Métriques, où d'inlassables arpenteurs mesurent et re-mesurent les distances entre les bornes disposées à chaque étape, et le camp militaire de la Gestion de Projets, et le riche plateau de la Programmation Fonctionnelle aux sillons si parallèles, et même la forêt des Systèmes Experts où, disent certains, plus d'un voyageur momentanément égaré a perdu la raison, mais que d'autres tiennent pour un simple mirage. Nous n'oublierons jamais tout à fait (le pourrions-nous?) le bruit de la cité et ce qu'il faut bien appeler sa pollution. Peut-être apprendrons-nous un peu, en portant sans cesse nos regards d'un côté et de l'autre, ce qui de la campagne finira bien par passer quelque jour dans la ville.

PREMIERE PARTIE :

PANORAMA

CHAPITRE 1

SURVOL PARTIAL D'UNE DISCIPLINE

1.1 - DU SLOGAN A LA REALITE

Au commencement ce ne fut guère qu'un jeu de mots, à tout le moins une provocation. Lorsqu'en 1968 le comité scientifique de l'OTAN se mit en tête d'inviter quelques-uns des plus grands spécialistes mondiaux de la programmation à réfléchir ensemble sur ce que l'on commençait d'appeler la crise du logiciel, le titre choisi pour la conférence sonna comme un défi : parler de *software engineering*, de génie logiciel, et donc suggérer l'analogie avec de respectables disciplines telles que le génie civil ou le génie chimique, ce ne pouvait être qu'une manière de dérision, un procédé pour alerter les informaticiens sur le caractère rudimentaire de leurs méthodes et de leurs outils quotidiens.

La conférence se déroula à Garmisch, en Bavière, à l'automne de 1968, et se termina sur un cri d'alarme. Face à l'extraordinaire développement de la puissance du matériel, le logiciel n'avait pas suivi. Projets en retard, budgets dépassés, informatisations manquées, systèmes monstrueux, graves erreurs de fonctionnement : innombrables étaient les exemples qui montraient l'urgence d'une réflexion en profondeur sur les moyens d'améliorer la qualité des logiciels et leur mode de production.

Dix-sept ans après la conférence de Garmisch, le terme de génie logiciel et son équivalent anglais ne sont plus ressentis comme oxymorons. C'est même plutôt leur succès qui frappe : on ne compte plus les enseignements universitaires, les congrès, les revues et les groupes de travail qui s'en réclament. Mais cette faveur même peut sembler suspecte : au-delà de l'engouement pour une appellation bien tournée, le métier qu'elle désigne est-il vraiment devenu un *génie*, une discipline d'ingénieur? On peut en débattre. Que la réflexion ait progressé, que des idées majeures aient fait leur chemin, il serait difficile de le nier ; mais l'effet réel de ces progrès dans les entreprises, là où se développe le logiciel qui devrait être de qualité industrielle, c'est une autre affaire. Les réflexions qui suivent apporteront peut-être quelques timides éléments de réponse.

1.2 - LOGICIEL, GENIE LOGICIEL

Avant de commencer notre promenade, il est utile de préciser l'objet de l'étude. Qu'appelle-t-on exactement "génie logiciel"?

Une définition parfois proposée (pas tout à fait sérieusement) est que cette expression sert à désigner ce que l'on appelait naguère tout bonnement "programmation", et doit être employée en présence de ses supérieurs hiérarchiques, des jeunes personnes à qui l'on fait la cour, et plus généralement chaque fois que l'on souhaite donner une apparence plus noble à ce que l'on fait, le terme de programmation étant réservé aux circonstances où l'on doit vraiment se faire comprendre, par exemple lorsqu'on affecte une tâche à l'un de ses subordonnés. Au-delà de la boutade, cette formule a le mérite de rappeler que la distance entre la simple notion de programmation et celle de génie logiciel n'est pas aussi grande que pourraient le faire croire certains clivages entre chercheurs.

A l'autre extrême, on rencontre aussi une définition très limitée qui ne s'intéresse qu'aux circonstances spécifiques de la production industrielle de logiciel : gestion de grands projets, gestion de configurations, organisation des équipes, mesures et modèles de coût et de fiabilité, etc. Cette définition exclut tout ce qui se rapporte aux problèmes classiques de la programmation (langages, outils de programmation, théorie) ; elle procède d'une vue selon laquelle les difficultés de la production de logiciel dans l'industrie sont dues à problèmes économiques et à des problèmes d'organisation beaucoup plus qu'à des questions strictement techniques. Nous ne partageons pas

cette opinion, et la définition qu'elle implique nous paraît exagérément restrictive.

Nous allons tenter de proposer une définition satisfaisante du génie logiciel ; mais il convient pour cela de préciser d'abord le sens du mot *logiciel*.

Définition : On regroupe sous le terme de *logiciel* les différentes formes des programmes qui permettent de faire fonctionner un ordinateur et de l'utiliser pour résoudre des problèmes, les données qu'ils utilisent, et les documents qui servent à concevoir ces programmes et ces données, à les mettre en oeuvre, à les utiliser et à les modifier.

Cette définition permet d'inclure dans le logiciel non pas seulement les programmes (sous leurs *différentes formes*, c'est-à-dire "sources" aussi bien qu'"objets"), mais aussi les données et la documentation associée : spécifications, documents de conception, manuels d'utilisation, etc. La définition couvre le logiciel de base (compilateurs, systèmes d'exploitation etc.) aussi bien que les applications.

Nous suivrons l'usage en utilisant aussi le mot "logiciel" comme adjectif et en écrivant "un logiciel" pour "un produit logiciel".

Munis de cette première définition, nous pouvons nous attaquer au "génie logiciel" proprement dit.

Définition : On appelle *génie logiciel* l'application de méthodes scientifiques au développement de théories, méthodes, techniques, langages et outils favorisant la production de logiciel de qualité.

Cette définition cherche à réunir les éléments qui nous paraissent essentiels : la présence d'une base scientifique, sans laquelle on ne peut parler de "génie" ; cinq directions de développement fondamentales, de la plus abstraite (les théories) à la plus pratique (les outils) ; et le rôle essentiel de la notion de *qualité*. Nous considérons le mot "production" suffisamment général pour qu'il ne soit pas nécessaire d'ajouter "et la maintenance" (ou "l'entretien").

Cette définition peut apparaître très générale ; on notera cependant qu'elle limite le génie logiciel aux activités de deuxième degré : développer un outil de construction de programmes, c'est faire du génie logiciel ; mais ce n'est pas nécessairement en faire que de développer un simple programme d'application (même de bonne qualité).

Les chapitres suivants de cette thèse sont consacrés à l'examen ponctuel de quelques-uns des éléments du génie logiciel, tels qu'ils apparaissent dans cette définition (théories, méthodes etc.). Il nous a cependant paru utile de faire précéder d'un panorama rapide cette série de vues inévitablement partielles.

1.3 - QUELQUES QUESTIONS FONDAMENTALES

Il est tout d'abord utile de préciser quelques-uns des grands axes qui définissent les degrés de liberté d'un projet logiciel.

Le premier axe est celui des *temps*. Un projet logiciel suit un déroulement chronologique ; un modèle de ce déroulement recueille aujourd'hui un accord à peu près général, au moins dans ses grandes lignes : le modèle du "cycle de vie". Nous le présenterons au paragraphe 1.4[1], et nous discuterons ses limitations et quelques variations possibles.

Le second axe correspond à l'un des mots essentiels de la définition précédente : le mot de *qualité*. Le but du génie logiciel étant de permettre la production de logiciel de qualité, il importe de savoir précisément ce que recouvre ce concept. Tel est l'objet du paragraphe 1.5.

[1] Pour éviter toute ambiguité, nous appelons "chapitres" et "paragraphes", respectivement, les divisions de premier et de second niveau.

Vient ensuite l'axe des *méthodes* : la mise en place d'une politique de génie logiciel exige des méthodes s'appliquant aux aspects techniques (spécification, conception, validation) et à l'organisation des projets. Quelques-unes de ces méthodes sont brièvement présentées au paragraphe 1.6.

Certaines des entités caractéristiques des projets logiciels (coûts, délais, taux d'erreurs) se prêtent à une analyse quantitative, et des techniques de modélisation mathématique ont été proposées pour estimer à l'avance la valeur de certains paramètres. C'est l'axe des *mesures et modèles* (paragraphe 1.7).

Un rôle fondamental est joué par les *langages* utilisés aux différentes étapes de la réalisation des produits logiciels. Cet axe est étudié au paragraphe 1.8.

Enfin, les *outils* ont une influence considérable en génie logiciel comme dans les autres disciplines. Cette question fait l'objet du paragraphe 1.9

Un dernier axe important est orthogonal à tous les précédents : c'est l'axe des *applications*, qui permet de parcourir les problèmes spécifiques dûs au fait que les produits logiciels ont des objets différents : calcul scientifique, gestion, contrôle-commande, informatique système, etc. Ce chapitre ne traite pas directement des applications, mais on trouvera aux chapitres 9, 10 et 11 de la deuxieme partie et dans les articles correspondants de la troisième partie une étude des problèmes posés par trois domaines d'application particuliers : la programmation des super-ordinateurs [80b, 81a, 84b][2] ; la programmation des applications interactives [82a] ; et l'édition structurelle [84a, 85b].

1.4 - LE CYCLE DE VIE

1.4.1 - Définition et interprétations

Un paradigme, dans une discipline scientifique, est un concept fondamental impliquant tout un mode de raisonnement, comme la notion d'invariant en physique ou celle de structure en mathématiques. L'un des paradigmes du génie logiciel est la notion de cycle de vie, qui sert de référence pour l'étude chronologique des projets.

Le cycle de vie est une modélisation conventionnelle de la succession d'étapes qui préside à la mise en oeuvre d'un produit logiciel. Le cycle de vie est souvent représenté sous la forme dite **"modèle de la cascade"** en raison de l'analogie qu'évoque le schéma correspondant (figure 1).

Les huit étapes indiquées sur ce schéma se répartissent en deux époques : jusqu'à la phase d'écriture des programmes, il s'agit de construire, au-delà, de mettre en place.

On notera que le détail de la division en étapes varie dans les différentes présentations publiées. Une norme de l'IEEE propose cependant une terminologie de référence[3].

[2] Les références entre crochets renvoient aux textes reproduits dans la troisième partie de cette thèse, et dont la liste est donnée page iii.

[3] ANSI/IEEE Standard 730-1981. *IEEE Standard for Quality Assurance Plans*, 1981.

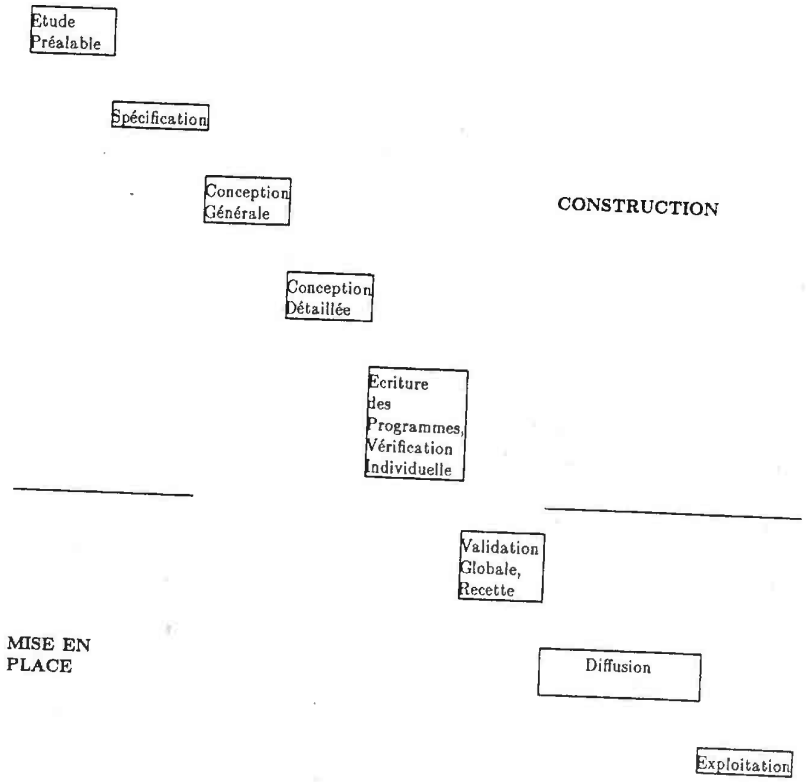


Figure 1 : Le modèle de la cascade

La notion de cycle de vie se prête à deux interprétations :

• l'interprétation "neutre" selon laquelle le cycle de vie est une description plus ou moins idéalisée (un modèle, au sens mathématique du terme) de ce qui se passe dans la plupart les projets logiciels ;

• l'interprétation "volontariste" qui propose le modèle de la cascade ("modèle" signifiant ici exemple à suivre) comme une **méthode** à respecter dans l'organisation d'un projet, impliquant en particulier[4] :

- que toutes les étapes, sans exception, soient exécutées ;
- que l'ordre indiqué soit respecté ;

[4] Barry W. Boehm. *Les Facteurs de Coût du Logiciel*, trad. E. Girard. TSI (*Technique et Science Informatiques*), Vol. 1, n° 1, 1982.

- que l'on ne passe pas à l'étape n sans avoir terminé l'étape $n-1$ et fait avaliser formellement ses résultats par un comité technique différent de l'équipe de développement, selon une procédure définie à l'avance ;
- que les remises en cause, inévitables en pratique, se limitent à un niveau (on ne peut changer à l'étape n que des décisions prises à l'étape $n-1$) et passent obligatoirement par la même procédure de validation.

1.4.2 - Les étapes

Définissons maintenant plus précisément l'objet des différentes étapes citées ; le résultat de chacune d'elles (ce qui doit en sortir concrètement) est résumé sur le tableau de la figure 2. Nous suivons ici les auteurs qui se rattachent à ce que nous venons d'appeler l'interprétation volontariste ; même si l'on ne s'impose pas la discipline stricte qu'implique cette interprétation, cependant, les définitions suivantes fournissent un cadre de référence utile.

Etude préalable : il s'agit tout d'abord (*phase exploratoire*) de déterminer s'il y a lieu de réaliser l'application et, si oui, d'en fixer les conditions générales, débouchant (*phase conceptuelle*) sur un cahier des charges et un plan du projet.

Spécification : passage de la description non formelle que représente le cahier des charges à une définition précise des objets manipulés par le système, des tâches qu'il doit effectuer sur ces objets (spécification fonctionnelle), des contraintes de performance, et de la planification détaillée des étapes suivantes.

Conception générale : on passe ici de la définition à la réalisation. Il s'agit de concevoir l'architecture du système, c'est-à-dire de décrire les principales structures de données (internes aussi bien qu'externes, la seconde catégorie incluant les fichiers et les bases de données) et la décomposition du système en un certain nombre de modules (qui, dans la méthode "par objets", sont précisément bâtis autour des structures de données ; cf. plus loin 1.6.4 et le chapitre 4).

Conception détaillée : affinage des éléments précédents (structures de données, modules) jusqu'à l'obtention d'une forme permettant d'écrire immédiatement les programmes.

Ecriture des programmes et vérification individuelle : écriture des textes des programmes et mise en oeuvre des structures de données ; vérification interne de chaque module par ses auteurs.

Validation globale, recette : validation de l'ensemble du système ; il s'agit cette fois de valider par rapport aux fonctions à assurer (on distingue la *vérification*, qui a pour objet de contrôler la cohérence interne d'un élément de logiciel, de la *validation*, qui contrôle le respect de conditions fixées de façon extérieure à l'objet lui-même). Cette phase s'appuie sur le plan de validation défini à l'étape de spécification ; elle est normalement confiée à une équipe différente de l'équipe de développement.

Diffusion : préparation et distribution des différentes versions.

Exploitation : mise en place du système dans son environnement opérationnel ; prise en compte des rapports d'incident ; correction des erreurs.

Dans ce modèle, il convient de noter l'absence d'étapes associées à la documentation (activité nécessaire tout au long du projet, et non pas phase indépendante du cycle de vie) et à la maintenance (qui est couverte par l'"exploitation" pour ce qui est de la correction des erreurs, et doit être considérée comme relevant d'un nouveau projet pour ce qui est des modifications de spécification).

Etape	Résultat
Etude préalable	*Phase exploratoire :* dossier d'entretiens ; décision (ne pas faire, acheter, faire faire, faire) ; budget approximatif.
	Phase conceptuelle : cahier des charges ; plan général du projet ; budget précis ; définition des contraintes.
Spécification	Document de spécification (fonctions et performances) ; première version du manuel d'utilisation ; plan détaillé du reste du projet ; plan de validation (tests en particulier).
Conception Générale	Définition des principales structures de données ; décomposition du système en modules (architecture) et description du rôle de chaque module.
Conception Détaillée	Description détaillée des structures de données et des modules
Ecriture des Programmes, Vérification Individuelle	Texte des programmes, chaque module vérifié séparément.
Validation Globale, Recette	Compte rendu de recette ; rapports d'inspection et de validation.
Diffusion	Versions des programmes et de leur documentation adaptées aux différentes catégories d'utilisateurs.
Exploitation	Programme en fonctionnement. Rapports d'incidents et de correction.

Figure 2 : Résultats des étapes du cycle de vie

1.4.3 - Variantes : le prototypage

La notion de cycle de vie, avec ce qu'elle implique de rigidité dans le déroulement séquentiel des étapes, a donné lieu à un certain nombre de critiques ; des schémas plus souples ont été proposés. Il est en particulier beaucoup question depuis quelques années de **prototypage rapide**[5]. Les tenants de cette méthode considèrent qu'il est vain de vouloir figer à un stade précoce des spécifications qui, de toute façon, vont être remises en cause. Ils recommandent donc de travailler par approximations successives, en construisant d'abord, le plus rapidement possible, une première version sommaire, un **prototype**.

Les avis diffèrent sur le rôle exact qui est assigné au prototype ; on peut distinguer deux grandes tendances :

- Dans certains cas, le prototype est simplement destiné à mettre certaines hypothèses à l'épreuve, hypothèses qui peuvent porter par exemple sur l'interface du système avec ses utilisateurs, l'efficacité de certains algorithmes, la faisabilité de certaines tâches etc ; une fois le résultat obtenu, on ne conservera pas le prototype et le développement repartira de zéro, fort cependant de l'expérience gagnée. Avec cette méthode, le prototype peut être réalisé avec des outils et un langage différents de ceux qui seront retenus pour le produit final ; certains langages de très haut niveau, de mise en oeuvre peu efficace mais permettant d'obtenir très vite des programmes exécutables, peuvent être mis à profit ici (voir plus loin le paragraphe 1.8).
- Un prototype de la seconde forme ne comporte que certaines des fonctions de base du système envisagé ; il s'agit de le faire évoluer vers un produit complet par additions successives, sans rupture de continuité. Comme l'a fait remarquer E. Girard, cette interprétation du prototypage correspond en quelque sorte à une variation sur le modèle classique de la cascade donné à la figure 1 : le modèle de la "Cascade-Escher" (figure 3).

On peut utiliser les termes de *prototype jetable* et de *prototype incrémental* pour distinguer ces deux variantes (certains auteurs parlent de "maquette" dans le premier cas et réservent le terme de prototype au second, mais cette terminologie n'est pas universelle).

Le débat entre partisans de la méthode séquentielle et du prototypage n'est pas clos. Il est intéressant de noter que les arguments des deux écoles sont nourris par la même constatation de base : tous les résultats d'études sur le coût des erreurs en logiciel montrent que les erreurs commises tôt dans le cycle de vie (aux étapes d'étude préalable et de spécification) et décelées tard ont des conséquences énormes sur le coût des projets. Ce phénomène est illustré de façon particulièrement frappante par une courbe résumant les résultats d'un certain nombre d'analyses de grands projets et publiée par B. W. Boehm (figure 4).

On voit pourquoi ces résultats peuvent servir à justifier des méthodes très différentes :

- une conclusion possible est qu'il faut consacrer une attention considérable aux phases initiales du cycle de vie afin d'éviter à tout prix que des erreurs commises lors de la définition du système passent inaperçues jusqu'aux étapes de mise en place : c'est l'un des arguments majeurs des recherches sur la spécification formelle ;
- on peut au contraire en déduire que, les erreurs de définition étant inévitables, la seule chose raisonnable est d'essayer d'en limiter les conséquences ; il convient donc de passer le plus vite possible à l'expérimentation séparée sur des segments du système, segments suffisamment bien délimités pour que les phénomènes décrits par la figure 4 n'aient pas d'effet catastrophique sur le projet dans son ensemble. C'est la voie qui mène au prototypage sous l'une de ses deux formes.

[5] Le prototypage rapide s'oppose non seulement à la méthode du cycle de vie, mais aussi à celle du prototypage lent, qui est certainement l'une des façons les plus courantes de développer du logiciel.

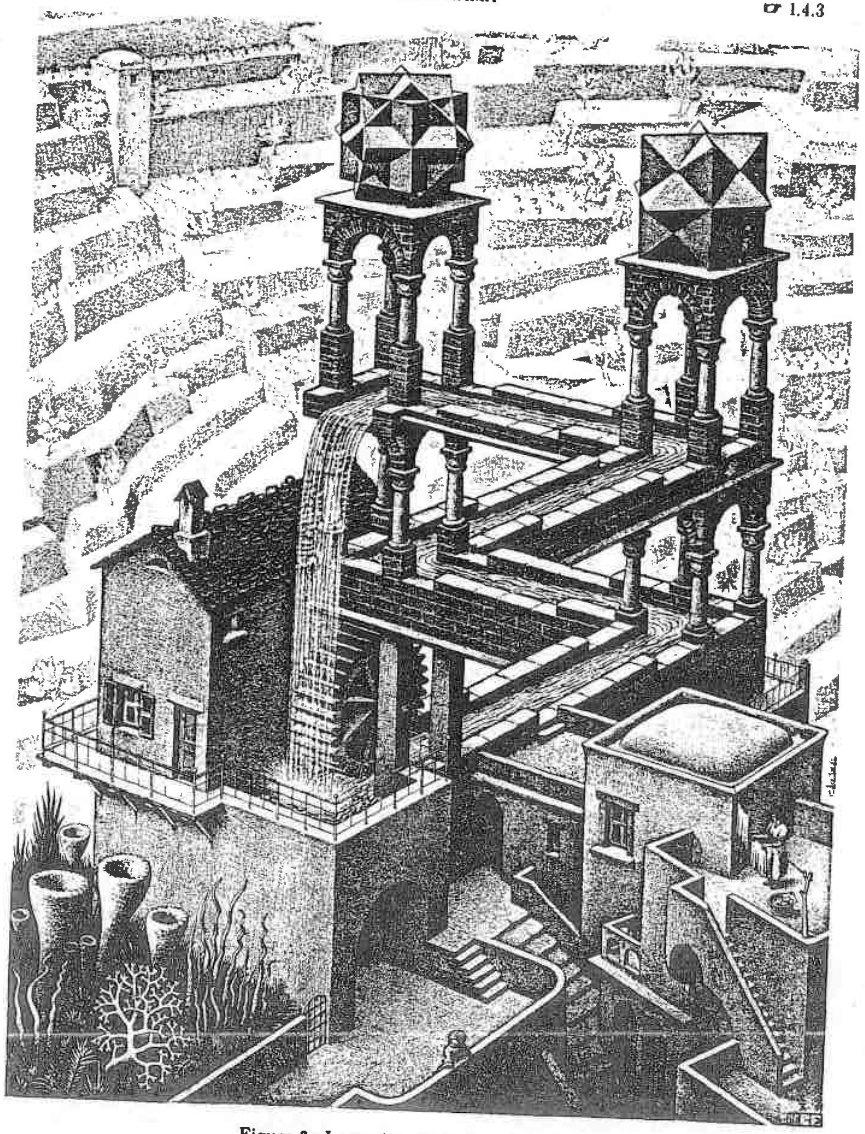


Figure 3 : Le modèle de la Cascade-Escher

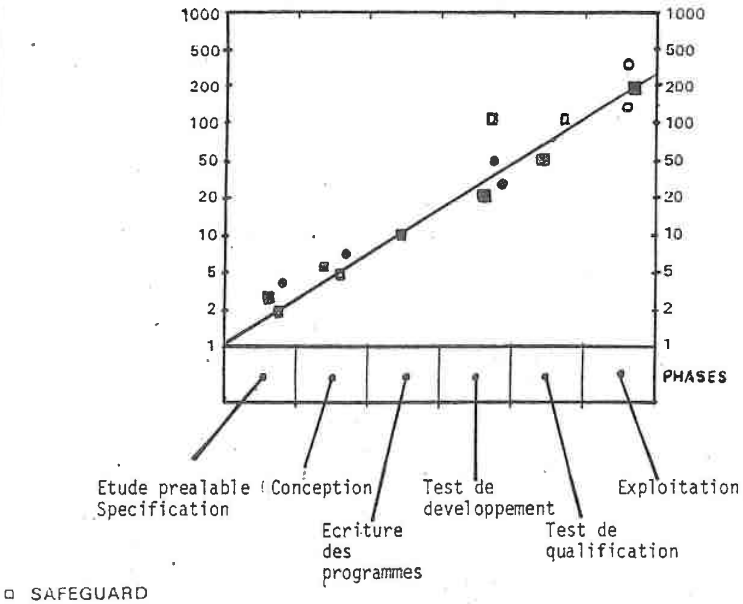


Figure 4 : Coût de correction d'une erreur de définition en fonction de la phase où elle est détectée

Les deux méthodes (prototypage et spécification complète avant toute réalisation) ne sont cependant pas aussi totalement incompatibles que cette discussion peut le laisser supposer. En particulier, un prototype jetable peut dans certains cas faire bon ménage avec une spécification : s'il est naïf de croire que la réalisation d'un prototype dispense d'une réflexion initiale approfondie, il peut être tout à fait raisonnable d'intégrer à cette réflexion une composante expérimentale, sous la forme d'un prototype destiné à valider certaines hypothèses qu'il est difficile d'analyser autrement.

Comme il apparaît clairement au chapitre 5 de cette thèse et dans les articles correspondants de la troisième partie, l'auteur a porté ses recherches dans la direction de la spécification plus que dans celle du prototypage (encore qu'il ait, comme tout le monde, pratiqué sans l'avoir cherché le prototypage incrémental lent). Mais on ne peut nier que la technique du prototypage rapide puisse rendre des services lorsque les conditions de son application sont réunies[6].

1.5 - L'OBJECTIF MAJEUR : LA QUALITE DU LOGICIEL

La définition que nous avons donnée du génie logiciel présente comme but principal l'obtention de logiciel de qualité. Il est nécessaire de définir un peu plus précisément ce que signifie cette notion dans le cas du logiciel.

Un certain nombre d'auteurs en ont proposé des définitions ; l'une des premières à reposer sur une étude systématique était celle de Boehm et de ses collaborateurs à TRW[7]. La définition dont nous nous inspirerons ici est celle de James McCall, résultat d'une étude effectuée en 1977 par General Electric pour le compte de l'armée de l'air américaine.

McCall prend bien soin de distinguer les deux sortes de conditions qui déterminent la qualité d'un produit logiciel :

- les caractéristiques ***externes*** de la qualité d'un logiciel, comme l'extensibilité (facilité d'adaptation à des changements de spécifications), qui peuvent être perçues par les utilisateurs du produit ;
- les caractéristiques ***internes*** de la qualité, qui peuvent être analysées par les informaticiens à l'examen du programme et des autres documents techniques associés : un exemple est la modularité du système, c'est-à-dire sa division en unités logiques simples et cohérentes.

Cette distinction, qui n'apparaissait pas toujours clairement dans les travaux antérieurs sur la qualité du logiciel, est importante car elle évite de confondre causes et symptômes. McCall appelle **facteurs** les caractéristiques externes et **critères** les caractéristiques internes.

Seuls les facteurs comptent en dernier ressort, puisqu'ils déterminent l'utilisation du produit ; mais c'est le respect des critères au cours du développement qui conditionne la production d'un logiciel conforme à ces facteurs. Les exemples précédents illustrent cette remarque : la modularité (critère) est un élément important pour assurer l'extensibilité (facteur).

Plus généralement, McCall propose une matrice montrant l'influence de chacun des critères sur chacun des facteurs. Cette influence peut être positive, comme dans l'exemple de la modularité et de l'extensibilité, négative ou nulle. Cette analyse est complétée par la définition d'un certain nombre de **mesures**, permettant d'associer à chaque critère des vérifications quantitatives.

Le tableau de la figure 5 donne la définition de dix facteurs voisins de ceux de McCall.

[6] Pour une liste de conditions nécessaires, cf. Barry W. Boehm, *Software Engineering Economics*, in *IEEE Transactions on Software Engineering*, Vol. SE-10, n° 1, pages 4-21, juillet 1984.

[7] Barry W. Boehm, J.R. Brown, H. Kaspar, Myron Lipow, G. Macleod, M. Merrit : *Characteristics of Software Quality* ; TRW Series of Software Technology, North-Holland, Amsterdam, 1978.

Facteur	*Définition*
Validité :	L'aptitude d'un produit logiciel à remplir exactement ses fonctions, définies par le cahier des charges et la spécification.
Fiabilité :	L'aptitude d'un produit logiciel à fonctionner dans des conditions éventuellement anormales.
Extensibilité :	La facilité avec laquelle un logiciel se prête à une modification ou à une extension des fonctions qui lui sont demandées.
Réutilisabilité :	L'aptitude d'un logiciel à être réutilisé, en tout ou en partie, dans de nouvelles applications.
Compatibilité :	La facilité avec laquelle un logiciel peut être combiné avec d'autres.
Efficacité :	L'utilisation optimale des ressources matérielles (processeurs, mémoires internes et externes, dispositifs de communication, etc.).
Portabilité :	La facilité avec laquelle un produit peut être transféré dans différents environnements matériels et logiciels.
Vérifiabilité :	La facilité de préparation des procédures de recette et de validation (particulièrement des jeux d'essai).
Intégrité :	L'aptitude d'un logiciel à protéger son code et ses données contre des accès non autorisés.
Facilité d'emploi :	La facilité d'apprentissage, d'utilisation, de préparation des données, de correction des erreurs d'utilisation, d'interprétation des résultats.

Figure 5 : Dix facteurs de la qualité en logiciel

L'une des caractéristiques essentielles d'une telle définition de la qualité est qu'elle est plurielle : ce sont des qualités possibles que l'on met ici en évidence.

Or un simple examen montre que ces qualités ne sont pas nécessairement deux à deux compatibles. on ne saurait par exemple atteindre l'optimum tout à la fois en matière de fiabilité et de facilité d'emploi : le premier but entraîne inévitablement l'inclusion de protections, de barrières, qui sont préjudiciables au second.

Une telle constatation n'a rien de vraiment surprenant : dans toute branche d'ingénierie, la qualité s'obtient au prix d'un compromis entre des objectifs souvent contradictoires, coûts, délais de réalisation, étendue des fonctions offertes, sécurité d'emploi etc. Mais en logiciel ce compromis est trop souvent encore réalisé de façon inconsciente, l'un des facteurs (l'efficacité par exemple) étant exagérément privilégié.

1.6 - METHODES

De nombreux travaux ont été menés depuis une quinzaine d'années sur la méthodologie de construction des programmes. Nous avons retenu quelques étapes particulièrement marquantes.

1.6.1 - Programmation structurée

Le terme de "programmation structurée" est un de ceux qui ont eu le plus de succès, à la suite de travaux de Dijkstra et Hoare (à partir de 1967 environ). Pourtant, les idées de ces précurseurs n'ont souvent atteint les praticiens de la programmation que sous une forme tronquée.

Il est trop courant de voir la programmation structurée présentée comme un simple ensemble d'interdictions relatives aux structures de contrôle dans les langages de programmation : ne pas employer de branchements, se limiter à trois figures de base (enchaînement, choix, boucle) et à quelques variantes.

En réalité, la programmation structurée se voulait une remise en cause profonde des méthodes habituelles de construction de logiciel, et ses créateurs insistaient sur la nécessité d'une démarche rigoureuse, mathématique. Un rôle important était dévolu à la notion de démonstration de validité des programmes : un programme doit être construit, selon Dijkstra et Hoare, comme on prouve un théorème ; sa validité doit être évidente, et mathématiquement vérifiable.

Ces méthodes rigoureuses exigent que l'on parte d'une description du problème qui soit elle aussi mathématiquement irréprochable ; elles mènent donc au problème de la spécification formelle. Nous en donnerons quelques exemples ci-après après avoir abordé ce thème (1.6.3).

1.6.2 - Méthodes de conception : descendantes, ascendantes

Les méthodes de conception proposent des principes généraux pour guider le concepteur d'un système complexe à travers le labyrinthe des problèmes à résoudre.

Les méthodes le plus souvent associées à la "programmation structurée" sont de type **descendant**, c'est-à-dire qu'elles appliquent une démarche systématique dite "par affinages successifs" qui part de l'expression la plus générale du problème à résoudre et décompose répétitivement les tâches à effectuer en sous-tâches plus simples, jusqu'à ce que tout ait été exprimé en termes d'opérations assez élémentaires pour être mises en oeuvre directement dans un langage de programmation.

Les méthodes **ascendantes** ne doivent pas être négligées pour autant : elles cherchent à favoriser la réutilisation de logiciel existant et la construction de nouveaux systèmes par combinaison d'éléments prédéfinis. Elles répondent ainsi à l'un des problèmes-clés du génie logiciel, celui de la réutilisabilité : trop d'investissements sont perdus en raison du faible taux de réutilisation des programmes, et de l'habitude (qui, dans l'état actuel de la technique, est souvent une nécessité) de repartir de zéro pour chaque nouveau produit.

La vérité en matière de méthodes de conception consiste sans doute en une judicieuse combinaison de "descendant" et d'"ascendant", permettant d'appliquer à chaque nouveau problème une étude systématique partant de la spécification, tout en favorisant au maximum la réutilisation d'éléments de logiciel précédemment réalisés.

1.6.3 - Méthodes de spécification

Les méthodes de spécification sont destinées à faciliter l'analyse des problèmes à résoudre et la description externe des systèmes.

Dans le cycle de vie, la spécification précède les phases de conception et de mise en oeuvre, au cours desquelles les objets du système (programmes et structures de données) seront effectivement construits. La spécification a pour but de définir auparavant les propriétés précises que devront respecter ces objets.

Au-delà d'une certaine taille de projet, cette tâche se révèle souvent la plus importante et la plus ardue. Plusieurs méthodes ont été proposées pour la faciliter. Parmi celles qui sont réellement employées dans certaines branches de l'industrie, les unes (SADT, Merise) cherchent essentiellement à faciliter le dialogue entre les informaticiens et leurs clients, en proposant un support commun de description et de discussion (graphique dans le premier cas cité). D'autres (SREM, PSL/PSA) mettent l'accent sur les outils informatiques permettant d'exercer un contrôle serré de l'évolution du projet, de ses spécifications et de sa documentation.

A côté de ces méthodes issues en général de l'industrie, de nombreux chercheurs ont proposé des **langages formels** permettant, grâce à des notations de type mathématique, d'exprimer les spécifications sous une forme rigoureuse et non ambiguë. Les méthodes associées sont en général plus difficiles à mettre en oeuvre, car elles exigent de la part des spécificateurs une culture mathématique et une habitude du raisonnement abstrait que tous les informaticiens ne possèdent pas ; c'est, selon les partisans des méthodes formelles, le prix à payer pour obtenir des systèmes informatiques dont la validité et la fiabilité pourront être garanties. On trouvera au chapitre 5 de cette thèse et dans les articles correspondants [78c, 79a, 85a, 85g] de la troisième partie des discussions plus approfondies sur les méthodes formelles et leur application.

On peut distinguer, dans les travaux sur la spécification formelle, trois directions principales : *validation, construction, exécution.* Ces trois façon de considérer la spécification ne sont pas nécessairement incompatibles ; elles se distinguent par l'utilisation prioritaire à laquelle on destine les spécifications.

Les méthodes de la première catégorie (par exemple FDM, HDM, Affirm) mettent l'accent sur la production de spécifications qui puissent servir de base à la **validation** automatisée des programmes ; des progrès importants ont été réalisés dans ce domaine depuis quelques années, et l'on sait désormais valider mathématiquement (au prix d'un effort qui reste considérable) des programmes de plusieurs dizaines de milliers d'instructions, incluant des processus parallèles.

D'autres méthodes ont pour but essentiel non pas de permettre la validation de programmes développés indépendamment de la spécification, mais plutôt d'utiliser la spécification elle-même comme point de départ de la **conception** du programme, le processus de validation étant intégré au processus de construction. Partant de la spécification, on peut procéder par déduction, par transformation, ou par utilisation de schémas prédéfinis (ou encore par une combinaison de ces techniques) pour obtenir des versions exécutables. Les travaux de Dijkstra[8] ont donné une forte impulsion à ce domaine ; de nombreuses équipes s'y intéressent : projet CIP à Munich, méthodes développées à Nancy (méthode déductive, SPES), méthode VDM (Jones, Bjørner), travaux menés par Abrial, Manna, Sintzoff, van Laamswerde, et par de nombreux chercheurs gravitant autour du groupe de travail 2.1 de l'IFIP (et en France du groupe Anna Gram).

[8] Edsger W. Dijkstra, *A Discipline of Programming*, Prentice-Hall 1976.

L'idée d'utiliser la spécification comme point de départ de la construction des programmes sert de base à une série de réflexions présentées ci-après au chapitre 6 ("Construction systématique"), qui se réfère à des articles [80a, 81a, 84b, 85c] de la troisième partie.

La figure 7 cherche à donner une idée, sur un petit exemple, de ce que peut être une telle méthode de construction des programmes par affinage de leur spécification.

Le problème traité ici est celui de la recherche d'un élément dans une table supposée triée. Il s'agit bien entendu d'un exemple de petite taille, mais qui présente quelques pièges. On s'en persuadera en consultant d'abord la figure 6, qui présente quatre tentatives de solution du même problème : le lecteur est invité à vérifier que toutes ces solutions, à première vue acceptables, sont en fait erronnées ; il suffit pour cela de montrer pour chacune d'entre elles que certaines valeurs du tableau t et de l'élément x produiront un résultat erronné (*présent* mis à **vrai** alors que l'élément est absent, ou inversement) ou entraîneront une erreur à l'exécution (débordement de mémoire, bouclage infini). Un contre-exemple est fourni en annexe pour chacun de ces quatre programmes.

Le programme (correct) de la figure 7 calcule le plus grand indice i tel que $t[i] \leq x$ (ou 0 s'il n'existe pas de tel indice ; cf. la spécification plus précise donnée sur la figure). Pour obtenir la réponse à la question "x apparaît-il dans t ?", on doit le faire suivre d'une instruction de la forme :

si $i \geq 1$ **et** $i \leq n$ **alors**
    présent := $(x = t[i])$
sinon
    présent := **faux**
fin

Figure 6

QUATRE (faux) PROGRAMMES DE RECHERCHE DICHOTOMIQUE

(P1)

$i := 1 ; j := n ;$
tant que $i \neq j$ **faire**
    $m := \lfloor i + j \ / \ 2 \rfloor ;$
    si $x \leq t[m]$ **alors**
        $j := m$
    sinon
        $i := m$
    fin
fin ;
présent := $(x = t[i])$

(P2)

$i := 1 ; j := n ;$ *trouvé* := **faux** ;
tant que $i \neq j$ **et non** *trouvé*
faire
    $m := \lfloor i + j \ / \ 2 \rfloor ;$
    si $x < t[m]$ **alors**
        $j := m - 1$
    sinon si $x = t[m]$ **alors**
        trouvé := vrai
    sinon $i := m + 1$ **fin**
fin ;
présent := *trouvé*

(P3)

$i := 0 ; j := n ;$
tant que $i \neq j$ **faire**
    $m := \lfloor i + j \ / \ 2 \rfloor ;$
    si $x \leq t[m]$ **alors**
        $j := m$
    sinon
        $i := m + 1$
    fin
fin ;
si $i \geq 1$ **et** $i \leq n$ **alors**
    présent := $(x = t[i])$
sinon
    présent := **faux**
fin

(P4)

$i := 0 ; j := n+1 ;$
tant que $i \neq j$ **faire**
    $m := \lfloor i + j \ / \ 2 \rfloor ;$
    si $x \leq t[m]$ **alors**
        $j := m$
    sinon
        $i := m + 1$
    fin
fin ;
si $i \geq 1$ **et** $i \leq n$ **alors**
    présent := $(x = t[i])$
sinon
    présent := **faux**
fin

Figure 7

DEVELOPPEMENT D'UN PROGRAMME SELON UNE METHODE RIGOUREUSE : L'EXEMPLE DE LA RECHERCHE DICHOTOMIQUE

Donnée : un tableau $a[1..n]$;
un élément x.

(Bien entendu, le programme ne peut modifier les valeurs de ces objets).

Hypothèse : a trié, c'est-à-dire

$a[i] \leq a[j]$ pour $1 \leq i \leq j \leq n$.

Résultat cherché : un entier i tel que

$0 \leq i \leq n$ et

$$\begin{cases} a[k] \leq x \text{ pour } 1 \leq k \leq i \\ a[k] > x \text{ pour } i+1 \leq k \leq n \end{cases}$$

Note : il est important de vérifier qu'il existe toujours un index i unique conforme à ces propriétés. En particulier, i vaudra 0 si $x < a[1]$; i vaudra n si $x \geq a[n]$ (on se souviendra qu'une propriété de la forme "pour tout x appartenant à E, $P(x)$" est toujours vraie, quelle que soit la propriété P, si l'ensemble E est vide).

METHODE

Le but recherché peut s'écrire sous la forme

c_1 **et** c_2

où c_1 est défini comme

$$\begin{cases} 0 \leq i \leq j \leq n \text{ et} \\ a[k] \leq x \text{ pour } 1 \leq k \leq i \text{ et} \\ a[k] > x \text{ pour } j+1 \leq k \leq n \end{cases}$$

et c_2 est la condition $i=j$.

Considérons c_1 comme un "**invariant**" et c_2 comme un "but". Le programme peut s'écrire :

établir c_1 de façon simple ,
tant que *c_2 n'est pas vérifié* **faire**
    rapprocher i de j
    en conservant c_1
fin

MISE EN OEUVRE

Pour *établir c_1 de façon simple*, il suffit de prendre $i = 0$, $j = n$. Pour *rapprocher i de j*, considérons la valeur

$$m = \lfloor (i + j) / 2 \rfloor$$

(division entière). Pouvons-nous affecter à i ou à j la valeur de m sans pour autant remettre en cause la validité de l'invariant c_1 ? Ceci dépend de la valeur relative de x et $a[m]$:

- Si $a[m] \leq x$, nous savons que $a[k] \leq x$ pour $1 \leq k \leq m$ puisque a est trié. Nous pouvons donc choisir m comme nouvelle valeur de i.

- Si $a[m] > x$, nous savons que $a[k] > x$ pour $k \geq m$, c'est-à-dire pour $k \geq (m-1)+1$. Nous pouvons donc choisir $m-1$ comme nouvelle valeur de j.

Il convient dans les deux cas de vérifier que le nouveau choix fait effectivement décroître la valeur de $j-i$, ce que le lecteur est invité à faire (attention : m est le résultat d'une division entière).

SOLUTION

Nous obtenons finalement le programme suivant :

$i := 0$; $j := n$;
tant que $i \neq j$ **faire**
    $m := \lfloor (i + j) / 2 \rfloor$;
    si $a[m] \leq x$ **alors** $i := m$
    sinon $j := m-1$ **fin**
fin

La troisième voie de développement citée à propos de la spécification formelle est celle qu'empruntent certains auteurs qui cherchent à abattre la cloison entre la spécification, forme *descriptive* du système, et le programme, forme *prescriptive* (exécutable). On obtient ainsi la notion de **spécification exécutable**. Cette approche a été particulièrement illustrée par les utilisateurs de certaines méthodes issues de l'Intelligence Artificielle et en particulier du langage **Prolog** (cf. 1.8 ci-après) : en utilisant des notations formelles soumises à certaines restrictions, on peut obtenir des textes interprétables comme spécifications mais pouvant également être exécutés sur un ordinateur. On doit accepter pour cela une perte de puissance expressive (du point de vue de la spécification) et d'efficacité (du point de vue de l'exécution), mais on gagne l'unicité de la description. C'est une des méthodes qui tendent à réconcilier la spécification, et le prototypage.

1.6.4 - Méthodes à objets, types abstraits

Il convient de mentionner un ensemble de méthodes qui peuvent s'appliquer aux différentes étapes de la première partie du cycle de vie (la phase de construction) et s'opposent assez nettement à la démarche traditionnelle. Dans une programmation (ou conception, ou spécification) par objets, on décrit un système de façon globale, moins par *"la"* fonction qu'il remplit que comme un ensemble de classes d'**objets**, caractérisés par leurs propriétés abstraites.

Le bien-fondé de cette approche est particulièrement évident dans des domaines comme la conception des systèmes d'exploitation ou des systèmes de contrôle-commande, où les objets coopérants sont les gestionnaires des différentes ressources disponibles ; mais la même idée se transpose avec bonheur aux autres domaines d'application.

La programmation par objets a été introduite en 1967 par le langage de programmation Simula 67, dont les principales idées ont depuis été reprises par Smalltalk. Certains éléments de cette méthode ont également influencé la conception du langage Ada. La base théorique a été fournie par les travaux menés à partir de 1974 autour de la notion de **type abstrait**, modélisation mathématique des concepts informatiques de type et d'objet (cf. [76a, 78a]).

1.6.5 - Méthodes empruntées à l'intelligence artificielle

Les spécialistes d'intelligence artificielle ont depuis longtemps (plus précisément depuis 1959, date de diffusion de Lisp) utilisé des méthodes et des langages les plaçant quelque peu en marge du génie logiciel. L'une de leurs contributions les plus importantes est la mise au points d'*environnements de programmation* avenants, tel Interlisp, disponibles dans les laboratoires d'intelligence artificielle bien avant que l'industrie se préoccupât de la question.

> Les langages de l'intelligence artificielle et les systèmes associés se distinguent en particulier par le caractère **dynamique** des objets manipulés : il est possible dans ces environnements de créer librement de nouveaux objets de programme à l'exécution. Les langages utilisés en génie logiciel classique sont bien plus *statiques* ; il faut en général prévoir dès avant l'exécution le nombre et la taille des objets dont on aura besoin (ceci est vrai de Fortran et de Cobol, mais aussi dans une large mesure de langages comme Pascal, C et Ada dont les possibilités de création dynamique d'objets voient leur utilité pratique limitée par l'absence de mécanismes automatiques de récupération de la mémoire). Ceci nous paraît une des limitations techniques majeures des langages actuels. On notera que les langages à objets tels que Simula et Smalltalk n'en souffrent pas.

Plus récemment, le bruit fait autour des **systèmes experts** et de la programmation heuristique a soulevé quelques espoirs : peut-on espérer que des systèmes "intelligents" d'aide à la programmation fourniront une solution radicale aux grands problèmes du génie logiciel? Quelques applications expérimentales de systèmes experts d'aide à la construction ou à la correction de programme ont été réalisés, mais aucune expérience en vraie grandeur n'a encore fourni de réponse définitive.

1.6.6 - Méthodes de gestion de projets

A côté des problèmes purement techniques, la réalisation de logiciel soulève de nombreuses difficultés quant à la gestion des projets. Dans une certaine mesure, ces difficultés sont les mêmes que dans les autres applications de l'ingénierie ; mais les caractéristiques propres du logiciel et la complexité des systèmes que l'on réalise dans ce domaine ont rendu nécessaire la mise au point de méthodes scientifiques.

Les travaux en ce domaine ont porté par exemple sur l'organisation des équipes : comment éviter le phénomène souvent observé selon lequel, au-delà d'une certaine taille de l'équipe, les problèmes de communication l'emportent sur les problèmes techniques? Certaines méthodes d'organisation très strictes ont été préconisées pour résoudre ce problème, comme la méthode des "équipes à programmeur en chef" d'IBM[9].

> Brooks, l'architecte principal du système OS 360, résume ainsi, dans son livre intitulé *Le Mythe de l'Homme-Mois*[10], l'acuité des problèmes de personnel dans un projet de quelque taille : *Si l'on ajoute du personnel à un projet en retard, on ne fait que le retarder encore.*

Une autre question liée à la gestion des projets est celle de la maîtrise des coûts et des délais. Un élément important est la possibilité d'effectuer des estimations à l'avance ; c'est ici qu'interviennent les modèles.

1.7 - MODELES ET MESURES

L'une des idées essentielles de la science moderne, formulée avec éclat par Lord Kelvin au siècle dernier, est que l'on ne connaît bien que ce que l'on sait mesurer. Le génie logiciel ayant pour ambition d'asseoir la production de logiciel sur des bases scientifiques, il est naturel que de nombreux spécialistes se soient demandé si l'on pouvait appliquer à ce domaine des techniques quantitatives. Des méthodes de modélisation et de mesure ont ainsi été proposées ; elles sont encore loin de constituer une véritable "physique du logiciel" mais contiennent des éléments qui peuvent être utiles au praticien. Voyons quelles sont les principales idées en ce domaine.

Les modèles applicables au logiciel se répartissent en deux catégories principales : modèles de **coût**, modèles de **fiabilité**. Les premiers permettent d'évaluer a priori les dépenses liées à un projet ; les seconds, d'estimer le taux d'erreurs dans un système.

1.7.1 - Modèles de coût

Les modèles de coût ont été proposés en grand nombre. Beaucoup d'entre eux se présentent, au moins dans leur version la plus simple, sous la forme

$$E = a\, I^b$$

où a et b sont des constantes ; une telle formule donne l'effort nominal E nécessaire à la construction du logiciel, exprimé en hommes-mois, en fonction du nombre de milliers d'instructions du programme final, I.

Ainsi, selon le modèle Cocomo de Boehm[11], qui donne (pour un logiciel intégré à un système complexe) $a = 2{,}8$ et $b = 1{,}2$, l'effort total nécessaire à la construction d'un produit dont le code comporte 20 000 lignes est

$$2{,}8 \times 20^{1{,}20} = 102 \text{ hommes-mois} \simeq 9 \text{ hommes} \times \text{années}$$

étant entendu qu'il s'agit de l'effort total, incluant toutes les phases de construction (spécification, etc.) et la rédaction de la documentation ; on se place dans l'hypothèse d'un logiciel réalisé dans

[9] F. Terry Baker, *Chief Programmer Team Management of Production Programming*, in *IBM Systems Journal*, Vol. 11, n° 1, 1972.

[10] *F. P. Brooks : The Mythical Man-Month*, Addison-Wesley, 1974.

[11] Barry W. Boehm : *Software Engineering Economics*, Prentice-Hall, 1981. Article de titre identique dans *IEEE Transactions on Software Engineering*, Vol. SE-10, n° 1, pages 4-21, juillet 1984.

des conditions industrielles.

Les formules de ce genre (obtenues par identification à partir de diverses bases de données contenant des informations collectées sur des projets industriels) appellent plusieurs remarques.

Il est bien connu que l'"homme-mois" est une unité douteuse (c'est l'un des thèmes du livre de Brooks cité plus haut, qui explique son titre) ; pour prendre un cas extrême, 365 personnes ne font pas en un jour le travail d'une en un an. Il convient donc d'interpréter le résultat E avec précaution. Cocomo (comme certains autres modèles) permet d'ailleurs d'estimer non seulement l'effort total E mais aussi le **délai** nominal donné, selon ce modèle (et toujours pour la catégorie des logiciels intégrés à des systèmes complexes), par la formule

$$D = 2{,}5\, E^{0{,}32}$$

qui donne pour l'exemple précédent 11,5 mois (en l'absence des multiplicateurs de l'effort mentionnés plus loin). Par division, on obtient la taille moyenne nominale T de l'équipe de développement, 8,5 personnes environ dans l'exemple choisi.

Le terme "nominal" appliqué aux résultats précédents (effort E, délai D, taille de l'équipe T) signifie que, pour Boehm, il existe pour tout produit logiciel une série de valeurs optimales, celles que fournissent les formules du modèle de base. Que se passe-t-il dans le cas bien réel où le responsable du projet souhaite soit aller plus vite (en mettant plus de monde au travail), soit diminuer la taille de l'équipe (en acceptant de retarder la fin du projet)?

Comme tout gestionnaire de projet le sait d'expérience, ce genre d'aménagement n'est pas gratuit en termes d'effort total et ne peut de toute façon s'opérer que dans certaines limites. Cocomo fournit des formules pour réviser en conséquence les estimations précédentes. On notera que la formule correspondant au premier cas (diminution du délai) ne s'applique qu'à une plage de délais allant de 75% à 100% du délai nominal D : pour Boehm, il est vain d'espérer gagner plus de 25% sur D, quelles que soient les ressources supplémentaires en personnel dont on dispose. C'est le "théorème" de Brooks cité plus haut qui réapparaît sous une forme plus précise.

Plusieurs commentaires viennent naturellement à l'examen d'un modèle de ce type.

On notera tout d'abord le caractère assez fruste de la mesure que constitue I, nombre d'instructions du programme (en milliers). La définition précise est "nombre de kilo-lignes d'instructions-source livrées" ; en d'autres termes, on ne considère que les instructions du programme-source, écrit dans un certain langage, dont le niveau d'abstraction influera inévitablement sur cette mesure ; le mot "instruction" couvre aussi bien les déclarations que les instructions exécutables, mais exclut les commentaires ; enfin seules sont prises en compte les instructions *livrées* : les programmes de test ou les outils de développement n'entreront dans le calcul que s'ils font partie du produit livré au client ; un appel de sous-programme compte pour une instruction, mais les instructions du sous-programme lui-même ne seront pas comptabilisées s'il appartient à une bibliothèque précédemment définie.

Cette mesure prête le flanc à la critique et, de fait, nombreux sont les auteurs qui ont proposé des critères de mesure plus fins que la simple ligne d'instruction-source livrée. Certains de ces critères cherchent ainsi à prendre en compte la complexité du graphe de contrôle des programmes. Mais il faut bien admettre que les mesures plus ambitieuses ne donnent pas, dans les études qui ont été menées sur la corrélation entre les divers critères proposés et l'effort réel observé expérimentalement sur les logiciels correspondants, des résultats très supérieurs à celui de la simple mesure I, qui présente l'avantage de pouvoir être calculée facilement par un simple programme de comptage.

On notera par ailleurs que l'application de modèles de ce type suppose que l'on connaît la taille du programme final, qui n'est pas toujours facile à déterminer à l'avance, même pour un responsable de projet expérimenté.

Une autre objection évidente est que les formules universelles comme celle de Cocomo ne permettent pas de rendre compte des conditions particulières à chaque projet : il est certainement irréaliste de traiter de la même façon un système de pilotage automatique d'avion, réalisé sur un microprocesseur à mémoire strictement limitée, par une équipe qui connaît mal le matériel et le

langage, et un programme de comptabilité écrit en Cobol sur IBM 3081 par une société de service rompue à ce genre d'exercice - même si le nombre d'instructions final des deux programmes est le même. Mais Cocomo permet de prendre en compte cette remarque : d'une part, le modèle comporte non pas une mais trois formules de base (celle que nous avons citée pour les logiciels intégrés à un système complexe, une autre pour les logiciels autonomes et moins critiques, et la troisième pour les cas intermédiaires). Mais surtout les estimations de base obtenues grâce à ces formules sont ajustées à l'aide d'un ensemble de multiplicateurs associés aux caractéristiques spécifiques de chaque projet.

Le tableau de la figure 8 donne les plages de variation pour chacun de ces multiplicateurs.

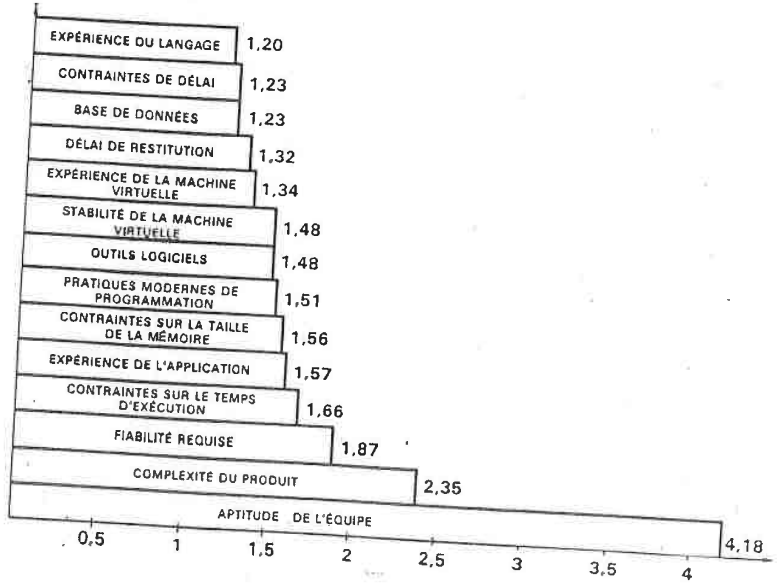


Figure 8 : Facteurs influant sur le coût réel d'un logiciel

Source : Barry W. Boehm, *Les Facteurs de Coût du Logiciel*, trad. F. Girard, *Technique et Science Informatiques (TSI)*, Vol. 1, n° 1, 1982.

Ainsi, selon ce tableau, un multiplicateur allant de 1 à 1,34 est associé au facteur "expérience de la machine virtuelle", c'est-à-dire que l'effort nominal doit être divisé ou multiplié par une valeur appartenant à cette plage selon que l'équipe de développement connaît particulièrement bien ou particulièrement mal la machine virtuelle (terme sous lequel Boehm regroupe l'ordinateur, le système d'exploitation, le compilateur, et les autres outils qui constituent ensemble la "machine" vue par les membres du projet).

L'examen de ces multiplicateurs (par exemple du dernier, relatif à la compétence de l'équipe, qui peut aller jusqu'à 4,2!) peut évidemment laisser sceptique sur l'applicabilité réelle de modèles tels que Cocomo : à quoi sert un modèle qui donne, d'un côté, des formules mathématiques précises, et de l'autre des coefficients de variation aussi importants, se rapportant à des facteurs dont l'analyse est inévitablement subjective?

Ce type de modèle est en effet difficile à défendre comme source de prévisions *absolues*. Comment déterminer (même si Cocomo donne quelques indications générales sur ce point) si l'*aptitude de l'équipe* doit être évaluée à 2,3 ou 2,7 par rapport à la moyenne? Mais la réponse change si l'on considère plutôt des valeurs *relatives*. Une société qui a entrepris sérieusement de collecter des mesures sur ses projets pendant un certain temps peut comparer les valeurs mesurées avec les prédictions du modèle, et donc *calibrer* celui-ci en fonction de ses caractéristiques propres. Il devient alors beaucoup plus facile d'estimer les paramètres non pas dans l'absolu mais par comparaison avec les valeurs retenues pour des projets précédents, sur lesquels le modèle a été essayé et affiné.

C'est de cette façon qu'il faut comprendre un modèle tel que Cocomo : non pas un oracle qui donne une réponse infaillible à une question précise, mais un élément d'estimation qui suppose une politique de collecte de mesures et la mise en place d'une base de données sur les coûts de développement de logiciel dans l'entreprise.

Notons pour terminer sur ce sujet que nous avons seulement donné un aperçu du modèle Cocomo, qui inclut aussi des techniques de prévision plus fines, dont on trouvera la description dans l'article et (surtout) le livre de Boehm cités plus haut.

1.7.2 - Modèles de fiabilité

Les modèles mathématiques de fiabilité[12], transposition au logiciel de la théorie classique de la fiabilité des systèmes matériels, permettent d'obtenir un certain nombre d'informations sur les erreurs qui subsistent dans un programme à partir d'informations sur le projet (particulièrement sur les erreurs précédémment détectées), de données de référence, et d'hypothèses statistiques sur la répartition des erreurs.

Implicitement, ces modèles traitent donc les défaillances des systèmes logiciels de la même façon que celles des dispositifs matériels. On peut trouver cette assimilation hasardeuse, puisque les défaillances des systèmes matériels proviennent non seulement de fautes de conception ou de réalisation mais aussi de la simple usure physique, phénomène qui bien sûr n'existe pas en logiciel.

Dans la suite, nous appelons *erreur* une anomalie présente dans le logiciel, et *défaillance* un cas observé de mauvais fonctionnement du logiciel. A la différence des défaillances matérielles, les défaillances d'un logiciel sont toujours dues à des erreurs.

Comme tout modèle mathématique, un modèle de fiabilité calcule un certain nombre de résultats en fonction d'un certain nombre de paramètres d'entrée. Ces paramètres d'entrée peuvent être :

- le temps (selon les modèles, on prend en compte soit le temps calendaire, soit le temps pendant lequel le programme s'exécute) ;
- les taux de défaillances observés en phase de mise au point ou (si l'on est déjà en phase d'exploitation) jusqu'à l'instant courant ;
- des résultats antérieurs relatifs à d'autres projets comparables ;
- des hypothèses statistiques (par exemple, l'un des modèles les plus connus, celui de Musa[13], suppose que les erreurs sont uniformément distribuées parmi les instructions du programme) ;

[12] Pour une référence très complète sur cette question (mais moins satisfaisante, malgré son titre très général, sur les autres aspects du génie logiciel), voir l'ouvrage *Software Engineering : Design, Reliability and Management*, de Martin L. Shooman, McGraw-Hill, 1984.

• des paramètres spécifiques du projet.

En sortie, on obtiendra par exemple :

• une estimation du nombre d'erreurs résiduelles ;

• le taux prévisible d'erreurs $z(t)$ (où t est le temps) ;

• le temps moyen jusqu'à la première défaillance (MTTF, *Mean Time To Failure*) : $MTTF(t) = \frac{1}{z(t)}$.

On notera que le dernier paramètre est choisi de préférence au "temps moyen entre défaillances" (MTBF), qui n'a pas de sens si l'on suppose que l'exploitation s'arrête après toute défaillance afin de permettre la correction de l'erreur, ou des erreurs, qui l'ont causée. Le modèle de Musa, qui utilise le temps d'exécution τ, a pour formule de base :

$$MTTF(\tau) = \frac{1}{f\ K\ N_0}\ e^{f\ K\ \tau}$$

où N_0 est le nombre d'erreurs restant au début de la phase considérée, f est la fréquence moyenne d'exécution des instructions du programme (fréquence moyenne d'exécution des instructions divisée par le nombre d'instructions du programme), et K est une constante (facteur de détection des erreurs).

Il est nécessaire de préciser ici (comme pour le modèle de coût Cocomo) que les modèles effectivement utilisés en pratique sont plus fins que ne laisserait supposer cette brève présentation, et que seule une politique de mesure systématique, aboutissant à une base de données propre à l'entreprise, permet de calibrer convenablement les paramètres.

On notera une technique amusante qui se rapproche des modèles de fiabilité : celle du *bebugging*, que l'on peut appeler en francais "pêche aux bogues" (la bogue, avant d'être le terme recommandé par le Premier Ministre pour traduire l'anglais "bug", désignait une variété de poisson). Supposez que vous ayez à compter le nombre de poissons dans un lac : vous aurez peut-être l'idée d'y introduire m poissons bagués et, après un temps suffisant, de pêcher n poissons, dont p seront bagués. Le rapport $\frac{m \times (n - p)}{p}$ vous donnera une idée du nombre total de poissons dans le lac.

Si maintenant, au lieu de poissons, vous voulez compter le nombre d'erreurs dans un programmes, la méthode correspondante consiste à introduire m erreurs volontaires ; si le processus de mise au point est poursuivi normalement, la détection de n erreurs, dont p sont des erreurs artificiellement introduites, permet d'estimer le nombre d'erreurs "naturelles". Une telle méthode n'a de sens que si l'on dispose de données expérimentales fiables sur la nature et le taux de répartition des erreurs ordinairement commises par les programmeurs, afin de garantir que les erreurs introduites leur sont bien statistiquement comparables (les poissons bagués doivent avoir la même probabilité que les poissons du lac de passer à travers les mailles du filet).

Pour intéressantes qu'apparaissent ces différentes techniques, on peut cependant ressentir un certain malaise vis-à-vis de la conception qu'elles reflètent des erreurs en programmation et plus généralement de la programmation elle-même. S'il apparaît naturel de traiter par des méthodes statistiques les défaillances de systèmes matériels, l'idée selon laquelle la présence d'erreurs dans un logiciel est un phénomène qui peut et doit être mesuré, donc finalement un phénomène normal, peut choquer.

Les erreurs dans les programmes sont des erreurs de raisonnement, et l'analyse quantitative a ses limites dans ce domaine. En phase de mise au point, le nombre d'erreurs restantes n'est pas nécessairement un bon indicateur du travail qui reste à faire (une seule erreur bien corsée peut en valoir beaucoup de vénielles) ; et en phase d'exploitation, le passager d'un avion ne sera pas nécessairement réconforté, surtout s'il est lui-même programmeur, par l'annonce que d'après les prédictions du modèle il ne doit rester "que" 0,005% d'erreurs résiduelles dans le logiciel d'atterrissage automatique.

[13] John D. Musa, *A Theory of Software Reliability and its Application*, in *IEEE Transactions on Software Engineering*, Vol. SE-1, n° 3, pages 312-327, juillet 1984.

1.8 - LANGAGES

Les langages de programmation ont pendant longtemps occupé une place sans doute exagérée dans les discussions sur le logiciel, mais l'excès inverse serait tout aussi regrettable : on ne peut négliger l'importance de l'outil d'expression fondamental en pour l'écriture des programmes et la description des données. Il convient d'ailleurs d'inclure dans cette discussion les notations dont se servent les informaticiens aux autres étapes du cycle de vie, particulièrement en amont : les langages de spécification et de conception.

La situation en matière de langages de programmation a pendant longtemps été caractérisée par une coupure presque complète entre la réalité industrielle, dominée par les langages des années cinquante et soixante (langages d'assemblage, Fortran, Basic, Cobol, PL/I), et les recherches universitaires qui tournaient autour des dérivés d'Algol 60, d'une part, et de ceux de Lisp, d'autre part. Depuis 1975 environ, cette division a été remise en cause par la percée industrielle, encore limitée mais indéniable, de plusieurs nouveaux langages venus d'horizons divers : **Pascal**, issu de la recherche universitaire et d'abord destiné à l'enseignement, qui a atteint une diffusion importante en mini- et en micro-informatique ; **C**, développé aux Laboratoires Bell (AT&T) et qui gagne du terrain en même temps que le système Unix ; **APL**, ancien dans sa conception (1960), mais qui n'est utilisable en vraie grandeur que depuis quelques années.

Un langage plus récent a fait couler beaucoup d'encre : Ada, développé par une équipe de CII-HB (aujourd'hui Bull) en réponse à un appel d'offres du Ministère Américain de la Défense (DoD). Ada est sans doute le premier langage d'ambition industrielle à avoir été conçu selon des critères qui ressortissent incontestablement du génie logiciel : modularité, avec la notion de "paquetage" ; compilation séparée, importante pour la programmation en équipe et pour la gestion des configurations ; généricité, qui permet d'utiliser une même structure (procédure, type) dans des contextes différents et favorise donc les facteurs de réutilisabilité et de compatibilité ; sévérité des contrôles de types, dans la tradition de Pascal, qui permet d'améliorer la validité et la fiabilité (les erreurs de conception se traduisent souvent, au moment de la mise en oeuvre, par des incompatibilités de types, que des langages laxistes toléreront abusivement). L'un des domaines d'application privilégiés envisagé pour Ada est la programmation des applications "temps réel", et le langage possède aussi des caractéristiques adaptées à ce type d'applications (primitives pour le calcul parallèle).

Dans l'attente de compilateurs qui soient à la fois facilement accessibles, de qualité industrielle et adaptés aux principaux matériels du commerce, Ada reste un pari[14]. Un échec de ce pari aurait sans doute un influence considérable sur l'évolution du génie logiciel.

L'apparition d'Ada aura de toute façon marqué une étape dans l'évolution des langages. On peut dire qu'Ada constitue le couronnement de la lignée des langages de programmation classiques, dans la filiation Fortran—Algol 60—Algol W—PL/I—Algol 68—Pascal (filiation certes illégitime à certaines étapes) ; l'effort consacré à la conception du dernier-né semble avoir presque complètement fermé, au moins pour un temps, toute recherche dans cette famille de langages, avec quelques exceptions comme la création du langage Modula 2 par Niklaus Wirth, l'auteur de Pascal.

Cela ne signifie pas, bien sûr, que les recherches aient cessé sur le thème des langages en général ; on peut noter que ce domaine connaît au contraire depuis quelques années un regain d'intérêt. Les travaux actuels s'orientent pour la plus grande part vers des langages s'écartant notablement de la famille citée précédemment ; il s'agit en général de langages de plus haut niveau, moins **impératifs** que les langages courants. Comme on l'a souvent fait remarquer, les langages

[14] Au moment où sont rédigées ces lignes, il existe des compilateurs répondant à l'un de ces critères, parfois à deux, mais non aux trois. Voir la table fournie régulièrement par C. Robert Morgan sous le titre *Matrix of Ada Language Implementation* dans le bulletin *Ada Letters* du groupe Ada de l'ACM ; dernier état en date dans le Vol V, n° 1, juillet-août 1985.

classiques possèdent maintes caractéristiques directement inspirées de l'architecture typique des ordinateurs actuels :

- faible niveau d'**abstraction** des objets manipulés (malgré les primitives de construction de nouveaux types offertes par la plupart des langages depuis Algol W) ;
- importance de la notion de **variable** et donc d'effet de bord (modification dynamique et répétée de la valeur du même objet) ;
- nécessité de raisonner en termes de **commandes** plutôt que de conditions logiques, qui accroît la distance de la spécification au programme ;
- obligation pour le programmeur d'indiquer précisément l'**ordre** séquentiel du déroulement des instructions ;
- description "procédurale" du traitement, fondée sur les **actions** à effectuer et non sur la vie individuelle des objets du système.

De nombreux langages proposés depuis quelques années cherchent à s'affranchir de ces différents défauts :

- les langages **ensemblistes** tels que SETL[15] offrent les ensembles, les listes etc. comme structures de données primitives ;
- les langages **fonctionnels**, nouvelles versions de Lisp[16] ou créations plus récentes comme FP[17] et SASL[18], s'appuient sur la notion de fonction de préférence à celle de variable ;
- les langages **logiques**, en particulier Prolog[19] permettent d'assimiler les programmes à des clauses de logique mathématique (calcul des prédicats du premier ordre) exécutables, en rapprochant leur forme de celle de spécifications formelles ;
- les langages dits **à flot de données** ou à "affectation unique" tels que VAL, LAU, SISAL ou Lucid, permettent de ne pas préciser l'ordre d'exécution des instructions lorsqu'il n'a pas d'influence sur la sémantique finale du programme, et d'autres langages (fondés sur les concepts des "processus séquentiels coopérants" de Hoare[20] ou ceux du "calcul des systèmes communicants" de Milner[21]) s'orientent vers le parallélisme véritable ;
- enfin, les langages dits **à objets**, dérivés de Simula 67[22] tels que Smalltalk[23], C++[24] ou Objective C[25] permettent de décrire un système non pas comme une suite d'instructions à exécuter mais comme un ensemble d'objets dynamiques, autonomes et communicants.

[15] R. B. K. Dewar, A. Grand, S. C. Liu, J.T. Schwartz et E. Schonberg : *Programming by Refinements, as exemplified by the SETL Language*, in *ACM Transactions on Programming Languages and Systems*, Vol. 1, n° 1, pages 27-49, juillet 1979.

[16] Guy L. Steele : *Common Lisp*, Prentice-Hall, 1984.

[18] John Backus, *Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs*, in *Communications of the ACM*, Vol. 21, n° 8, pages 613-641, août 1978.

[18] David Turner : *SASL Language Manual*, St. Andrews University, 1981.

[19] Alain Colmerauer, Henry Kanoui, Michel van Caneghem : *Prolog, Bases théoriques et Développements actuels*, in *Technique et Science Informatiques (TSI)*, Vol. 2, n° 2, pages 271-311, 1983.

[20] C.A.R. Hoare, *Communicating Sequential Processes*, in *Communications of the ACM*, Vol. 21, n° 8, pages 666-677, août 1978.

[21] Robin Milner, *A Calculus of Communicating Systems*, Lecture Notes in Computer Science 92, Springer-Verlag, 1980.

[22] Graham Birtwistle, Ole-Johan Dahl, Bjorn Myrhaug et Kristen Nygaard : *Simula Begin* ; Studentliteratur et Auerbach Publishers, 1973.

[24] Bjarne Stroustrup : *Data Abstraction in C*, in *AT&T Bell Laboratories Technical Journal*, Vol. 63, n° 8 (2ème partie), octobre 1984.

[23] Adele Goldberg et David Robson : *Smalltalk-80, the Language and its Implementation*, Addison-Wesley, 1983.

[25] Brad J. Cox : *Message/Object Programming : An Evolutionary Change in Programming Technology*, in *IEEE Software*, Vol. 1, n° 1, pages 50-69, janvier 1984.

Il reste à savoir ce qui, dans ce bouillonnement, va émerger. Les langages fonctionnels et logiques jouent un rôle essentiel dans les recherches sur l'intelligence artificielle ; leur applicabilité au génie logiciel reste à démontrer. Ils sont parfois proposés comme langages pour le prototypage rapide ; SETL a été utilisé avec succès dans ce domaine, ayant servi de langage pour le premier compilateur Ada officiellement validé[26]. Les perspectives offertes par les langages à objets nous semblent plus prometteuses ; leur rôle peut à notre sens être fondamental pour résoudre quelques-uns des problèmes cruciaux du génie logiciel, en particulier la réutilisabilité et l'extensibilité. Ce point de vue est développé au chapitre 4 de cette thèse et dans les textes correspondants [78b, 79b, 85d] de la troisième partie.

1.9 - OUTILS

Dans toute branche de l'ingénierie, les outils communément disponibles jouent un rôle considérable. Il ne s'agit pas seulement de leur utilité concrète et quotidienne : à plus long terme, il se forme autour des outils, chez les praticiens du domaine, une *culture* aussi importante que la culture proprement scientifique. Devient homme du métier celui qui a non seulement maîtrisé les concepts et les connaissances jugés nécessaires à un certain moment de l'évolution des techniques, mais aussi appris à manier et à dominer les outils qui, à ce même moment, constituent le support ordinaire des professionnels.

En logiciel, l'outil est d'abord matériel, devenu depuis quelques années de moins en moins lointain grâce à la diffusion de la micro-informatique et à la décentralisation des fonctions sur les grands systèmes ; nous allons y revenir. Mais les outils qui nous intéressent ici au premier chef sont les **outils logiciels** : programmes destinés à favoriser le développement, la modification et la diffusion d'autres programmes.

Un catalogue des différents types d'outils de génie logiciel dépasserait les limites de cette introduction[27]. Nous nous bornerons à citer quelques-uns des domaines les plus prometteurs.

1.9.1 - Outils d'aide à la construction des programmes

Parmi les outils qui interviennent lors des phases de construction, certains sont classiques, comme les "éditeurs de textes" sur les systèmes interactifs. Parmi les développements plus originaux, on peut citer :

- les outils d'aide à la conception, en général associés à un langage de conception ou "pseudo-code" (également appelé PDL pour *Program Design Language*), et plus généralement tous les outils ("préprocesseurs" etc.) grâce auxquels les programmeurs peuvent faire semblant de croire qu'ils disposent de langages de programmation moins primitifs qu'ils ne le sont en réalité ;
- les "générateurs de programmes" ou "langages de quatrième génération"[28], progiciels paramétrables permettant de produire, dans un domaine d'application bien délimité, des programmes adaptés aux besoins de chaque utilisateur, à qui il suffira de préciser les paramètres spécifiques de son application (de tels outils existent en informatique de gestion où ils commencent à concurrencer sérieusement Cobol pour les applications de nature répétitive et bien connue, mais aussi dans d'autres domaines où l'on peut traiter les applications les plus courantes dans un cadre normalisé, comme l'analyse syntaxique) ;

[26] Philippe Kruchten et Edmond Schonberg : *Le Système Ada/Ed : une expérience de prototype utilisant le langage SETL*, in *Technique et Science Informatiques (TSI)*, Vol. 3, n° 3, pages 193-200, 1984.

[27] Cf. Jean-Marc Nerson, *Panorama des outils d'aide à l'amélioration de la qualité des logiciels*, Note Atelier Logiciel n° 41, Electricité de France, Direction des Etudes et Recherches, avril 1983. L'auteur et J.-M. Nerson ont entrepris un ouvrage de synthèse sur la question.

[28] Ellis Horowitz, Alfons Kemper et Balaji Narasimhan : *A Survey of Application Generators*, in *IEEE Software*, Vol. 2, n° 1, pages 40-54, Janvier 1985.

• les éditeurs structurels[29] qui permettent de créer, de manipuler et de modifier des textes structurés (programmes, spécifications, etc.) en fonction de leur structure, et non pas comme de simples suites de caractères ; ces outils devraient aujourd'hui sortir des milieux de la recherche et gagner l'industrie.

1.9.2 - Outils de gestion de projet et de configurations

L'ordinateur permet aujourd'hui de gérer bien des tâches humaines ; le développement de logiciel ne devrait pas faire exception. De nombreux outils ont été proposés dans ce domaine, allant de simples outils ponctuels (permettant par exemple la prise en compte et le contrôle des emplois du temps individuels dans une équipe) à de véritables systèmes intégrés de gestion de projet, regroupant autour d'une base de données centrale l'ensemble des données : le chef de projet dispose ainsi à chaque instant du tableau de pilotage complet, comprenant des informations techniques et d'autres relatives aux coûts, aux tâches, aux délais, à l'ordonnancement. Il est clair que l'utilisation de tels systèmes n'a de sens que si elle s'accompagne de méthodes rigoureuses pour la gestion et le suivi de projet.

Une activité complémentaire de la gestion de projets est la **gestion de configurations**, dont le but est de contrôler l'évolution des différents composants d'un logiciel (programmes, spécifications, documents de conception, jeux d'essai, données, documents divers) tout au cours du projet.

On rencontre encore rarement dans la pratique des systèmes intégrés de gestion de projet et et de configurations, bâtis autour d'une base de données complète, mais il existe des outils individuels qui n'en rendent pas moins d'appréciables services. Dans le domaine de la gestion de configurations, en particulier, de nombreux environnements offrent des outils inspirés de deux produits disponibles sous Unix, SCCS (*Source Code Control System*), qui permet d'archiver les versions successives d'un document (non pas nécessairement un programme malgré le nom du produit), et Make qui permet de reconstruire un logiciel en exécutant automatiquement les actions de recréation rendues nécessaires par l'évolution de certains composants (par exemple un programme-source qui a été modifié postérieurement à la création du module-objet correspondant doit être recompilé).

Plusieurs équipes développent aujourd'hui des outils de gestion de projets et de configurations qui s'appuient sur des systèmes de gestion de bases de données. On trouvera dans l'article [85f] (voir aussi le chapitre 3 de la thèse) la description d'un outil en cours de réalisation qui s'appuie sur un modèle binaire des relations entre composants logiciels et sur la notion de contrainte sémantique.

Toujours dans le domaine des outils de gestion, nous avons cité plus haut les modèles de coût ; des outils logiciels sont associés à certains de ces modèles (comme Wicomo de l'Institut Wang pour Cocomo, Price-S de RCA pour le modèle du même nom, etc.)

1.9.3 - Outils de contrôle et de validation

La méthode de validation la plus répandue, malgré ses limitations évidentes, est le test dynamique, c'est-à-dire l'exécution du système dans des situations prédéterminées et la comparaison des réponses obtenues avec un scénario-type établi à l'avance.

Il est d'autant plus surprenant de constater que peu d'outils permettant d'automatiser le processus de test existent en pratique. Des générateurs automatiques de jeux d'essai ont été développés par des chercheurs, mais sont rarement applicables en pratique à l'exception de quelques domaines bien définis comme le test des compilateurs, où des batteries de tests

[29] Véronique Donzeau-Gouge, Gérard Huet, Gilles Kahn et Bernard Lang : *Programming Environments Based on Structured Editors : the MENTOR Experience* ; in *Interactive Programming Environments*, réd. David R. Barstow, Howard E. Shrobe, Erik Sandewall. McGraw-Hill, 1984 ; Nico Habermann et al., *The Second Compendium of Gandalf Documentation*, Carnegie-Mellon University, Pittsburgh, 1982 ; Bertrand Meyer et Jean-Marc Nerson : *Cépage : un éditeur structurel pleine page*, in *Second Colloque de Génie Logiciel*, AFCET, Nice, juin 1984.

normalisées existent pour certains langages comme Pascal[30] et Ada.

Des outils de validation de portée théorique limitée mais qui peuvent rendre de grands services sont disponibles dès aujourd'hui. Il s'agit des **analyseurs statiques**, qui permettent d'examiner un texte de programme (indépendamment de toute donnée et donc de toute exécution) pour y détecter des anomalies possibles et fournir des éléments de documentation automatisée, et des **analyseurs dynamiques** (ou "moniteurs de test") qui permettent de contrôler l'exécution d'un programme sur des jeux d'essai et d'en déduire un certain nombre de résultats et de mesure ("taux de couverture"). Un exemple de produit commercial combinant l'analyse statique et l'analyse dynamique sur des programmes Fortran est RXVP de General Research, installé avec succès à la Direction des Etudes et Recherches d'EDF.

1.9.4 - Au-delà des outils : les environnements

Quand les outils deviennent nombreux et complexes, leurs qualités individuelles ne suffisent plus : encore faut-il qu'on puisse les employer les uns avec les autres. Ce problème de compatibilité a souvent été très mal résolu par les systèmes d'exploitation classiques : le compilateur, l'éditeur de textes, l'interprète de commandes s'y présentent en général comme des entités distinctes, construites et utilisées selon des conventions différentes, voire contradictoires, qui provoquent chez l'utilisateur gêne et parfois danger (comme sur tel système où deux outils permettent de copier des fichiers : l'un exige que l'on nomme d'abord la source, puis la cible ; l'autre attend l'inverse).

La notion d'**environnement logiciel intégré** est née d'une réaction contre les multiples incompatibilités qui polluent les systèmes classiques.

La plupart des environnements actuels s'inspirent au moins en partie d'Unix. Unix était au départ un système d'exploitation plus qu'un environnement au sens défini ci-dessus ; il serait en outre exagéré de dire que ce système est complètement intégré. Mais les outils disponibles sous Unix bénéficient d'un niveau de compatibilité encore à peu près unique dans la confrérie des systèmes. Ce remarquable succès a été obtenu grâce à la simplicité de la conception de base du système, et à la combinaison d'un petit nombre d'idées fructueuses :

• la structure des fichiers est uniforme : les fichiers "normaux" sont de simples suites de caractères, accessibles soit en séquence soit par leur index ;

• l'interactivité est ramenée au schéma précédent : on considère le terminal comme un fichier, qui peut être utilisé comme entrée ou comme sortie par tout programme apte à lire ou à écrire sur des fichiers ;

• un programme simple sous Unix est en général un *filtre* qui prend en entrée un fichier du type précédent et produit en sortie un fichier du même type ;

• deux ou plusieurs programmes de ce type peuvent donc être composés en "soudant" la sortie de l'un à l'entrée de l'autre : c'est la notion de **tuyau**. La notation

refer | *tbl* | *eqn* | *troff*

désigne un tuyau de cette sorte, obtenu en composant les outils Unix de traitement de texte : *refer* (traitement des références bibliographiques), *tbl* (traitement des tables), *eqn* (traitement des textes de type mathématique), *troff* (photocomposition). La sortie de chacun d'entre eux est utilisée comme entrée du suivant[31].

[30] Jacqueline Sidi : *Validation de Compilateurs : Application à Pascal*. in *Technique et Science Informatiques (TSI)*, Vol. 2, n° 5, pages 345-354.

[31] Ce texte étant préparé à l'aide de certains des outils en question, comme l'ont été plus généralement la première et la deuxième partie de cette thèse et tous les articles de la troisième partie postérieurs à septembre 1983, l'auteur doit à la vérité de préciser que la réalité quotidienne, faite d'un long combat contre les bizarreries individuelles de ces outils et les subtiles incompatibilités qui se cachent derrière leur accord de façade, n'est pas toujours idyllique.

La simplicité et l'élégance de ces structures de base ont favorisé l'éclosion sous Unix d'une quantité considérable d'outils qui (à la différence de ce qui se passe sur d'autres systèmes) sont souvent deux à deux compatibles. La table de la figure 9 indique quelques-uns de ces outils (non nécessairement tous présents dans toutes les versions commerciales du produit).

Application	Outils
Compilateurs et interprètes	C, Pascal, Fortran 77, Lisp, Prolog, FP ; *yacc* et *Lex* (construction de compilateurs).
Courrier électronique	*mail*, UUCP (réseau)
Utilitaires de base	coquille . (interprète de commandes) ; gestion de fichiers (copie, déplacement, etc.) ; éditeurs de texte (*ed, vi, emacs*) ; création de processus parallèles.
Traitement de textes	*nroff/troff* (formatage de textes et photocomposition), description de tables (*tbl*), textes mathématiques (*eqn*), bibliographies *(refer)*, contrôle de l'orthographe et du style (*spell, diction, style*).
Traitement de données	recherche de modèles dans un texte (*grep*), comparaison de fichiers (*diff*), tri (*sort*), transformation de fichiers (*sed, awk*).
Développement de logiciel	gestion de versions et de configurations (*sccs, make*)

Figure 9 : Quelques outils sous Unix

Pour les nombreux spécialistes qui cherchent actuellement à développer des environnements logiciels, Unix et ses contemporains (tel Interlisp, dans lequel la structure de référence commune est, plutôt que le fichier, la liste au sens du langage Lisp) ne méritent pas véritablement le qualificatif "intégré". De Drouas et Nerson[32] les appellent environnements de *deuxième génération* ; ces environnements évitent les incompatibilités inhérentes à leurs prédécesseurs, mais sont essentiellement des boîtes à outils. Pour certains auteurs, seuls peuvent être dits intégrés les environnements dans lesquels les outils sont liés par un fil conducteur plus solide qu'un simple ensemble de conventions homogènes : par exemple un langage (comme dans les environnements Apse définis autour d'Ada par le DoD) ou une méthode intégrée de développement.

Cette vision selon laquelle les environnements doivent être "totalitaires", bâtis autour d'une idée centrale très forte, ne fait pas cependant l'unanimité ; les ateliers "boîtes à outils" de type

[32] Eric de Drouas et Jean-Marc Nerson : *Les Ateliers Logiciels Intégrés : Développements français actuels*, in *Technique et Science Informatiques (TSI)*, Vol. 1, n°3, pages 211-232, 1982.

Unix, moins intégrés mais plus souples et ouverts, ont fait leurs preuves et conservent de chauds partisans.

Une chose est certaine pour l'avenir : les nouveaux ateliers intégrés prendront de plus en plus en compte les progrès du **matériel**. L'élément important ici est le développement des **postes de travail** qui, grâce à la puissance croissante des micro-processeurs et aux progrès des réseaux (locaux et à distance), permettent de concilier les avantages des micro-ordinateurs et ceux des grands sysèmes.

Un aspect remarquable de ces nouveaux postes de travail est la qualité de l'interface, utilisant des écrans de haute résolution et des dispositifs d'entrée rapide (souris, tablette). En particulier, nombre d'entre eux (reprenant les idées introduites à l'origine par l'environnement Smalltalk de Xerox, bâti autour du langage de même nom) permettent de diviser un écran en plusieurs zones rectangulaires ou fenêtres ; si le logiciel est à la hauteur, c'est-à-dire s'il permet de gérer plusieurs processus actifs en parallèle, affectés chacun à une fenêtre (les possibilités d'Unix en matière de création de processus concurrents font que ce système est bien adapté à des situations de ce genre), le confort des utilisateurs est singulièrement amélioré : ils peuvent en effet poursuivre simultanément plusieurs activités, chacune d'entre elles restant visible grâce à la présence de sa fenêtre sur l'écran.

Ces possibilités, que le lancement du MacIntosh par Apple ont fait connaître à un large public, sont particulièrement intéressantes pour les informaticiens, que leur tâche force souvent à être tout à la fois au four, au moulin et à la rivière. Dans une situation typique, par exemple celle de la mise au point d'un module, on peut être conduit à alterner constamment entre le compilateur, l'"éditeur" et le langage de commande ; même si l'on peut espérer qu'à l'avenir ces outils seront mieux "intégrés" qu'il n'est aujourd'hui de règle, la nécessité subsistera de pouvoir passer rapidement d'une vue à une autre. Les systèmes à multi-fenêtrage fournissent ici un élément de solution particulièrement intéressant. Ceci n'a de sens, bien entendu, que si la définition de l'écran est suffisante, et (répétons-le) si le système d'exploitation fournit le substrat logiciel adéquat en matière de gestion des processus concurrents.

Ce n'est pas par hasard que nous avons choisi de mentionner les évolutions du matériel au terme de cette brève visite guidée : malgré toute leur superbe, les spécialistes de logiciel sont bien obligés de reconnaître que, pour une large part, l'informatique reste traînée par le matériel (*hardware-driven*), selon le mot de Niklaus Wirth. Sans matériel, il n'y aurait pas de logiciel, c'est une platitude ; mais si le matériel n'avait pas évolué de façon aussi phénoménale, le génie logiciel ne serait peut-être pas aussi nécessaire, et serait de toute façon beaucoup moins intéressant.

1.10 - VERS UNE DISCIPLINE SCIENTIFIQUE

On nous permettra, au terme de cette promenade rapide, un point de vue plus personnel. Pour l'auteur, le défi majeur auquel les informaticiens sont aujourd'hui confrontés vient de trois des "facteurs de la qualité" cités plus haut : la validité, l'extensibilité, la réutilisabilité. Le logiciel fabriqué actuellement n'est pas assez sûr ; il est trop difficile à modifier ; il est trop spécifique.

> La plupart des autres problèmes du génie logiciel peuvent être rattachés, au moins en partie, à ces trois-là. La question de la maintenance, par exemple, ne sera pas résolue par des recherches visant à améliorer les méthodes de maintenance elle-même, mais par des progrès dans les méthodes de construction, permettant de produire des systèmes plus corrects et plus souples. Les coûts du logiciel, pour prendre un autre exemple, ne peuvent diminuer qu'au prix d'une industrialisation véritable, c'est-à-dire de la mise au point de composants normalisés, réutilisables et combinables. Or en logiciel, nous l'avons déjà signalé, on repart trop souvent de zéro : à l'instant où vous lisez ces lignes, combien de personnes de par le monde sont-elles en train d'écrire, pour la mille-et-unième fois, un programme de tri ou de recherche en table?

Il nous semble donc que les problèmes-clés du génie logiciel (ceux dont la solution peut apporter des sauts quantiques, et non pas seulement des améliorations marginales) sont pour l'heure des problèmes **techniques** : problèmes de méthodes, de langages, d'outils.

Cette opinion, il est juste de le préciser, est loin d'être universelle ; pour toute une école, c'est du côté de la gestion des projets que le bât blesse. B. W. Boehm, par exemple, estime que plus de 50% des problèmes du développement de logiciel sont des problèmes de gestion. Si l'on part de telles prémisses, il est clair que les solutions seront recherchées en priorité du côté de l'étude des pratiques industrielles, de la collecte de mesures ; si l'on met l'accent sur des méthodes, il s'agira de méthodes de gestion de projet ou d'organisation des équipes plutôt que de spécification formelle ou de conception par objets.

L'importance des problèmes de gestion apparaît clairement à quiconque a observé le déroulement d'un projet, mais dans l'état actuel du métier ils nous semblent plus un symptôme qu'une cause première. Le génie logiciel n'a pas atteint un niveau suffisant pour que les aspects techniques puissent être considérés comme secondaires par rapport aux questions d'organisation. Il est à peu près aussi fructueux d'espérer faire progresser la qualité du logiciel par l'analyse des méthodes de travail employées dans les entreprises qu'il l'eût été de fonder les progrès de l'épidémiologie, dix ans avant Pasteur, sur l'étude des pratiques hygiéniques en vigueur dans les hôpitaux.

Où sont donc les progrès potentiels? Ce tour d'horizon a fait ressortir quelques domaines particulièrement importants, dans lesquels il ne nous semble pas déraisonnable d'espérer des améliorations considérables si l'on s'en donne les moyens :

- la nécessité d'une démarche plus rigoureuse vis-à-vis de la construction de programme, fondée sur les mathématiques et plus particulièrement sur la logique : c'est à ce prix que des améliorations significatives pourront être apportées à la validité des logiciels, à travers l'utilisation de spécifications formelles et de techniques de construction systématique ;
- la formation des informaticiens, qui apparaît essentielle en regard du point précédent (les méthodes formelles, on l'a vu, font appel à la culture mathématique), et dont le multiplicateur 4.2 attribué au facteur "aptitude de l'équipe" dans le modèle Cocomo, résultat d'une analyse de la réalité actuelle, fait cruellement ressortir l'importance ;
- l'utilisation de langages et d'outils modernes, en particulier dans le domaine de la conception et de la programmation par objets (meilleure solution connue, à notre sens, au problème de la réutilisabilité et à celui de l'extensibilité), des outils d'aide à la construction des programmes (générateurs de programmes et éditeurs structurels tout particulièrement), de la gestion de configurations ;
- l'utilisation des possibilités des postes de travail modernes comme composants essentiels d'environnements véritablement intégrés.

Le reste de cette thèse marque quelques modestes jalons sur ces différentes voies.

ANNEXE

Contre-exemples pour la recherche dichotomique (cf. Figure 6)

Pour chacun des quatre programmes, on trouvera ci-après une ou plusieurs valeurs du tableau t et de l'élément x pour lequel le programme est incorrect. Dans tous les cas, t contient un ou deux éléments.

- Programme **P1** :
  $t = [1\ 2]$, $x = 2$ (bouclage infini)

- Programme **P2** :
  $t = [1\ 2]$, $x = 0$ (débordement du tableau)
  $t = [1]$, $x = 1$ (résultat erronné : *présent* mis à **faux** alors que l'élément est là)

- Programme **P3** :
  $t = [1\ 2]$, $x = 0$ (débordement du tableau)

- Programme **P4** :
  $t = [1\ 2]$, $x = 0$ (débordement du tableau)

DEUXIEME PARTIE :

PRESENTATION DES ARTICLES

CHAPITRE 2

LES TEXTES ET LEURS THEMES

2.1 - UNE CERTAINE IDEE DE LA PROGRAMMATION

La troisième partie de cette thèse, qui en est la composante essentielle, reprend un ensemble de textes écrits entre 1975 et 1985.

Les textes considérés traitent de domaines variés de la programmation, mais un fil solide les relie. Leur objet commun peut être défini comme la recherche de méthodes rigoureuses de construction de programmes, modérée par le souci constant de prendre en compte les contraintes d'un environnement industriel.

Cette double exigence, cette tentative de synthèse entre principes scientifiques et impératifs de production s'expliquent en partie par les circonstances dans lesquelles la plus grande partie des travaux présentés ici ont été rédigés : la direction d'un petit groupe d'informaticiens, soucieux de participer activement aux progrès de leur discipline, mais sachant qu'elle restait marginale pour leur entreprise et qu'on attendait surtout d'eux des prestations plus immédiatement utiles. En dépit de ses inconvénients évidents, cette situation ne laissait pas de comporter quelques bénéfices : la fréquentation quotidienne de collègues qui voyaient dans la programmation plus une corvée qu'un objet d'étude nous obligeait à examiner de près les effets concrets des méthodes proposées par les théoriciens ; elle nous interdisait d'oublier la distance qui sépare la recherche des applications et l'université de l'industrie ; elle nous offrait enfin une source régulière de problèmes intéressants.

Mais les circonstances n'expliquent pas seules le compromis (ou, si l'on préfère, l'hésitation) entre rigueur et pragmatisme qui caractérise les réflexions reprises ci-après. Il s'agit aussi, plus profondément, de toute une attitude vis-à-vis de la programmation. Nous reconnaissons le caractère essentiellement mathématique de cette discipline, mais nous admettons aussi l'importance des aspects non directement mathématisables : problèmes d'organisation, rôle des outils, qualité des notations. L'expression *génie logiciel* est éclairante à cet égard, puisqu'une discipline d'ingénieur se caractérise par une solide base scientifique tempérée par l'importance des considérations pratiques.

2.2 - THEMES

Les textes reproduits dans la troisième partie se caractérisent, au-delà de leurs sujets immédiats, par la présence d'un petit nombre de thèmes récurrents. C'est sur ces thèmes, non sur l'ordre de publication, qu'a été fondée la structure de cette partie de la thèse.

On notera dans l'énumération de ces thèmes le va-et-vient entre la vue "macroscopique" (*programming-in-the-large*) des systèmes logiciels et l'étude "microscopique" des objets de la programmation (structures de données, algorithmes, programmes).

- **Structure et Systèmes :** la notion (macroscopique) essentielle pour les applications industrielles est celle de système chargé d'assurer tout un ensemble de services. Cette notion s'oppose à celle de simple programme assurant une fonction unique. Les systèmes réels pouvant être complexes et, surtout, susceptibles d'une longue évolution, il est indispensable de procéder à une étude méthodique des principales lois qui régissent leur structure.

- **Modularité :** une direction de recherche importante, conséquence des principes précédents, a été l'étude des mécanismes permettant de décomposer des systèmes complexes (étudiés à différents niveaux d'abstraction : spécification, conception, mise en

oeuvre) en sous-systèmes autonomes et cohérents. Le chapitre et les articles qui présentent ces idées plaident avec vigueur pour l'utilisation de méthodes de construction par objets.

• **Spécification :** un thème régulièrement repris est l'idée selon laquelle en programmation, comme dans bien d'autres disciplines scientifiques, l'étape cruciale de la résolution d'un problème est souvent sa formulation précise. En informatique, cette formulation s'appelle spécification, et plusieurs articles traitent des méthodes et outils qui servent à la faciliter. Deux directions principales peuvent être dégagées : les premiers travaux ont porté sur la spécification d'objet individuels, en particulier de structures de données, utilisant la notion maintenant classique de type abstrait ; plus récemment, l'accent a été mis sur les techniques macroscopiques qui permettent de spécifier des systèmes entiers.

• **Construction systématique :** dans le domaine du microscopique, une série de travaux appliquent à la *construction* de programmes corrects les techniques développées par Floyd et Hoare pour permettre les *démonstrations* de validité des programmes. Ce renversement de point de vue est l'oeuvre de Dijkstra, dont nous nous inspirons directement. Ces travaux incluent quelques éléments de théorie, destinés à asseoir la méthode sur une base solide, et l'étude de quelques *heuristiques* de dérivation de programmes qui semblent particulièrement fructueuses.

• **Langages :** la mise en oeuvre de toute technique de génie logiciel soulève, d'une façon ou d'une autre, des problèmes de langage. De nombreux éléments des réflexions présentées ont trait à ces problèmes : langages de spécification, langages de programmation. Bien qu'elles ne soient pas entièrement exemptes du péché mignon des informaticiens - le goût immodéré pour la discussion de problèmes de syntaxe et l'invention de nouvelles notations -, ces réflexions ont en général porté sur le fond : comment les langages informatiques permettent-ils de mettre en oeuvre les principes méthodologiques essentiels? On trouvera tout à la fois des études comparatives de langages de programmation favorisant la mise en oeuvre des concepts modernes tels que la conception par objets, et des propositions de langages de spécification formelle.

• **Outils :** avec les méthodes et les langages, les outils forment le troisième volet de la trilogie fondamentale du génie logiciel. L'existence d'outils adéquats est indispensable pour le renouvellement des pratiques de programmation ; nous présentons notre expérience dans ce domaine. En retour, la notion d'outils logiciel soulève d'importants problèmes méthodologiques ; quelques éléments de solution sont proposés.

Les thèmes que nous venons de présenter forment l'objet des six premiers chapitres de cette partie de la thèse. Les trois derniers chapitres sont consacrés à des domaines d'application particuliers, où nous avons cherché à mettre en oeuvre les principes précédents.

• **La programmation vectorielle** a été étudiée (avec Alain Bossavit) comme un champ d'application des idées modernes sur la programmation. Il s'agissait d'aborder la construction de programmes efficaces sur le Cray-1 de façon méthodique, alors qu'elle nous avait été présentée à l'origine comme une ensemble de recettes peu rigoureuses. Les techniques utilisées incluent la construction à partir d'assertions, les types abstraits et la transformation de programmes.

• **Les systèmes interactifs** jouent un rôle essentiel dans l'utilisation des ordinateurs. Ils fournissent un objet d'étude particulièrement riche, puisqu'ils soulèvent des problèmes de méthode (comment structurer convenablement des applications interactives ?), d'outils (nous décrirons quelques réalisations) et de spécification. Nous abordons également la question de la qualité des interfaces humaines.

• **Un éditeur structurel, Cépage,** fait l'objet du dernier chapitre : nous décrivons les caractéristiques de ce produit (conçu en collaboration avec J.-M. Nerson), dont l'objet était de transférer une technique maintenant bien connue des chercheurs pour obtenir un outil répondant aux besoins des praticiens. La qualité de l'interface était l'un des points essentiels. Ce travail reprend nombre des éléments exposés par ailleurs dans cette thèse ; sa

conception résulte d'une approche semi-formelle qui nous semble refléter un compromis effort-bénéfice assez bien adapté à la situation de la programmation aujourd'hui.

CHAPITRE 3

STRUCTURE ET SYSTEMES

3.1 - UNE SYSTEMATIQUE DU LOGICIEL

Le thème "structure et systèmes" (ou structure *des* systèmes) regroupe un ensemble de réflexions que l'on peut qualifier d'horizontales par rapport à l'axe "vertical" du cycle de vie des projets logiciels. Il s'agit d'étudier dans son principe la structure des systèmes logiciels, considérés à un niveau de "coupe" quelconque (spécification, conception, mise en oeuvre etc.) en analysant les relations entre éléments d'un même niveau ou de niveaux différents.

Ce thème est donc le plus général de tous ceux que nous allons aborder, et c'est pour cela que nous commençons par lui ; mais les publications qui en traitent directement figurent parmi les plus récentes. On trouvera tout d'abord une étude des relations entre éléments de **programmes**, appliquée au cas du langage Simula 67, à la fin du texte [85d][1] ; l'analyse du concept de modularité en programmation, objet principal de ce texte (cf. chapitre suivant), conduisait naturellement à l'examen des types de relations qui peuvent exister entre modules.

Les travaux en cours sur la méthode de spécification M [85g] ont également conduit à définir des relations entre modules, mais cette fois-ci à un niveau plus abstrait, celui de la description fonctionnelle des systèmes.

L'article [85f], enfin, étudie d'un point de vue encore plus général la question des relations pouvant exister entre éléments logiciels. Il s'agit d'ébaucher une véritable **théorie** des relations en logiciel, pouvant servir entre autres à fonder sur des bases rigoureuses un système de gestion de projets et de gestion de configurations.

L'objet commun de ces réflexions peut donc être défini comme une tentative de créer une **systématique** des logiciels, c'est-à-dire une classification précise des objets logiciels et de leurs relations[2].

3.2 - RELATIONS EN LOGICIEL

Les éléments de théorie présentés dans l'article [85f] s'appuient sur un modèle relationnel restreint où toutes les relations considérées sont binaires.[3] De se limiter aux relations binaires simplifie les notations et donne accès à un jeu d'opérateurs puissants (composition, inverse, fermetures transitives, etc.), sans pour autant diminuer la portée théorique du formalisme puisque toute relation n-aire (pour $n > 2$) peut s'exprimer comme jointure de relations binaires. Cet argument de principe ne suffit pas (car la décomposition d'une relation quelconque en relations binaires peut ne pas être très naturelle), mais il se double d'une considération pratique : il semble en effet rarement utile, dans la modélisation des systèmes logiciels, d'employer des relations d'ordre supérieur à deux.

---

[1] Les références entre crochets renvoient aux publications dont la liste est donnée au début de cette thèse, pages iii-iv. On trouvera à partir de la page 69 une liste de publications plus complète.

[2] On notera avec plaisir dans tout dictionnaire usuel que le terme de *systémique* est superflu.

[3] Une référence pourtant évidente a été oubliée dans [85f] : *Data Semantics*, de Jean-Raymond Abrial (dans *Data Base Management*, réd. J.W. Klimbie et K.L. Koffeman, North-Holland, 1974, pages 1-59), omission d'autant plus fâcheuse que cet article contient de nombreuses remarques directement applicables à la modélisation de relations entre éléments de logiciel. On trouvera une présentation des principaux travaux sur les systèmes binaires dans le chapitre 9 de *Data Models*, de Dionysios C. Tsichritzis et Frederick H. Lochovsky, Prentice-Hall, 1982.

L'article cite cependant le contre-exemple d'une relation ternaire importante, qui a été prise en compte dans le langage LM associé à la méthode de spécification modulaire M [85g] ; cette relation, qui peut s'exprimer sous la forme "Toutes les propriétés de *A* sont vraies de *B*, sauf *P*", a de nombreuses applications en logiciel.

Nous considérons donc un ensemble très général de relations entre objets logiciels. Les objets en question sont tous ceux qui peuvent intervenir dans un développement de système : éléments de spécifications, de programmes, de documents, de budgets, d'échéanciers, etc. Les relations peuvent s'appliquer soit à des objets de même nature (comme la relation *appelé_par* qui donne les procédures pouvant appeler une procédure donnée), soit à des objets de types différents (comme la relation *mis_en_oeuvre_par* qui exprime le lien entre les éléments de spécification et leurs réalisations).

La simple énumération des relations ne fournit qu'une description syntaxique des systèmes ; seuls les noms de ces relations donnent quelques indications sur leurs propriétés. Pour prendre en compte la sémantique des systèmes modélisés, il faut pouvoir décrire formellement les propriétés des relations ; nous utilisons pour cela la notion de **contrainte**.

Une contrainte est un prédicat sur une ou plusieurs relations. Par exemple, la contrainte

$$date^{-1} \; ; \; dépend\_de \; ; \; date \subseteq après$$

où *dépend_de* est une relation générale de dépendance (entre programme-source et programme-objet, spécification et mise en oeuvre, etc.), *date* est une relation entre objets et temps indiquant à quel moment un objet a été modifié pour la dernière fois, et *après* est la relation de postérité entre temps, exprime une propriété de cohérence que doit normalement satisfaire tout ensemble d'objets logiciels. C'est cette loi qu'appliquent implicitement des outils simples mais utiles de gestion de systèmes tels que Make sous Unix.

L'article présente ainsi un certain nombre de lois, qui ne couvrent qu'une partie du génie logiciel ; certaines, du reste, sont peut-être fausses. Mais il nous semble nécessaire de mener une analyse de ce genre pour comprendre la structure des systèmes logiciels. Cette analyse nous paraît en particulier indispensable si l'on veut pouvoir donner aux ateliers logiciels de l'avenir une assise solide, ce qui implique que l'on dispose d'une théorie précise des propriétés des objets logiciels.

3.3 - LE SYSTEME SKB

C'est précisément un élément d'atelier logiciel que décrit l'article [85f]. Le système intitulé SKB (*Software Knowledge Base*) se veut un outil général pour la gestion des projets et la gestion des configurations, appliquant les idées précédentes (utilisation de relations binaires et de contraintes) à la modélisation des propriétés essentielles des objets logiciels.

Le système SKB permet donc à ses utilisateurs de définir les propriétés des logiciels à gérer à l'aide d'un langage appelé Calcul des Relations, Attributs et Contraintes ou CRAC (les attributs sont une classe particulière de relations fonctionnelles). Le CRAC, et donc le noyau du système SKB, sont entièrement généraux : ils ne connaissent que les notions mathématiques liées aux relations et aux fonctions, et non les concepts spécifiques du logiciel.

Pour faire du SKB un système vraiment opérationnel, il est donc nécessaire d'ajouter au noyau universel une ou plusieurs **bibliothèques** spécifiques, écrites elles-mêmes en CRAC, et contenant la définition de notions utiles pour telle ou telle application. On pourra ainsi disposer de bibliothèques spécialisées pour le test, la conception, la gestion de configurations etc. Le langage CRAC contient des primitives simples permettant d'empaqueter (d'"encapsuler") des définitions éventuellement complexes de relations ou de contraintes pour les inclure dans une bibliothèque.

Deux prototypes du système ont été construits par des étudiants de Santa Barbara au printemps de 1985. L'un est écrit en Prolog, l'autre utilise Ingres, le SGBD relationnel d'Unix, avec (dans ce second cas) traitement des contraintes par des programmes C. Ces prototypes n'ont pas encore été essayés sur des exemples en vraie grandeur, mais leur comparaison semble

assez nettement favoriser l'utilisation d'un langage de programmation peu procédural tel que Prolog.

D'autres membres du projet (il s'agit dans tous les cas cités d'étudiants préparant des thèses de Master's Degree) travaillent sur des aspects complémentaires : modélisation exhaustive des relations et contraintes utiles dans la réalisation de projets industriels, et écriture des bibliothèques CRAC correspondantes ; définition précise du CRAC et étude d'algorithmes efficaces pour la vérification incrémentale des contraintes ; interface graphique interactive permettant aux utilisateurs de "voir" les objets logiciels et leurs relations.

3.4 - DIRECTIONS DE TRAVAIL

Les réflexions présentées sur la structure des systèmes logiciels dans [85d, 85f, 85g] doivent bien évidemment être complétées et approfondies.

Il est tout d'abord nécessaire de poursuivre l'étude des lois qui gouvernent la structure des systèmes. Deux applications nous semblent ici particulièrement importantes :

- l'analyse théorique de la structure des grands systèmes, sorte de **sémantique formelle macroscopique** (*in the large*), s'intéressant aux relations intermodulaires, par opposition à la sémantique classique qui est plutôt de type microscopique (l'accent y étant mis sur la formalisation des propriétés des objets de base internes aux modules : instructions, variables, types, déclarations etc.) ;
- l'étude expérimentale citée plus haut des propriétés qui caractérisent les projets logiciels dans l'industrie, étude menée à la lumière des concepts du système SKB et s'appuyant sur la modélisation en CRAC.

Pour se transformer en système utile, le SKB a besoin non seulement d'une mise en oeuvre efficace et de bonnes interfaces visuelles, mais aussi d'un large jeu de bibliothèques répondant aux applications les plus importantes.

On notera enfin que les réflexions sur les moyens d'exprimer grâce à la méthode de spécification M (cf. 5.5 ci-après) la structure abstraite des systèmes logiciels suscitent (comme nous l'avons signalé plus haut) des problèmes très voisins de ce qui a été étudié autour du SKB, et que certaines des relations mises en évidence à propos du SKB ont leur équivalent direct dans les constructions du langage (LM) associé à M.

CHAPITRE 4

MODULARITE

Les réflexions résumées au chapitre précédent conduisent tout naturellement à une discussion de la modularité en programmation : une fois analysés les divers types de relations qui peuvent intervenir dans la structure d'un logiciel, se pose la question plus concrète de choisir une bonne décomposition.

Trois des textes présentés étudient les critères qui peuvent servir de guide dans cette recherche : un bref extrait du chapitre 8 des *Méthodes de Programmation* [78b], un article de 1978 sur la conception des "progiciels" [82b], et un extrait du livre *Applied Programming Methodology* (en préparation) [85d].

4.1 - CRITERES ET PRINCIPES

L'extrait [85d] présente tout d'abord une tentative de classification des facteurs de la modularité. On y soutient que la notion de modularité ne peut pas être décrite par une définition unique ; c'est une description multivoque qui est donc proposée, s'appuyant sur cinq *critères* et cinq *principes*. Les principes sont aussi importants en pratique que les critères ; la différence est que les critères sont indépendants les uns des autres (pour n'importe lequel d'entre eux, on peut imaginer un système qui le viole mais satisfait aux quatre autres), alors que les principes découlent logiquement des critères.

Bien que nous nous soyons surtout intéressé à la modularité des programmes, les critères et principes considérés peuvent dans une large mesure se généraliser à d'autres types d'éléments logiciels (spécifications, documents de conception).

Les critères considérés sont :

- la **décomposabilité** ou facilité de division d'un système en sous-systèmes ;
- la **composabilité** ou facilité de combinaison d'éléments en systèmes ;
- la **lisibilité** ou facilité de compréhension autonome des éléments ;
- la **continuité** ou localité des effets du changement de spécification ;
- la **protection** ou localité des effets des erreurs.

Les principes sont les suivants :

- l'existence d'une **structure syntaxique** correspondant à la forme de modularité utilisée ;
- la limitation du **nombre d'interfaces** entre modules ;
- la limitation de la **taille des interfaces** ;
- le caractère **explicite** des interfaces (pas de communication en tapinois) ;
- l'**opacité des interfaces** (*Information hiding*), ou impossibilité d'utiliser un module autrement que par ses propriétés officielles[1].

On notera que certaines de ces conditions rejoignent la définition de la modularité donnée dans les travaux de Constantine et Yourdon (*Structured/Composite Design*). Mais ces travaux souffrent d'une limitation importante due à ce que leurs auteurs tiennent pour évidente l'identité des notions de module et de sous-programme. Ce postulat implicite (courant il y a quelques années encore) les conduit à des règles directement opposées à ce que nous préconisons : ils recommandent ainsi, pour assurer la "cohésion temporelle" du système, de regrouper toutes les initialisations dans un même module. Cette pratique

[1] Le parler des professionnels de l'informatique emploie généralement "transparence" pour opacité.

nous paraît au contraire néfaste car elle contredit les principes relatifs aux interfaces (petit nombre, petite taille, opacité) : un tel module général d'initialisation doit pouvoir accéder à la plus grande partie des structures de données du système. Dans la décomposition "par objets" que nous décrirons plus loin, chaque module gère un petit nombre de structures de données : il en est entièrement responsable et doit donc assurer leur initialisation.

Notre définition de la modularité est complétée, dans le texte cité, par une liste de cinq "mots-clés" qui définissent des techniques de programmation particulièrement propices à la mise en oeuvre des critères et principes : *rémanence ; généricité ; synchronisation ; communication ; compilation séparée.*

4.2 - CONCEPTION ET PROGRAMMATION PAR OBJETS

La mise au point des critères et principes précédents a suivi, plutôt qu'elle n'a précédé, l'adhésion à une méthode bien précise de décomposition des systèmes, appelée aujourd'hui "conception (ou programmation) par objets"[2]. Les fondements de cette méthode sont tout entiers contenus dans deux articles de Parnas (1972 et 1974). On trouvera une présentation des concepts essentiels dans le texte [78b] (extrait du chapitre 8 des *Méthodes de Programmation*) ; la discussion se poursuit dans l'article sur Simula [79b], et dans le chapitre d'*Applied Programming Methodology* [85d].

4.2.1 - Un renversement copernicien

Les présentations actuelles de la programmation par objets utilisent souvent le principe d'opacité (*Information hiding*) comme concept fondamental. Cette idée (incluse plus haut dans notre définition de la modularité) nous paraît certes fort importante, mais non pas vraiment caractéristique de la méthode par objets, puisqu'on peut tout aussi bien appliquer le principe d'opacité à une conception procédurale classique : il suffit pour cela d'associer à chaque procédure une interface bien conçue et bien documentée, en interdisant tout accès au sous-programme autrement qu'à travers cette interface.

Les principes qui définissent véritablement la méthode des objets nous semblent être les deux idées suivantes :

- tout d'abord, la remarque (développée dans le texte [85d]) selon laquelle aucun système de quelque ampleur ne peut être décrit comme assurant une fonction unique : c'est tout un ensemble de tâches qu'il faut considérer ;
- deuxièmement, l'idée (qui découle en partie de la précédente) selon laquelle la structure fondamentale adoptée pour la construction du système (son "architecture") doit, pour avoir quelque chance de résister aux inévitables évolutions du projet, être déduite non pas des tâches qu'effectue le système mais des **structures de données** qu'il manipule.

Par rapport à la méthode procédurale classique il y a donc un renversement que l'on pourrait dire copernicien : au lieu de décomposer en fonctions et d'attacher chaque classe d'objets aux fonctions qui l'utilisent, on décompose le système autour de ces classes d'objets et l'on attache chaque élément fonctionnel à l'une d'elles. Bien entendu, les classes d'objets en question reprennent au niveau de la construction les catégories qui, au niveau de la spécification, peuvent être décrites par des types abstraits.

Ces principes nous semblent aussi opposer assez nettement la conception par objets aux méthodes qui fondent la construction des programmes sur l'analyse "du" résultat, comme Médée[3] et Spes[4]. On peut certes, grâce à un artifice théorique, considérer qu'une méthode de conception procédurale permet de

[2] L'emploi de l'adjectif "orienté" n'apporte rien.

[4] Claude Pair : *La Construction des Programmes*, in *RAIRO Informatique*, Vol.13, n° 2, pages 113-118, juillet 1979.

[4] Jean-Pierre Finance, Monique Grandbastien, N. Lévy, Alain Quéré, Janine Souquières : *SPES, un système pour spécifier et transformer*, in *Second Colloque de Génie Logiciel*, Nice, juin 1984, AFCET, pages 345-356.

traiter des modules tels que ceux de la conception par objets : il suffit de définir "le" but d'un tel module comme "*calculer l'une quelconque des fonctions de la spécification du type abstrait correspondant*". Mais cette apparente réconciliation nous semble masquer une opposition conceptuelle fondamentale : l'objectif de la construction d'un système est vu, dans un cas, comme étant de réaliser un traitement, dans l'autre, d'implanter un certain nombre de structures de données assurant des services. Cette différence de point de vue est confirmée par l'examen des exemples traités dans les articles sur les méthodes citées : il s'agit dans la plupart des cas de programmes calculant une seule fonction.

Une fois ses principes acquis, la méthode peut déployer toutes ses possibilités, qui sont considérables.

Nous décrirons ces principes en nous appuyant sur les structures linguistiques offertes par Simula et Smalltalk ; mais l'emploi fréquent du terme "langage" ci-après ne doit pas tromper le lecteur : c'est bien d'une méthode qu'il s'agit, même s'il est commode d'en présenter les concepts à l'aide des structures offertes par les langages correspondants. En outre c'est au niveau de la conception, plus encore qu'à celui de la mise en oeuvre, que l'application de ces idées est déterminante.

Il doit donc être bien clair lorsque nous parlons ci-après de programmes et de langages de programmation que ce sont les structures modulaires de Simula, de Smalltalk et des quelques langages comparables qui nous intéressent, non leurs propriétés spécifiques de langages directement exécutables, comme la façon dont ils permettent d'écrire une affectation, une boucle, un appel de procédure etc. Le paradoxe est ici que ces langages de "programmation" sont, pour ce qui compte vraiment (la structure macroscopique), de *plus haut niveau* que la plupart des langages de "conception" proposés aujourd'hui ; la remarque nous en avait été faite par un stagiaire à qui nous avions demandé, sans trop y réfléchir, d'écrire un document de conception en pseudo-code (PDL) pour un programme final devant être réalisé en Simula.

4.2.2 - Objets dynamiques

Tout d'abord, on ne peut véritablement parler de programmation par objets que si les objets en question sont dynamiques. C'est l'une des trouvailles majeures des concepteurs de Simula 67 : avoir compris qu'il fallait combiner les structures modulaires fondées sur la notion d'objet et la liberté pour les programmes de créer *à l'exécution* (comme en Lisp par exemple) toutes les structures de données dont ils ont besoin. Ce principe découle naturellement de la conception même de Simula : si un langage d'abord destiné à la simulation s'est révélé un remarquable langage de programmation général, c'est bien que la programmation n'est rien d'autre que de la simulation généralisée, de la *modélisation opérationnelle* d'une certaine part du monde réel. Or le monde réel est presque toujours dynamique : il est impossible de savoir à l'avance quels sont les objets dont on aura besoin.

A l'étape de mise en oeuvre sur machine, cette notion d'objet dynamique exige une politique appropriée de gestion et de récupération de la mémoire. Il est regrettable de constater que les mises en oeuvre habituelles de Pascal et de C ne comportent pas de ramassage de miettes automatique, et que le problème a été complètement délaissé dans la conception d'Ada (qui relègue les solutions éventuelles dans les "pragmas" facultatifs). Pour les applications d'"informatique système" au sens large (éditeurs, compilateurs, outils de génie logiciel, etc.), qui manipulent nécessairement des structures de données dynamiques, ce problème est essentiel ; il est de ceux qui peuvent conduire à augmenter d'un ordre de grandeur la complexité des programmes (car si le système-support ne résout pas le problème, il faut bien que quelqu'un s'en charge). Quand on constate que des problèmes techniques aussi communs, aussi bien connus sur le plan théorique (les solutions se trouvent dans le volume 1 de Knuth), ne sont pas résolus par les systèmes courants, on ne peut qu'être sceptique vis-à-vis de l'approche essentiellement économique et gestionnaire du génie logiciel (voir les remarques de la fin du "panorama" liminaire, paragraphe 1.10).

4.2.3 - Module ≡ type ≡ classe

La seconde caractéristique essentielle des véritables langages à objets est la fusion des notions de module et de type. D'autres langages modernes (Ada, Modula) offrent à la fois des structures modulaires non strictement procédurales et des types d'objets dynamiques ; Simula et Smalltalk identifient les deux idées. Une *classe* est à la fois la déclaration d'un type (mise en oeuvre d'un type abstrait : structure de données et opérations) et un module du système.

Cette conception de la modularité, qui oblige à identifier chaque module (classe) à un type unique, semble parfois trop contraignante à qui vient d'un autre horizon ; Ada paraît de prime abord plus général, puisqu'un module peut y inclure la définition d'un ou de plusieurs types. Une fois les concepts bien compris, cependant, on découvre tous les bénéfices qu'apportent la simplicité et l'uniformité de la solution Simula, particulièrement en matière de réutilisabilité, de compatibilité et d'extensibilité : comme chaque module est la mise en oeuvre d'un type de données (tout ce type, et rien que ce type), il forme une entité autonome qu'on peut librement combiner avec d'autres.

Contrairement aux structures modulaires d'Ada et de Modula, qui sont essentiellement des moyens syntaxiques de grouper des éléments de programme, sans véritables propriétés sémantiques, les classes des langages à objets sont donc des entités sémantiques à part entière, susceptibles (grâce au caractère dynamique du langage) de s'incarner en autant d'exemplaires que le commandera l'exécution du programme.

4.2.4 - L'héritage

On peut tirer des idées de base de la conception par objets une troisième conséquence majeure qui, bien présentée, semble évidente, mais nous apparaît en fait comme un véritable coup de génie de la part des concepteurs de Simula 67 : c'est la notion d'héritage.

Le principe des structures à héritage découle de la remarque selon laquelle les classes d'objets ne sont pas indépendantes : le plus souvent, au contraire, une nouvelle classe reprend de nombreux éléments d'une ou de plusieurs classes précédemment définies.

Prenons un exemple simple, celui d'une structure de données correspondant à un élément de liste doublement chaînée, avec une certaine information associée à l'élément et un moyen d'accéder à l'élément précédent et au suivant. Tout langage de programmation permet de définir une telle structure de données de façon plus ou moins élégante. Mais cette définition dépend du type de l'information associée aux éléments : pour chaque variante (liste d'entiers, liste de booléens, liste d'articles de telle ou telle nature), il faudra une nouvelle définition de la structure de données et des procédures de manipulation associées.

Il suffit de regarder un programme C ou Pascal de quelque importance (un programme Fortran ferait tout aussi bien l'affaire, mais l'observation en est plus ardue du fait de l'absence de mécanismes adéquats de description de données) pour se persuader de l'ampleur de ce problème : d'une page à l'autre, d'une structure de données ou d'une procédure à la suivante, les ressemblances abondent. Ressemblances et non pas répétitions à l'identique (qui feraient de la réutilisabilité du logiciel un problème facile à résoudre) : on ne cesse de rencontrer telle ou telle déclaration qui rappelle sa voisine *et qui n'est chaque fois ni tout à fait la même, ni tout à fait une autre.*

> Un texte qui n'a pas été repris ici comparait le métier de programmeur non pas à la tâche de Sisyphe ni à celle des Danaïdes mais plutôt (il avait dû être rédigé au retour d'une réunion Simula à Oslo) à l'oeuvre du peintre norvégien Edward Munch qui, toute sa vie, s'est attaché à peindre et repeindre des variantes d'un petit nombre de tableaux sur des sujets très simples, toujours les mêmes. (le cri, l'angoisse, la passion...), chaque nouvelle incarnation ne se distinguant des précédentes que par quelques détails.

Certains langages de programmation proposent de timides éléments de réponse à ce problème majeur : on peut citer les déclarations d'articles avec variantes en Pascal, qui présentent le grave inconvénient d'être *fermées* (il faut avoir prévu à l'avance toutes les variantes possibles), et les procédures génériques d'Algol 68 et d'Ada, qui permettent d'utiliser le même nom pour des opérations s'appliquant à des types différents (ce qui traite seulement un aspect superficiel du problème puisque seuls les noms sont réutilisés, non le code).

La solution de Simula 67, reprise par Smalltalk, est lumineuse dans sa simplicité : on peut déclarer une nouvelle classe comme *sous-classe* d'une classe existante, ce qui lui permet d'**hériter** automatiquement de toutes les propriétés de cette classe, tout en restant libre de leur ajouter toutes qui lui sont spécifiques. Plus précisément, on écrira quelque chose comme (nous adaptons et francisons la syntaxe pour la circonstance) :

classe *A* **début**

... *attributs et propriétés des objets de classe A* ...

fin ;

classe *B* **reprend** *A* **début**

... *attributs et propriétés des objets de classe B* ...

fin

avec pour résultat que tout objet de classe *B* a les attributs et propriétés qui figurent dans sa déclaration, augmentés de celles des objets de classe *A*. Si par exemple *A* décrit des éléments de listes doublement chaînées, et *B* des structures de données quelconques (disons pour fixer les idées des objets graphiques à visualiser), les objets de la classe *B* pourront automatiquement être mis en listes doubles sans que la déclaration des propriétés spécifiques des listes doubles vienne polluer la description de *B*.

Bien entendu, ce mécanisme n'est pas limité à un niveau : une nouvelle classe *C* pourrait être déclarée comme sous-classe de *B*, héritant alors des propriétés de sa mère *B* et de sa grand-mère *A*. Par opposition à la technique des articles avec variantes, il est **ouvert** : une fois *A* déclarée, on peut toujours lui donner de nouvelles sous-classes sans changer sa déclaration initiale. Enfin son corollaire naturel (ou qui du moins paraît tel lorsqu'on l'a compris!) est la notion de **déclaration virtuelle** : une procédure peut être déclarée comme applicable à tout objet de classe *A*, sa réalisation pratique dépendant de la sous-classe de *A* à laquelle appartient son argument effectif. *A* pourrait être ainsi (pour changer d'exemple) la classe "figure plane", à laquelle est associée la procédure virtuelle "rotation" ; la façon précise dont s'effectue une rotation, et donc le corps de la procédure, seront différents selon la sous-classe de *A* considérée (segment, cercle, polygone etc.).

> On voit ici comment l'identité **module ≡ type ≡ classe** permet à Simula de résoudre avec une seule notion primitive puissante, l'héritage, un problème auquel Ada, par exemple, consacre trois caractéristiques orthogonales : les articles avec variantes (pour le sous-problème des types de données différents mais possédant des éléments communs), la généricité (qui correspond dans une certaine mesure à la notion de procédure virtuelle) et les clauses de visibilité **use** (pour ce qui est de permettre à un module d'accéder directement à des objets définis dans un autre module).

Seule une minorité d'informaticiens peuvent accéder aujourd'hui à ce genre de techniques : ceux qui disposent de Simula, de Smalltalk, de C++, d'Objective-C et de quelques autres systèmes de ce type. Et ce ne sont pas les succès croissants de C, de Modula ou d'Ada (ni d'ailleurs ceux de Prolog ou de FP) qui vont améliorer les choses. Or la seule question qui nous semble se poser ici est : pourquoi tout le monde ne conçoit-il pas et ne programme-t-il pas ainsi?

4.2.5 - De la théorie à la pratique

Les principes précédents ne sont pas faciles à enseigner. Dans un premier temps, les idées de base paraissent évidentes ; mais lorsqu'on cherche à les faire appliquer, elles se heurtent bien vite à la tradition "procédurale" ancrée dans la pratique des programmeurs d'aujourd'hui. (Paradoxalement, elles se heurtent aussi à la mode suscitée par Smalltalk, et à l'assimilation étrange et réductrice qu'opèrent de nombreux novices entre la programmation par objets et la simple utilisation d'interfaces graphiques avec écran de haute résolution et souris). Pourtant, la méthode des objets nous paraît la plus apte à produire des programmes réutilisables et modifiables.

Le texte [85d] présente les arguments qui justifient le recours à cette méthode. Notre prise de position sans nuances ne résulte pas seulement d'une réflexion abstraite, résumée plus haut ; elle a été confirmée par l'expérience, et particulièrement par les éléments suivants :

- l'observation, au fil des années, de nombreux programmes industriels d'une certaine taille, dont l'*objet* officiel n'a cessé de changer, mais dont l'identité restait assurée, à travers leurs avatars, par la permanence des principales classes de données manipulées ;
- l'impossibilité que nous avons constatée de caractériser un logiciel de quelque importance, sauf dans quelques cas de plus en plus rares aujourd'hui, par une fonction unique : presque tous les logiciels auxquels nous avons eu affaire, comme utilisateur ou comme concepteur, sont, plutôt que de simples transformateurs *Entrée* → *Sortie*, de véritables **systèmes** capables d'assurer un ensemble de services relatifs à un ensemble d'objets (selon les termes utilisés dans le texte [85d] pour montrer les limites de la démarche procédurale descendante, *Real systems have no top*) ;
- le succès partiel, mais aussi les difficultés, de l'application d'une partie des principes les plus simples à la réalisation de différents progiciels en Fortran, avec un mécanisme d'allocation dynamique mis en oeuvre en langage d'assemblage, selon des méthodes décrites dans l'article *Principles of Package Design* [82b].
- enfin, la pratique d'un langage à objets, Simula 67, et la constatation concrète des avantages considérables de la méthode : facilité de conception, de relecture, de correction, d'extension ; on trouvera une discussion de l'application de Simula à la méthode des objets dans [85d] et [79b].

Notre expérience de Simula à EDF a bénéficié de la qualité du compilateur développé par le Norsk Regnesentral d'Oslo pour les systèmes IBM-MVS. Malheureusement, Simula n'est pas disponible partout ; Smalltalk, pour sa part, reste dans une large mesure un outil de recherche disponible seulement sur quelques environnements avancés.

Le problème s'est donc posé, au moment de mettre en place le développement d'un éditeur structurel de qualité industriel fondé sur les idées du prototype de Cépage (chapitre 11), de nous frayer un chemin menant aux merveilles de la programmation par objets. Nous nous orientons actuellement vers l'utilisation d'un langage défini pour la circonstance et appelé provisoirement Ilse, qui sera traduit en C (pour des raisons de portabilité) par un pré-processeur. Ilse hérite (ce terme est particulièrement bien venu) des propriétés essentielles de Simula ; mais ce nouveau langage inclut aussi quelques propriétés supplémentaires :

- Simula et Smalltalk ne permettent que l'héritage linéaire : on ne peut être fille que d'une seule mère. Cette clause limite parfois les avantages de la méthode : si l'on a déclaré par exemple deux classes, décrivant l'une la notion de liste double et l'autre la notion de figure plane, le mécanisme ne permet pas en principe de déclarer une nouvelle classe qui est à la fois liste et figure (en Simula, cependant, cette impossibilité peut être tournée assez facilement au prix d'une petite astuce). L'héritage multiple nous paraît indispensable à une véritable politique de réutilisabilité systématique et sera donc inclus dans Ilse. Le prix à payer n'est pas exorbitant : le graphe d'héritage à parcourir n'est plus nécessairement un arbre mais un dag (graphe acyclique orienté).

- L'héritage à la mode Simula-Smalltalk n'est pas sélectif : on hérite de tout - meubles, immeubles et dettes. Ceci oblige à un effort méthodologique supplémentaire pour mettre en oeuvre certains des principes de modularité définis plus haut (limitation des interfaces, opacité). En Ilse, si l'on est fille d'une ou plusieurs sous-classes, on n'en recevra que les legs explicitement demandés : il faut nommer ce qu'on veut récupérer.
- Ilse, langage pour la construction de programmes selon les règles du génie logiciel, comprend aussi d'autres éléments, par exemple des notations pour les structures de contrôle encourageant l'inclusion explicite d'assertions et d'invariants de boucle (cf. le chapitre 6 ci-après et les notations du texte [85c]).

CHAPITRE 5

SPECIFICATION

5.1 - DEFINITION

La spécification d'un système informatique est la description précise des objets qu'il manipule et des fonctions qu'il assure.

Plusieurs raisons expliquent le rôle central que joue cette notion dans les recherches de génie logiciel.

• 1 - Du point de vue *chronologique*, la spécification est l'une des étapes du "cycle de vie" des projets logiciels, celle qui permet le passage d'une expression approximative des besoins à une définition précise du rôle du système projeté. Tous les auteurs s'accordent pour reconnaître l'importance technique et économique de cette étape.

• 2 - Du point de vue *méthodologique*, la spécification a pour objet de définir le but d'un système avant le passage à la réalisation. Ce problème se pose inévitablement à la racine de toute méthode de développement, puisqu'il faut bien partir d'une description de la tâche à résoudre avant de proposer une solution. La difficulté de la spécification vient des impératifs contradictoires qui doivent être conciliés : être précis et complet sans pour autant surspécifier (c'est-à-dire sans remplacer l'énoncé des caractéristiques externes du système par la description d'une réalisation particulière) ; être rigoureux tout en restant clair ; servir les informaticiens sans exclure les utilisateurs.

• 3 - Du point de vue de la *validité* du logiciel, l'existence d'une spécification précise est une condition indispensable à toute tentative de démonstration de programme.

• 4 - Du point de vue de la *réutilisabilité* et de la **compatibilité**du logiciel, il est nécessaire de disposer d'une description exacte de tout élément logiciel si l'on veut pouvoir l'utiliser un grand nombre de fois dans des circonstances variées.

• 5 - Du point de vue voisin de la **compatibilité** des éléments logiciels, on ne peut combiner sans risque des éléments développés indépendamment que si l'on est certain qu'ils ne font pas d'hypothèses contradictoires sur leur environnement, ce qui, à nouveau, exige une spécification précise.

• 6 - De la même façon, le facteur de qualité que nous avons appelé *extensibilité* au chapitre 1 (facilité de modification) implique que l'on connaisse précisément le rôle de chaque élément pour pouvoir le modifier dans de bonnes conditions.

• 7 - Du point de vue *théorique*, enfin, la tâche de spécification rejoint les travaux visant à fournir des modèles mathématiques adéquats pour les concepts logiciels, pour les structures de données par exemple.

5.2 - LA DESCRIPTION DES STRUCTURES DE DONNEES

La première publication reproduite ci-après sur le thème de la spécification [76a] est précisément intitulée *Description des Structures de Données*. Ce travail, directement suscité par le premier article de Liskov et Zilles sur les types abstraits (1974), était par contre indépendant de la thèse de J. Guttag.

L'article, dont les principales idées sont reprises et appliquées à de nombreuses structures de données dans le chapitre 5 des *Méthodes de Programmation* [78a], proposait de séparer complètement, dans la description des structures de données, les aspects "physiques", liés à la représentation, des aspects "fonctionnels", liés à l'utilisation.

La méthode proposée distinguait en fait non pas deux, mais trois niveaux de description : les deux niveaux extrêmes, *spécification fonctionnelle* et *représentation physique*, mais aussi un niveau intermédaire appelé *description logique*.

La spécification fonctionnelle correspond à ce qui est couramment appelé aujourd'hui "spécification par des types abstraits algébriques". Il s'agit de fournir une vue purement externe de la structure de données considérée, caractérisée seulement par la liste des *fonctions* disponibles et celle des *propriétés* abstraites de ces fonctions.

A l'autre bout, la représentation physique est la mise en oeuvre de la structure et des fonctions correspondantes sur un support matériel donné (mémoires, processeurs).

Spécification fonctionnelle et représentation physique s'opposent en fait de deux façons différentes. Non seulement la première est abstraite et la seconde tout à fait concrète ; mais il faut aussi noter que l'une repose sur une description *implicite* d'une structure de données, caractérisée uniquement par ses propriétés (ce qu'elle *a*), là où la seconde est parfaitement explicite (la structure étant donnée pour ce qu'elle *est*).

Cette remarque explique pourquoi nous avons jugé bon d'introduire le niveau intermédiaire, celui de la description "logique". Il s'agit d'une description à la fois abstraite et explicite, qu'on pourrait aussi caractériser comme une *implantation mathématique*. Le niveau de la description logique est aussi celui de certaines méthodes de spécification comme VDM, où les objets sont décrits de façon abstraite mais constructive.

Si le terme "logique" est assez mal choisi, comme nous l'avait fait remarquer J.-C. Boussard en 1977 ("constructif" aurait sans doute été préférable), la notion nous paraît importante. Elle sert en effet à préserver par contraste le caractère implicite des spécifications fonctionnelles, qui nous semble essentiel.

Les différents travaux sur la spécification repris ci-après ont tous en effet pour propriété de ne jamais chercher à dire d'abord ce que "sont" les objets considérés, ni même d'en proposer des modèles (comme le font la plupart des théories actuelles des types abstraits algébriques). A la même réunion GROPLAN de juin 1977, où l'article [76a] avait été présenté, B. Robinet nous en avait fait le reproche : comment peut-on savoir si les spécifications données décrivent vraiment quelque chose?

A notre sens, ce problème peut être négligé à l'étape de spécification, dont le but n'est pas de construire mais de décrire. Le rôle de la description logique est précisément de faciliter l'étape ultérieure, celle où l'on passe d'une spécification implicite à une réalisation effective, en proposant un moyen terme, abstrait mais déjà constructif.

> Un article récent sur la méthode M [85g] reprend l'idée émise dans une brève communication de 1980 (*A Three-level Approach...*, cf. liste de publications) selon laquelle le passage de la spécification à la conception peut précisément être caractérisé comme un passage de l'implicite à l'explicite : chaque type abstrait peut alors être représenté comme le produit cartésien de ses attributs.

5.3 - Z, 1977

Les étapes suivantes en matière de spécification sont marquées par l'influence des travaux d'Abrial, tout d'abord à travers ce qu'on peut appeler la version 1977 du langage Z. Décrite à travers une série d'articles d'Abrial jamais vraiment publiés, cette version (correspondant à ce qui est décrit dans le chapitre 8 de *Méthodes de Programmation* [78c]) se caractérise par les éléments suivants :

- distinction entre un langage de spécification implicite, appelé Z0, et un langage de conception, Z1, assez proche de langages de programmation de très haut niveau comme SETL ;
- dans Z0, utilisation d'un petit nombre de concepts directement empruntés à la théorie élémentaire des ensembles (ensembles, relations, fonctions, prédicats), avec une notation très simple ;

- méthode "transformationnelle", visant à fournir des schémas de traduction de Z0 vers Z1 (pour remplacer par exemple une quantification universelle sur un ensemble fini par une boucle, etc.).

Ce cadre très simple avait le mérite de s'enseigner facilement ; il a permis de convaincre un grand nombre de personnes de l'intérêt des spécifications formelles.

L'un des articles reproduits ci-après, la synthèse de 1979 sur les langages de spécification [79a], écrite avec Michel Demuynck, est directement sous l'influence de Z 1977. L'une des caractéristiques de ce travail, qui (vu en 1985) lui donne un caractère quelque peu superficiel mais aussi une certaine originalité, est la volonté de comparer des méthodes "industrielles" de spécification telles que SREM, SADT, ISDOS ou les réseaux de Petri avec des méthodes formelles telles que Z. Ce mélange des genres nous a été reproché à propos de cet article et d'autres et, de fait, les deux écoles ne communiquent guère ; mais nous continuons de penser que la combinaison d'une clarté à la SADT et d'une précision telle que celle de Z est souhaitable.

5.4 - Z, 1979-1980

Face à certaines limitations du cadre Z0-Z1, J.-R. Abrial et S.A. Schuman se sont orientés, à partir de 1978, vers un formalisme à la fois plus rigoureux quant à sa syntaxe et plus ambitieux (l'influence du développement d'Ada, contemporain de ce travail, est évidente). Le langage résultant est un langage de spécification pur : les travaux sur la transformation avaient été provisoirement interrompus après que le catalogue des transformations nécessaires pour rendre compte des processus les plus élémentaires de construction de programmes était apparu démesurément long. Le nouveau langage possède les caractéristiques suivantes :

- syntaxe définie précisément, sur le modèle de celles des langages de programmation de la série Algol (en fait, de style Ada très net) ;
- primitives de construction modulaire de spécifications ("chapitre", rappelant le paquetage d'Ada) ;
- base mathématique simple, toujours fondée sur la théorie des ensembles (à l'exception des classes, cf. ci-après) ;
- notion de classe avec préfixation, directement reprise de Simula, enrichie par la possibilité de filiation non linéaire et de composition ascendante ;
- rôle important des définitions génériques (là encore très proches de ce que propose Ada) ;
- possibilité, largement utilisée dans le document introductif[1], d'écrire des spécifications très abstraites, avec des fonctions de haut niveau en un style voisin de celui de la programmation en FP ;
- langage à structure "en couches", avec un noyau en principe très réduit, des extensions syntaxiques et une série de "chapitres de base" qui présentent de nombreux concepts fort utiles en pratique (suites, synchronisation de processus, etc.) mais pouvant être décrits en termes des couches précédentes.

> C'est cette version de Z qu'utilisait l'article *Sur le Formalisme dans les Spécifications* publié en 1980 dans le bulletin Globule. L'utilisation d'un formalisme spécifique nuisait en fait à l'article. Aussi avons-nous préféré reproduire ci-après l'adaptation anglaise récente de cet article [85a], qui s'appuie sur une notation mathématique commune plutôt que sur un langage de spécification particulier.

Le destin de la version 1980 de Z a été assez curieux. Il s'agissait d'une très belle construction intellectuelle, supérieure à notre sens par sa portée et sa simplicité à d'autres développements contemporains (comme CLEAR), mais dont l'application directe s'est limitée à

---

[1] J.-R. Abrial, S.A. Schuman, B. Meyer. *A Specification Language*. in *On the Construction of Programs*. Cambridge University Press, 1980. Une description plus systématique est donnée dans *The Specification Language Z : Syntax and "Semantics"*, par J.-R. Abrial, Oxford University, Programming Research Group, avril 1980.

quelques exemples ; Z 1977 a eu, et (semble-t-il) continue d'avoir plus d'utilisateurs. L'influence **indirecte** de Z 1980 a été considérable, particulièrement en Grande-Bretagne, à travers les travaux de l'Ecole d'Oxford et de sa collaboration avec IBM. Il est significatif cependant de constater que le "Z" de Sufrin et Sorensen est plus modeste d'apparence que l'imposant langage décrit par les documents de 1980-81 : il s'agit pour l'essentiel d'annotations formelles introduites ici et là dans des descriptions de systèmes en langue naturelle.

5.5 - M

Le dernier article sur la spécification reproduit ici est récent [85g]. Il présente des réflexions en cours sur une méthode de spécification appelée M.

Nous avons essayé de présenter aussi clairement que possible dans l'article lui-même les principes qui ont conduit à la mise au point de cette méthode et nous ne répéterons donc pas cette discussion ; nous nous contenterons de quelques remarques sur le contexte de cette recherche et particulièrement sur les différences entre M et Z.

- Nous nous sommes efforcé de prendre en compte la remarque (émise en particulier par C. Pair) selon laquelle Z était trop "langage" et pas assez "méthode". M se veut bien une méthode de spécification et non pas seulement une notation (même s'il est difficile d'éviter les questions de langage).
- L'accent a été mis sur les techniques de spécification modulaire et itérative, qui nous paraissent essentielles pour le succès pratique des spécifications formelles. Les techniques de décomposition et de réutilisation de spécifications procèdent évidemment de l'analyse contenue dans [85f] sur les relations entre éléments de logiciel, citées au chapitre 3 de cette thèse.
- Par rapport à Z, qui est essentiellement une notation pour la logique et la théorie des ensembles, M est plus ouvertement destiné à la spécification des les problèmes informatiques. Le substrat mathématique reste très proche de la surface (et affleure par endroits), mais les notations correspondent aux concepts utilisés pour la description de systèmes logiciels ; par exemple, les fonctions sont explicitement divisées en "attributs" et "transformateurs". (Il est intéressant de noter que partant du même point de départ, Z 1979-80, l'équipe d'Oxford aboutit à un résultat opposé et "re-mathématise" la notation ; comme l'écrit gentiment C.A.R. Hoare : *We have made a lot of efforts to avoid the kind of formal clutter which M does not escape from*).
- Il nous paraît indispensable de pouvoir disposer d'outils pour l'application courante de la méthode à des systèmes importants.
- M est une méthode de description plus que de construction, malgré la présence de techniques pour guider le passage d'une spécification à une architecture de système (chapitre 7 de [85g]) et à une mise en oeuvre. Ce souci de spécification presque pure peut paraître schizophrénique. Il nous semble cependant nécessaire de disposer de techniques et de notations adéquates pour la simple description des problèmes, avant toute velléité de résolution.

CHAPITRE 6

CONSTRUCTION SYSTEMATIQUE

6.1 - LES ARTICLES

Une série de travaux a porté sur l'application systématique des méthodes de démonstration de programmes, issues des articles de Hoare et Dijkstra, à la construction de programmes. Il s'agit pour l'essentiel de variations sur les idées contenues dans le livre de Dijkstra, *A Discipline of Programming* (Prentice-Hall, 1976). Le problème traité est presque toujours de synthétiser une boucle à partir de son invariant. Les divers textes sur ce sujet résultent pour une large part de discussions avec A. Bossavit et de travail en commun sur des exemples empruntés aux algorithmes numériques et au calcul vectoriel.

La méthode avait été utilisée de façon déjà assez systématique dans le livre *Méthodes de Programmation*, à la fois au chapitre 3 ("Structures de Contrôle") et, dans les chapitres suivants, pour présenter de nombreux algorithmes (recherche dichotomique, quicksort, heapsort, etc.)

L'article [80a] cherche à fournir une base formelle à la méthode. Il utilise la notation Z (1980), qui rend la lecture assez lourde mais permet (grâce aux classes) de montrer les différentes formes de programmes commes cas particuliers d'une même structure générale. La définition générale d'un programme (fonction) répondant à une spécification (relation) est donnée sous la forme

program $\subseteq$ *problem*

ce qui n'est pas assez fort puisque la fonction vide satisfait, selon cette définition, à n'importe quelle spécification. Il faut ajouter que **domaine** *(program)* = **domaine** *(problem)*. Cette correction a été faite dans la présentation des mêmes concepts donnée dans l'article [85a] (bas de la page 16)[1]. Une forme correcte est donnée ci-après.

La dernière partie (4) de l'article [80a] étudiait quelques stratégies d'affinage des spécifications en vue d'obtenir des invariants : plongement (*embedding*), relaxation de constantes et découplage. Toutes sont des heuristiques générales d'affaiblissement de postconditions. Nous avons utilisé ces idées à nouveau dans le chapitre 3 du livre *Applied Programming Methodology* en préparation [85c], et dans des notes de cours plus élémentaires (*An Introduction to the Art of Writing Correct Programs*, rapport UCSB TRCS-8403, 1984).

Les articles [81a], [85b] et [85e] présentent des applications de ces idées. Il s'agit dans chaque cas de descriptions, soigneusement justifiées par des invariants, d'algorithmes en général assez délicats : factorisation vectorielle de Choleski dans [81a], visualisation sur écran d'un arbre syntaxique abstrait dans [85b] et recherche incrémentale de patrons dans un texte dans [85e].

[1] Cette propriété ne nous porte pas chance puisque sa seconde formulation dans le même article, en haut de la page 17, est légèrement inexacte. Une rectification est parue (juillet 1985).

6.2 - LES PRINCIPES

L'idée de base de ces travaux est de considérer les différentes structures de contrôle, et tout d'abord les structures séquentielles classiques (enchaînement, choix, boucle) comme des techniques de **résolution de problèmes** adaptées chacune à une certaine classe de questions.

On part donc d'une spécification et, de la forme de cette spécification, l'on déduit (dans les cas favorables) une structure de programme. Les spécifications considérées ici sont plus limitées que celles qui peuvent être exprimées avec les formalismes généraux du chapitre précédent (Z, M) : il s'agit essentiellement de considérer des problèmes définis par une transformation simple (*données* → *résultats*). La spécification est alors une relation binaire r ; un programme satisfaisant à cette spécification sera une fonction f, construite à partir d'un jeu de fonctions de base qu'on sait calculables, et telle que

domaine $(f) \supseteq$ **domaine** (r) **et** $f \mid$ **domaine** $(r) \subseteq r$

où le symbole $\mid$ désigne la restriction d'une fonction à un sous-ensemble de son domaine ; en d'autres termes, f calcule un résultat compatible avec r partout où il en existe au moins un. (On peut aussi exprimer cette propriété sous la forme plus concise **domaine** (r) = **domaine** $(r \cap f)$

Dans ce cadre général (restrictif mais permettant cependant de rendre compte de nombreux problèmes de base), la méthode essaye de rattacher toute spécification rencontrée en pratique, c'est-à-dire toute relation r particulière, à l'un des schémas pour lesquels une forme de solution, c'est-à-dire de fonction f, est connue.

Les trois structures de contrôle fondamentales correspondent à de tels schémas. Par exemple, l'**enchaînement** correspond à la composition, c'est-à-dire au cas où la relation r est de la forme

$$r = t_m \circ t_{m-1} \circ \ldots\ldots \circ t_1$$

et où les problèmes représentés par les relations t_1, ..., t_{m-1}, t_m paraissent plus simples à résoudre que le problème initial représenté par r.

Le **choix** (à n branches) correspond au cas où la relation r peut s'exprimer simplement comme l'union ensembliste de n relations qui, là encore, paraissent individuellement susceptibles de solutions plus simples :

$$r = r_1 \cup r_2 \cup \ldots\ldots \cup r_n$$

Si cette union n'est pas disjointe, on obtient le choix non déterministe introduit par Dijkstra à l'aide des commandes gardées.

La **boucle**, enfin, répond (on ne s'en étonnera guère) à un schéma un peu plus compliqué. Cette méthode de résolution de problèmes s'applique lorsque la relation r peut être mise sous la forme

$$r = inv \cap sortie$$

où les deux composantes ont un rôle différent (il peut être utile de suivre cette discussion sur la figure 3.5, page 27, de la référence [85c]). La relation *inv* (pour "**invariant**") doit correspondre à un problème assez "facile" pour qu'il existe une fonction d'initialisation *init* permettant de l'établir directement ; le problème associé à *inv* doit tout au moins être notablement plus facile que le problème initial r. Mais par ailleurs la relation *inv* doit être assez forte ("difficile") pour qu'il existe une fonction *trans* qui maintient *inv* tout en rapprochant l'état courant d'un état où la relation *sortie* est aussi satisfaite.

Ce rapprochement peut être mesuré à l'aide d'une quantité appelée le **variant** associé à la boucle. Si les conditions nécessaires sont réunies, une fonction solution de r sera de la forme

$$boucle = trans^* \circ init$$

où * représente la fermeture réflexive transitive. Cette solution correspond à la notion usuelle de boucle ; la convergence au sens informatique exige que le variant appartienne à un ensemble bien fondé.

Plusieurs exemples de synthèse de boucles sont développés dans [80a] et [85c]. La démarche générale est toujours la même : partant de la relation décrivant le résultat souhaité, on cherche un invariant de boucle par **affaiblissement** de cette relation.

Diverses stratégies d'affaiblissement sont étudiées ; là encore, des figures, pages 33 et *passim* de [85c], cherchent à suggérer (là s'arrête leur ambition) les idées sous-jacentes :

- la **relaxation de constantes** consiste à remplacer une relation qu'exprime un prédicat de la forme $P(n)$, où n est un paramètre constant pour chaque cas du problème, par la forme équivalente $P(x)$ **et** $(x = n)$, où x est une nouvelle variable introduite pour la circonstance. Le premier opérande du **et** servira d'invariant et le second de condition de sortie. Cette forme correspond par exemple aux boucles de type "**pour**".
- Le **découplage** consiste à remplacer une relation qu'exprime un prédicat de la forme $P(n, n)$, où n intervient plus d'une fois, par la forme équivalente $P(m, n)$ **et** $(m = n)$, et à prendre là encore le premier opérande comme invariant. L'idée est qu'il sera plus facile d'assurer l'initialisation si les deux apparitions de n dans P sont "découplées", et que l'on pourra ensuite travailler itérativement à rapprocher m de n tout en maintenant invariante la propriété $P(m, n)$.
- Le **plongement** consiste à remplacer un problème de la forme $P(x, y)$ où y, la solution, doit appartenir à un certain espace E, par la forme équivalente $P(x, y)$ **et** $y \in E$, où y est redéfini comme appartenant à un certain sur-ensemble E' de E ; on espère ici qu'il sera plus facile initialement de trouver un y satisfaisant à P dans E' que dans E, et que l'on "convergera" ensuite vers le sous-espace E tout en maintenant la validité de P généralisée, qui sert d'invariant. On notera que les stratégies précédentes peuvent s'exprimer comme cas particuliers de celle-ci.

Ces stratégies sont complétées par quelques variantes, mais les schémas de base restent toujours assez proches.

6.3 - PERSPECTIVES

A défaut d'avoir permis l'invention d'algorithmes révolutionnaires (mais il n'est jamais interdit d'espérer), les principes précédents offrent tout au moins un indéniable intérêt pédagogique, en permettant de de présenter sous un jour original la notion de structure de contrôle, d'inclure dans la présentation des algorithmes la démonstration de leur validité, d'introduire dans un cadre simple les méthodes élémentaires de construction de programmes à partir de spécifications formelles, et d'éveiller l'intérêt des praticiens pour l'étude de la sémantique des langages.

De nombreuses extensions restent évidemment nécessaires : application à des systèmes qui ne sont pas seulement de naïves transformations de données à résultats (cf. chapitre 4) ; prise en compte de structures de données évoluées et du parallélisme. Il est en outre indispensable de prendre en compte, non pas seulement l'héritage des travaux de Dijkstra et de son école, mais aussi tous les travaux sur la construction de programmes (Sintzoff, Pair, Finance, Anna Gram, Manna-Dershowitz) et l'apport des systèmes de synthèse de programmes issus de l'intelligence artificielle (Barstow, Green, Low).

On notera dans le chapitre 7 de l'article sur la méthode M [85g] une première approche du problème de la construction de programmes vu au niveau macroscopique : il s'agit de partir de la spécification modulaire d'un système, obtenue par application de la méthode, et d'en déduire la structure du programme correspondant et de ses structures de données. L'aspect algorithmique est par contre laissé dans l'ombre : il faut pouvoir combiner cette méthode pour la construction systématique de ce qu'on peut appeler le *gros oeuvre* d'un programme avec les techniques de "remplissage des détails" (détails non nécessairement triviaux, bien sûr) qui ont été esquissées dans ce chapitre.

CHAPITRE 7

LANGAGES

Pendant longtemps, les discussions sur les langages ont constitué l'essentiel des recherches en programmation. Le retour de bâton était inévitable : si les années soixante peuvent être caractérisées comme l'époque des langages, les années soixante-dix ont été celle des méthodes et la décennie 80 semble devoir être celle des outils.

Les langages ne sont pas toute la programmation ; mais cette évidence ne doit pas conduire à négliger leur rôle. Il nous paraît vain, par exemple, de prétendre (comme on l'entend assez souvent) que le choix du langage importe peu et que ce qui compte, c'est d'utiliser une bonne méthode ou (selon l'interlocuteur) de bons outils.

Nous pouvons nous appuyer, pour réfuter ces affirmations, sur l'expérience acquise à EDF, où nous avons précisément cherché (pendant près de dix ans) à permettre à un public industriel de tirer profit des progrès de la science de la programmation tout en se débrouillant avec les moyens du bord, c'est-à-dire avec Fortran (et avec IBM-MVS-TSO-SPF). C'est dans cet esprit qu'a été mis au point (avec Gérard Brisson) le programme des cours de logiciel à la Direction des Etudes et Recherches ; c'est lui aussi qui a guidé la rédaction de nombreux rapports internes EDF, des chroniques du Bulletin du centre de calcul ("J'ai même rencontré des programmeurs heureux"), et de l'ouvrage *Méthodes de Programmation* (en collaboration avec C. Baudoin) qui présentait en détail l'application des méthodes modernes à la programmation en Fortran, PL/I et Algol W.

Le bilan de cette tentative est clair : il est certes recommandé de bien programmer même avec un mauvais langage, mais c'est encore mieux si l'on peut bien programmer dans un bon langage. Les efforts de génie logiciel qui s'appuient sur des langages périmés se heurtent à des limites conceptuelles infranchissables.

Deux des articles ci-après ~~ont~~ traitent directement de langages de programmation. L'article [80c] est une synthèse sur ce sujet ; il contient quelques tentatives de classification systématique (sur la structure des langages, sur les critères de conception, sur les styles syntaxiques, sur l'histoire des langages, sur la notion de généralité). L'article [79b] analyse Simula 67 en détail, montrant l'application des principes de programmation par objets, en particulier de la préfixation et des procédures virtuelles.

CHAPITRE 8

OUTILS

Méthodes, outils, langages : le triptyque a déjà été cité. Nos efforts sur les outils ont débuté en 1976 avec la mise en place d'une structure appelée "Atelier logiciel" à la Direction des Etudes et Recherches d'EDF[1]. De nombreux outils (et plus de de quarante notes) ont été réalisés par l'équipe de l'Atelier Logiciel, allant de simples utilitaires pour le traitement des fichiers sur IBM à un système de construction, manipulation et visualisation de programmes, Cépage (dernier chapitre), en passant par de nombreux "progiciels" :

- Ensorcelé, ensemble de procédures d'entrée et des sortie ;
- Chronos, pour la mesure de temps ;
- Errare, outil de traitement des erreurs ;
- Gescran, progiciel de gestion d'écrans, complété par un système interactif de fabrication d'écrans, Conscran, écrit en Simula.

L'expérience apportée par la spécification, la réalisation et la mise en oeuvre de ces produits a été rassemblée dans l'article [82b], *Principles of Package Design* (Principes de Conception des Progiciels). Cet article prône une méthode de conception fondée sur l'idée selon laquelle un progiciel est la mise en oeuvre d'un ou plusieurs types abstraits. Il présente des règles précises quant à la conception des primitives offertes aux utilisateurs (nom, nombre d'arguments, choix entre fonctions et procédures) et une méthode de traitement des erreurs.

L'activité de l'équipe de l'Atelier Logiciel en matière d'outils ne s'est pas limitée à la construction d'outils : il est au moins aussi important de savoir trouver et réutiliser des outils de qualité réalisés ailleurs. L'Atelier Logiciel a ainsi introduit à EDF un grand nombre de produits venus de l'extérieur :

- compilateurs et systèmes d'exécution : Algol W, Simula 67, Pascal, Prolog ;
- préprocesseurs : Mortran ;
- systèmes de calcul symbolique : Reduce (avec Lisp en prime) ;
- bibliothèques numériques : IMSL (acquis antérieurement à la création de l'équipe), Linpack, Eispack, NAG (un problème important en logiciel numérique, qu'abordait la réflexion entamée avec A. Bossavit, est d'augmenter l'utilisation pratique de ces bibliothèques) ;
- outils de validation et de documentation : à la suite d'une étude comparative détaillée des principaux outils disponibles, menée avec J.-M. Nerson[2], notre choix s'était porté sur RXVP, qui combine des possibilités d'analyse statique, d'analyse dynamique (mesure du taux de couverture de tests) et de documentation de programmes, et qui a rendu des services appréciables aux utilisateurs du centre de calcul des Etudes et Recherches.

C'est également dans le cadre de l'Atelier Logiciel qu'a été menée (essentiellement par J.-M. Nerson et E. de Drouas) l'étude sur les ateliers logiciels intégrés (généralisation naturelle de la notion d'outil), dont les principaux éléments se trouvent dans la thèse de docteur-ingénieur de J.-M. Nerson[3] et dans un article de synthèse sur les ateliers français[4].

---

[1] Cf. *L'Atelier Logiciel*, Actes des Journées BIGRE 1977 ; également note Atelier Logiciel n° 1, EDF, Direction des Etudes et Recherches, Service IMA.

[2] *Quatre Outils d'Analyse statique*, bulletin GLOBULE no. 4, 1982.

[3] Jean-Marc Nerson : *Panorama des outils d'aide à l'amélioration de la qualité des logiciels*, note Atelier Logiciel n° 40, avril 1983.

[4] E. de Drouas et J.-M. Nerson, *Les Ateliers Logiciels Intégrés : Développements français actuels* ; TSI, Vol. 2, no. 3. L'auteur et J.-M. Nerson ont en chantier un ouvrage sur les *Outils et Ateliers de Génie logiciel*.

CHAPITRE 9

PROGRAMMATION VECTORIELLE

La recommandation du groupe de travail EDF qui avait étudié en 1979-80 les besoins des grands utilisateurs scientifiques était d'acquérir un ordinateur Cray-1. L'installation ne devant avoir lieu qu'au printemps 1981, tout le temps nécessaire nous était laissé pour préparer le terrain et former les utilisateurs.

Les premiers contacts avec la documentation du constructeur nous avaient quelque peu inquiété ; la principale référence alors disponible sur la programmation de cette puissante machine[1] contenait par exemple la la phrase suivante : *Des programmes par trop modulaires ou structurés ne marcheront pas bien sur le Cray...*

Un examen plus attentif avait rapidement montré, cependant, qu'il n'était pas nécessaire d'abandonner tout esprit de méthode pour programmer efficacement un Cray.

La note [80b] (révisée à plusieurs reprises depuis lors, la version incluse ci-après étant celle de janvier 1982) est le résultat de ce travail préliminaire. Il ne s'agit pas d'un article de recherche mais d'un document interne qui présentait le Cray au public EDF. Il cherche à expliquer sous une forme précise et systématique les conditions qui doivent être respectées par un programme Fortran pour qu'il puisse s'exécuter vectoriellement. Ces conditions sont décrites par cinq règles précises ; on notera que les deux règles les plus intéressantes, relatives aux conditions de dépendance, font en annexe l'objet d'une spécification en Z.

Au-delà de cette première clarification, les réflexions sur la vectorisation se sont poursuivis en collaboration avec A. Bossavit (et avec l'aide d'E. de Drouas, R. Butel et M. Farge). Cet effort, continuation d'un travail de plus longue haleine sur les algorithmes numériques, peut être défini comme une tentative de réconciliation du génie logiciel et du calcul sur super-ordinateurs.

La première tâche était d'approfondir la remarque simple selon laquelle le calcul vectoriel est le calcul sur des vecteurs : de façon plus formelle, il convient de définir précisément le type abstrait "vecteur" adapté à la modélisation de ce que sait bien faire un calculateur tel que le Cray.

> La note [80b] contenait une timide allusion sous la forme d'un'un conseil simple : *mettez la boucle dans le sous-programme plutôt que le sous-programme dans la boucle.* L'explication immédiate de cette maxime est qu'elle reflète simplement une des limitations du compilateur Fortran sur le Cray : une boucle ne peut être vectorisée si elle contient appel de sous-programme. Mais son sens profond est qu'il faut décomposer les programmes en sous-programmes correspondant à des opérations sur des vecteurs.

L'article [81a] développe cette notion et présente des techniques d'affinage d'algorithmes permettant d'obtenir des programmes vectoriels efficaces.

Le troisième texte sur le calcul vectoriel [84b] applique à ce domaine une autre technique de génie logiciel : la transformation de programmes. Le cadre formel n'est pas parfaitement rigoureux, mais il permet d'obtenir de façon systématique un algorithme non trivial, dont la justification directe paraît difficile.[2] Nous ignorons s'il existe un système de transformation de programmes capable d'accepter les transformations présentées dans cet article (qui propose également quelques réflexions sur l'utilisation d'Ada en calcul vectoriel).

---

[1] Lee Higbie, *Vectorization and Conversion of Fortran Programs for the Cray-1*, document Cray 2240207, juin 1979.

[2] Une correspondance avec E.W. Dijkstra est éclairante à cet égard : Dijkstra nous propose un invariant pour chacune des deux boucles du programme final, mais aucune justification pour ces invariants quelque peu abscons.

Le travail sur le calcul vectoriel n'est pas terminé : il resterait à effectuer une synthèse systématique des algorithmes de ce domaine. La difficulté de programmer efficacement les machines de la génération du Cray-1 donne un avant-goût des problèmes qui vont se poser lorsque des architectures vraiment parallèles vont devenir largement disponibles.

CHAPITRE 10

SYSTEMES INTERACTIFS

Les publications reproduites ci-après traitent essentiellement de deux aspects de l'interactivité : les *outils* de construction de programmes interactifs et les *méthodes* de construction de programmes interactifs. Quelques éléments de réflexion sont également proposés sur le thème de l'ergonomie des dialogues[1].

L'article [82a] est une sorte de tour d'horizon de l'expérience acquise entre 1980 et 1982 sur ces thèmes. La partie "outils" décrit les premières réalisations de l'Atelier Logiciel en matière de systèmes interactifs : bibliothèque de procédures "AL", progiciel Gescran de gestion d'écrans, avec le "constructeur" associé Conscran.

Ces outils, le premier en particulier, sont quelque peu marqués par l'environnement TSO-3278 dans lequel ils ont été développés, mais leurs principales caractéristiques restent intéressantes, nous semble-t-il, dans un cadre plus large. En matière d'ergonomie, la notion de *profil* associé à chaque utilisateur par le système nous paraît faire particulièrement bon ménage avec l'utilisation de systèmes "pleine page" : en offrant un grand nombre d'options, on satisfait l'utilisateur expérimenté ; en proposant pour chacune d'entre elles, dans la zone d'écran correspondante, une valeur par défaut raisonnable, on évite de dérouter le novice ; en utilisant lors des sessions suivantes, au lieu de la valeur par défaut initiale, le dernier choix de l'utilisateur, on fait plaisir à tout le monde. L'aspect visuel est ici essentiel : il ne suffit pas de connaître le profil de l'utilisateur ; encore faut-il lui montrer ce qu'on croit savoir de ses goûts.

Le concept fondamental de Gescran (utilisation d'un type abstrait de base,"fenêtre") nous paraît toujours valable et nous réfléchissons actuellement à un nouveau produit fondé sur les mêmes idées mais beaucoup plus souple et général ; il s'adaptera à une classe très large de terminaux, y compris des terminaux matriciels (*bit-mapped*) à possibilités graphiques.

L'éditeur structurel et visuel Cépage, décrit au chapitre suivant, reprend les principaux éléments de notre expérience des systèmes interactifs.

[1] Voir également sur ce thème l'analyse bibliographique intitulée *Le Facteur humain a sonné trois fois* et parue dans TSI, vol. 3, no. 3.

CHAPITRE 11

CEPAGE

Nous terminerons cette revue par la présentation d'un outil plus ambitieux que les précédents : Cépage, un éditeur structurel qui se veut le noyau d'un environnement de programmation.

Le but du premier projet Cépage (à EDF en 1983-84, en collaboration avec Jean-Marc Nerson) était d'étudier comment la notion d'éditeur structurel, bien acceptée dans les milieux de la recherche, pouvait se transposer dans un environnement de production. Il s'agissait cependant de réaliser non pas un produit, mais un prototype.

Deux critères avaient joué un rôle important dans les réflexions initiales :

- l'importance de l'interface : il était nécessaire d'utiliser au mieux les terminaux modernes ;
- la nécessité de pouvoir s'adapter à des langages très divers, et à des variantes locales de langages.

L'effort principal a porté sur ces deux points. La principale innovation technique de Cépage est l'algorithme de visualisation [85b] (perfectionné en 1984 à Santa Barbara avec l'aide d'une étudiante, Soon Hae Ko), qui cherche à utiliser au mieux la taille disponible sur l'écran (ou la fenêtre). L'indépendance vis-à-vis des langages est assurée par l'utilisation d'une structure de données interprétée, le "graphe de grammaire", pouvant être changée facilement.

Les réflexions actuelles autour de Cépage tendent à en faire un produit véritablement industriel et à en étendre les fonctionnalités : possibilité d'analyse syntaxique, à la fois pour offrir plus de souplesse dans la construction interactive de documents et pour permettre la réutilisation de documents existants ; définition complète d'un "langage de description de langages" (LDL) facilitant la prise en compte de nouveaux formalismes ; possibilités de contrôle sémantique ; possibilité d'exécution de programmes même non complètement affinés, offrant une aide à la mise au point, aux tests et au prototypage rapide ; lien avec le projet SKB pour les aspects de gestion ; ouverture du système, enfin, les fonctions internes de Cépage devenant accessibles au monde extérieur et pouvant ainsi constituer le noyau d'un ensemble d'outils capables d'effectuer des manipulations très diverses sur toutes sortes de textes structurés.

Nous ne cachons pas ici notre ambition : faire de Cépage nouvelle manière le véhicule grâce auquel la "technologie" de l'édition structurelle, si riche de potentiel mais qui n'a jusqu'ici touché que quelques cercles restreints, atteindra les constructeurs de logiciel dans l'industrie.

TROISIEME PARTIE :

LES ARTICLES

LISTE DE PUBLICATIONS

Bertrand Meyer

Avril 1985

Les publications marquées d'un double point •• sont reproduites dans cette thèse (un point unique indique un texte reproduit en partie).

LIVRES

• avec Claude Baudoin, *Méthodes de Programmation*, Eyrolles, Paris, 1978. 687 pages. Nouvelle édition, 1984. Traduction russe, Mir, Moscou, 1983.

(Rédacteur) *Logiciel et Matériel, Applications et Implications (actes du congrès AFCET-INFORMATIQUE 1980)*, Nancy, novembre 1980.

"An Introduction to the Theory of Programming Languages," Notes du cours CS 262, Computer Science Department, University of California, Santa Barbara, décembre 1984. (Version en préparation sous forme de livre).

Applied Programming Methodology, 1985. En cours.

ARTICLES DE REVUES

avec Claude Kaiser et Etienne Pichat, "L'enseignement de la Programmation à l'IIE," *Zéro-Un Informatique*, 1975.

•• "Description des Structures de Données," *Bulletin de la Direction des Etudes et Recherches d'Electricité de France, Série C (Informatique)*, no. 2, p. 81-90, Clamart, 1976. Egalement dans Bulletin GROPLAN n° 2, AFCET, 1978 ; version russe, Novossibirsk, 1978.

"Initiation à la programmation en milieu industriel," *RAIRO, série bleue (informatique)*, vol. 11, no. 3, pp. 21-34, 1977.

"A Note on Computing Multiple Sums," *Software, Practice and Experience*, vol. 8, pp. 3-8, 1978.

- "Quelques concepts importants des langages de programmation modernes et leur expression en Simula 67," *Bulletin de la Direction des Etudes et Recherches d'Electricité de France, Série C (Informatique)*, no. 1, pp. 89-150, Clamart, 1979. Egalement dans le bulletin GROPLAN, AFCET, 1979 (actes de la réunion de travail sur les langages de programmation modernes, Cargèse, Corse, 14-22 mai 1979).

- avec Michel Demuynck, "Les Langages de Spécification," *Bulletin de la Direction des Etudes et Recherches d'Electricité de France, Série C (Informatique)*, no. 1, pp. 39-60, Clamart, 1979.

  avec Alain Bossavit, "Sur la Programmation rationnelle des Algorithmes numériques," *Bulletin de la Direction des Etudes et Recherches d'Electricité de France, Série C (Informatique)*, no. 2, Clamart, 1979.

  avec Jean-Raymond Abrial et Stephen A. Schuman, "A Specification Language," in *On the Construction of Programs*, ed. R. McNaughten and R.C. McKeag, Cambridge University Press, 1980.

- "Article *Langages de Programmation*," in *Encyclopédie "Techniques de l'Ingénieur"*, Paris, décembre 1980.

  avec Michel Demuynck, "Les Langages de Spécification", suivi de "A la Recherche de la Spécification Idéale," *Zéro-Un Informatique*, no. 150 et 151, pp. 62-70, 65-70, Avril et mai 1981.

- "Principles of Package Design," *Communications of the ACM*, vol. 25, no. 7, pp. 419-428, juillet 1982.

  avec Alain Bossavit, "Transformation de Programmes: Une Application à la Programmation des Super-Ordinateurs," in *Journées AFCET-Ada*, Paris, décembre 1984. (version révisée de la présentation VAPP, Oxford, Août 1984)

- avec Jean-Marc Nerson et Soon Hae Ko, "Showing Programs on a Screen," *Science of Computer Programming*, vol. 5, no. 2, 1985.

- "Incremental String Matching," *Information Processing Letters*, 1985. A paraître.

- avec Alain Bossavit, "An Application of Program Transformation to Supercomputer Programming," *Computer Physics Communications*, 1985. Version révisée de la présentation VAPP, Oxford, Août 1984, à paraître.

- "On Formalism in Specifications," *IEEE Software*, vol. 3, no. 1, pp. 6-25, janvier 1985.

  Article Génie Logiciel, Encyclopédie "Techniques de l'Ingénieur", 1985 (à paraître).

ARTICLES DE CONFERENCES AVEC REVISION

"L'Atelier Logiciel," in *Journées BIGRE*, INRIA (Rocquencourt), 1977.

avec Michel Demuynck, "Les Langages de Spécification: Vers une meilleure Compréhension des Problèmes, et des Programmes plus fiables," in *Convention Informatique*, Paris, septembre 1978.

avec Alain Bossavit, "On the Constructive Approach to Programming: The Case of "Partial Choleski Factorization" (A Tool for Static Condensation in Numerical Analysis)," in *Advances in Computer Methods for Partial Differential Equations III (IMACS Symposium, Bethlehem, Pennsylvania)*, ed. Vichnevetsky et Stepleman, pp. 287-291, 1979.

avec Gérard Brisson, "Two Program Manipulation Systems Using Simula: SATRAPE and ZAIDE," in *Ecole d'Eté Simula*, Oslo (Norvège), juillet 1980.

- "A Basis for the Constructive Approach to Programming," in *Information Processing 80 (Actes du congrès IFIP, Tokyo, Japon, 6-9 octobre 1980)*, ed. S. H. Lavington, pp. 293-298, North-Holland Publishing Company, Amsterdam (Pays-Bas), 1980.

  "A Three-Level Approach to Data Structure Description, and Notational Framework," in *ACM-NBS Workshop on Data Abstraction, Databases and Conceptual Modelling*, ed. Michael Brodie and Steven Zilles, pp. 164-166, Pingree Park, Colorado, 25-26 juin 1981. Numéro de janvier des bulletins SIPLAN, SIGMOD, SIGART.

  "Some Methodological Aspects of Using a Cray-1 (Abstract)," in *Cray User's Group Meeting*, Bad-Schliersee, Germany, septembre 1981.

- avec Alain Bossavit, "The Design of Vector Programs," in *Algorithmic Languages*, ed. Jaco de Bakker and R.P. van Vliet, pp. 99-114, North-Holland Publishing Company, Amsterdam (Pays-Bas), 1981.

  avec Bertrand Heilbronn et Alain Poujol, "Avantages et Limites des Spécifications formelles: Une expérience industrielle," in *Journées AFCET-AUTOMATIQUE sur la Validation des Spécifications fonctionnelles*, AFCET, Paris, 23 octobre 1981.

  "Vers un Environnement Conversationnel à deux dimensions," in *Journées BIGRE*, Grenoble, 27-29 janvier 1982.

  "Tools for Two-Dimensional Programming," in *Proceedings 6th ICSE (Poster Session)*, pp. 83-85, Tokyo, octobre 1982.

- "Towards a Two-Dimensional Programming Environment," in *Proceedings of the European Conference on Integrated Computing Systems (ECICS 82), Stresa (Italie), 1-3 septembre 1982*, ed. Pierpaolo Degano and Erik Sandewall, pp. 167-179, North-Holland, Amsterdam (Pays-Bas), 1983. Version anglaise de l'article des journées Bigre.

  "Software Engineering for Engineering Software (conférence invitée)," in *ACM-INRIA Conference on Tools, Methods and Languages for Scientific Computation*, Paris, 16-18 mai 1983.

  avec Alain Bossavit, "An Application of Program Transformation to Supercomputer Programming (conférence invitée)," in *Proceedings of VAPP Conference (Vector And Parallel Processors in Computational Science)*, Oxford (Great Britain), 20-24 août 20-24, 1984.

  "A System Description Method," in *International Workshop on Models and Languages for Software Specification and design*, ed. Robert G. Babb II and Ali Mili, pp. 42-46, Orlando (Fl.), mars 1984.

- avec Jean-Marc Nerson, "Cépage : Un Editeur structurel Pleine Page," in *Second Colloque de Génie Logiciel (E. Girard, réd.)*, pp. 153-158, AFCET, Nice, 1984. Traduction anglaise:

"CEPAGE, a full-screen structured editor" in *Software Engineering: Practice and Experience*, North Oxford Academic, Oxford, 1984, pp. 60-65.

•• "The Software Knowledge Base," in *8th International Conference on Software Engineering*, London, août 1985. A paraître.

ARTICLES NON SOUMIS A REVISION (BULLETINS, ETC.)

"Sur le Formalisme dans les Spécifications," *Globule (Bulletin du groupe AFCET-Génie Logiciel)*, no. 1, pp. 81-122, 1979.

avec Jean-Marc Nerson, "Quatre Outils d'Analyse Statique," *GLOBULE*, no. 4, Paris, 1982.

"Some Mistakes are Worse than Others," *Software Engineering Notes (ACM)*, vol. 8, no. 5, octobre 1983.

"A Note on Iterative Hanoi," *SIGPLAN Notices (ACM)*, vol. 19, no. 12, pp. 38-40, décembre 1984.

QUELQUES RAPPORTS NON PUBLIES

"Un Ramasse-Miette par Tri," Atelier Logiciel no. 8, Electricité de France, Clamart, août 1978.

avec B. Lalandeet P. Gaudron, "Reduce: Calcul Symbolique," Atelier Logiciel 20, Electricité de France, Clamart, 1979.

• "Un Calculateur Vectoriel: Le Cray-1 et sa Programmation (Version 2)," Atelier Logiciel no. 24, HI-34552/01, Electricité de France, Clamart, juin 4, 1980.

avec Eugène Audin, Gérard Brisson et Françoise Vapné-Ficheux, "Gescran, Manuel de Référence," Atelier Logiciel 22, Electricité de France, Clamart, 1980.

"Le langage Fortran 77," Atelier Logiciel 10, Electricité de France, Clamart, juin 1980.

avec Gérard Brisson, "TRI: tri interne rapide," Atelier Logiciel no. 13, Electricité de France, Clamart, avril 1981.

"La Spécification," Atelier Logiciel 28, Electricité de France, Clamart, avril 1981.

"Ensorcelé: Entrées et Sorties Sans Format (1ère partie)," Atelier Logiciel no. 4, Electricité de France, Clamart, avril 1981. (Quatrième Edition).

avec Gérard Brisson et Françoise Vapné-Ficheux, "Ensorcelé: Entrées et Sorties Sans Format (2ème partie)," Atelier Logiciel no. 6, Electricité de France, Clamart, décembre 1982.

"Errare, un Outil de Traitement des Erreurs," Atelier Logiciel 30, Electricité de France, Clamart, 1983.

avec Eric de Drouas, "Ada: Introduction et Bibliographie," Atelier Logiciel 21, Electricité de France, Clamart, 1983.

• "Modularity," Notes du cours CS 272, Computer Science Department, University of California, Santa Barbara, janvier 1985. (Chapitre du livre "Applied Programming Methodology", en cours).

• "Control Structures," Notes du cours CS 272, Computer Science Department, University of California, Santa Barbara, janvier 1985. (Chapitre du livre "Applied Programming Methodology", en cours).

ANALYSES BIBLIOGRAPHIQUES

Nombreuses analyses bibliographiques dans TSI, 1982-1985.

avec Olivier Lecarme, "Revue de "Compilation", par Cunin, Voiron, Griffiths," *ACM Computing Reviews*, 1982.

"Le Facteur humain a sonné trois fois (Analyses bibliographiques: Card, Moran, Newell; Ledgard, Singer, Whiteside; Shneiderman)," *TSI*, vol. 3, no. 3, pp. 227-230, juin 1984.

"Revue de *Computer Systems Methodologies*, par C.A. Ziegler," *Science of Computer Programming*, vol. 5, no. 1, janvier 1985.

NOTES DE COURS

Introduction à la Programmation en Algol W, IIE (Institut d'Informatique d'Entreprise), Paris, 1975.

avec Abrial, Jean-Raymond, *Notes sur la spécification formelle, étude de cas.*, Ecole d'été INRIA-EDF-CEA, Le-Bréau-sans-Nappe, juillet 1978.

Le Génie Logiciel, Notes de cours, EDF, 1981-83.

"An Introduction to the Art of Writing Correct Programs," Rapport TRCS84-05, Computer Science Department, University of California, Santa Barbara, mars 1984 (Revisé en janvier 1985). (Notes du cours CS130A).

FILM

avec Eric de Drouas, *Un ordinateur vectoriel: le Cray-1 et sa Programmation*, Electricité de France, Clamart, 1981.

ARTICLE ACTUELLEMENT SOUMIS A UNE REVUE

avec Jean-Marc Nerson, "A Visual and Structural Editor," Rapport TRCS84-03, Computer Science Department, University of California, Santa Barbara, mars 1984.

Service Informatique et Mathématiques Appliquées

Description des structures de données

B. MEYER *

1. – INTRODUCTION

La nécessité de séparer, dans la description d'une structure de données, les propriétés abstraites des objets et leur représentation concrète, est reconnue depuis longtemps sur le plan des idées. Dans [7], par exemple, d'Imperio faisait déjà la distinction entre "structures de données" et "structures de mémoire". En pratique, cependant, cette distinction reste souvent un voeu pieux, et la manipulation concrète des structures de données mélange caractéristiques logiques et problèmes de représentation physique.

Il est souvent important, pourtant, de dégager les propriétés "intrinsèques" d'une structure de données des caractéristiques de sa représentation. Dans un traité récent sur la conception et l'analyse des algorithmes [2], les "files" (*queues*) sont introduites initialement (§ 2.1) par le biais d'une représentation formée d'un tableau et de deux pointeurs. Or, le premier algorithme qui utilise des files (figure 3.1 de [2]) effectue l'opération "concaténation de 2 files" dont l'implémentation efficace requiert une représentation chaînée (bien que l'utilisation d'une représentation contiguë ne change pas la complexité globale de l'algorithme en question, qui est $0((m + n)\,k)$). Il semble donc qu'on doive utiliser une notion "abstraite" de file, préexistente à toute représentation physique.

Le but du présent article est de proposer, dans la lignée des travaux de Liskov et Zilles [8] [9], une définition des structures de données reposant sur trois niveaux de description que nous appellerons *spécification fonctionnelle*, *description logique*, et *représentation physique*, et de montrer la nécessité de cette distinction en l'appliquant à la définition de quelques structures classiques.

NOTATIONS

Nous aurons à manipuler des fonctions bivaluées, de la forme :

$$f : A \to B \times C$$

Pour $x \in A$, nous noterons $f_B^{(x)}$ et $f_C^{(x)}$ les projections de $f(x)$ sur B et C respectivement.

Soit une fonction f de la forme

$$f : B \times A \to A$$

pour $x \in A$, et $y_1, y_2, \ldots, y_n \in B$, nous dénoterons par

$$f^{(n)}(y_1, y_2, \ldots, y_n ; x)$$

la valeur de $\quad f(y_1, f(y_2, f(y_3, \ldots, f(y_n, x)\ldots)))$

(*) Ingénieur-Chercheur au Département Méthodes et Moyens de l'Informatique.

E.D.F. – BULLETIN DE LA DIRECTION DES ÉTUDES ET RECHERCHES
SERIE C – MATHÉMATIQUES, INFORMATIQUE N° 2 1976 pp. 81-90

2. – SPECIFICATION FONCTIONNELLE

Une "structure de données" est un objet composé vérifiant certaines propriétés. La description d'une telle structure est la définition du domaine des objets vérifiant ces propriétés, c'est-à-dire la définition d'un nouveau *type* de données.

On appellera *spécification fonctionnelle* d'une structure de données la définition externe de ses propriétés, c'est-à-dire l'ensemble des caractéristiques qu'elle doit présenter pour qu'un programme puisse l'utiliser. La spécification fonctionnelle donne une description logiquement complète du domaine D considéré, mais n'implique aucune décision quant à l'organisation interne des objets appartenant à ce domaine.

Une spécification fonctionnelle est formée de :

a) un certain nombre de fonctions $f_i (i = 1, 2, \ldots, n)$:

$$f_i : X_{i_1} \times X_{i_2} \times \ldots \times X_{i_p} \rightarrow X_{i_{p+1}} \times \ldots \times X_{i_q}$$

où l'un au moins des X_{i_j} est le domaine D que l'on cherche à définir, et

b) un certain nombre de relations faisant intervenir ces fonctions et des fonctions définies sur les domaines $X_{i_1} \ldots X_{i_q}$ autres que D. Ces relations seront en général des formules du calcul des prédicats du premier ordre, avec égalité ; elles servent à définir les propriétés abstraites que doivent vérifier les fonctions définies sur D.

On distinguera en pratique trois sortes de fonctions

$$f_i : X_{i_1} \times X_{i_2} \times \ldots \times X_{i_p} \rightarrow X_{i_{p+1}} \times \ldots \times X_{i_q}$$

a) fonctions de création : D n'apparaît qu'à droite de la flèche. La fonction peut ne pas avoir d'arguments ($p = 0$).

b) fonctions d'accès : D n'apparaît qu'à gauche de la flèche.

c) fonctions de modification : D apparaît à gauche et à droite.

3. – DESCRIPTION LOGIQUE

Le niveau suivant de décomposition, que nous appelons *description logique*, peut être pour le programmeur le niveau terminal s'il dispose d'un langage de programmation évolué (ALGOL W, SIMULA 67, PASCAL, ALGOL 68 . . .) ; dans le cas contraire (programmation en langage machine, FORTRAN etc.) un niveau d'affinage supplémentaire sera nécessaire.

La description logique consiste à fournir une décomposition (éventuellement récursive) des objets du domaine en objets d'autres domaines, et une décomposition (éventuellement récursive) des fonctions intervenant dans les spécifications fonctionnelles en termes d'autres fonctions. Ces décompositions doivent évidemment satisfaire les axiomes de la spécification fonctionnelle.

La description logique fait intervenir un certain nombre de types de base et d'opérations de composition. Plusieurs choix sont possibles pour cet ensemble d'éléments de base :

1) Un premier exemple, proche de ce que proposent des langages comme ALGOL W, PASCAL, SIMULA, ALGOL 68, etc. consiste à prendre un certain nombre de types connus (entiers, réels, caractères. . .), et les trois opérations de composition suivantes :

a) **juxtaposition** (produit cartésien) : t_1 et t_2 étant des types, on notera :

$$t_1 \; ; \; t_2$$

le type dont les éléments sont des couples d'objets de types respectifs t_1 et t_2.

b) **union** : on notera

$$t_1 \mid t_2$$

le type dont les éléments sont, soit de type t_1, soit de type t_2.

c) **énumération** :

$$\alpha_1 , \alpha_2 , \ldots , \alpha_n$$

dénotera le type dont les éléments peuvent être égaux à l'une des constantes

$$\alpha_i , i = 1 , \ldots , n$$

Du point de vue de la notation, on pourra utiliser des parenthèses pour lever les ambiguïtés dans la description de types complexes, et étiqueter les différents composants dans une juxtaposition. Exemples :

type *DATE = an : ENTIER ; mois : ("janvier", . . . , "décembre") ; jour : ENTIER*

type *PERSONNE = nom : TEXTE ; âge : (DATE\|ENTIER) ; sexe : LOGIQUE*

Les définitions peuvent être récursives. On utilisera en général dans ce cas une "union" avec un type spécial appelé *VIDE* (contenant un seul élément que nous noterons également *VIDE* par abus de langage) :

type *LISTE D'ENTIERS = VIDE \|(premier : ENTIER ; suite : LISTE D'ENTIERS)*

type *ARBRE BINAIRE = VIDE \|(racine : INFO ; gauche : ARBRE BINAIRE ; droite : ARBRE BINAIRE)*

Notons à cet égard qu'il est prématuré d'introduire à ce niveau la notion de pointeur [6], de même qu'il serait prématuré d'introduire la notion de branchement dans la description logique d'un algorithme.

2) On peut réduire le nombre d'éléments de base en se limitant, comme [3], à un ensemble de départ contenant :

– un seul type de base (qui est la juxtaposition de 0 types) ;

– 2 opérations de composition : l'union et la juxtaposition.

Avec une interprétation adéquate, on peut alors retrouver de façon simple tous les types usuels (booléens, entiers, . . .) et complexes (listes, arbres. . .).

3) Un troisième ensemble d'éléments de base d'une description logique pourrait être emprunté aux méthodes de descriptions des bases de données, en particulier à un modèle relationnel : modèle de Codd [4] ou modèle binaire d'Abrial [1].

4) A titre de dernier exemple d'outils de description logique, mentionnons l'approche de Gedanken [13], où les structures de données sont décrites comme des fonctions. Les opérations de base sont alors la composition de fonctions et l'expression conditionnelle.

4. – REPRESENTATION PHYSIQUE

La représentation physique d'une structure de données est l'implantation concrète, en machine, d'objets et de sous-programmes conformes à une description logique. Les descriptions logiques étant en général récursives, on peut considérer la représentation physique comme une sorte de "point fixe" de la description logique. Son élaboration sera, selon le langage de programmation utilisé, à la charge du compilateur ou à celle du programmeur.

Nous ne développerons pas dans cet article les problèmes de la représentation physique, où interviennent des méthodes telles que l'utilisation des pointeurs (pour la juxtaposition récursive), de "drapeaux" (pour l'union), les algorithmes de "ramasse-miettes" et de gestion de la mémoire etc. Signalons simplement un point important : l'approche "fonctionnelle" utilisée jusqu'ici n'interdit

en aucune façon de représenter certaines des fonctions de manipulation par des "procédures non typées" (*SUBROUTINE*) opérant par effet de bord sur certains arguments, si cela conduit à une implantation plus efficace ; on exigera simplement que les relations intervenant dans la spécification fonctionnelle soient vérifiées entre les valeurs initiale et finale de l'argument correspondant.

5. – UN EXEMPLE

Soit à définir le type "compte en banque", ou *COMPTE*, et le type "ensemble de comptes en banque", ou *BANQUE*. Nous supposons connus les types de base *ENTIER*, *ENTIER POSITIF*, *TEXTE* (= chaîne de caractères), *LOGIQUE* (= booléen), et le type *PERSONNE* (qui peut être un type simple ou complexe). Une spécification fonctionnelle possible pour *COMPTE* et *BANQUE* comprend :

fonctions de création

créer-banque : → *BANQUE*

créer-compte : *PERSONNE* × *ENTIER POSITIF* → *COMPTE*

fonctions d'accès

titulaire : *COMPTE* → *PERSONNE*

solde : *COMPTE* → *ENTIER*

trouver-compte : *BANQUE* × *PERSONNE* → *COMPTE* ∪ {*VIDE*}

fonctions de modification

inscrire-compte : *BANQUE* × *COMPTE* → *BANQUE*

ajouter : *COMPTE* × *ENTIER POSITIF* → *COMPTE*

retirer : *COMPTE* × *ENTIER POSITIF* → *COMPTE*

relations

$\forall b \in BANQUE, \forall c, c' \in COMPTE, \forall n, n' \in ENTIER\ POSITIF, \forall p, p' \in PERSONNE$:

(C_1) *titulaire (créer-compte (p , n)) = p*

(C_2) *solde (créer-compte (p , n)) = n*

(C_3) $c = c' \Leftrightarrow$ *titulaire (c) = titulaire (c')*

(C_4) *trouver-compte (créer-banque, p) = VIDE*

(C_5) *trouver-compte (inscrire-compte (b, créer-compte (p , n)) , p) = créer-compte (p , n)*

(C_6) *inscrire-compte (inscrire-compte (b , c) , c') = inscrire-compte (inscrire-compte (b , c') , c)*

(C_7) *solde (ajouter (c , n)) = solde (c) + n*

(C_8) *solde (retirer (c , n)) = solde (c) – n*

(C_9) *titulaire (ajouter (c , n)) = titulaire (c)*

(C_{10}) *titulaire (retirer (c , n)) = titulaire (c)*

De ces propriétés, on peut déduire un grand nombre d'autres. Par exemple :

(C_1) et (C_3) entraînent

(C_{11}) $p \neq p' \Rightarrow$ *créer-compte (p , n)* $\neq$ *créer-compte (p' , n')*

de même, on peut montrer à partir de (C_1),(C_3), (C_4), (C_5) et (C_6) que

(C_{12}) $p \neq p' \Rightarrow$ *trouver-compte (b , p) = trouver-compte (inscrire-compte (b , créer-compte (p' , n')) , p)*

à condition toutefois de supposer que les seules opérations effectuées sur les *COMPTES* et les *BANQUES* sont celles qui entrent dans la définition fonctionnelle (démonstration par récurrence sur le nombre d'opérations *inscrire-compte* effectuées).

Une relation du type (C_3) indique que l'un des "composants" d'un type (ici le titulaire) est une "clé primaire" [4].

Une description logique des types *COMPTE* et *BANQUE* (conduisant sans doute à une représentation physique peu efficace) pourrait être :

type *BANQUE = VIDE | (premier : COMPTE ; suite : BANQUE)*

type *COMPTE = possesseur : (PERSONNE | SOCIETE); numéro : ENTIER POSITIF ; solde : ENTIER*

type *PERSONNE = PERSONNE PHYSIQUE | SOCIETE*

type *PERSONNE PHYSIQUE = (nom : TEXTE ; adresse : TEXTE)*

type *SOCIETE = nom : TEXTE ; siège-social : TEXTE ; capital : ENTIER POSITIF*

avec pour les fonctions associées des descriptions logiques de la forme :

sous-programme *trouver-compte (p : PERSONNE ; b : BANQUE) : COMPTE*

```
si   b = VIDE alors
                VIDE
sinon si titulaire (premier [b]) = p alors
                premier [b]
sinon    trouver-compte (p , suite [b])
```

Notons qu'il s'agit bien là d'une description logique, et que la représentation physique ne fera pas nécessairement intervenir des sous-programmes récursifs. Un autre point important est qu'à une spécification fonctionnelle donnée peuvent correspondre plusieurs descriptions logiques, et réciproquement ; ainsi, une même structure logique de "banque" pourra être "vue" fonctionnellement de diverses façons pour des raisons de protection (certaines opérations étant absentes de certaines spécifications). De la même manière, une structure logique peut avoir plusieurs représentations physiques adéquates.

6. – SPECIFICATION FONCTIONNELLE DE QUELQUES STRUCTURES CLASSIQUES

Nous allons préciser la notion de spécification fonctionnelle à propos de quelques structures usuelles : pile, listes, tableaux, ensembles, etc. La simplicité de ces spécifications, qui permettent cependant de démontrer toutes les propriétés habituelles des structures considérées, nous paraît un argument important en faveur de l'utilisation de spécifications fonctionnelles avant toute description détaillée de l'organisation des données d'un programme.

Dans ce qui suit, nous considérons des structures complexes bâties à partir d'un type de base que nous appellerons T : des piles d'objets de type T, des files d'objets de type T, etc. Le type T est quelconque (ce pourrait être *ENTIER*, *TEXTE* etc. ou un type complexe, comme *PILE D'ENTIERS* etc.) ; nous le considèrerons cependant comme connu, c'est-à-dire que nous n'aborderons pas le problème de la définition de types complexes paramétrés.

Nous considèrerons d'abord deux variétés de structures que l'on peut appeler "structures séquentielles à lecture destructrice" : les *piles* ou "structures à une seule tête de lecture/écriture", et les *files*, à "une tête de lecture et une tête d'écriture" ; nous étudierons ensuite les *tableaux*, qui sont à "lecture non destructrice, accès direct, et une seule tête de lecture/écriture".

6.1. – Piles.

Une pile a la spécification fonctionnelle suivante :

fonctions

créer-pile :	$\rightarrow PILE$	(création)
vide :	$PILE \rightarrow LOGIQUE$	(accès)
dépiler :	$PILE \rightarrow T \times PILE$	(modification)
empiler :	$T \times PILE \rightarrow PILE$	(modification)

relations

$\forall t \in T,\ \forall p \in PILE$:

(P_1) *vide (créer-pile)*

(P_2) $\sim$ *vide (empiler (p , t))*

(P_3) *dépiler (empiler (t , p))* = [t , p]

Ces trois axiomes permettent de retrouver toutes les propriétés habituelles des piles. La plus importante de ces propriétés peut s'énoncer de façon très intuitive, "on retrouve les éléments dans l'ordre inverse de celui où on les a mis", ou encore

$$\forall p \in PILE,\ \forall t_1, t_2, \ldots, t_n \in T :$$
$$0 \leqslant m < n \Rightarrow dépiler_T(dépiler^{(m)}_{PILE}(empiler^{(n)}(t_n, t_{n-1}, \ldots, t_1 ; p)) = t_{n-m}$$

C'est-à-dire que si l'on dépile $m + 1$ éléments d'une pile sur laquelle on a auparavant empilé n éléments $t_1, t_2, \ldots, t_n$, $(0 \leqslant m < n)$, le dernier élément dépilé est t_{n-m}. Cette propriété se démontre immédiatement à partir de (P_3) par récurrence sur m.

Plus généralement, on démontre facilement à partir de (P_3) la propriété suivante : soit

$$X \equiv a_n(a_{n-1}(\ldots a_1(p)\ldots))$$

une expression où chaque fonction a_i est :

{ soit $dépiler_{PILE}$
{ soit la fonction associant à toute pile y la pile $empiler(t_i, y)$ $(t_i \in T)$

et telle que pour i, $1 \leqslant i \leqslant n$, il y ait parmi $a_i, a_{i+1}, \ldots, a_n$ au moins autant d'opérations *empiler* que d'opérations *dépiler*.

Alors, on obtient une expression égale à X en rayant successivement, à partir de la droite, chaque opération *dépiler* et l'opération *empiler* non encore rayée qui se trouve (par construction) immédiatement à sa droite. Exemple :

$$empiler\ (t_6, dépiler_{PILE}(empiler(t_4, empiler(t_3, dépiler_{PILE}(empiler(t_1 , p)))))) =$$
$$= empiler(t_6 , empiler(t_3 , p))$$

Les axiomes (P_1), (P_2), (P_3) n'interdisent pas expressément de *dépiler* une pile vide, mais empêchent de connaître quelque propriété que ce soit sur l'élément "dépilé" d'une pile vide. Une pile vide dépilée reste cependant une pile légale, sur laquelle on pourra ensuite *empiler* des éléments de T en leur appliquant l'axiome (P_3). Si l'on veut préciser au niveau de la spécification fonctionnelle que le dépilage d'une pile vide est une erreur et rend la pile inutilisable, on associera à chaque type X un élément spécial "indéfini" noté ω_X ; on ajoutera alors l'axiome suivant :

(P_4) *vide (p)* $\Rightarrow$ *dépiler (p)* = $[\omega_T, \omega_{PILE}]$

et l'on spécifiera que les fonctions caractéristiques de la structure sont "strictement monotones", ou encore [11, p. 359] des "extensions naturelles" au sens de la théorie du point fixe, c'est-à-dire qu'elles valent ω si l'un au moins de leurs arguments est ω :

(P_5) *vide* (ω_{PILE}) = $\omega_{LOGIQUE}$

(P_6) *dépiler* (ω_{PILE}) = $[\omega_T, \omega_{PILE}]$

(P_7) *empiler* (t, ω_{PILE}) = ω_{PILE}

(P_8) *empiler* (ω_T, p) = ω_{PILE}

La plupart des descriptions de piles publiées font intervenir la notion de "taille" d'une pile. Il s'agit là à notre sens d'un concept qui ne doit pas intervenir au niveau d'une spécification fonctionnelle : il est important dans une implantation contigue (tableau et pointeur), mais beaucoup moins dans le cas d'une implantation chaînée. De fait, si toute implantation oblige à prendre en compte une taille maximale, on peut considérer qu'elle ne fait pas partie du type abstrait *PILE*. Ceci dit, la notion de taille s'introduit toutefois facilement dans le formalisme ci-dessus : une pile de taille n est caractérisée par l'axiome supplémentaire :

(P_9) $\forall m > n$, $empiler^{(m)}_{PILE}(p ; t_1, t_2, \ldots, t_m) = \omega_{PILE}$

ou simplement, si l'on suppose que les fonctions de base sont des "extensions naturelles" (axiomes (P_5) à (P_8)) :

(P'_9) $empiler^{(n+1)}(t_{n+1}, t_n, \ldots, t_2, t_1 ; créer\text{-}pile) = \omega_{PILE}$

6.2. – Files.

La spécification fonctionnelle d'une *file* (anglais *queue*) est donnée par

fonctions

créer-file :	$\rightarrow FILE$	(création)
vide :	$FILE \rightarrow LOGIQUE$	(accès)
défiler :	$FILE \rightarrow T \times FILE$	(modification)
enfiler :	$T \times FILE \rightarrow FILE$	(modification)

relations

$\forall t \in T,\ \forall f \in F$:

(F_1) *vide (créer-file)*

(F_2) $\sim$ *vide (enfiler (t , f))*

(F_3) $\sim$ *vide (f)* $\Rightarrow$ *défiler (enfiler (t , f))* = $[défiler_T(f), enfiler(t , défiler_{FILE}(f))]$

(F_4) *vide (f)* $\Rightarrow$ *défiler (enfiler (t , f))* = [t , f]

L'axiome (F_3) affirme la commutativité des opérations *défiler* et *enfiler* sur une file non vide (seul cas où cette propriété ait un sens) ; il est évident intuitivement puisque l'on "enfile à un bout" et que l'on défile à l'autre.

Il est intéressant de noter que nos définitions de piles et de files ne diffèrent que par le 3e axiome (P_3 , F_3), l'axiome correspondant à (F_4) étant vrai sur les piles. Les axiomes (F_1 – F_4) permettent de retrouver toutes les propriétés habituelles des files. La plus importante s'exprime intuitivement par le fait que "l'on retrouve les éléments dans l'ordre où on les a mis". Par exemple, on montre immédiatement par récurrence que pour $0 \leqslant m < n$:

$$vide(f) \Rightarrow défiler_T(défiler^{(m)}_{FILE}(enfiler^{(n)}(t_n, t_{n-1}, \ldots, t_1 ; f)) = t_{m+1}$$

Plus généralement, si f est une file vide, et si l'on considère l'expression

$$X = a_n(a_{n-1}(\ldots(a_1(f)\ldots))$$

ou chaque a_i est

{ soit $défiler_{FILE}$
{ soit la fonction qui associe à toute file y la file $enfiler(t_i , y)$ $(t_i \in T)$

et telle que pour tout i, $1 \leqslant i \leqslant n$, il y ait parmi $a_i, a_{i+1}, \ldots, a_n$ au moins autant d'*enfiler* que de *défiler*, alors on ne change pas la valeur de X en rayant toutes les opérations $défiler_{FILE}$, et autant d'opérations *enfiler* **à partir de la droite.** Exemple :

si *vide (f)* alors
$enfiler(t_6, défiler_{FILE}(enfiler(t_4, enfiler(t_3, défiler_{FILE}(enfiler(t_1, f)))))))) =$
$= enfiler(t_6, enfiler(t_4, f))$

Si f n'est pas vide, on peut d'après (F_2) et (F_3) ramener toutes les opérations $défiler_{FILE}$ à gauche dans l'expression X. Ainsi, l'expression de l'exemple ci-dessus est égale pour tout f à

$$défiler_{FILE}(défiler_{FILE}(enfiler(t_6, enfiler(t_4, enfiler(t_3, enfiler(t_1, f)))))))$$

Comme pour une pile, on peut spécifier qu'il est illégal de *défiler* une file vide par des axiomes semblables à (P_4) et $(P_5 - P_8)$ ("extension naturelle"). On peut de même préciser qu'une file est de "taille" n par un axiome transposé de (P_9) ou (P_9').

6.3. – Tableaux.

Par opposition aux structures précédentes, un tableau est une structure à *accès direct* indexé.

La spécification fonctionnelle d'un tableau à une dimension de bornes m et n est donnée ci-dessous (on note $[m : n]$ l'ensemble $\{i \mid i \in \mathbf{N},\ m \leqslant i \leqslant n\}$, et TABLEAU $[m : n]$ l'ensemble des tableaux de bornes m et n) :

fonctions

créer-tableau : $\rightarrow$ *TABLEAU* $[m : n]$ (création)
accéder-élément : $[m : n] \times$ *TABLEAU* $[m : n] \rightarrow T$ (accès)
modifier-élément : $T \times [m : n] \times$ *TABLEAU* $[m : n] \rightarrow$ *TABLEAU* $[m : n]$ (modification)

relations

$\forall t, t' \in T, \forall tab \in TABLEAU\ [m : n], \forall i, j \in [m : n]$:

(T_1) *accéder-élément (i, modifier-élément (t, i, tab)) = t*

(T_2) $i \neq j \Rightarrow$ *modifier-élément (t, i, modifier-élément (t', j, tab))*
= modifier-élément (t', j, modifier-élément (t, i, tab))

(T_3) *modifier-élément (t, i, modifier-élément (t', i, tab)) = modifier-élément (t, i, tab)*

Notons que le langage LISP 1.5 [10] introduit précisément les tableaux à l'aide de deux fonctions distinctes, accès et modification.

On pourrait aussi considérer que les fonctions *accéder-élément* et *modifier-élément* font intervenir le domaine N plutôt que $[m : n]$, avec la convention que le résultat est indéfini (ω_T ou $\omega_{tableau}$) si leur argument entier n'appartient pas à $[m : n]$.

6.4. – Ensembles.

Même au niveau de la spécification fonctionnelle, on peut définir les caractéristiques d'une structure de données de façon plus ou moins précise. Ainsi, piles, listes et tableaux (de même qu'un grand nombre d'autres structures, comme les *arbres binaires de recherche* dont la spécification fonctionnelle est facile à donner en supposant T muni d'une relation d'ordre) peuvent servir de réalisations particulières à une structure générale, celle d'*ensemble*, dont la spécification fonctionnelle est :

fonctions

créer-ensemble : $\rightarrow$ *ENSEMBLE* (création)
membre : $T \times$ *ENSEMBLE* $\rightarrow$ *LOGIQUE* (accès)
ajout : $T \times$ *ENSEMBLE* $\rightarrow$ *ENSEMBLE* (modification)
retrait : $T \times$ *ENSEMBLE* $\rightarrow$ *ENSEMBLE* (modification)

relations

$\forall e \in ENSEMBLE, \forall t, t' \in T$:

(E_1) *membre (t, créer-ensemble)* = **faux**
(E_2) *membre (t, ajout (t , e))* = **vrai**
(E_3) *membre (t, retrait (t , e))* = **faux**
(E_4) *ajout (t, ajout (t',e)) = ajout (t', ajout (t , e))*
(E_5) $t \neq t' \Rightarrow$ *membre (t, retrait (t' , e)) = membre (t , e)*

7. – CONCLUSION

Nous avons jusqu'ici négligé un problème lié aux structures de données : celui de la complexité des algorithmes qui les manipulent. On peut remarquer par exemple que les *files* peuvent servir à une description logique des *tableaux*, à l'aide de l'algorithme ci-dessous :

sous-programme *accéder (i : ENTIER ; tab : TABLEAU$_T$ [m : n]) : T*

```
{tab est représenté par une file}
variable x : T ;
si i < m ou i > n alors
                    ω_T
sinon | pour j variant de m à i répéter
      |   | [x , tab] ← défiler (tab) ;
      |   |   enfiler (x , tab) ;
      | x {résultat renvoyé par le sous-programme}
```

et d'un algorithme similaire pour *modifier (t , i , tab)*. L'inconvénient est évidemment que cette décomposition conduit à une implantation physique de complexité *0(i)*, alors qu'on attend d'un tableau qu'il soit à "accès direct". Un des rôles de la spécification fonctionnelle est donc précisément de mettre en lumière les opérations fondamentales sur la structure, que l'on désire ensuite réaliser de la façon la plus efficace possible.

Pour ce problème, comme pour tous ceux qui se posent à propos des structures de données, la distinction préalable entre les trois niveaux de description dégagés dans cet article nous paraît une méthode de décomposition indispensable à la clarification des concepts fondamentaux.

BIBLIOGRAPHIE

[1] ABRIAL J.R. – *Data Semantics.* Université Scientifique et Médicale de Grenoble, Laboratoire d'Informatique, 1974.

[2] AHO A.V., HOPCROFT J.D. et ULLMANN. – *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[3] BURGE W. – *Recursive Programming Techniques.* Addison-Wesley, 1976.

[4] CODD E.F. – A Relational Model for Large Shared Data Banks ; *Communications of the ACM*, Vol. 13, n° 6, Juin 1970, pp. 377-387.

[5] HOARE C.A.R. – Notes on Data Structuring ; *in* DAHL, DIJKSTRA, HOARE. – *Structured Programming.* Academic Press, 1972.

[6] HOARE C.A.R. – *Recursive Data Structures.* Stanford University, rapport STAN-CS-73-400, Octobre 1973.

[7] d'IMPERIO M.E. – Data Structures and their Representation in Storage ; *Annual Review in Automatic Programming*, 1969, pages 1-75.

[8] LISKOV B. et ZILLES S. – Programming with Abstract Data Types ; *Sigplan Notices*, 9, 5, Mai 1974.

[9] LISKOV B. et ZILLES S. – Specification Techniques for Data Abstractions ; *IEEE Transactions on Software Engineering*, vol. 1 n° 1, pages 7-19.

[10] McCARTHY J., *et al. : LISP 1.5 Programmer's Manual.* The MIT Press, 1962.

[11] MANNA Z. – *The Mathematical Theory of Computation.* McGraw-Hill, 1975.

[12] PARNAS D.L. – A Technique for Software Module Specification with Examples ; *Communications of the ACM*, vol. 6 n° 1, mai 1972, pages 330-336.

[13] REYNOLDS J.C. – GEDANKEN, a Simple Typeless Language Based on the Principle of Completeness and the Reference Concept ; *Communications of the ACM*, vol. 13 n° 5, mai 1970, pages 308-318.

CHAPITRE V

Structures de données

Comment localiser le vénérable et secret hexagone qui l'abritait ? Une méthode rétrograde fut proposée : pour localiser le livre A, on consulterait au préalable le livre B, qui indiquerait la place de A ; pour localiser le livre B, on consulterait au préalable le livre C, et ainsi jusqu'à l'infini... C'est en de semblables aventures que j'ai moi-même prodigué mes forces, usé mes ans.

Jorge Luis Borges,
La Bibliothèque de Babel, in *Fictions*

Structures de données

Ce chapitre vise à appliquer la démarche d'analyse systématique, ébauchée au chapitre III à propos des structures de contrôle, à la seconde composante fondamentale de la programmation, "duale" des programmes : les données. Parler de "structures de données", c'est déjà affirmer la nécessité d'une description méthodique et hiérarchisée des objets manipulés par les programmes. C'est le but que poursuit la méthodologie développée ici, et appliquée ensuite à une revue, longue mais nécessaire, de quelques-unes des principales structures qui font partie de l'arsenal du programmeur.

Les *structures de données* sont les *informations organisées* que peuvent *décrire, créer* et *manipuler* les programmes. C'est par la construction de structures de données qu'il est possible de donner un sens externe aux représentations concrètes de l'information sur les supports physiques, qui ne sont par elles-mêmes que des suites de bits amorphes.

Les structures de données fondamentales ont été vues au chapitre II : il s'agit des types de base présents dans les langages de programmation – ENTIER, TEXTE, CARACTERE, REEL etc. Le but de ce chapitre est de fournir des procédés de construction et de manipulation de structures de données, ou *types*, plus complexes.

Dans le schéma général du traitement de l'information (figure V.1) ce chapitre concerne non seulement le problème de la correspondance entre les informations externes et leur représentation en machine dans le passage d'un algorithme abstrait f à un programme concret f', c'est-à-dire le problème de la construction des codes d'entrée et de sortie φ et ψ, mais aussi celui de l'accès du programme f' aux informations qu'il doit manipuler.

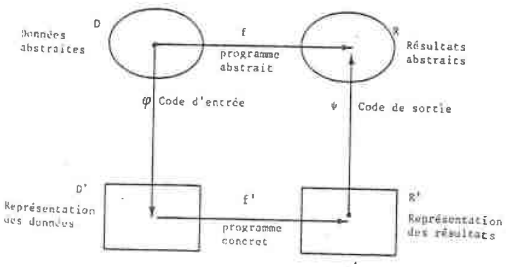


Figure V.1. – Traitement de l'information

Nous nous conformerons dans ce chapitre à l'usage, en tenant pour synonymes les mots de *données* et d'*informations*. Rappelons qu'au sens strict les "données" ne sont que l'un des trois types d'"informations" manipulées par un programme, les deux autres étant les *variables intermédiaires* et les *résultats*.

V.1 DESCRIPTION DES STRUCTURES DE DONNEES

V.1.1 Niveaux de description

La présentation des structures de données est souvent obscurcie par un mélange très gênant entre les propriétés abstraites des structures décrites et des problèmes de représentation associés à une machine ou à un langage. Il importe de bien distinguer ces problèmes par une description en plusieurs niveaux.

Définition : On appelle **structure de données** l'association d'un ou plusieurs *noms* et d'un *ensemble d'informations* auxquelles ce ou ces noms permettent d'accéder.

A titre d'exemple, considérons les informations représentant l'état des comptes des clients dans une banque. **Physiquement**, ces données sont représentées par un certain nombre de bits, d'octets ou de mots répartis en mémoire centrale ou sur des supports périphériques. **Logiquement**, le programmeur désirera accéder aux informations associées au compte d'un client particulier comme à une entité désignée par un nom, par exemple :

COMPTE (MEYER)

sur laquelle on doit pouvoir effectuer un certain nombre d'opérations bien définies (retrait, dépôt, recherche du solde....).

Une structure de données, donc, est à la fois un objet abstrait (un ou plusieurs noms) et un objet physique (des informations). Pour assurer la correspondance entre ces deux niveaux extrêmes, le programmeur doit fournir une décomposition de la structure de données en termes de structures plus élémentaires. Toute structure de données peut donc être décrite à trois niveaux distincts, qui sont, du plus abstrait vers le plus concret :

- La *spécification fonctionnelle*, qui précise pour une classe de noms les **opérations** permises sur ces noms, et les **propriétés** de ces opérations ; il s'agit donc d'une définition **externe**.
- La *description logique*, qui fournit une décomposition des objets répondant à la définition fonctionnelle en objets plus élémentaires, et une décomposition des opérations associées en opérations plus élémentaires.
- La *représentation physique*, enfin, qui est le mode d'implantation, dans la mémoire d'un ordinateur, des informations composant la structure et des relations entre elles, et le mode de codage des opérations dans un langage de programmation.

Les trois paragraphes suivants seront consacrés au développement de ces trois notions de spécification fonctionnelle, de description logique et de représentation physique. Notons tout de suite que la démarche naturelle, quand on cherche à construire une structure de données pour la résolution d'un problème, consiste à la définir d'abord du point de vue fonctionnel (quelles opérations veux-je pouvoir effectuer sur mes données et quelles sont les propriétés de ces opérations ?), ensuite du point de vue logique (quelles données de types connus, et quelles relations construites sur ces données, me permettent de respecter la spécification fonctionnelle ?), enfin du point de vue physique (comment puis-je représenter et utiliser tout cela dans ma machine, avec le langage dont je dispose ?).

A une même décomposition fonctionnelle peuvent, nous le verrons, correspondre plusieurs descriptions logiques, qui pourront être elles-mêmes réalisées par plusieurs représentations physiques. On doit cependant pouvoir s'assurer que la décomposition de chaque niveau "représente" bien la décomposition de niveau immédiatement supérieur : en d'autres termes, l'analyse au niveau *fonctionnel* fournit un "cahier des charges" qui doit être respecté au niveau *logique*, et celui-ci guide de la même façon la construction de la représentation *physique* de la structure de données.

V.1.2 Spécification fonctionnelle (types abstraits de données)

Supposons que nous ayons été chargés, en tant que programmeurs, de construire une structure de données permettant aux employés d'un banque, qui ont par exemple accès à une console conversationnelle, d'effectuer un certain nombre de manipulations sur les comptes des clients.

La première étape consiste à définir précisément quelles sont ces manipulations ; on peut supposer que l'employé pourra, par exemple, retirer de l'argent d'un compte, chercher à savoir quel est son solde, déterminer s'il est créditeur ou non, et connaître le nom de son titulaire. Un employé de rang supérieur – disons le directeur de l'agence – pourrait en outre créer un nouveau compte, ou ajouter de l'argent à un compte existant ; ceci correspond à une autre spécification fonctionnelle, et nous aurons l'occasion d'y revenir.

La tâche à laquelle nous sommes confrontés est donc de définir la notion de COMPTE de façon qu'on puisse ensuite nommer et créer des objets de type COMPTE comme on manipule des objets de type TEXTE ou REEL, c'est-à-dire grâce à des propriétés abstraites et non pas à des informations sur les adresses où ils sont rangés ou sur le nombre de bits exigé par leur représentation.

Définir une structure de données équivaut donc à définir un nouveau *type*, et ses propriétés. Un type défini de cette façon est appelé **type abstrait de données**, ou plus simplement **type abstrait**.

La même démarche pourrait être utilisée pour définir abstraitement un type connu. Ainsi, le type "entier positif ou nul" ou ENTIER NATUREL, est défini comme un ensemble d'éléments avec certaines opérations de base : l'addition, qui associe à deux éléments de cet ensemble leur somme $x + y$; la soustraction ; l'opération de comparaison, notée $\leqslant$, qui associe à deux éléments une valeur de type LOGIQUE, vrai ou faux. Le type ENTIER NATUREL peut être entièrement défini par la donnée d'un certain nombre de fonctions de ce genre, et d'un certain nombre de propriétés comme :

$x + y = y + x$ (commutativité)

$x \leqslant y \Rightarrow (y - x) + x = y$ (soustraction et addition sont des opérations inverses l'une de l'autre)

etc. La systématisation de cette démarche conduit en mathématiques à des définitions axiomatiques (axiomes de Peano pour les entiers). Ici, nous la généraliserons pour définir les structures de données, ou types, de façon externe.

Pour revenir à notre "compte en banque" : le type abstrait COMPTE sera défini à l'aide d'un certain nombre d'opérations, que nous déclarerons à l'aide d'une notation semblable à celle des sous-programmes :

- nom-du-titulaire : COMPTE → TEXTE

(opération définie sur des objets de type COMPTE, et donnant comme résultat un TEXTE, le nom du titulaire du compte) ;

- solde : COMPTE → ENTIER

(opération permettant de connaître le solde d'un compte. Nous supposons pour simplifier les sommes exprimées en centimes, donc entières).

- créditeur : COMPTE → LOGIQUE

(opération permettant de savoir si un compte est créditeur, en un sens qui reste à préciser).

- retirer : COMPTE x ENTIER NATUREL → COMPTE

(opération donnant le nouveau compte obtenu en retirant à un compte une somme supposée positive).

Les propriétés suivantes sont vraies sur ces opérations ; pour tout compte c et tout entier naturel x :

(C_1) créditeur (c) ⇔ (solde (c) ⩾ 0)

(C_2) solde(retirer(c, x)) = solde (c) − x
{l'opération "retirer" fait bien ce que dit son nom}

(C_3) nom-du-titulaire (retirer (c, x)) = nom-du-titulaire (c)
{l'opération "retirer de l'argent à un compte" ne change pas le nom de son titulaire !}

Cette dernière relation montre qu'on est parfois obligé, lorsqu'on spécifie fonctionnellement une structure de données, de préciser des propriétés auxquelles on ne penserait pas nécessairement de prime abord.

Les opérations intervenant dans la spécification fonctionnelle d'une structure de données, ou type abstrait, T , sont de la forme :

$$f : X_1 \times X_2 \times \ldots \times X_n \rightarrow Y_1 \times \ldots \times Y_m$$

où les X_i et Y_i sont des types abstraits ; au moins l'un d'entre eux doit être T. Nous distinguerons :

a) Les *fonctions d'accès* qui sont telles que T apparaisse seulement à gauche de la flèche : elles donnent des valeurs caractérisant les objets de type T. Exemples : nom-du-titulaire, solde, créditeur.

b) Les *fonctions de modification* qui sont telles que T apparaisse à gauche et à droite de la flèche. Elles permettent de créer de nouveaux objets de type T à partir d'objets de ce type déjà créés (et d'autres éléments éventuels). Exemple : retirer.

c) Les *fonctions de création* qui sont telles que T apparaisse seulement à droite. Elles permettent de créer des éléments de type T à partir d'éléments d'autres types (ou à partir d'aucun élément ; il n'y aura rien alors à gauche de la flèche). Nous n'en avons pas encore rencontré.

La spécification fonctionnelle permet de traiter une structure de données comme un objet abstrait, sans s'occuper de problèmes de réalisation concrète. Nous pourrons dans la suite de ce chapitre caractériser complètement des structures comme les "piles", les "files", les tableaux etc. sans aucune hypothèse sur leur décomposition en objets de base ni a fortiori sur leur représentation en machine.

Dans la démonstration de certaines propriétés, il sera parfois nécessaire de supposer que les seules opérations autorisées sur ces structures sont celles qui interviennent dans la spécification fonctionnelle. Soit par exemple un type T tel qu'une certaine propriété P soit vraie sur tous les objets résultant des *fonctions de création* intervenant dans la spécification de T. Si toute *fonction de modification* f appliquée à tout objet t de type T laisse invariante la propriété P, c'est-à-dire si

$P(t) \Rightarrow P(f(t, \ldots))$

nous en déduirons que P est vraie pour tous les objets de type T. Cette méthode de démonstration se justifie parce que les spécifications fonctionnelles données décrivent complètement les structures.

Une objection peut être présentée à propos des *fonctions de modification* : la fonction retirer, par exemple, telle que nous l'avons présentée, n'est pas très élégante. Pourquoi mettre en jeu deux comptes, l'ancien et le nouveau ? On sait bien que, dans la réalité, le compte qui résulte de l'opération "retirer" est le même que le compte initial, dont on a simplement changé le solde ; en termes informatiques, retirer a pour type :

retirer : COMPTE x ENTIER NATUREL → VIDE

où le premier argument est une donnée modifiée. Mais avec une telle représentation, il devient difficile d'énoncer précisément les propriétés des opérations. Dans la pratique, le sous-programme qui représentera retirer pourra ne pas donner de résultat, mais modifier son premier argument, considéré comme donnée modifiée, pour peu que les relations (C_2) et (C_3) soient vérifiées entre les valeurs de tout argument réel avant et après l'appel de ce sous-programme. Nous avons vu au chapitre IV qu'un argument de type donnée modifiée est toujours décomposable en une donnée et un résultat.

Notons aussi qu'on pourrait ajouter la condition

nom-du-titulaire (c) = nom-du-titulaire (c') ⇒ c = c'

(pour tous comptes c et c') afin de restreindre l'emploi de retirer à des instructions de la forme :

c ← retirer (c, x)

La spécification fonctionnelle ci-dessus est volontairement restreinte : elle ne permet que quelques manipulations bien précises sur des comptes existants : leur retirer de l'argent, connaître le nom de leur titulaire, leur solde, ou savoir s'ils sont créditeurs. En particulier, elle n'autorise pas la création de nouveaux comptes, ni le dépôt de sommes positives ; ces opérations pourraient être réser-

vées à des employés de niveau hiérarchique plus élevé, et donc faire l'objet d'une *autre* spécification fonctionnelle plus complète. Dans cette nouvelle spécification, la notion correspondant à celle de "créditeur" pourrait être plus raffinée, et faire intervenir d'autres éléments que le caractère positif ou négatif du solde du client : l'histoire antérieure de son compte, l'état de ses crédits et de ses emprunts (on ne prête qu'aux riches), etc. Si cela est possible, on s'efforcera d'utiliser une même *description logique* (paragraphe suivant) pour ces différentes définitions fonctionnelles.

V.1.3 Description logique ; juxtaposition ; séparation de cas ; énumération

La nature fonctionnelle d'une structure de données étant fixée, on est amené à en trouver une représentation en termes d'éléments de base et de relations entre ces éléments.

Définir une structure de données revient, nous l'avons vu, à créer un nouveau type ; jusqu'ici, nous avons montré comment un tel type était décrit abstraitement par les opérations associées. Pour pouvoir créer et manipuler des objets du type nouveau, il faut maintenant donner une définition de ce type en fonction de types précédemment connus.

Ces types "connus" comprennent des types précédemment définis par les mêmes méthodes, et des *types élémentaires* qu'on suppose prédéfinis. Dans l'exemple du COMPTE en banque, les types "élémentaires" pourraient être :

ENTIER
ENTIER NATUREL
LOGIQUE {type à deux éléments : vrai et faux}
TEXTE {les éléments de ce type sont des suites finies de caractères}

Le choix des types élémentaires, il importe de le noter, est en partie arbitraire. Tel type considéré comme élémentaire dans un certain problème sera pour une autre application une structure de données complexe, décomposable en éléments d'un type de plus bas niveau. Ainsi, pour qui veut mettre au point les circuits d'une machine, ENTIER ne sera pas un type élémentaire, mais un type complexe défini en fonction de types élémentaires comme BIT ou OCTET ; de même, TEXTE sera considéré, selon le cas, comme un type élémentaire ou comme une structure de données obtenue à partir du type CARACTERE.

Comment décrire un COMPTE au niveau logique ? Nous pouvons utiliser la décomposition suivante :

```
                ⎧ valeur : ENTIER
                ⎪                ⎧ nom : TEXTE
                ⎪ titulaire :    ⎪ âge  : ENTIER NATUREL
COMPTE =        ⎨                ⎩ adresse : TEXTE
                ⎪                ⎧ jour : ENTIER NATUREL
                ⎩ date-de-création : ⎨ mois : ENTIER NATUREL
                                 ⎩ an   : ENTIER NATUREL
```

Cette description (dont il est clair intuitivement qu'elle nous fournit *plus* d'informations qu'il est nécessaire pour la spécification fonctionnelle donnée) indique que tout COMPTE sera caractérisé par un ENTIER (la somme qui lui est attachée à un certain moment), trois informations (un TEXTE, un ENTIER NATUREL et un autre TEXTE) décrivant son titulaire, et trois informations

donnant sa date de création. Chacune des informations entrant dans la description de COMPTE a un nom (comme valeur, titulaire, an, etc.) ; nous pourrions parler, si c est un compte, des différents éléments qui le composent, ou "composants", à l'aide d'une notation semblable à celle qui permet de désigner les éléments d'un tableau :

valeur (c)
âge (titulaire (c))

Pour pouvoir noter de façon linéaire la description ci-dessus, nous écrirons plutôt, à l'aide de parenthèses et du séparateur "point-virgule" :

```
type COMPTE = (valeur : ENTIER ;
               titulaire : (nom : TEXTE ;
                 âge : ENTIER NATUREL ; adresse : TEXTE) ;
               date-de-création : (jour : ENTIER NATUREL ;
               mois : ENTIER NATUREL ; an : ENTIER NATUREL))
```

Comme toute notation imbriquée, les deux notations ci-dessus (proches de ce qu'offre PL/1) sont assez lourdes et les déclarations de types difficiles à suivre. Nous préférerons donc en général nous limiter à un seul niveau de parenthèses en définissant des types intermédiaires PERSONNE et DATE :

```
type PERSONNE = (nom : TEXTE ; âge : ENTIER NATUREL ; adresse : TEXTE)
                {une personne est caractérisée par deux TEXTES et un ENTIER NATUREL}

type DATE = (jour : ENTIER NATUREL ; mois : ENTIER NATUREL ;
             an : ENTIER NATUREL)
            {une date comprend trois entiers positifs ou nuls}

type COMPTE = (valeur : ENTIER ; titulaire : PERSONNE ; date-de-création :
               DATE)
```

c étant un compte, on pourra comme précédemment utiliser :

valeur (c) {qui joue le rôle d'une variable de type ENTIER}
date-de-création (c) {de type DATE}
jour (date-de-création (c)) {de type ENTIER NATUREL}

Nous venons donc de voir un premier moyen de définir un type en fonction d'autres types, que nous pouvons appeler la **juxtaposition**, et que nous avons symbolisé par le point-virgule : pour obtenir un élément de type COMPTE, il faut rassembler une PERSONNE (plus exactement, sa représentation "bancaire"!), un ENTIER et une DATE (figure V.2).

Figure V.2. – Compte

La juxtaposition n'est pas la seule opération de composition possible. Affectons notre notion de COMPTE : le "titulaire" d'un COMPTE pourrait être traité

différemment suivant qu'il s'agit d'une PERSONNE ou d'une société. Le type SOCIETE sera par exemple défini par :

type SOCIETE = (raison-sociale : TEXTE ; capital : ENTIER NATUREL ; siège social : TEXTE)

(le capital d'une société fournissant à un banquier la même présomption de solvabilité que l'âge d'une personne). Nous définirons alors COMPTE par :

type COMPTE = (valeur : ENTIER ; titulaire : (PERSONNE|SOCIETE) ; date-de-création : DATE)

où la barre verticale (lue "*ou*") indique la séparation de cas : le second composant d'un COMPTE sera soit de type PERSONNE, soit de type SOCIETE. Là encore, nous pouvons nous limiter à une structure plus simple en nommant un type intermédiaire :

type PERSONNE-MORALE-OU-PHYSIQUE = (PERSONNE|SOCIETE)

type COMPTE = (valeur : ENTIER ; titulaire : PERSONNE-MORALE-OU-PHYSIQUE ; date de création : DATE)

Un algorithme accédant aux composants d'un compte devra pouvoir tester si le composant "titulaire" d'un objet de type COMPTE est une PERSONNE ou une SOCIETE. Nous noterons cette opération à l'aide de l'instruction décider entre . . . du chapitre III et de l'opérateur est un (ou est une) :

décider entre
| titulaire (c) est une PERSONNE : action 1,
| titulaire (c) est une SOCIETE : action 2 ①

En action 1 (mais là seulement) on a accès aux champs nom (titulaire (c)), âge (titulaire (c)), adresse (titulaire (c)) ;

En action 2 (mais là seulement) on a accès à raison-sociale (titulaire (c)), etc. On imposera que *tout algorithme manipulant le composant* "titulaire" *d'un compte ait la structure de* ① *ci-dessus*, ou une structure équivalente.

Il nous reste à voir une troisième règle de composition de types qui, à dire vrai, est en fait un cas particulier de la "séparation de cas" : il s'agit de l'énumération, qui nous permet de définir un type pour lequel le nombre de valeurs possibles est fini (comme le type LOGIQUE) en donnant la liste de ces valeurs. Ainsi, plutôt que de définir le "mois" d'une date comme un ENTIER NATUREL, ce qui nous oblige à des contrôles de validité chaque fois qu'une valeur de "mois" est fournie, nous pourrions définir un type MOIS par énumération, en notant :

type MOIS = ("janvier", "février", "mars", "avril", "mai", "juin", "juillet", "aout", "septembre", "octobre", "novembre", "décembre")

Les virgules indiquent l'énumération. Plus généralement, un type sera défini par *énumération* comme suit :

type T = (const 1, const 2, . . . , const n)

où const 1, const 2, . . . const n sont des constantes.

Nous exigerons que toutes ces constantes aient le même type. Cette restriction ne nous gênera pas dans cet ouvrage. Elle peut sembler inélégante : dans l'exemple ci-dessus, nous sommes obligés de manipuler "janvier" comme un TEXTE alors qu'il s'agit d'un *nom* et que les caractères j, a, n, etc. ne nous intéressent pas par eux-mêmes. Le problème, cependant, est que dans les langages comme PASCAL où nous pourrions écrire (avec un syntaxe légèrement différente) :

type MOIS = (janvier, ... , décembre)
ou *type FEU = (vert, orange, rouge)*

le statut de mots comme *janvier* ou *vert* n'est pas très clair : ce ne sont pas des constantes au sens habituel, car les constantes entières, réelles, etc. ont un type qui se déduit immédiatement de leur mode d'écriture, alors que rien dans *vert* ne rappelle le type *FEU* ; ce ne sont pas non plus des identificateurs puisqu'il n'y a pas de variables associées ; pourtant, aucune variable ne peut être déclarée avec pour identificateur *vert* ou *janvier* après les déclarations ci-dessus.

La *déclaration* complète du type COMPTE s'écrit, d'après ce qui précède :

type PERSONNE = (nom : TEXTE ; âge : ENTIER NATUREL ; adresse : TEXTE)

type SOCIETE = (raison-sociale : TEXTE ; capital : ENTIER NATUREL ; siège social : TEXTE)

type JOUR-DU-MOIS = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31)

type MOIS = ("janvier", "février", "mars", "avril", "mai", "juin", "juillet", "août", "septembre", "octobre", "novembre", "décembre")

type PERSONNE-MORALE-OU-PHYSIQUE = (PERSONNE|SOCIETE)

type DATE = (jour : JOUR-DU-MOIS ; mois : MOIS ; an : ENTIER NATUREL)[1]

type COMPTE = (valeur : ENTIER ; titulaire : PERSONNE-MORALE-OU-PHYSIQUE ; date-de-création : DATE)

Nous pouvons maintenant manipuler des objets de type COMPTE :

variables c, c', compte-dupont : COMPTE

Notons que la démarche adoptée ici est semblable à celle du chapitre III ("Structures de contrôle"). De même que nous avions alors indiqué comment on construit des **programmes** complexes à partir d'*actions élémentaires* et de *règles de composition* (enchaînement, répétition, choix), nous appliquons ici un principe similaire aux **données**, en construisant des **types** complexes à partir de *types élémentaires* et de règles de composition, la juxtaposition, la séparation de cas et l'énumération.

Nous nous sommes donnés le moyen de créer de nouveaux types et de nommer des *variables* de ces types. Nous devons aussi pouvoir manipuler des constantes d'un type complexe. Pour cela, il suffit d'indiquer le type de la constante et les valeurs de ses différents composants. Par exemple, voici deux constantes de types respectifs DATE et PERSONNE-MORALE-OU-PHYSIQUE :

DATE (25, "mars", 1976)

PERSONNE-MORALE-OU-PHYSIQUE (PERSONNE ("DUPONT", 25, "20 RUE DES ORMES"))

(1) Le fait que nous ayons donné le même nom (mois) à un composant de structure et à un type ne devrait pas entraîner de confusion. Sans prendre position quant à ce problème de syntaxe, rappelons que nous développons une notation simple et non un langage de programmation.

Ainsi notée, une constante de type complexe ressemble un peu à un appel de sous-programme : dans le cas d'un type défini par juxtaposition, le "nom du sous-programme" est remplacé par le nom du type, et les "arguments réels" par les valeurs des différents composants figurant dans la constante (voir la constante de type DATE ci-dessus) ; pour un type défini par séparation de cas (comme PERSONNE-MORALE-OU-PHYSIQUE) on a en fait un "double appel de sous-programme" précisant le cas choisi.

En fait, la notion de "constante" est moins claire pour les types complexes que ne peuvent le laisser supposer les exemples ci-dessus.

On aimerait en effet pouvoir écrire :

variables x : PERSONNE, c : COMPTE, y : PERSONNE-MORALE-OU-PHYSIQUE ;
x ← PERSONNE (" DUPONT ", 25, " 20 RUE DES ORMES ") ;
y ← PERSONNE-MORALE-OU-PHYSIQUE (x)
c ← COMPTE (30000, y, DATE (25, "mars", 1976))

Le deuxième composant de l'expression affectée à c se réfère à la variable x ; on ne peut donc véritablement dire que l'expression est une constante ; pour désigner les "valeurs" structurées que peut prendre une donnée de type complexe, on parlera plutôt d'*enregistrement*. Notons que l'analogie avec les sous-programmes peut se poursuivre assez loin : le deuxième composant de l'enregistrement affecté à c pourrait être implanté "par valeur", c'est-à-dire contenir une *référence* à l'enregistrement désigné par y. Mais ceci est du niveau de la réalisation physique, et nous y reviendrons.

La *description logique* de la structure COMPTE n'est évidemment complète que si l'on fournit une description des *opérations* de la définition fonctionnelle sous forme de *sous-programmes* manipulant des objets de type COMPTE et vérifiant les propriétés postulées dans cette définition.

Une description logique complète de la structure COMPTE est fournie à la figure V.3. Notons que les sous-programmes qu'elle contient permettent d'en faire plus qu'il n'est requis par la définition fonctionnelle du paragraphe V.1.2 ; ils pourraient donc être utilisés pour remplir des spécifications fonctionnelles plus exigeantes.

La figure V.4 est une autre description logique possible pour réaliser la même définition fonctionnelle : elle suppose qu'on garde en mémoire 10 opérations récentes, plutôt que de mettre à jour le "solde" à chaque opération "retirer" ou "ajouter". Dans la pratique, le tableau OPERATIONS serait plutôt géré comme une *file* (voir V.5) ; nous avons seulement voulu montrer qu'à une même définition fonctionnelle pouvaient correspondre des réalisations logiques très différentes.


```
type PERSONNE          = (nom : TEXTE ; âge : ENTIER NATUREL ; adresse :
                          TEXTE) ;
type SOCIETE           = (raison-sociale : TEXTE ; capital : ENTIER NATU-
                          REL ; siège-social : TEXTE) ;
type JOUR-DU-MOIS      = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17,
                          18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31} ;
type MOIS              = {"janvier", "février", "mars", "avril", "mai", "juin",
                          "juillet", "août", "septembre", "octobre", "novembre",
                          "décembre"} ;
type PERSONNE-MORALE-OU-PHYSIQUE = (PERSONNE|SOCIETE) ;
type DATE              = (jour : JOUR-DU-MOIS ; mois : MOIS ; an : ENTIER
                          NATUREL) ;
type COMPTE            = (valeur : ENTIER ; titulaire : PERSONNE-MORALE-
                          OU-PHYSIQUE ; date-de-création : DATE) ;

programme créer-compte : COMPTE
          (données p : PERSONNE-MORALE-OU-PHYSIQUE,
                   d : DATE,
                   v : ENTIER NATUREL)
    | créer-compte ← COMPTE (v, p, d)

programme ajouter : COMPTE
          (données c : COMPTE, x : ENTIER)
    | {ajouter x, positif, négatif ou nul}
    | ajouter ← c ; valeur (ajouter) ← valeur (ajouter) + x

programme retirer : COMPTE
          (données c : COMPTE, x : ENTIER NATUREL)
    | retirer ← ajouter (c, – x)

programme nom-du-titulaire : TEXTE (donnée c : COMPTE)
    | décider entre
    |   | titulaire (c) est une PERSONNE :
    |   |         nom-du-titulaire ← nom (titulaire (c)),
    |   | titulaire (c) est une SOCIETE :
    |   |         nom-du-titulaire ← raison-sociale (titulaire (c))

programme solde : ENTIER (donnée c : COMPTE)
    | solde ← valeur (c)

programme créditeur : LOGIQUE (donnée c : COMPTE)
    | créditeur ← (solde (c) ≥ 0)
```

Figure V.3. – Une description logique de COMPTE

```
type COMPTE = (nombre-d'-opérations : ENTIER NATUREL ;
               tableau opération [1 : 10] : ENTIER ; nom : TEXTE ;
programme créer-compte : COMPTE
          (données n : TEXTE,
                   d : DATE,
                   v : ENTIER)
    | tableau t [1 : 10] : ENTIER ;
    | t [1] ← v ;
    | créer-compte ← COMPTE (1, t, n)
programme retirer (donnée modifiée c : COMPTE ;
                   donnée x : ENTIER NATUREL)
    | nombre-d'-opérations (c) ← nombre-d'-opérations (c) + 1 ;
    | si nombre-d'-opérations (c) = 11 alors
    |       | pour i variant de 2 à 10 répéter
    |       |       | opération (c) [i] ← opération (c) [i]
    |       |       |                   + opération (c) [i]
    |       | nombre-d'-opérations (c) ← 2
    | opérations (c) [nombre-d'-opérations (c)] ← – x
programme solde : ENTIER (donnée c : COMPTE)
    | solde ← 0 ;
    | pour i variant de 1 à nombre-d'-opérations (c) répéter
    |       |solde ← solde + opération (c) [i]
{programmes nom-du-titulaire, créditeur comme sur la figure V.3}
```

Figure V.4. – Une autre description logique de COMPTE

Définitions récursives

Une possibilité extrêmement utile dans les descriptions logiques, que nous n'avons pas mentionnée jusqu'ici, est celle de *définition récursive* : dans la définition d'une structure T obtenue par juxtaposition, l'un des composants de T peut avoir pour type T lui-même.

Ainsi, nous pourrions modifier (encore) notre type COMPTE en ajoutant à tout COMPTE un composant appelé "garantie", qui est un autre compte (servant par exemple de garantie en cas de catastrophe). Nous écrirons :

type COMPTE = (valeur : entier ; titulaire : PERSONNE-MORALE-OU-PHYSIQUE ; date-de-création : DATE ; garantie : COMPTE)

c étant un COMPTE,

{ garantie (c) sera un COMPTE,
  date-de-création (garantie (c)) sera une DATE,
  an (date-de-création (garantie (c))) sera un ENTIER NATUREL, etc.

On appelle **récursion** la présence dans la définition d'un objet d'une référence à l'objet lui-même, et *récursivité* la propriété correspondante. La récursivité des *sous-programmes* sera étudiée en détail au chapitre VI.

La récursion peut être *indirecte* : un type T_1 est défini en fonction d'un type T_2, qui est défini en fonction de T_1 ; le nombre de niveaux pourrait être quelconque.

> En fait, la récursivité indirecte qui aura donné le plus de mal aux auteurs de ce livre est . . . la récursivité de la notion de récursion ! Nous n'avons pu éviter d'inclure dans le présent chapitre un certain nombre de sous-programmes récursifs, qui s'introduisent naturellement à propos des structures de données récursives. Plus généralement, il est clair qu'on ne peut parler de récursion (chapitre VI) sans avoir exposé auparavant les structures de données (ce chapitre) ; mais comment peut-on développer décemment les structures de données avant d'avoir analysé le problème de la récursion ? En lisant les paragraphes suivants, le lecteur jugera si nous avons su appliquer les techniques d'implantation physique des structures de données récursives (pointeurs, références mutuelles . . .).

La définition suivante résume ce qui a été dit sur la *description logique* d'une structure de données :

Définition

Une **description logique** d'une structure de données, dont la spécification fonctionnelle est connue, comprend :

a) La définition d'un **type** par une "identité logique" faisant intervenir des types définis par ailleurs et éventuellement le nouveau type lui-même (récursion), et utilisant les opérations de juxtaposition, séparation de cas et énumération.

b) Un certain nombre de **sous-programmes** opérant sur les éléments du type ainsi défini. A chaque opération de la définition fonctionnelle doit être associé un de ces sous-programmes, vérifiant les propriétés correspondantes. Les sous-programmes peuvent manipuler les différents composants du type défini en a) ; si un composant p est défini par *séparation de cas*, c'est-à-dire si son type est de la forme $(T_1 | T_2 | \ldots | T_n)$, toute partie d'un sous-programme accédant au composant p d'un objet x doit être de la forme [1]

décider entre
- p(x) est un T_1 : . . . {manipulation d'objets de type T_1},
- p(x) est un T_2 : . . . {manipulation d'objets de type T_2},
- ---
- p(x) est un T_n : . . . {manipulation d'objets de type T_n}

Mentionnons enfin, pour terminer l'exposé de la description logique d'une structure, qu'il est souvent utile de définir une structure en faisant intervenir dans une *séparation de cas* le type spécial VIDE (*NIL* dans le langage *LISP*, *NULL* en ALGOL W ou PL/1, etc.) qui permet d'"arrêter" une structure récursive. Par exemple, si nous supposons COMPTE défini comme à la figure 4 (sans récursivité), nous pouvons vouloir définir une structure de données permettant de manipuler *plusieurs* comptes.

Nous écrirons :

type LISTE-DE-COMPTES = (VIDE | LISTE-COMPTES-NON-VIDE)
type LISTE-COMPTES-NON-VIDE = (COMPTE ; LISTE-DE-COMPTES)

C'est-à-dire qu'une LISTE-DE-COMPTES est soit VIDE, soit un COMPTE juxtaposé à une autre LISTE-DE-COMPTES. Il s'agit d'une définition récursive ; elle revient à exprimer qu'une LISTE-DE-COMPTES est la juxtaposition d'un nombre quelconque, éventuellement nul, de COMPTES, et d'un élément vide.

(1) Lorsqu'il n'y aura pas d'ambiguïté possible, nous utiliserons des alternatives si . . . alors . . . sinon . . . plutôt que des décider entre . . . si l'écriture s'en trouve allégée.

Les définitions récursives de ce dernier type, obtenues à partir d'une "séparation de cas" où l'un des choix possibles est VIDE et l'autre récursif, fournissent des structures de données en général plus faciles à manipuler que les définitions récursives non restreintes comme celle de la page 220 où rien ne permet de garantir qu'on évite une structure *cyclique*, c'est-à-dire :

garantie $(c_1) = c_2$, garantie $(c_2) = c_3$, ..., garantie $(c_{m-1}) = c_m$
garantie $(c_m) = c_1$

Avec la définition donnée pour LISTE-DE-COMPTES il est par contre possible, moyennant certaines hypothèses sur les opérations de construction effectuées, de garantir que l'on ne peut pas construire de listes cycliques. Bien que les structures cycliques soient parfois indispensables, la certitude qu'une structure n'est pas cyclique simplifie en général considérablement l'étude de ses propriétés.

V.1.4 Représentation physique. Notion de pointeur

Le paragraphe précédent nous a montré que, pour définir une structure de données, on partait de données de types élémentaires et de relations entre elles. Nous supposons connue la représentation des types élémentaires comme ENTIER, CARACTERE etc. ; nous nous intéresserons donc dans ce paragraphe à la représentation des relations : juxtaposition, séparation de cas, énumération.

Les techniques de représentation décrites dans ce paragraphe sont, suivant les cas, du ressort du programmeur ou de celui d'un compilateur. La confusion qui peut en résulter est difficile à éviter : selon le langage qu'il utilise, le programmeur peut ou non se reposer sur le compilateur pour régler un certain nombre de problèmes de représentation. De ce point de vue, le programmeur en ALGOL W ou PL/1 est incontestablement favorisé par rapport à l'utilisateur de FORTRAN. Dans l'un et l'autre cas, cependant, l'étude des méthodes de représentation est profitable pour une compréhension approfondie des algorithmes.

V.1.4.1 Séparation de cas, énumération

Commençons par ce qui ne fait pas vraiment problème, la séparation de cas et l'énumération. Soit un type T défini par **séparation de cas:**

type $T = (T_1 | T_2 | \ldots | T_n)$

où $T_1, T_2, \ldots T_n$ sont des types connus. Un objet de type T sera représenté par deux éléments :

- un *indicateur de type* : un entier i, $1 \leqslant i \leqslant n$, indiquant à quel T_i appartient l'objet considéré ;
- un objet de type T_i.

Pour représenter l'indicateur de type, il suffit de $\lceil \log_2 n \rceil$ bits (où $\lceil x \rceil$ représente le plus petit entier supérieur ou égal à x).

Les types T_i, $i = 1, \ldots n$ pourront avoir des représentations demandant des tailles de mémoire variables. Ce problème sera examiné un peu plus loin à propos des structures de données récursives. La solution la plus élémentaire consiste à utiliser systématiquement la taille maximale.

- Un objet d'un type T défini par **énumération :**

type $T = (c_1, c_2, \ldots, c_m)$

sera tout simplement représenté par un indicateur : un entier compris entre 1 et m, qu'on peut mettre en place sur $\lceil \log_2 m \rceil$ bits.

V.1.4.2 Juxtaposition

La **juxtaposition** sans récursivité directe ni indirecte ne soulève guère de difficulté : si T est défini par :

type $T = (x_1 : T_1 ; x_2 : T_2 ; \ldots x_n : T_n)$

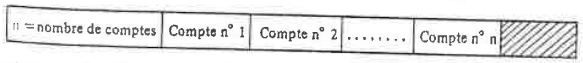
et si la taille de mémoire requise par des objets de types $T_1, T_2, \ldots T_n$ est connue, la taille de mémoire requise pour représenter un objet de type T est fixe : c'est-la somme des précédentes. Ainsi, sur IBM 360/370, (ENTIER = 4 octets, DOUBLE PRECISION = 8 octets, CARACTERE = 1 octet), 13 octets suffisent pour représenter un objet de type :

type EXEMPLE = (e : ENTIER ; d : DOUBLE PRECISION ;
c : CARACTERE)

Supposons par contre que nous voulions manipuler des listes de comptes dans une banque. Nous prenons COMPTE défini comme précédemment, sans récursivité (donc occupant une place fixe) ; nous définissons :

type LISTE-DE-COMPTES = (VIDE | LISTE-NON-VIDE)
type LISTE-NON-VIDE = (COMPTE ; LISTE-DE-COMPTES)

Une LISTE-DE-COMPTES est un ensemble de comptes. Or la représentation en ordinateur de tels ensembles se heurte au problème de leur taille variable : rien ne nous permet de fixer une limite a priori. On peut toujours certes utiliser un tableau de dimension plus ou moins arbitraire et aboutir au schéma



où le premier élément, un entier n, indique le nombre de blocs effectivement utilisés dans la zone allouée au tableau. Avec cette solution, cependant, on ne peut garantir qu'on ne débordera pas de cette zone si le nombre de COMPTES devient trop grand, ni qu'on n'aura pas réservé de la place inutile pendant la majeure partie du temps.

Au lieu de cela, on peut concevoir la liste

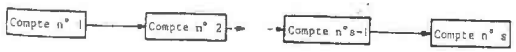
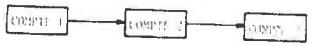


Figure V.5 — Liste de comptes

comme une suite de blocs d'information *multiples* mais de taille *fixe*, reliés entre eux ; or une suite de blocs de dimension fixe et en nombre variable est souvent plus facile à représenter que l'inverse. Pour pouvoir représenter les relations entre ces différents blocs, symbolisées par des flèches sur la figure ci-dessus, il suffira d'adjoindre à chacun d'eux une information repérant le suivant : son *adresse*.

Une telle adresse employée dans la représentation d'une structure de données est appelée **pointeur** (à la suite d'une traduction peu appropriée, mais passée dans les habitudes, du mot anglais *pointer*, qui signifie "indicateur" ou "désignateur"). On dit aussi *référence*. Nous dirons d'un pointeur qu'il *désigne* une certaine donnée, pour exprimer que sa valeur est l'adresse de cette donnée.

Nous avons vu qu'un élément spécial, que nous avons noté VIDE, jouait un rôle particulièrement important dans les structures de données récursives. En particulier, il pourra servir de marqueur de fin de liste, de telle sorte qu'une liste (plus précisément une "liste linéaire" ; cf. V.6)



sera souvent représentée comme :

où la "masse" représente un "pointeur vers VIDE". On représente en machine un tel pointeur par une valeur spéciale qui ne peut être une adresse, par exemple une valeur négative.

Ainsi, en supposant qu'un objet de type COMPTE prend 6 éléments de mémoire et qu'une adresse en prend un, on pourrait utiliser la représentation suivante :

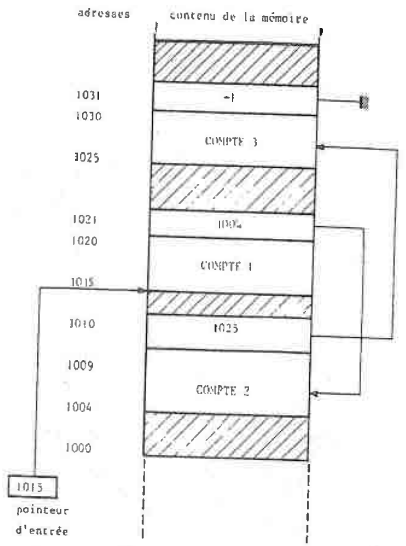


Figure V.6. – Représentation d'une liste linéaire (de comptes)

La liste, en tant qu'entité nommable, sera également accessible grâce à un pointeur (à valeur 1015 dans l'exemple ci-dessus). Cependant, alors qu'un pointeur représentant une relation "interne" entre éléments d'une structure de données (par exemple, les blocs d'une liste) est rangé contigûment à un élément, un pointeur fournissant un point d'entrée dans une structure devra être accessible au programme et donc rangé en un endroit connu de celui-ci : le pointeur est associé à une variable du programme, et demande un emplacement de mémoire de taille fixe, celle qui est nécessaire à la représentation d'une adresse ; cette taille est connue dès la compilation.

On perçoit donc une solution au problème de la gestion des objets de types demandant à l'exécution une taille de mémoire variable et non prévisible à la compilation ; ce problème concerne à la fois les structures de données récursives et, comme nous l'avons vu en V.1.4.1, les types obtenus par "séparation de cas" ; il apparaît dans les langages comme ALGOL W, PASCAL, PL/1, ALGOL 68 qui offrent ce genre de structures. Une solution possible est la suivante : l'objet de type complexe est considéré par le programme comme demandant une taille mémoire *fixe*, celle qu'exige la représentation d'un pointeur et éventuellement d'un indicateur (s'il y a séparation de cas). L'emplacement correspondant peut donc être géré de façon classique (allocation statique, piles, etc.), comme l'emplacement attribué à une variable de type entier, réel, etc. ; les pointeurs considérés désigneront des adresses d'une zone spéciale, appelée tas et uniquement consacrée aux structures de données spéciales, de taille non prévisible (figure V.7).

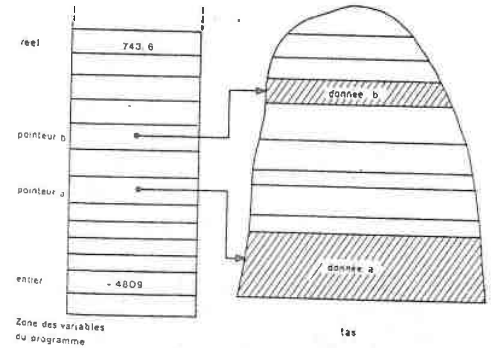


Figure V.7. – Utilisation d'un tas

La gestion d'un tas requiert des techniques particulières. Le problème essentiel est que certains éléments du tas deviennent inutiles à certains moments, lorsqu'ils ne sont plus désignés par aucun pointeur (les pointeurs qui les désignaient

auparavant ayant changé de valeur) ; on peut désirer récupérer la place qu'ils occupent. Il y a deux techniques fondamentales :

- associer à chaque élément du tas un entier dit **compteur de références**, donnant le nombre de pointeurs qui le désignent, et mis à jour à chaque opération ; si sa valeur devient nulle, on peut réutiliser l'emplacement attribué à l'élément ;
- laisser le tas se remplir jusqu'à ce qu'une demande de place ne puisse être satisfaite ; appeler alors un programme spécial, dit **ramasse-miettes**, qui explore toute les structures de données accessibles à partir du programme, pour déterminer quels sont les emplacements occupés, et par conséquent les emplacements libres, du tas (les Anglo-saxons, moins délicats, appellent cette opération *garbage collection*, ramassage d'ordures).

La première méthode, celle des compteurs de références, est assez rarement employée parce qu'elle exige un mécanisme de comptage assez lourd. La seconde pose également de nombreux problèmes, dont le moindre n'est pas que le programme ramasse-miettes doit se contenter d'une zone de mémoire forcément restreinte puisque, par définition, on l'appelle quand il n'y a presque plus de place ! Un phénomène d'émiettement de la mémoire, familièrement connu comme la "gruyérisation", se produit en outre après quelques appels du ramasse-miettes, rendant impossible l'allocation de "gros" blocs de mémoire même si la place totale demandée est disponible (figure V.8.). Il faut donc "recompacter" périodiquement la mémoire. Sur tous ces problèmes de gestion de la mémoire, nous renvoyons le lecteur à [Knuth 69]. On trouvera quelques notions complémentaires sur les ramasse-miettes et leur réalisation pratique au paragraphe VI.3.5.2.

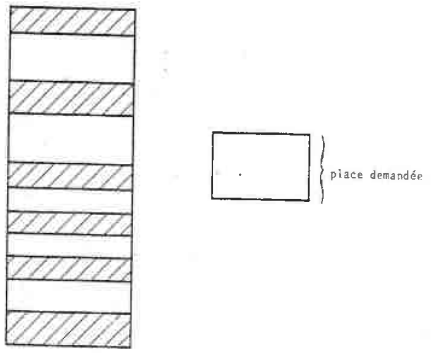


Figure V.8. – Emiettement de la mémoire

Notons que la technique du tas, qui sépare nettement la gestion des *variables* du programme et celle des *structures de données* permet en général d'éviter

le problème dit de la **référence folle** : une "référence folle", dans un langage à structure de blocs dynamique comme ALGOL W ou PL/1 (IV.6.2), est une référence vers une zone de la mémoire qui a été désallouée du fait des règles de l'allocation dynamique. Ces problèmes seront évoqués de nouveau, et la discussion des techniques de gestion de la mémoire sera complétée, au paragraphe VI.3.4.3.

V.1.5 Discussion – Problèmes

Nous avons volontairement insisté sur la séparation entre les trois niveaux de description d'une structure de données : spécification fonctionnelle, description logique, représentation physique. Les confusions entre ces niveaux sont en effet fréquentes ; il nous paraît pourtant capital, dans la construction d'une structure de données, de bien distinguer :

- ce qu'on veut en faire ;
- la structure logique (algébrique) d'un ensemble qui permet de le faire ;
- la représentation en machine d'un tel ensemble.

La structure logique permettant de réaliser une description fonctionnelle donnée est rarement unique, de même que la représentation physique correspondante. Le choix sera guidé par plusieurs critères, en particulier le critère de *généralité* : on fera en sorte que, réciproquement, la même description logique puisse servir à plusieurs spécifications fonctionnelles ; nous en avons vu un exemple à propos des "comptes en banque".

Nous avons présenté la construction d'une structure de données comme une démarche *descendante* : on part des concepts pour aller vers les objets, on examine ce qu'on *veut* faire avant de chercher *comment* on le fera. La même démarche, appliquée à la construction des programmes, a été esquissée au chapitre III et nous y reviendrons au chapitre VIII. Il est clair que, dans la pratique, on suit rarement une démarche descendante pure : notre esprit procède souvent, qu'il le veuille ou non, par aller et retour, essai et erreur ; en outre, on ne "peut" pas toujours faire tout ce qu'on "veut", et les problèmes de représentation imposent parfois des retours en arrière déchirants sur les spécifications. Il faut ajouter que le programmeur peut être amené à tempérer sa démarche d'une composante "ascendante", lorsqu'il cherche en bon artisan à adapter son programme aux outils existants, qui ont pu être conçus pour un autre problème, mais sont assez généraux pour répondre à des spécifications plus larges : l'expérience de chacun montre que les spécifications, comme le monde, changent.

Le langage de programmation qui a ouvert la voie à la conception moderne des structures de données est sans aucun doute SIMULA 67, dont nous avons déjà parlé en IV.7. En SIMULA 67, une structure de données est définie par une **classe** ; une classe, entre autres choses – nous avons vu qu'elle pouvait jouer le rôle de "coprogramme" – combine : une déclaration de type semblable à notre niveau de "description logique", avec sa décomposition en champs élémentaires ; la déclaration de tous les sous-programmes habilités à opérer sur les objets de ce type, c'est-à-dire des réalisations des opérations de la spécification fonctionnelle ; enfin, une séquence d'initialisation devant être appliquée lors de la création de tout objet de la classe.

VIII.3.4 Notion de module

Pour construire un programme, donc, une stratégie de décomposition est nécessaire, et nous venons d'en voir des exemples. Il reste cependant à approfondir la question : quel doit être l'élément de base dans la construction d'un programme, l'unité de décomposition obtenue à chaque étape d'une conception ascendante, descendante, ou mixte ?

La réponse la plus simple et la plus classique est qu'il s'agit simplement d'un élément de programme défini par ses entrées, ses sorties, et éventuellement ses "données modifiées", c'est-à-dire, comme nous venons de le voir, d'un sous-programme – tout au moins conceptuellement, la réalisation pratique pouvant bien entendu être différente.

Cette solution n'est cependant pas le fin mot de l'histoire. En effet, la grande difficulté de l'organisation d'un projet de programmation tient souvent beaucoup moins aux rapports entre les éléments de *programme* qu'à la gestion et à la communication des *données*. Or, même si l'on définit rigoureusement les modes de communication de données entre unités de programmes comme nous l'avons vu ci-dessus, il subsiste un difficile problème de répartition des responsabilités : quelle unité de programme doit assurer la gestion de base d'un ensemble de données utilisées par plusieurs éléments ?

La prise de conscience du fait que ce problème est finalement le plus ardu conduit à un découpage du projet fondé sur la notion d'"unité de données" plutôt que sur celle d'"unité de programme".

Nous appellerons **module** un tel élément de base ; des termes voisins sont ceux de "classe" en SIMULA 67, "forme" en ALPHARD [Wulf 75], "grappe" (*cluster*) en CLU [Liskov 77] voir aussi LIS [Ichbiah 75], etc. Notons cependant que nous nous écartons d'un sens souvent attribué au mot "module", celui d'unité de programme aux interfaces bien définies ; les partisans de la "programmation modulaire" limitée à ce sens restreint insistent sur la nécessité d'un découpage lo-

gique en petites unités aux rapports bien définis, point que les lecteurs de ce ouvrage, nous l'espérons, considéreront comme une évidence, mais qui ne permet pas de savoir *comment* effectuer ce découpage.

Pour nous, la programmation modulaire sera fondamentalement une méthode de décomposition dont l'élément de base, ou module, est une *structure de données* (chapitre V), accessible de l'extérieur par le biais d'un ensemble de sous-programmes, et par eux seuls.

Reprenons l'exemple du compilateur. L'approche la plus simple consiste à considérer les différents éléments de programme, aux divers niveaux d'abstraction : le "compilateur" ; l'"analyseur syntaxique" ; l'"analyseur lexical" ; le programme de "gestion de la table des symboles" ; le "traducteur en forme polonaise postfixée" ; le "producteur de code" ; le "lecteur de caractères" ; etc. Or une telle décomposition pose de sérieux problèmes quant à l'accès aux données. Par exemple, l'analyseur syntaxique appelle répétitivement l'analyseur lexical pour connaître les éléments successifs d'une instruction. Or il doit pouvoir décrypter ces éléments, c'est-à-dire connaître leur type, leur position éventuelle dans la table des symboles, etc. Analyseur lexical et analyseur syntaxique partagent donc une structure de données assez complexe, celle qui permet de décrire les éléments lexicaux, ou "lexèmes", et leurs différentes propriétés. La difficulté de communication entre ces deux éléments de programme n'est pas due à la définition précise de leurs tâches – définition relativement triviale –, mais à la délimitation de leur droit d'accès à cette structure de données communes : il s'agit de préciser quels composants d'un lexème chacun d'entre eux peut *consulter*, quels composants il peut *modifier*, s'il peut *créer* un nouveau lexème, etc.

En programmation modulaire, on attaquera de front ces problèmes en prenant précisément comme unité de décomposition les structures de données de base (et non les programmes qui les manipulent) :

- le **caractère**, unité de base du processus de lecture (et non le programme de lecture) ;
- le **lexème**, unité de base du processus d'analyse lexicale (et non l'analyseur lexical) ;
- le **syntagme** (et non l'analyseur syntaxique) ;
- le **symbole** et la **table des symboles** ;
- l'**expression** en forme polonaise postfixée ;
- l'**instruction en langage machine** ;
- le **programme source** ;
- le **programme objet** ;

Chacun de ces éléments est au centre d'un module (figure VIII.3) comprenant en outre tous les *programmes d'accès*. Ces programmes d'accès sont les seuls moyens permettant de manipuler les éléments en question. Par exemple, le module "lexème" correspond, dans la terminologie du chapitre V, à la définition d'un nouveau type LEXEME, et des fonctions :

textelex : LEXEME → TEXTE

{ℓ étant un lexème, textelex (ℓ) est le texte formé de la chaîne des caractères composant ℓ}

typelex : LEXEME → ("constante entière", "identificateur", "opérateur", "constante texte", ...)

{ℓ étant un lexème, typelex (ℓ) peut valoir "constante entière", "identificateur", etc., selon ce que ℓ représente}

analex : → LEXEME

{à partir de l'"entrée", c'est-à-dire des caractères formant un programme, analex analyse les caractères constituant un lexème, et renvoie ce lexème comme résultat}

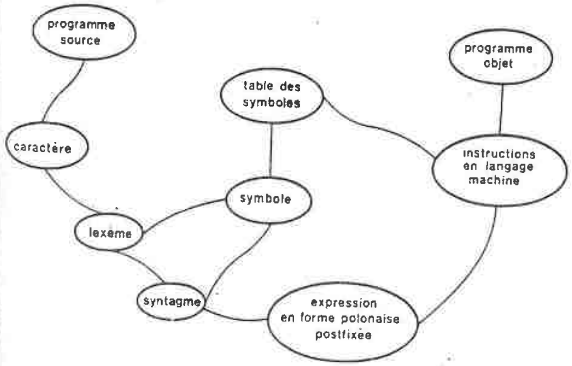


Figure VIII.3. – Modules d'un compilateur

Remarquons que, comme il est naturel, l'"analyseur lexical" réapparaît ici sous la forme d'analex, mais ce n'est plus qu'un des programmes composant le module lexème. L'hypothèse de la programmation modulaire est que le programme textelex joue un rôle tout aussi important, dans la construction du compilateur, qu'analex – même si l'écriture de textelex est comparativement triviale, puisqu'il s'agira en général simplement de renvoyer la valeur d'un élément de tableau, d'un champ d'"enregistrement" (ALGOL W, ALGOL 68, PASCAL, SIMULA 67, etc.), ou d'un composant de "structure" (COBOL, PL/1).

Tous les modules sont organisés de cette façon comme une réunion de programmes d'accès, pouvant communiquer avec les autres modules par les programmes d'accès à ces modules. Par exemple, le module syntagme comprendra sans doute un programme :

analinstr : → SYNTAGME

chargé d'élaborer à partir des données un syntagme représentant une instruction ; ce programme pourrait comporter des éléments du genre :

```
variables ℓ1, ℓ2 : LEXEMES ;
ℓ1 ← analex ; {lire un lexème, grâce au module LEXEME}
si typelex (ℓ1) = "identificateur" alors
    ℓ2 ← analex ;
    si typelex (ℓ2) = "opérateur" et textelex (ℓ2) = "←"
    alors
        {l'instruction analysée est une affectation : continuer le décodage}
        .....
```

Le but de cette ébauche d'exemple est de montrer que des modules doivent ne pouvoir communiquer que par l'utilisation des programmes associés, à l'exclusion absolue d'un accès direct aux données elles-mêmes (cette restriction est la condition *sine qua non* de la validité de la méthode).

Cette méthode offre plusieurs avantages importants :

a) chaque module est imperméable aux modifications internes des autres modules : ne communiquant avec eux que par des programmes d'accès, il n'a pas à connaître des choix de représentation successifs et contradictoires qui peuvent être faits, pourvu que les programmes d'accès soient modifiés en conséquence.

b) un point également très important dans le développement "modulaire" d'un grand projet est qu'à des modifications restreintes des spécifications – qui ne sauraient manquer de se produire – doivent correspondre des modifications restreintes des modules. Cette condition est bien remplie ici, puisqu'une modification légère doit se traduire par la suppression, l'ajout ou la modification d'un seul programme d'accès, ou d'un petit nombre de tels programmes.

On notera par contraste combien ce problème est aigu dans une décomposition fondée sur les programmes plutôt que sur les données. Dans le cas du compilateur, par exemple, l'élément "analyseur syntaxique" pourrait dans cette approche communiquer avec l'"analyseur lexical" de façon similaire à ce qui est décrit ci-dessus, par le biais d'un appel de sous-programme

analexic (tx, ty)

où tx et ty représentent des arguments de mode résultat, permettant d'obtenir le texte et le type du lexème analysé. Mais supposons maintenant qu'une modification des spécifications intervienne : on demande à l'analyseur lexical de renvoyer, outre le texte d'un lexème et son type, une valeur entière ; celle-ci correspondra, pour une constante entière, à sa valeur numérique, pour un opérateur, à un code permettant de le désigner (par exemple, 1 pour "+", 2 pour "←", etc.), pour un identificateur, à un indice dans la table des symboles, etc. Comme analexic a maintenant un argument de plus, tous les appels de ce sous-programme, dans tous les modules qui l'utilisent, doivent être modifiés ! La

tâche peut être considérable, et les possibilités d'erreur ou d'omission sont nombreuses. Dans notre approche "modulaire", par contre, point n'est besoin de changer ce qui existe : il suffit d'ajouter à la spécification du module LEXEME un programme

vallex : LEXEME → ENTIER

Les programmes qui ont effectivement besoin de connaître la valeur associée à un lexème ℓ pourront alors y accéder par vallex (ℓ).

De semblables modifications ou renforcements des spécifications sont monnaie courante dans le développement d'un grand projet. Il est donc particulièrement important que la méthode en prévoie et permette de les prendre en compte sans dommage excessif.

Notons que l'un des premiers à avoir exprimé clairement les idées de base de cette méthode, Parnas, va jusqu'à proposer [Parnas 72], pour mieux assurer la protection contre les accès illégitimes, de donner aux modules et aux fonctions de manipulation des noms sans aucune valeur mnémonique ; ainsi, suggère-t-il, l'utilisateur d'un module ne sera-t-il pas tenté de faire des hypothèses a priori sur l'organisation interne des données du module, et devra-t-il s'en remettre entièrement à la spécification fonctionnelle abstraite. Dans notre exemple, analex pourrait s'appeler xx7tz02 ; textelex, uuvrs45 ; etc. Ceci ne nous paraît pas sérieux : la suggestion de Parnas suppose que les programmeurs sont des êtres parfaits, doués d'un esprit logique infaillible et d'une capacité jamais mise en défaut de compréhension immédiate de toute formule abstraite. En fait, il suffit d'avoir utilisé un système d'exploitation sur lequel le compilateur FORTRAN répond au doux nom d'*IFEAAB* pour apprécier *a contrario*, pour le restant de sa vie, la vertu des noms clairs et significatifs.

c) un troisième avantage de la méthode modulaire est qu'elle permet de bien circonscrire les communications entre éléments de programme, puisque les interfaces sont explicitement définies par les programmes d'accès. On a donc une vue claire de la situation, permettant de savoir "qui appelle qui". En particulier, la construction d'environnements de tests consiste à remplacer certains des programmes d'accès par des "échafaudages".

La méthode que nous avons proposée présente cependant certains inconvénients :

a) l'utilisation répétée de sous-programmes, dont certains se contentent de lire ou de modifier un champ d'une structure de données, entraîne une perte d'efficacité due à la lourdeur des appels de sous-programmes dans bien des systèmes actuels. Certes, on pourrait considérer que l'utilisation des programmes d'accès intervient au niveau de la conception des programmes, et qu'à celui de leur réalisation pratique on peut manipuler directement les champs désirés ; mais ce serait se priver de tous les avantages de la méthode, pour ce qui est de la mise au point des tests, des diagnostics à l'exécution, etc.

b) la méthode ne résoud pas le problème du choix des modules de base, pour lesquels il reste difficile d'énoncer des principes à la fois généraux, valides et directement applicables. La méthode modulaire fournit cependant quelques indices qui peuvent servir de guide : construire les modules autour des structures de données qui sont les plus utilisées et jouent le rôle le plus crucial (mais il reste à les déterminer) ; essayer

de minimiser les interactions entre modules, et préserver l'unité fonctionnelle de chacun d'entre eux.

On notera que l'approche modulaire ne favorise ni n'interdit en aucune façon la conception descendante ou la conception ascendante ; l'une et l'autre peuvent en fait déterminer la stratégie de développement des modules. Tout au plus l'approche modulaire permet-elle de faire entrevoir des horizons plus larges où le schéma de développement serait moins strictement hiérarchisé, plus "autogestionnaire", que les schémas ascendant et descendant traditionnels ; mais l'élaboration de méthodes permettant de réaliser un tel schéma, tout en préservant la rigueur de développement et en évitant le désordre conceptuel, est un problème ouvert.

[78c]

VIII.3.5 Programmation statique. Le langage Z[1]

VIII.3.5.1 Définitions : le langage Z_0

Dans tout ce qui précède, nous avons insisté sur l'idée de *spécification fonctionnelle*. Appliquée aux données, nous avons vu au chapitre V que cette notion permettait de définir une structure de données de façon parfaitement externe, à l'aide d'une liste de fonctions et de leurs propriétés abstraites, sans aucune référence aux modes de représentation en mémoire. Une caractérisation semblable a été appliquée aux programmes à partir du chapitre III, et en particulier aux sous-programmes à partir du chapitre IV, lorsque nous avons intercalé, sous forme de commentaires, des *assertions* permettant d'énoncer certaines propriétés sur l'état du programme ; si l'on s'efforce de rendre ces assertions exhaustives, une unité de programme est complètement caractérisée par son assertion initiale et son assertion finale.

Lorsqu'on cherche à développer ces assertions et à les détailler jusqu'au bout, on s'aperçoit vite qu'il s'agit d'un travail difficile ; d'un travail *aussi difficile que la programmation elle-même.*

A partir de cette remarque, une idée décisive, due en particulier à J.R. Abrial, consiste à franchir un pas supplémentaire et à poser que le développement des assertions *est une activité de programmation* : en d'autres termes, qu'un programme peut être entièrement décrit par une suite de relations statiques, qui fournissent simplement une image se situant à un niveau d'abstraction différent de celui de la description habituelle – et plus fondamental.

Le langage Z [Abrial 77] [Abrial 77a] pour la conception et la description des programmes est, ainsi, fondé sur l'idée qu'un programme peut et doit être décrit à plusieurs niveaux. Le premier de ces niveaux, appelé Z_0, permet une description purement statique de la tâche à résoudre ; il est entièrement non-algorithmique, c'est-à-dire que les programmes Z_0 ne contiennent pas d'instructions – d'ordres gouvernant le comportement dynamique d'une machine réelle ou virtuelle – mais seulement une liste de fonctions et de leurs propriétés. Il s'agit donc d'une spécification fonctionnelle, non plus d'une structure de données, mais, plus généralement, de programmes et de problèmes[2].

(1) Le langage Z a considérablement évolué à partir de la version présentée ici. Voir J.R. Abrial, S.A. Schuman, B. Meyer ; *Specification Language* : dans *Proceedings of Summer School on the Construction of Programs* (Belfast, 1979) ; Cambridge University Press, Cambridge (Grande-Bretagne), 1980.

(2) La description de Z donnée ici est sommaire et partielle ; nous avons pris quelques libertés avec les notations d'Abrial pour assurer la compatibilité avec celles de cet ouvrage.

VIII.3.5.2 Le niveau Z_0

Z_0 est entièrement fondé sur un langage de communication et de description qui a eu l'occasion de faire ses preuves au cours des ans : le langage mathématique, et en particulier le langage de la théorie des ensembles, augmenté d'un petit nombre d'extensions correspondant aux problèmes précis rencontrés en programmation. Les éléments de base manipulables en Z_0 sont ainsi des "objets", des ensembles, et des ensembles ordonnés ou "tuples" ; les opérations de base sont les opérations ensemblistes habituelles (test d'appartenance d'un objet à un ensemble, test d'inclusion d'un ensemble dans un autre, union de deux ensembles, intersection, différence, produit cartésien, etc.) et des opérations sur les tuples comme la concaténation, notée * : la concaténation des tuples $t_1 = [a, b, c]$ et $t_2 = [d, e, f, g]$ est $t_1 * t_2 = [a, b, c, d, e, f, g]$; enfin, un programme Z_0 contient des définitions de fonctions et leurs propriétés. Par exemple, A et B étant deux ensembles, l'existence d'une fonction f de A dans B est notée :

$$A \xrightarrow{f} B$$

L'intérêt de cette notation (par rapport à celle que nous avons utilisée : $f : A \to B$) est qu'elle permet de préciser visuellement un certain nombre de propriétés possibles des fonctions, importantes pour le programmeur. Par exemple, il pourrait exister une fonction inverse $f^{-1} = g$; on notera :

$$A \underset{g}{\overset{f}{\rightleftarrows}} B$$

f pourrait être une fonction partielle, c'est-à-dire ne pas donner de résultat pour tous les arguments ; il s'agit d'une information importante, qu'on notera en indiquant la borne inférieure 0 du nombre de résultats de f :

$$A \underset{g}{\overset{f(0)}{\rightleftarrows}} B$$

Plus généralement, f pourrait être éventuellement multivaluée, c'est-à-dire associer à tout élément de A un sous-ensemble f(A) de B (plutôt que 0 ou un seul élément). Les noms des fonctions multivaluées, comme ceux des ensembles, commencent par des majuscules ; si F associe à tout élément de A entre 0 et 10 éléments de B, on notera :

$$A \underset{G(1,\ -)}{\overset{F(0,\ 10)}{\rightleftarrows}} B$$

(F est ici surjective). Il faut ici préciser deux bornes, l'inférieure et la supérieure. Nous avons supposé en outre que la fonction réciproque G était totale, mais que la borne supérieure du nombre de ses résultats était inconnue, ce que représente le tiret. Là où une fonction partielle comme f ou F n'est pas définie, on conviendra que sa valeur est l'élément spécial vide. Les relations portant sur les fonctions sont des propriétés logiques, exprimées à l'aide des opérateurs habituels $\forall$ (pour tout), $\exists$ (il existe), et, ou, non, etc.

VIII.3.5.3 Le niveau Z_1 et les transformations.

Le niveau suivant de langage, Z_1, permet une "algorithmisation" du programme Z_0 en ce sens qu'il introduit des instructions. En fait Z_1 est très proche

de la notation algorithmique utilisée dans cet ouvrage, avec la particularité qu'il emploie largement des expressions ensemblistes (si $x \in A$ alors ...) et des boucles ensemblistes du type (cf. III.3.1.3) :

```
pour x ∈ A répéter
    | .........
```

ou

```
pour x ∈ A tant que c(x) répéter
    | .........
```

L'intérêt tout particulier de cette caractéristique de Z_1 est qu'elle permet d'obtenir de façon systématique des formes "algorithmisées" des programmes statiques exprimés en Z_0. [Abrial 77a] donne ainsi toute une série de règles de transformation de relations Z_0 en instructions Z_1. Par exemple, pour évaluer la condition (expression logique) Z_0 suivante :

$$\forall x \in A \,.\, c(x)$$

on écrira en Z_1 :

```
| e ← vrai ;
| pour x ∈ A tant que e répéter
|     | e ← c(x)
```

Le programme Z_1 ainsi obtenu n'est pas définitif : il faut encore se rapprocher, par des transformations successives, d'un niveau proche de celui des langages usuels. Le niveau suivant de langage, qui n'a pas reçu de nom particulier (on serait tenté de l'appeler Z_2), est débarrassé de toute référence ensembliste : les ensembles, représentés par des fichiers, des tableaux, des listes, etc., sont explicitement parcourus grâce à des ordres ouvrir (ouverture du "fichier" et accès au premier élément) et lire (accès aux éléments suivants) ; les éléments ont été épuisés si et seulement si l'élément lu est l'élément spécial vide. Par exemple :

```
| pour x ∈ A répéter
|     | p(x)
```

s'écrit maintenant :

```
| x ← ouvrir(A) ;
| tant que x ≠ vide répéter
|     | p(x) ;
|     | x ← lire(A)
```

Les transformations suivantes permettront de préciser encore la réalisation du programme : élimination éventuelle de la récursivité, choix de représentation des ensembles par des structures de données plus précises (cf. V.4) en vue de permettre des accès efficaces selon les besoins du problème, etc. Le but est, bien sûr, d'arriver finalement à une version FORTRAN, ALGOL, PL/1, COBOL, etc.

VIII.3.5.4 Un exemple

Pour pouvoir évaluer la méthode d'Abrial, il convient de l'esquisser sur un exemple – hélas trop simple. Il s'agit du traitement d'un fichier documentaire, destiné à en permettre l'"inversion". (1)

(1) Cet exemple a été étudié en collaboration avec M. Demuynck, C. Maillard et P. Moulin.

Soit un fichier B consitué d'"enregistrements". On veut écrire dans un fichier D des "enregistrements" déduits de certains de ceux de B selon des "formats de sortie" contenus dans un fichier C. Un fichier A contient une liste de "clés" permettant de sélectionner les éléments de B qui doivent être traités (figure VIII.4).

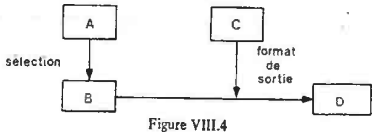


Figure VIII.4

Un élément de B est un "article" composé d'une "clé" et d'un nombre quelconque de couples [nom de champ, valeur] ; un "nom de champ" est un code comme C10, C20, etc ; une "valeur" est un qualificateur, comme "pompe", "turbine", "explosion", etc. Deux éléments distincts ne peuvent avoir la même clé.

Si le fichier A est vide, tous les articles de B seront traités ; sinon, A contient des clés, et seuls les articles de B dont la clé se trouve dans A seront traités.

Dans le traitement d'un article b de B, des enregistrements seront écrits dans D pour chacun des couples [nom de champ, valeur] appartenant à b ; ces enregistrements auront la forme [clé, valeur, numéro de sous-fichier], où la "clé" est celle de l'article b, la "valeur" est celle qu'on trouve dans le couple traduit, et le "numéro de sous-fichier" est déduit du "nom de champ" grâce au fichier C, qui contient une liste de couples [nom de champ, numéro de sous-fichier].

Nous avons volontairement exposé le problème dans son intégralité avant de passer en Z ; pour le traiter, il est utile de séparer les niveaux. Cherchons d'abord à l'exposer en Z_0 au niveau global. (Nous numéroterons les définitions d'ensembles : E_1, E_2, etc. ; les définitions de fonctions : F_1, etc ; les relations : R_1, etc.)

Les ensembles manipulés sont d'abord les fichiers A, B, C, D. B (le fichier de départ) contient des "articles" ; nous postulons donc l'existence d'un ensemble :

Articles

et l'on a :

$$B \subseteq \text{Articles}$$

Nous avons vu que tout article était formé d'une clé et d'un nombre quelconque de couples [nom de champ, valeur]. Ceci s'écrit comme un produit cartésien :

$$\text{Articles} = \text{Clés} \times \text{Tuple}\,(\text{Nomchamps} \times \text{Valeurs}) \qquad (E5)$$

où Clés est l'ensemble des clés, Nomchamps celui des noms de champs, Valeurs celui des valeurs. X étant un ensemble, la notation Tuple(X) désigne en Z_0 l'ensemble :

$$\text{Tuple}(X) = X \cup X \times X \cup X \times X \times X \cup \ldots$$

c'est-à-dire l'ensemble des suites finies d'éléments de X. Ainsi, [3], [8, 5, 9], [12, 50, 10], etc. appartiennent à Tuple (Entiers).

Enfin, un élément de D est un triplet [clé, numéro de sous-fichier, valeur] ... on aura donc ;

$$D \subseteq \text{Clés} \times \text{Numsousfich} \times \text{Valeurs} \qquad (E6)$$

où Numsousfich est l'ensemble des numéros de sous-fichier possibles.

En récapitulant, les ensembles manipulés sont donc :

Clés	(E1)
Nomchamps	(E2)
Valeurs	(E3)
Numsousfich	(E4)
Articles = Clés x Tuple(Nomchamps x Valeurs)	(E5)
$B \subseteq$ Articles	(E6)
$A \subseteq$ Clés	(E7)
$C \subseteq$ Nomchamps x Numsousfich	(E8)
$D \subseteq$ Clés x Valeurs x Numsousfich	(E9)

Au niveau global, le traitement à réaliser peut s'exprimer comme une fonction sans argument, qu'on exprimera comme une fonction dont l'ensemble de départ est l'ensemble spécial à un élément noté **1** :

$$1 \xrightarrow{\text{Trait}\,(0,-)} D \qquad (F1)$$

Trait est une fonction multivaluée, comme l'indique l'emploi de la majuscule : on crée 0, 1 ou plusieurs enregistrements de D.

Les articles de B sont traités, "traduits", un à un, ce qui suggère d'écrire :

$$\text{Trait} = \bigcup_{b \in B} \text{Traduction}(b) \qquad (R'1)$$

avec la fonction

$$B \xrightarrow{\text{Traduction}\,(0,-)} D \qquad (F2)$$

où Traduction est bien multivaluée puisque chaque article de B peut donner lieu à la création de plusieurs articles de D. La relation (R'1) ci-dessus est cependant incomplète car nous avons vu que tous les éléments de B ne sont pas traités si A n'est pas vide ; elle doit être remplacée par :

$$\text{Trait} = \bigcup_{b \in B\,.\,\gamma(b)} \text{Traduction}(b) \qquad (R1)$$

notation qui signifie que l'"union" n'est opérée que sur les b qui vérifient la condition $\gamma(b)$, avec :

$$\text{Article} \xrightarrow{\gamma(1)} (\text{vrai}, \text{faux}) \qquad (F3)$$

et :

$$\forall\, b \in \text{Articles}\,.\,\gamma(b) = (A = \phi \text{ ou } b(1) \in A) \qquad (R2)$$

Articles étant le produit cartésien Clés x Tuple (Nomchamps x Valeurs), b(1) désigne le premier composant d'un article b, sa clé.

Nous devons maintenant exprimer la fonction Traduction. Elle portera sur chacun des couples [nom de champ, valeur] formant le second composant, b(2), d'un article b. Ces couples sont traduits séparément, ce que nous exprimerons en introduisant la fonction de traduction d'un couple :

$$\text{Nomchamps} \times \text{Valeurs} \xrightarrow{\text{tradélém (1)}} \text{Valeurs} \times \text{Numsousfich} \qquad (F'3)$$

et en exprimant Traduction par

$$\forall\, b \in B\,.\ \text{Traduction}(b) = \bigcup_{x \in b(2)} b(1) * \text{tradélem}(x) \qquad (R'3)$$

L'astérisque, rappelons-le, représente la concaténation de tuples.

Pour exprimer la fonction tradélem, associant à tout couple [nom de champ, valeur] un couple [valeur, numéro de sous-fichier], nous introduisons la fonction permettant d'associer un numéro de sous-fichier à un nom de champ, soit numsous:

$$\text{Nomchamps} \xrightarrow{\text{numsous (1)}} \text{Numsousfich} \qquad (F4)$$

Cette fonction, rappelons-le, est représentée par le fichier C ; D'après (E8), $C \subseteq$ Nomchamps x Numsousfich, et l'on a précisément :

$$\forall\, c \in C\,.\ \text{numsous}(c(1)) = c(2) \qquad (R4)$$

On notera que l'existence du fichier C et celle de la fonction numsous sont deux propriétés équivalentes ; cette équivalence est exprimée par la relation (R4). Une telle redondance est une des caractéristiques de la programmation au niveau Z_0 : ce n'est qu'au moment de l'"algorithmisation" qu'on choisit entre une représentation par une fonction (= sous-programme) ou par un ensemble (= structure de données). Au niveau Z_0, on se refuse à choisir entre l'espace et le temps car cette question est prématurée, et l'on garde les deux formulations redondantes.

numsous nous permet d'expliciter tradélem :

$$\forall\, b \in B.\ \ \forall x \in b(2).\ \text{tradélem}(x) = [x(2), \text{numsous}(x(1))] \qquad (R'4)$$

d'après la description donnée du traitement d'un couple.

Il reste à exprimer que A contient, s'il n'est pas vide, des clés permettant d'accéder aux éléments de B, et ceci de façon univoque, deux éléments distincts de B ayant des clés différentes. Cela se traduit par l'existence d'une fonction injective :

$$B \xrightarrow[\text{article de clé (0)}]{\text{clé (1)}} \text{Clés} \qquad (F5)$$

La fonction importante, ici, est la fonction inverse article de clé ; en l'écrivant comme une fonction monovaluée, nous avons exprimé qu'on pouvait accéder de façon unique à un élément de B à partir de sa clé. Cette fonction est partielle, comme l'indique le 0 dans sa définition : toutes les clés possibles ne sont pas nécessairement représentées dans B. Notez que, comme précédemment, l'affirmation de l'existence d'une fonction clé totale est redondante avec les relations

ensemblistes précédentes (E5) et (E6) ; la correspondance est exprimée par la relation :

$$\forall\, b \in B\,.\ \text{clé}(b) = b(1) \qquad (R5)$$

Finalement, nous pouvons récrire (R'3) en nous dispensant de la fonction intermédiaire tradélem grâce à (R'4), ce qui permet d'obtenir, en récapitulant le programme Z_0 final suivant :

ENSEMBLES

Clés	(E1)
Nomchamps	(E2)
Valeurs	(E3)
Numsousfich	(E4)
Articles = Clés x Tuple (Nomchamps x Valeurs)	(E5)
$B \subseteq$ Articles	(E6)
$A \subseteq$ Clés	(E7)
$C \subseteq$ Nomchamps x Numsousfich	(E8)
$D \subseteq$ Clés x Numsousfich x Valeurs	(E9)

FONCTIONS

$1 \xrightarrow{\text{Trait (0,–)}} D$	(F1)
$B \xrightarrow{\text{Traduction (0,–)}} D$	(F2)
$\text{Article} \xrightarrow{\gamma\,(1)} (\text{vrai, faux})$	(F3)
$\text{Nomchamps} \xrightarrow{\text{numsous (1)}} \text{Numsousfich}$	(F4)
$B \xrightarrow[\text{article de clé (0)}]{\text{clé (1)}} \text{Clés}$	(F5)

RELATIONS

$\text{Trait} = \bigcup_{b \in B\,.\,\gamma(b)} \text{Traduction}(b)$	(R1)
$\forall\, b \in \text{Articles}\,.\ \gamma(b) = (A = \phi \text{ ou } b(1) \in A)$	(R2)
$\forall b \in B.\ \text{Traduction}(b) = \bigcup_{x \in b(2)} [b(1), x(2), \text{numsous}(x(1))]$	(R3)
$\forall\, c \in C\,.\ \text{numsous}(c(1)) = c(2)$	(R4)
$\forall\, b \in B\,.\ \text{clé}(b) = b(1)$	(R5)

On notera que si l'ordre des enregistrements dans D était important, et nécessairement le même que celui des éléments de B, il faudrait remplacer les unions par des concaténations (*).

Pour passer à un programme Z_1, nous traduirons d'abord (R1) par la version (V1) suivante :

```
(V1)  Trait ← ∅ ;
      pour b ∈ B répéter
          si γ (b) alors
              Trait ← Trait ∪ Traduction (b)
```

c'est-à-dire, en explicitant la condition γ (b) :

```
(V2)  Trait ← ∅ ;
      pour b ∈ B répéter
          si A = ∅ ou b(1) ∈ A alors
              Trait ← Trait ∪ Traduction (b)
```

Comme le test "A = ∅ ?" est indépendant de b, on peut le "sortir de la boucle" pour traiter séparément les cas A = ∅ et A ≠ ∅. Il s'agit de l'une des "transformations" standard répertoriées dans [Abrial 77a]. Elle nous donne la version (V'3) suivante :

```
(V'3) Trait ← ∅ ;
      si A = ∅ alors
          pour b ∈ B répéter
              Trait ← Trait ∪ Traduction (b)
      sinon {A ≠ ∅}
          pour b ∈ B répéter
              si b(1) ∈ A alors
                  Trait ← Trait ∪ Traduction (b)
```

Nous pouvons cependant poursuivre ici le développement du programme sur une voie légèrement différente. L'existence de la fonction monovaluée partielle article de clé (propriété (F5)) nous montre en effet que, dans le cas où A ≠ ∅, on peut accéder à un élément b vérifiant b(1) ∈ A, c'est-à-dire clé (b) ∈ A (relation (R5)), c'est-à-dire encore de clé donnée, à partir de sa clé dans A. On peut donc fonder l'algorithme, dans le cas où A ≠ ∅, sur un parcours séquentiel de A plutôt que de B. C'est ce que nous choisissons de faire ici ; notez que cette décision ne se justifie que si l'on en espère un gain d'efficacité, c'est-à-dire s'il y a beaucoup moins de clés dans A que d'articles dans B.

Nous obtenons :

```
(V3)  Trait ← ∅ ;
      si A = ∅ alors      {comme précédemment ; B traité séquentiellement}
          pour b ∈ B répéter
              Trait ← Trait ∪ Traduction (b)
      sinon               {A ≠ ∅ : accès "direct" aux éléments de B
                           à partir de leur clé}
          pour a ∈ A répéter
              b ← article de clé (a) ;
              si b ≠ vide {c'est-à-dire si la fonction "article de
                           clé" est définie sur a} alors
                  Trait ← Trait ∪ Traduction (b)
```


Nous utilisons maintenant (R3) pour développer Traduction (b) : la notation "X :∋ x", où X est un ensemble et x un objet, désigne en Z l'instruction d'inclusion ajoutant l'élément x à l'ensemble $X^{(1)}$.

```
      Trait ← ∅ ;
      si A = ∅ alors
          pour b ∈ B répéter
              pour x ∈ b(2) répéter
                  Trait :∋ [b(1), x(2), numsous (x(1))]
(V4)  sinon
          pour a ∈ A répéter
              b ← article de clé (a) ;
              si b ≠ vide alors
                  pour x ∈ b(2) répéter
                      Trait :∋ [b(1), x(2), numsous
                               (x(1))]
```

L'étape suivante consiste à éliminer les boucles pour ensembliste en les exprimant par des parcours séquentiels explicites. D'après les définitions données plus haut des opérations ouvrir, lire, et de la valeur spéciale vide, on obtiendra la version (V5) :

```
      Trait ← ∅ ;
      a ← ouvrir (A) ;
      si a = vide alors
          b ← ouvrir (B) ;
          tant que b ≠ vide répéter
              x ← ouvrir (b(2)) ;
              tant que x ≠ vide répéter
                  Trait :∋ [b(1), x(2), numsous (x(1))] ;
                  x ← lire (b(2))
              b ← lire (B)
(V5)  sinon
          répéter
              b ← article de clé (a) ;
              si b ≠ vide alors
                  x ← ouvrir (b(2)) ;
                  tant que x ≠ vide répéter
                      Trait :∋ [b(1), x(2), numsous (x(1))] ;
                      x ← lire (b(2))
              a ← lire (A) ;
          jusqu'à a = vide
```

(1) Notez la correspondance formelle entre l'instruction d'affectation habituelle a := b, rendant vraie la condition "a = b", et l'affectation ensembliste X :∋ x, rendant vraie la condition "X ∋ x", ou "X contient x".

Remarquons que peu de décisions de mise en œuvre ont été prises, hormis la notation b(1) plutôt que clé (b). En particulier, **ouvrir** et **lire** n'impliquent pas nécessairement des entrées-sorties physiques, mais peuvent être l'accès à un tableau (i ← 1 pour **ouvrir** ; i ← i + 1 pour **lire**) ou à une structure chaînée ℓ (x ← premier (ℓ) ; x ← suivant (x)).

De nombreuses transformations resteraient à opérer sur ce programme : il faut choisir un mode d'accès à A et à B, remplacer l'affectation ensembliste à Trait par une instruction d'écriture, etc. La méthode devrait cependant être claire.

A titre de complément, nous fournissons ci-après (une fois n'est pas coutume) un programme COBOL donnant une version exécutable du programme Z ci-dessus, et obtenu par transformation systématique à partir de celui-ci. Seules la *PROCEDURE DIVISION* et une partie de l'*IDENTIFICATION DIVISION* sont fournies.

Pour comprendre ce programme, il pourra être utile de savoir que les représentations physiques adoptées dans cette réalisation sont les suivantes :

- A est un "fichier séquentiel" externe, qui pourrait être sur cartes, sur disque ou sur bande magnétique. On y accède par l'ordre de lecture *READ*, qui lit les enregistrements dans l'ordre jusqu'à la fin du fichier, signalée par la condition spéciale *AT END*. Les enregistrements lus sont placés dans une chaîne de caractères, ici *ENR-A* : l'association entre le fichier et sa "zone de lecture" se fait en COBOL à la compilation, grâce à la *FILE SECTION* de la *DATA DIVISION*.
- D est un fichier séquentiel analogue à A, que l'on constitue par des ordres *WRITE* successifs, qui écrivent chacun l'enregistrement présent dans *ENR-D* ; même remarque que ci-dessus en ce qui concerne l'association entre le fichier et sa "zone d'écriture".
- C est représenté par le tableau *TABC*, rangé par numéro de sous-fichier : la valeur de *TABC(I)* est le nom de champ associé au sous-fichier numéro *I*. Le chargement en mémoire du tableau n'a pas été représenté.
- B, enfin, est une "base de données" gérée par le programme IMS d'IBM, et résidant sur disque. Sa structure hiérarchique est la suivante : chaque enregistrement b de B, tel qu'il a été défini plus haut, est représenté par une "racine" contenant la clé b(1), à laquelle sont rattachés des "segments dépendants" en nombre quelconque, représentant chacun un couple [nom de champ, valeur] de b(2) ; leur "type de segment" est le nom du champ, et leur contenu est la valeur associée.

Les accès à la base de données se font par l'intermédiaire du programme *CBLTDLI*, auquel on fournit :

- un argument qui spécifie le type d'opération d'entrée-sortie désirée : *GU* signifie "lire la première racine de la base de données" ; *GN* veut dire "lire le segment ou la racine suivante" ; *GNP* signifie ici "lire en séquence les segments qui dépendent hiérarchiquement de la dernière racine lue" ;
- un argument, le *PCB*, qui fournit au système d'exploitation certains renseignements indispensables à la bonne gestion de la base de données, et dont deux sous-ensembles utiles au programme utilisateur sont le *CODE-RETOUR* (*'GE'* signifie "racine ou segment non trouvé", *'GB'* veut dire "fin de fichier atteinte") et le *TYPE-SEGMENT* qui, dans ce cas précis, fournit le nom de champ auquel est associé une valeur lue ;
- un argument spécifiant la zone à utiliser pour placer la racine ou le segment lu ;
- un argument facultatif qui permet de sélectionner certains enregistrements, et qui est formé d'une chaîne de caractères, écrite dans un langage de commande spécial appelé DL/1, et décodée à l'exécution par un interprète contenu

dans le système de gestion de bases de données IMS : ici, *SELECT-RACINE* impose de ne lire que des racines, et *SELECT-RACINE-CLE* requiert l'accès direct à une racine de clé donnée (lue dans le fichier A). C'est ce qui a été exprimé en Z par l'affirmation de l'existence d'une fonction monovaluée partielle "article de clé", permettant d'accéder à un élément de B à partir de sa clé contenue dans A.

D'un point de vue méthodologique, on notera que dans ce cas précis la structure concrète des fichiers était connue a priori : le programme en Z_0 représente donc un effort d'abstraction de ses propriétés caractéristiques. La forme en Z_0 est plus facilement utilisable, nous semble-t-il, que la description détaillée de la base dans le système de gestion de bases de données utilisé (IMS), et plus indépendante de choix techniques non fondamentaux, qui peuvent être remis en cause ultérieurement. Mais la programmation en Z permet de revenir au système concret, comme nous le faisons ici, tout en gardant une description de plus haut niveau d'abstraction.

COBOL

```
PROCEDURE DIVISION.
        .
        .
        .
    OPEN OUTPUT D
    OPEN INPUT A
    MOVE 'B1' TO NOM-SELECT
    READ A AT END GO TO A-VIDE.
ETIQ1.
    MOVE ENR-A TO CLE-SELECT
    CALL 'CBLTDLI' USING GU PCB ZONE-B1 SELECT-RACINE-CLE
    IF CODE-RETOUR NOT = 'GE' CALL 'TRAD' USING ZONE-B1 PCB
    READ A AT END CLOSE A GOBACK.
    GO TO ETIQ1.
A-VIDE.
    CALL 'CBLTDLI' USING GU PCB ZONE-B1 SELECT-RACINE
    IF CODE-RETOUR = 'GE' CLOSE A, D GOBACK.
ETIQ2.
    CALL 'TRAD' USING ZONE-B1 PCB
    CALL 'CBLTDLI' USING GN PCB ZONE-B1 SELECT-RACINE
    IF CODE-RETOUR = 'GB' CLOSE A, D GOBACK.
    GO TO ETIQ2.
```

COBOL

```
IDENTIFICATION DIVISION.
PROGRAM-ID.
    TRAD.
    .
    .
    .
PROCEDURE DIVISION USING ZONE-B1 PCB.
ETIQ1.
    CALL 'CBLTDLI' USING GNP PCB ZONE-X2
    IF CODE-RETOUR = 'GE' OR 'GB' GOBACK.
    PERFORM RECH-NUMSOUS THRU FINR IF I = 0 GO TO ETIQ1.
    MOVE ZONE-B1 TO ENR-D (1)
    MOVE ZONE-X2 TO ENR-D (2)
    MOVE NUMSOUS (I) TO ENR-D (3)
    WRITE ENR-D
    GOTO ETIQ1.
RECH-NUMSOUS.
    MOVE 0 TO I.
BOUCLE.
    ADD 1 TO I
    IF I > IMAX MOVE 0 TO I GO TO FINR.
    IF TYPE-SEGMENT NOT = TABC (1) GO TO BOUCLE.
FINR.
    EXIT.
```

VIII.3.5.5 Discussion

L'utilisation de Z a le mérite principal de permettre de poser clairement quelques-unes des vraies questions qui sont la clé de la réalisation d'un programme. Or la détection des vraies questions est souvent en programmation – comme plus généralement dans toutes les disciplines scientifiques – l'étape la plus délicate de la recherche des solutions.

Dans la réalisation d'un projet important de programmation, une grande partie de la difficulté tient souvent à l'imprécision de la tâche à résoudre : on se trouve confronté à un "cahier des charges" lourd, multiforme, incomplet, contradictoire, désordonné. Chacune des tâches élémentaires demandées est souvent simple ; presque toujours, en tous cas, les difficultés conceptuelles véritables sont limitées à un petit nombre de sous-problèmes. La véritable difficulté tient à la multiplicité des demandes, à la définition de leurs relations mutuelles, à cette espèce de grouillement de spécifications devant lequel programmeurs et chefs de projets se sentent désespérés.

Face à une telle situation, typique de l'informatique de gestion, mais qui se rencontre aussi dans tous les autres domaines d'application,

on a souvent le sentiment qu'il suffirait de poser complètement le problème pour que la suite du développement coule de source. Or c'est précisément de poser complètement et rigoureusement le problème que s'occupe Z_0. Mais, à la différence d'un cahier des charges rédigé en français (et par des non-spécialistes de l'informatique), un programme Z_0 est directement exploitable pour donner un programme au sens habituel, algorithmique, du terme, à l'aide des transformations mentionnées ci-dessus, de façon en principe mécanisable – bien que dans l'état actuel de la technique l'intuition et l'expérience du programmeur continuent bien sûr à jouer un rôle.

L'expérience de la programmation en Z montre que l'on est parfois tenté, dans la résolution d'un problème, d'abandonner assez vite le niveau Z_0 pour passer à un programme "algorithmique" en Z_1, alors même qu'on n'a pas entièrement spécifié le problème en Z_0 : l'habitude des langages de programmation usuels donne en effet à croire que les sous-problèmes que l'on a négligés d'exprimer en détail sont triviaux, et qu'on pourra les résoudre en cours de route en écrivant le programme Z_1. Or bien souvent, on se heurte en Z_1 à des difficultés ou à des contradictions qui paraissent inextricables ; si l'on cherche à en déceler les raisons, on s'aperçoit que la démarche "algorithmisante" tend à *surspécifier* constamment le problème, en obligeant à prendre des décisions arbitraires qui peuvent se révéler trop contraignantes, et lier le programmeur au point qu'il aboutira à une impasse lors d'une étape ultérieure.

Un exemple typique de ce genre de surspécification est l'ordre de parcours d'un ensemble : bien souvent, on veut effectuer une action $a(x)$ pour tous les éléments x d'un ensemble A, l'ordre dans lequel ces éléments sont traités étant sans importance. Dans l'écriture d'un programme au sens classique, on est obligé de prescrire un ordre de parcours de l'ensemble, même si celui-ci est implicite ; par exemple, si les éléments de l'ensemble sont triés selon une certaine clé, et si on les consulte séquentiellement, c'est cet ordre qui sera adopté. Pour une étape ultérieure, on peut s'apercevoir que l'ordre de parcours devient significatif, mais qu'un ordre différent est désiré : on se trouve alors lié par une décision de concention prise trop tôt. En Z_0, au contraire, aucun ordre de parcours n'est spécifié lorsqu'on représente la transformation f correspondant à a et ses propriétés :

$$\begin{cases} A \xrightarrow{f(\ldots)} B \\ \forall x \in A \,.\, f(x) = \ldots\ldots \end{cases}$$

Notons qu'ici le premier niveau de Z_1 permet encore de ne pas spécifier l'ordre de parcours s'il n'est pas nécessaire :

$$\left| \begin{array}{l} \textbf{pour } x \in A \textbf{ répéter} \\ \quad \left| a(x) \right. \end{array} \right.$$

Cependant, dès l'étape Z_1, chaque fois qu'on écrit l'enchaînement de deux instructions, on prescrit un ordre d'exécution qui peut ne pas être significatif.

Un problème similaire de surspécification se pose à propos des structures de données et du compromis espace-temps. Que de projets de programmation se retrouvent entravés par un choix de représentation trop hâtif ! Soit un exemple aussi simple qu'un fichier du personnel, destiné à la paye. Supposons qu'on ait choisi une fois pour toutes qu'il sera composé d'enregistrements de la forme :

nom	emploi	catégorie	sexe	état-civil	nom de l'époux	enf. 1	enf. 2	...
(20 carac.)	(10 carac.)	(4 carac.)	(2 carac.)		(10 carac.)	âges des enfants		

Que ce choix soit initialement bon ou mauvais importe peu. Mais, ce format étant fixé et connu, tous les programmes accèderont à l'emploi comme aux caractères 21 à 30 de chaque enregistrement, à l'état-civil comme au 36ème caractère, etc. (nous supposons que les auteurs de ces programmes n'ont pas lu le chapitre V).

Supposons maintenant que, les programmes étant corrects, on s'aperçoive au moment du chargement du fichier que celui-ci demande une taille de mémoire (interne ou périphérique) inacceptable. On cherchera à réduire la place nécessaire, et ceci peut très certainement se faire à peu de frais. Le nombre d'emplois possibles est certainement limité à quelques milliers au plus, et 14 bits représentant un pointeur vers une "table des emplois" suffisent sans doute. Les noms se répètent : une méthode de type de l'adressage associatif peut se justifier, voire des methodes spéciales utilisant la redondance naturelle des langages usuels (*"trie"* : voir [Knuth 73], 6.3) et permettant d'économiser de la place dans des proportions considérables ; la simple adoption d'un format variable peut déjà donner de bons résultats si la plupart des noms ont moins de 10 lettres. Le sexe peut se représenter sur un seul bit. Le nom de l'époux est en général le même que celui de l'employé : avec un format variable, on utilisera un bit indiquant si tel est le cas, et, seulement si la réponse est négative, une zone décrivant le nom de l'époux – ou encore un pointeur si l'époux figure également sur le fichier.

De telles modifications sont l'application de techniques de routine pour le programmeur professionnel ; elles peuvent réduire l'encombrement d'un fichier dans des proportions considérables, voir permettre dans certains cas de conserver en mémoire centrale tout un fichier qui paraissait demander un disque entier. Notez que la démarche inverse pourrait être de mise : si le temps de calcul est une denrée précieuse, mais l'espace une ressource abondante, on sera conduit à répéter en de multiples exemplaires des informations redondantes pour diminuer à l'extrême les calculs effectués.

Le point important est que ce sont là des décisons de représentation, qui dépendent de l'environnement, des tests, etc. Prendre ces décisions au moment de la conception est prématuré, et peut amener des réécritures multiples et des modifications en chaîne lorsqu'on les remet en cause – ce qui ne saurait manquer d'arriver.

Par contraste, soit le "programme" Z_0 :

$$\text{Employés} \xrightarrow[\text{Employé de nom } (1,-)]{\text{nom } (1)} \text{Noms}$$

{Employés est l'ensemble des employés, Noms celui des noms ; ils sont reliés par la fonction nom : tout employé a un nom et un seul ; un nom peut être porté par un ou plusieurs employés}

$$\text{Employés} \xrightarrow{\text{emploi } (1)} \text{Emplois}$$

$$\text{Employés} \xrightarrow{\text{catégorie } (1)} \text{Catégories}$$

$$\text{Employés} \xrightarrow{\text{sexe } (1)} (\text{masculin, féminin})$$

$$\text{Employés} \xrightarrow{\text{etat-civil } (1)} (\text{veuf, divorcé, marié, célibataire})$$

$$\text{Employés} \xrightarrow{\text{Enfants } (0,-)} \text{Ages-enfants}$$

Parmi les relations, on aura par exemple (nous supposons pour les besoins de la démonstration que l'entreprise est particulièrement sourcilleuse quant à la moralité de son personnel) :

$$\forall\, x \in \text{Employés}.\ \text{état-civil}(x) = \text{célibataire} \Rightarrow \text{Enfants}(x) = 0$$

Si à chaque emploi correspond une catégorie et une seule, on notera qu'il existe une fonction :

$$\text{Emplois} \xrightarrow[\text{Emplois associés } (1,-)]{\text{grade associé } (1)} \text{Catégories}$$

En termes de représentation physique, l'existence de cette fonction signifie qu'il n'est pas strictement nécessaire que chaque enregistrement contienne un champ "catégorie", puisque la catégorie peut être "calculée" à partir de l'emploi (en passant par une table). Mais le programme Z_0 ne prescrit rien : il fournit les éléments permettant de choisir la meilleure représentation physique. Parmi ces renseignements, ceux qui indiquent quels sont, pour une fonction f, les nombres minimal et maximal d'éléments associés à tout élément (par exemple (1, 1) pour la fonction emploi ci-dessus), et ceux qui indiquent les caractéristiques de la fonction inverse, sont parmi les plus importants pour guider les choix ultérieurs en Z_1, ce qui explique qu'une notation spéciale ait été développée pour eux (ils pourraient être exprimés par des relations, comme dans les spécifications fonctionnelles du chapitre V).

La leçon de ces remarques est qu'il faut résister à la tentation, mentionnée ci-dessus, de passer trop vite en Z_1, et *toujours spécifier complètement le problème en Z_0* avant d'aborder les niveaux suivants.

La méthode de programmation qui a été rapidement présentée, utilisant une approche initiale "statique" des programmes et des transformations répétées, nous paraît apporter des solutions intéressantes à plusieurs des problèmes importants de la programmation. De fait, le langage Z nous paraît refléter une évolution importante. Ceci étant, on peut cependant émettre un certain nombre de réserves :

- l'emploi d'un formalisme fondé sur les notations mathématiques relativement avancées de la théorie des ensembles pose un certain nombre de problèmes psychologiques et pédagogiques – moins peut-être, d'ailleurs, à cause de la difficulté intrinsèque de ces notations, finalement plus simples que FORTRAN, COBOL, ou (certainement !) PL/1, que du statut spécial ("totem et tabou") dont jouissent les mathématiques dans notre société ;
- l'importante séparation entre aspects statiques et dynamiques ne résoud pas le problème de la décomposition d'une tâche complexe. De fait,

un programme en Z_0 pose les mêmes problèmes de division de la complexité, s'il est assez gros, qu'un programme habituel. Les méthodes usuelles s'appliquent : on peut parler de programmation descendante et de programmation ascendante en Z_0, et une stratégie de résolution des problèmes faisant appel à ces notions s'impose dès que les projets dépassent une certaine taille. Schématiquement, dans le processus de programmation en Z_0, qui consiste à définir des ensembles, des fonctions, et des relations, l'approche descendante conduit à dégager les fonctions correspondant au traitement à effectuer avant d'avoir décrit en détail les ensembles auxquels elles s'appliquent ; l'approche ascendante, à définir au contraire complètement les ensembles avant de s'intéresser aux fonctions qui leur sont appliquées ;

– on peut se demander si la somme de travail supplémentaire que demande la construction du programme en Z_0, est justifiée. Ce travail est effectivement assez lourd. La seule réponse à cette objection est la constatation que le programme statique est de toute façon construit implicitement en programmation usuelle, et qu'il est important d'en posséder une version explicite, afin de pouvoir distinguer les choix fondamentaux et les choix de représentation, avec les avantages annexes que sont les possibilités d'optimisation et la présence d'une documentation rigoureuse et exploitable (le programme Z_0) correspondant vraiment aux programmes réalisés.

Les langages de spécification

M. DEMUYNCK* et B. MEYER**

[79a]

Ceux à qui le Roi m'avoit confié remarquant combien j'étois mal habillé, donnerent ordre à un Tailleur de venir le lendemain, & de me prendre mesure pour un habillement complet. Cet Ouvrier le fit, mais d'une manière toute difeRente de celle qui est en vogue en Europe. *Il prit d'abord ma hauteur à l'aide d'un quart de Cercle, & puis par le moyen d'une Regle & d'un Compas, il decrivit sur le papier toutes les dimensions de mon corps, & six jours après il m'apporta mes habits parfaitement mal faits, parce qu'il s'étoit mepris dans une Figure : Mais ce qui me consola, c'est que je remarquai que ces sortes d'accidens étoient fort ordinaires, & qu'on ne s'en mettoit guères en peine.*

[Swift] *Voyages du Capitaine Lemuel Gulliver en Divers Pays Eloignez, tome second, Première Partie, 1727.*

* Ingénieur-Chercheur au Centre de Documentation.

** Ingénieur-Chercheur au Département Méthodes et Moyens de l'Informatique.

E.D.F. – BULLETIN DE LA DIRECTION DES ETUDES ET RECHERCHES
SERIE C – MATHEMATIQUES, INFORMATIQUE n° 1, 1979, pp. 39-60.

TABLE DES MATIERES

I. INTRODUCTION

Les dernières années ont vu une prise de conscience croissante des résultats techniques et économiques en jeu dans ce qu'on a appelé la "crise du logiciel". Le seul chiffre de 250 milliards de francs – coût estimé de l'informatique dans le monde en 1975 –, et le fait que ces coûts sont à 75 %, et pour une proportion qui ne cesse de croître, des coûts de logiciel, montre qu'il est crucial d'améliorer les méthodes d'étude et de maintenance des systèmes informatiques [12]. Dans ce but, de nombreuses idées ont été proposées, qui ont nom "programmation structurée", "programmation modulaire", "programmation systématique", "analyse descendante", "analyse composite", etc. ; leurs partisans insistent sur la nécessité d'une démarche logique et rigoureuse, sur l'importance de la clarté et de la documentation des programmes, sur l'emploi de structures de contrôle et de données bien définies.

Bien que ces idées soient fructueuses et fondamentales, l'expérience montre qu'elles ne résolvent qu'une partie du problème. Lorsqu'en effet on en arrive à l'étape de programmation, tout est déjà joué pour une large part : les véritables difficultés se placent avant, au moment où l'on cherche à comprendre et décomposer le problème. L'importance de cette phase est particulièrement évidente dans un domaine comme l'informatique de gestion, où la plupart des tâches à effectuer sont conceptuellement simples ; cependant, même si les éléments du système, pris individuellement, semblent "faciles", leur nombre même et leur imbrication rendent extrêmement ardue la compréhension de l'ensemble.

Une des conséquences de cette situation, qui se produit dans tous les domaines d'application, est qu'il manque la plupart du temps un bon *Cahier des charges*. Le cahier des charges est trop souvent énorme, complexe, difficile à comprendre et difficile à contrôler.

Vérifier qu'il est cohérent et complet est une tâche impossible dans ces conditions. Le résultat le plus clair est l'apparition au stade de la programmation de difficultés et de choix qui auraient dû être réglés au cours de l'"analyse" et qui peuvent perturber gravement, voire mettre en péril, la réalisation du projet de programmatin.

Le concept de *langage de spécification* correspond à cette recherche d'une méthode pour *poser* les problèmes avant de commencer à les *résoudre*. Il s'agit de fournir un cadre précis pour constituer sous une forme fiable et rigoureuse, pouvant éventuellement être traitée automatiquement, l'exposé d'un problème que l'on envisage de résoudre sur ordinateur.

II. LES LANGAGES DE SPECIFICATION : UN ENSEMBLE DE CRITERES

II.1. Introduction

Depuis quelques années, plusieurs essais ont été faits pour proposer des langages de ce type et les travaux continuent dans ce domaine.

Les paragraphes suivants de cet article seront consacrés à discuter en quoi le concept de langage de spécification diffère du concept de langage de programmation (section III) ; à passer en revue quelques-uns des formalismes et systèmes proposés (section IV) ; à en décrire un autre, le langage Z de Jean-Raymond Abrial, que nous considérons comme l'approche la plus prometteuse (section V) ; et, en conclusion, à discuter la portée et les implications de cette approche, ses rapports avec les autres systèmes décrits et le bénéfice qu'elle peut en tirer.

II.2. Critères d'évaluation des langages de spécification

L'étude des différents formalismes sera fondée sur les critères suivants, qui nous paraissent les plus utiles pour notre approche comparative des langages de spécification.

II.2.1. Aspect statique/dynamique

Le langage décrit-il seulement des *problèmes* (aspect statique) ou est-il orienté vers l'indication de *processus* (aspect dynamique) ?

II.2.2. Niveau d'abstraction

Quel est le niveau d'abstraction du langage ? Permet-il à l'utilisateur d'oublier les détails des sous-sytèmes ? Jusqu'à quel point les spécifications prescrivent-elles une méthode d'implémentation particulière ? Est-ce qu'elles permettent au contraire de faire abstraction des techniques qui pourront être ultérieurement utilisées ?

II.2.3. Généralité

Le langage est-il conçu pour une classe particulière de machines, de systèmes, de langages de programmation ou d'applications ?

II.2.4. Modularisation

Le formalisme de spécification aide-t-il dans la décomposition d'un système en sous-unités, ou bien suppose-t-il que cette décomposition a déjà été faite ?

II.2.5. Méthodologie

Le langage inclut-il des règles méthodologiques qui aideront à produire des spécifications amenant des systèmes fiables, ou bien de telles règles doivent-elles être ajoutées comme des restrictions à l'utilisation de certaines facilités ?

II.2.6. Dispositifs automatiques

Le langage est-il prévu pour des spécifications "sur papier", ou bien est-il susceptible de donner lieu à un traitement automatique, en particulier à des fins de documentation ?

II.2.7. Base théorique et rigueur

Le langage a-t-il une base théorique saine ? Est-il défini seulement par des documents écrits en langue naturelle ? Existe-t-il une définition précise, éventuellement axiomatique ?

II.2.8. Cadre linguistique

Dans quelle sorte de formalisme sont exprimées les spécifications (langue naturelle, diagrammes, automates, tables, logique mathématique, théorie des ensembles...) ?

II.2.9. Description de la syntaxe/ de la sémantique

Certains formalismes ne permettent de décrire que la *syntaxe* d'un système, c'est-à-dire sa structure et les relations entre les éléments. Nous soutiendrons que la *sémantique*, à savoir le rôle et la signification de ces éléments, devrait aussi être incluse dans les spécifications, exprimée dans le formalisme lui-même.

II.2.10. Support graphique

Existe-t-il une aide visuelle (automatique ou manuelle) ? Nous discuterons le pour et le contre des formalismes graphiques au paragraphe VI.2.

II.2.11. Utilisateurs

A quel type d'utilisateurs est destiné le langage (non-informaticiens, analystes, programmeurs) ? La spécification d'un système sera-t-elle compréhensible par les personnes pour qui le système est développé ? Qui écrira et qui utilisera les spécifications ?

II.2.12. Apprentissage et facilité d'emploi

Le langage et la méthodologie associée sont-ils difficiles à apprendre/à enseigner ? L'utilisation du langage entraîne-t-elle un surcroît de travail fastidieux ?

II.3. Portée de l'étude

Plusieurs formalismes possibles seront brièvement examinés en rapport avec les critères ci-dessus mentionnés : langages de programmation, méthodes d'analyse (Warnier, Jackson), tables de décision, diagrammes de transition, réseaux de Pétri, diagrammes HIPO, ISDOS, SREM-SREP, SADT, types abstraits de données, et Z. Les limites imposées à un article font que nous nous sommes limités à des généralités sur tous les systèmes étudiés (sauf le dernier, que nous exposerons avec plus de détails) ; nous espérons cependant avoir dégagé les principaux apports de chacun. (Nous devons cependant mentionner qu'il est particulièrement difficile d'obtenir des informations spécifiques sur quelques-uns des systèmes américains les plus ambitieux ; ceci semble tout à fait courant dans ce domaine de recherche, sans doute pour des raisons commerciales, plus que scientifiques).

III. POURQUOI LES LANGAGES DE PROGRAMMATION NE SONT-ILS PAS ADAPTES A LA SPECIFICATION ?

Avant de commencer la description des langages de spécification, il est utile de discuter en quoi ce concept diffère de celui des langages de programmation. Pourquoi ne pourrait-on spécifier un problème en COBOL, PL/1, ALGOL, PASCAL ? Il est en effet intéressant de noter que les langages de programmation satisfont à une partie des critères requis. Ces langages sont définis de façon remarquablement précise et non ambiguë (au moins si on les compare à tout système non formel) ; leur nature les rend susceptibles d'un traitement automatique, et ils produisent des aides à la documentation intéressantes (tables de références croisées, listes de tables de symboles, etc.). Quelques-uns ont acquis un haut degré de normalisation, qui les rend indépendants d'une machine particulière ou d'un système d'exploitation. Enfin, l'expérience a montré qu'ils étaient assez facilement compréhensibles et pouvaient être enseignés sur une large échelle.

Ce qui rend les langages de programmation impropres à la spécification, c'est à la fois un trop grand *niveau de détail* et leur caractère essentiellement *dynamique*.

Le premier de ces deux points se traduit par l'obligation pour l'utilisateur de préciser trop de choses trop vite. Dans l'ordre des structures de données, par exemple, une déclaration PL/1 de la forme

```
DCL 1 COMPTE (1000),
    2 CREDIT,
    2 DEBIT,
    2 SOLDE ;
```

implique qu'on a choisi une fois pour toutes un certain mode de représentation de la structure des données ou du fichier, alors qu'il aurait pu être plus sage de recalculer les soldes à partir des

crédits et des débits chaque fois que l'on en avait besoin, ou d'utiliser une approche intermédiaire ("mémo-fonctions").

Le second problème mentionné (l'aspect dynamique, ou *procédural*, des langages de programmation usuels) est dû au fait que les langages de programmation obligent à décrire *comment* sont calculés les résultats, alors que l'objet d'une spécification se limite à la question "*que* calcule-t-on ?". Un des exemples les plus simples est, dans le domaine des structures de contrôle, l'obligation faite au programmeur, par la plupart des langages, de programmer séquentiellement, c'est-à-dire de choisir un ordre d'exécution, même si celui-ci n'a aucune importance pour le problème.

Ces deux "défauts" des langages de programmation (excès de détail et aspect dynamique) conduisent à se tourner vers d'autres formalismes pour les problèmes de spécification. Il est toutefois intéressant de noter que l'évolution des langages de programmation tend à réduire ces deux défauts par des *possibilités d'abstraction*. L'*abstraction procédurale*, fournie en particulier par les appels de sous-programmes, permet de donner un nom à un sous-processus en négligeant temporairement son contenu :

CALL XX (A, B, . . .)

La méthodologie actuelle de la programmation a systématisé cette idée sous le nom de *conception descendante*. L'*abstraction de données*, offerte par les langages de programmation les plus récents, procède de la même approche pour l'étude des données, en particulier en liaison avec la notion de *type abstrait de données* (cf. IV.6 ci-dessous). Enfin, les difficultés provenant du caractère trop "dynamique" des langages de programmation traditionnels ont été étudiées depuis longtemps en liaison avec le développement des *langages non-procéduraux*, particulièrement LISP et ses dérivés (et dans une moindre mesure, APL). Il est bien connu, par exemple, que pour certains types de problèmes, les définitions récursives permettent une expression de programmes plus proche de la notation mathématique que les formulations itératives. La programmation "fonctionnelle" non-procédurale a été discutée récemment par Backus [4].

IV. QUELQUES FORMALISMES DE SPECIFICATION

IV.1. Méthodes d'analyse et de conception (Warnier, Jackson)

Des *méthodes d'analyse et de conception* largement utilisées comme celles de *Warnier* [18] et *Jackson* [7] pourraient sembler avoir leur place ici. Leur examen et celui d'autres méthodes similaires montrent rapidement toutefois qu'elles ne sont pas adaptées à la spécification ; elles aident à organiser et structurer données et programmes (ceux-ci étant déduits de celles-là dans les deux méthodes citées), mais ne fournissent que peu de recours quant à la manière de poser le problème. Une bonne méthode de conception n'est pas nécessairement une bonne méthode de spécification.

IV.2. Tables de décision, diagrammes de transition, réseaux de Pétri

Un certain nombre de formalismes couramment utilisés pour la spécification et la conception des programmes ont en commun deux caractéristiques importantes : ils sont fondés sur une description par états de transitions et ils se prêtent aisément à une représentation graphique. Les tables de décision, les diagrammes de transition et les réseaux de Pétri appartiennent à cette classe.

Le principe des *tables de décision* [9] est de décrire un processus par les transitions à effectuer en fonction de la situation actuelle et d'une certaine combinaison de conditions externes (fig. 1).

Conditions					
	retrait demandé ≤ solde	oui	oui	non	non
	autorisation de dépassement	oui	non	non	oui
Actions					
	autoriser le retrait	oui	oui	non	oui
	envoyer un avertissement	non	non	oui	oui

Figure 1. – Table de décision

Les tables de décision ont plusieurs avantages. Leur principe est simple et aisé à enseigner ; il n'y a pas d'ambiguités ; elles obligent le concepteur à des vérifications, puisque toutes les combinaisons possibles d'entrée doivent être examinées (malheureusement, cela semble être une pratique courante que de laisser beaucoup de positions de la table en blanc, c'est-à-dire non spécifiées) ; elles se prêtent à la vérification automatique ; elles peuvent être traduites en programmes ("conversion des tables de décision") ; comme nous le verrons, il existe des moyens naturels de les représenter graphiquement. Du point de vue de la spécification, cependant, elles sont beaucoup trop procédurales et algorithmiques, prescrivant un enchaînement rigoureux d'actions. Par ailleurs, l'emploi des tables de décision amène dans tout problème un peu compliqué à considérer un nombre considérable de combinaisons, et il devient difficile de conserver le contrôle du processus conceptuel, même par l'emploi de sous-tables.

Une idée voisine est celle des *diagrammes de transition*, que l'on rencontre couramment dans le domaine de la compilation. Initialement appliquée à la description de la structure lexicale des langages de programmation, c'est-à-dire des langages réguliers (ou automates finis), elle a été étendue par Wirth, dans la description de Pascal [8], aux langages sans contexte (fig. 2).

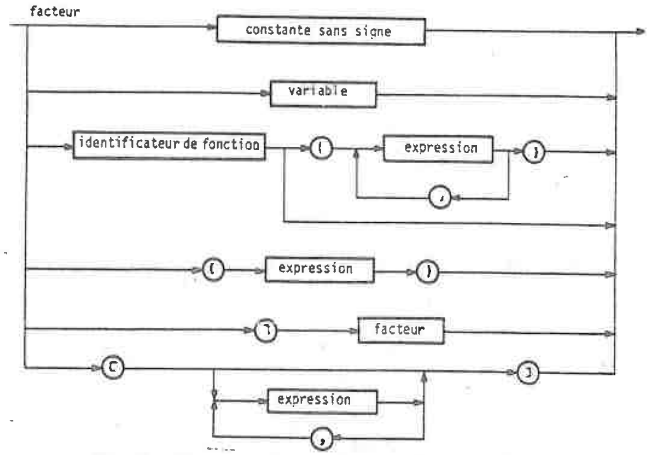


Figure 2. – Diagramme syntaxique (tiré de la définition de Pascal) [8]

Ce type de formalisme se prête bien à la représentation de tout processus de type "états-événements" (fig. 3).

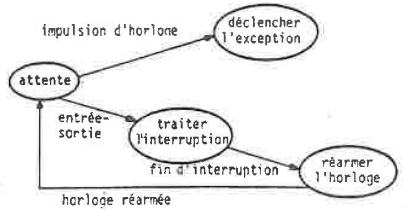


Figure 3

Les diagrammes de transition correspondent à une technique classique de la programmation, celle de la "commande par table" [12], permettant de reporter une partie de la commande du programme dans ses données, les avantages essentiels étant une grande souplesse, la possibilité de changer facilement certains paramètres du processus, et d'obtenir des systèmes évolutifs.

Pour le reste, ils présentent les mêmes avantages et les mêmes défauts que les tables de décision, auxquelles ils fournissent une représentation graphique commode.

Les concepteurs de systèmes parallèles et en temps réel préfèrent généralement utiliser un formalisme graphique particulier, les *réseaux de Pétri* [14]. Un réseau de Pétri (fig. 4) permet de décrire le fonctionnement d'un système, représenté par des *places* qui peuvent abriter un certain nombre de *jetons* associés à des processus ou des événements, et des *transitions* reliant différentes places. Une transition ne peut se "déclencher" que si toutes ses places en entrée sont remplies par un jeton ; son déclenchement envoie alors un jeton à chacune des places en sortie. Si plusieurs transitions peuvent se déclencher à un certain moment, on ne peut prévoir laquelle sera exécutée ; cet aspect non-déterministe est fondamental dans la modélisation des processus parallèles.

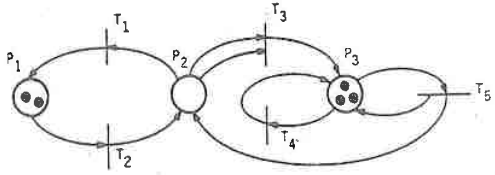


Figure 4. – Réseau de Pétri

Comme les tables de décision, les réseaux de Pétri ont l'avantage de pouvoir être traités et vérifiés automatiquement. Comme elles, cependant, ils conduisent aisément à des descriptions complexes et peu structurées. De même que les organigrammes en programmation, il s'agit d'outils très riches, dans lequels les mécanismes de structuration ne sont pas incorporés, mais doivent être ajoutés *a posteriori*.

IV.3. HIPO

HIPO [16] est une méthode de conception proposée par IBM et fondée sur l'emploi de diagrammes *H*iérarchisés décrivant, pour chaque élément d'un système, ses entrées (*I*nput), le traitement à effectuer (*P*rocess) et ses sorties (*O*utput) (fig. 5).

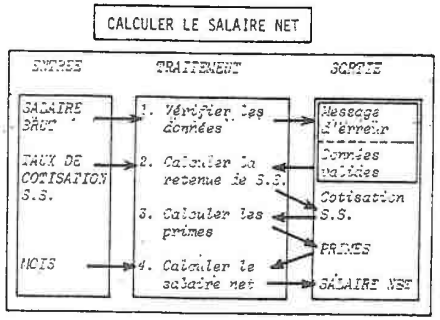


Figure 5. – Diagramme HIPO (schéma de principe)

HIPO permet de décrire un système de façon détaillée et complète. Du point de vue qui nous intéresse, cependant, on notera qu'il s'agit d'une description de caractère trop "procédural" pour être tout à fait satisfaite en tant que spécification. Par ailleurs, la notion de "traitement" demande à être précisée ; les diagrammes HIPO désignent les éléments de traitement par des phrases en langue naturelle ("calculer la somme à payer", etc.), avec leur inévitable manque de précision et leur risque d'ambiguité. D'un point de vue pratique, remplir les diagrammes HIPO semble assez fastidieux. Enfin, la méthode HIPO suppose que l'on a auparavant séparé le traitement en modules, bien que les critères pour une telle décomposition soulèvent des problèmes plus difficiles que ceux rencontrés lors de la description de modules simples ; la meilleure décomposition possible n'est de toute façon probablement pas fondée uniquement sur le traitement, mais aussi sur les structures de données (cf. "Types abstraits de données", section IV.6 et les références [13] et [12] ; le même problème se pose avec la méthode de conception IPT d'IBM). Ainsi HIPO peut être utilisé conjointement avec une autre technique de "modularisation".

IV.4. ISDOS, SREM-SREP

Plusieurs méthodes de spécification et de conception des systèmes informatiques ont été proposées aux Etats-Unis ces dernières années, parmi lesquelles deux paraissent relativement voisines : ISDOS, développée à l'Université du Michigan [17], et SREM-SREP développée par TRW [3]. Elles sont fondées sur l'utilisation d'organigrammes de fonction ("R-nets" dans [3]) assez voisins des formalismes de la section IV.2 (diagrammes de transition, réseaux de Pétri), mais visent à limiter la complexité par la restriction à un petit nombre de modes de combinaison imités des figures de base de la programmation structurée. L'un au moins de ces systèmes (SREM-SREP) offre une double forme d'expression : graphique et textuelle. L'un des objectifs essentiels de ces systèmes est de permettre un traitement automatique des spécifications, et en particulier de multiples vérifications de validité (vérifier que chaque objet est défini avant d'être utilisé, etc.).

ISDOS semble être fondé sur les mêmes principes qu'HIPO, mais permet en outre de traiter le problème de la décomposition en modules par la définition d'"interfaces", et de prendre en compte des informations sur l'environnement technique et humain. La "sémantique" du système est décrite par des commentaires en langage naturel.

Nous ne possédons pas suffisamment d'informations sur ces systèmes pour les analyser en profondeur. Il semble toutefois que, de notre point de vue de la spécification, ils soient trop "orientés traitement" et pas assez "statiques" pour être parfaitement adaptés à des spécifications véritables. Ils manquent en outre de rigueur dans leur définition (ISDOS en particulier inclut des concepts redondants d'"interface", "entrées et sorties" etc.), et semblent se limiter à la description de la "syntaxe" d'un système.

IV.5. SADT

SADT, développé par Softech [11] est en fait une "*T*echnique *S*tructurée" couvrant l'*A*nalyse et la Conception (*D*esign) hiérarchique, et incluant un langage de spécification. Ce langage est fondé sur des schémas duaux : les diagrammes de traitement appelés "actigrammes" et les diagrammes de données appelés "datagrammes". Chaque diagramme est constitué d'un nombre restreint (3 à 6) de boîtes reliées par des flèches. Pour chaque boîte, les flèches en entrée sont de trois types : *support* (machine réelle ou virtuelle), *entrée* et *contrôle* (la distinction entre ces deux derniers types ne semble pas très claire) ; les flèches en sortie représentent les *sorties* (fig. 6).

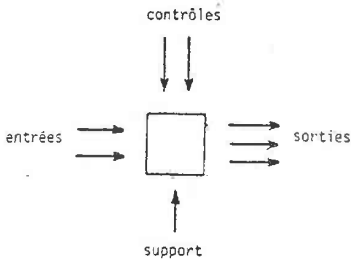


Figure 6. – Elément d'un schéma SADT

Le langage est orienté vers une conception hiérarchique modulaire et cherche à aider le concepteur dans le processus de décomposition. Ce système permet des sorties graphiques et, semble-t-il, des vérifications automatiques.

Les deux critiques principales qui peuvent être faites sont, d'abord, que l'utilisation des diagrammes SADT demande, comme dans le cas de HIPO, un codage assez lourd et fastidieux ; par ailleurs, et ceci est plus important, que SADT, comme les diagrammes de transition et les réseaux de Pétri, permet essentiellement la description de ce que nous avons appelé la syntaxe d'une spécification (c'est-à-dire l'organisation structurelle des modules et leurs relations), à l'exclusion de la sémantique, c'est-à-dire qu'on ne trouve pas d'informations sur la manière dont la sortie d'un élément peut être déduite des entrées et des contrôles. Bien sûr, cette information peut être, et parfois est, exprimée au moyen de "phrases" en langue naturelle accompagnant les diagrammes ; toutefois, le procédé consistant à surajouter ainsi la sémantique est peu pratique, et risque d'entraîner des erreurs du fait qu'il n'est pas cohérent avec le formalisme de base.

SADT pourrait sans doute servir de support graphique utile à la représentation des formalismes que nous allons décrire maintenant.

IV.6. Les types abstraits

La notion de *Type Abstrait de Données* [10] [11] [12] [6] permet de caractériser les objets manipulés par les programmes de façon indépendante des problèmes de représentation physique. Une structure de données sera ainsi définie par la liste des opérations qui permettent d'y accéder et de la manipuler, ainsi que par les propriétés formelles, externes, de ces opérations (fig. 7).

type : COMPTE
fonctions :
créer : → COMPTE
ajouter : COMPTE x ARGENT → COMPTE
retirer : COMPTE x ARGENT → COMPTE
solde : COMPTE → ARGENT
propriétés
pour tout compte c et toute somme a :
solde (créer) = 0
solde (ajouter (c, a)) = solde (c) + a
solde (retirer (c, a)) = solde (c) − a

Figure 7. – Une définition "fonctionnelle" d'un type abstrait

Une conséquence intéressante de cette approche est qu'elle conduit naturellement à une décomposition des systèmes fondée sur les structures de données plutôt que sur les éléments de traitement [13].

On obtient, par cette méthode, des spécifications précises, qui aident à décomposer et à concevoir des programmes "bien structurés". La plus grande partie de la recherche actuelle sur les langages de programmation est consacrée à la conception de langages qui mettent en application (en développant les concepts de SIMULA 67) l'idée de type abstrait de données. De tels langages devraient faciliter l'implantation de systèmes spécifiés de cette manière.

Nous ne discuterons pas plus avant ce type de spécifications, car les types abstraits peuvent être décrits dans le formalisme qui va être décrit ci-dessous, le langage Z.

V. LE LANGAGE Z

V.1. Introduction

Le langage Z, créé par J.R. Abrial [1] [2] est un formalisme pour la spécification, la conception et l'écriture des programmes, visant à permettre à l'utilisateur de conserver à tous les niveaux de l'analyse la maîtrise complète des processus en jeu. Z est fondé sur quelques idées importantes que nous allons maintenant décrire.

La première idée de base est qu'un programme doit être développé par *étapes successives*, permettant petit à petit de se rapprocher d'un système donné et d'un langage de programmation, en partant des spécifications du problème. Z permet de décrire le problème de façon totalement indépendante d'une implémentation éventuelle ; des transformations successives introduisent des concepts algorithmiques dans la spécification et permettent d'obtenir de façon sys-

tématique des programmes écrits dans un langage donné. La seconde idée fondamentale est l'importance attachée à la *spécification statique* d'un problème. L'expérience montre, nous l'avons vu, que l'une des principales difficultés d'un projet informatique est de produire une description statique, rigoureuse et complète, du problème à résoudre, matérialisée par un cahier des charges qui ne soit pas obscurci par un aspect trop analytique, par une trop grande abondance d'informations, par des incohérences fréquentes et par un mélange confus de propos provenant de critères aussi différents que la structuration des données, les contraintes de temps, et les méthodes d'implémentation. Z permet de telles descriptions et permet de les considérer comme des programmes bien que ne contenant aucun élément dynamique, c'est-à-dire rien qui spécifie plus qu'il n'est nécessaire le déroulement dans le temps du traitement informatique.

Une troisième idée importante est que, parmi tous les formalismes possibles pour représenter les objets informatiques et leurs relations, il en existe un plus simple, plus clair et plus général que les autres, parce que résultant de plusieurs siècles d'évolution et de mise à l'épreuve : c'est le formalisme mathématique. Z est donc fondé sur les concepts et les notations de la *théorie des ensembles*. Un certain nombre d'autres sources ont influencé la conception de Z, permettant de traiter les problèmes spécifiques rencontrés en programmation :

– les notations des *langages de programmation* de la famille ALGOL, qui caractérisent la forme externe des spécifications Z ;
– les *modèles relationnels* de bases de données, introduits par Codd ;
– le *lambda-calcul*, déjà utilisé dans la conception de LISP et BCPL ;
– les *types abstraits de données*, introduits par Liskov et Zilles ;
– et, bien sûr, la *programmation structurée*.

V.2. Structure d'une spécification en Z

A partir d'un cahier des charges, le "spécificateur" écrira des *clauses* Z, représentant chacune une formalisation partielle d'une petite partie de ce document.

Il y a 3 sortes de clauses :

V.2.1. Les clauses de TYPE qui introduisent les différentes sortes d'objets qui apparaissent dans le système. Par exemple :

```
type
    LIVRE ;
    LECTEUR some = (moi) ;
    CATEGORIE_DE_LIVRES = {nouvelle, monographie, roman, bande_dessinée} ;
    ordered MOIS = {janvier, février, . . . , décembre} ;
```

V.2.2. Les clauses de RELATION qui définissent les relations (ou fonctions) existant entre les types définis précédemment. Par exemple :

```
relation
    type_de_livre : LIVRE' → CATEGORIE_DE_LIVRES ;
    propriétaire_du_livre : LIVRE → LECTEUR ;
```

V.2.3. Les clauses d'ASSERTION qui expriment les lois logiques du système. Par exemple :

```
assertion
    forall b in LIVRE then
    |   if propriétaire_du_livre(b) = moi then
    |   |   type_de_livre(b) ≠ bande dessinée
    |   end
    end;
```


Du point de vue de la théorie des ensembles, on peut considérer grossièrement les "types" comme des ensembles d'objets ; les "relations", comme des fonctions (mono ou multivaluées) entre ces ensembles ou des combinaisons d'ensembles ; et les "assertions" comme des axiomes dans une théorie logique. Les clauses de type et de relation décrivent la structure, ou syntaxe, d'un système ; les assertions, sa sémantique.

Les seules règles sont qu'une clause doit être compréhensible à partir des précédentes et que deux clauses ne doivent pas se contredire.

La méthode à utiliser pour écrire une spécification Z est itérative et de nature descendante. La première étape sera consacrée à une lecture soignée et répétitive du cahier des charges. La première lecture permettra d'obtenir les types de base du problème, quelques relations et peu d'assertions. La lecture suivante déterminera tous ou presque tous les types, et quelques nouvelles relations et assertions. Des lectures ultérieures permettront d'affiner les types d'objets, les relations et les assertions.

Une des caractéristiques les plus agréables de Z est que, grâce à la manière dont le langage a été conçu, les questions qui surgissent au cours des étapes sont les questions véritablement importantes pour une compréhension complète du problème. Autrement dit, on doit répondre à la bonne question au bon moment. Ces questions pourront être des types suivants :

– Existe-t-il une *relation d'ordre intrinsèque* entre les objets d'un type (par exemple, dans les mois d'une année, mais peut-être pas dans l'ensemble des transactions bancaires) ?

– Une certaine relation est-elle *fonctionnelle* (c'est-à-dire monovaluée) ?

Par exemple :

PARENTS : *PERSONNE* → *PERSONNE* ne l'est pas ;
époux : *PERSONNE* → *PERSONNE* l'est.

– Une certaine relation est-elle *totale* ou *partielle* (*âge* et *sexe* sont totales, *époux* ne l'est pas) ?

– L'inverse d'une relation a-t-il une signification pour le problème, comme dans :

appartient_à : *LIVRE* ↔ *BIBLIOTHEQUE* : *set CATALOGUE*

(où la fonction inverse *CATALOGUE* est multivaluée).

– Quelles sont les propriétés ("assertions") d'une relation donnée ? Il est très instructif dans la pratique d'essayer de les écrire formellement et de découvrir des erreurs ou des incohérences dans le cahier des charges !

Quand les itérations sur le texte sont terminées, la spécification formelle est considérée comme une description complète et cohérente du système : ce sera la première *version* du système.

Toutefois, la spécification n'est pas terminée : des transformations et des améliorations vont suivre. Elle seront générales et systématiques ; la seule contrainte est que toutes les versions devront rester sémantiquement équivalentes. On trouvera notamment les transformations suivantes :

– Groupement de relations (représentant des fonctions totales) en une relation dont le domaine est le produit cartésien des domaines initiaux ;

– Elimination du non-déterminisme (un ensemble fini peut, par exemple, être "implémenté" comme une suite ordonnée finie) ;

– Elimination de la récursion ;

– Choix d'une méthode d'accès particulière pour les éléments d'un certain type (qui seront représentés par des fichiers ou des structures de données) ;

– etc.

De chaque transformation résulte une nouvelle *version* de la spécification formelle.

V.3. Description succincte du langage

Z comprend deux parties : le noyau, qui est théoriquement suffisant, et les extensions syntaxiques qui sont des constructions utiles en pratique, mais pouvant être exprimées en termes d'éléments du noyau.

V.3.1. Le noyau

V.3.1.1. Types

Les types en Z sont de 3 sortes :

– Certains types sont prédéfinis, à savoir *BOOL*, *NUM* et *CHAR* ;

– D'autres sont définis par leur occurence dans une clause de type ;

– D'autres encore peuvent être construits par composition à partir des précédents, à l'aide des opérateurs suivants :

- ***set***(X) est le type dont les éléments sont les sous-ensembles de X ;
- ***tuple***(X) est le type dont les éléments sont toutes les suites ordonnées d'objets du type X ;
- ***prod***$(X_1, \ldots, X_n)$ est le produit cartésien de $X_1, \ldots, X_n$;
- ***function***(X, Y) est le type dont les éléments sont toutes les fonctions monovaluées totales de X vers Y.

Tous les types sont disjoints.

V.3.1.2. Clauses

Les clauses de spécification sont soit des clauses de type, soit des clauses de relation, soit des clauses d'assertion. Deux autres sortes de clauses sont utiles en pratique : les clauses de définition et les clauses de partition.

a) *Les clauses de type* permettent la définition de nouveaux types, soit en extension, par exemple :

```
type
    FEUX_DE_CIRCULATION = {vert, orange, rouge};
```

soit en ne donnant que le nom de quelques éléments, par exemple :

```
type
    CLUB_DE_Z some = {demuynck, meyer} ;
```

soit en ne donnant que le nom du type si aucun élément n'est connu à l'avance :

```
type
    ENREG_EN_ENTREE ;
```

b) *Les clauses de relation* définissent des relations entre types. Les fonctions monovaluées (totales sauf si ***partial*** est précisé) sont écrites en minuscules :

```
relation
    impôt : prod (CONTRIBUABLE, ANNEE) → NUM ;
```

tandis que les fonctions multivaluées sont écrites en majuscules et précédées du mot ***set*** (ou ***ordered set*** si l'ordre a une signification pour le résultat) :


```
relation
    set FACTURES_DU_MOIS : CLIENT → FACTURE ;
    ordered set CANDIDATS_ACCEPTES : EPREUVE → CANDIDATS ;
```

Si la relation inverse a une importance pratique, elle apparaîtra, avec ses caractéristiques, à la droite de la relation :

```
relation
    numéro_de_voiture : VOITURE ↔ NUMERO : partial voiture_de_numéro
```

c) *Les clauses d'assertion* expriment les propriétés sémantiques des objets de spécifications. Elles correspondent en général à des prédicats. Les opérateurs comprennent ***forall*** :

```
assertion
    forall p in ENTREPRISE then
    |       age(p) ⩽ 65
    end ;
```

et ***if***, comme dans l'exemple suivant qui montre aussi l'utilisation de l'opérateur ***given*** :

```
assertion
    forall p in ENTREPRISE then
    |   given
    |   |   limite = if sexe(p) = masculin then
    |   |   |            65
    |   |   |        elsif sexe(p) = féminin then
    |   |   |            60
    |   |   |        end
    |   then
    |   |   age(p) ⩽ limite
    |   end
    end ;
```

D'autres opérateurs s'appliquent aux relations. ***lambda*** permet la définition de fonctions :

```
assertion
    valeur_absolue = lambda x in NUM ⇒
    |                    if x ⩾ 0 then
    |                    |   x
    |                    elsif x < 0 then
    |                    |   −x
    |                    end
    end ;
```

closure donne la fermeture transitive d'une relation. Cet opérateur est particulièrement important car il permet d'introduire le concept de boucle sans perdre le caractère statique de Z.

d) *Les clauses de définition* sont utilisées comme facilité pour nommer des objets, c'est-à-dire comme macros. Elles sont souvent employées en liaison avec l'opérateur ***subset*** :

```
définition
    REGULIERE = subset m in MATRICE where
    |               déterminant (m) > 0
    end ;
```

REGULIERE n'est pas un type, mais un sous-ensemble du type *MATRICE* ; une convention différente aurait contredit le principe de disjonction des types.

e) *Les clauses de partition* permettent d'exprimer un type comme une union de sous-ensembles disjoints :

```
partition
    EMPLOYES = (COLS_BLANCS, COLS_BLEUS) ;
    EMPLOYES = (EMPLOYES_MASCULINS, EMPLOYES_FEMININS) ;
```

V.3.2. *Extensions syntaxiques*

Les extensions syntaxiques servent à définir des constructions utiles en pratique, mais qui ne sont pas théoriquement indépendantes de celles du noyau. Par exemple, l'opérateur ***exist*** :

```
exist p in ENTREPRISE where
|     rang(p) = chef
end ;
```

est défini à partir de l'opérateur ***forall***. De même, l'opérateur "union d'ensembles" est défini à partir de l'opérateur "intersection d'ensembles" (du fait des contraintes de validité des types, l'union peut seulement être appliquée à des sous-ensembles d'un même type)

V.4. Un exemple

A titre d'exemple, nous incluons ici une courte spécification en Z. Les commentaires sont parenthésés par /* et */.

```
spécification paie ;
/* Le but de l'exemple est d'obtenir une liste des salaires dans une entreprise, un mois donné,
triés par nom d'employé */
    type
        ENTREPRISE, MOIS, EMPLOYE ;
    relation
        salaire : prod (EMPLOYE, MOIS) → NUM ;
        nom : EMPLOYE → tuple (CHAR) ;
        appartenance : EMPLOYE ↔ ENTREPRISE : set PERSONNEL ;
/* On introduit ici une description de l'enregistrement de sortie et des moyens de l'obtenir */
    type
        ENREG ;
    relation
        nom_enr : ENREG → tuple (CHAR) ;
        salaire_enr : ENREG → NUM ;
        liste_salaire : prod (ENTREPRISE, MOIS) → tuple (ENREG) ;
/* pour obtenir une "liste_salaire" triée, définissons la notion de "trié" : */
    relation
        trié : tuple (ENREG) → BOOL ;
/* pour ceci, définissons tout d'abord l' "ordre alphabétique" : */
```



```
    definition
        CHAINE = tuple (CHAR) ;
    relation
        ordalph : prod (CHAINE, CHAINE) → BOOL ;
    assertion /* "ordalph (t1, t2)" signifie "t1 est inférieur ou égal à t2 dans l'ordre alphabétique" */
        ordalph =
            lambda s1, s2 in prod (CHAINE, CHAINE) ⇒
            |   s1 = null or
            |   (s1 ≠ null and s2 ≠ null and
            |     (first (s1) < first (s2) or
            |     (first (s1) ≠ first (s2) and ordalph (tail (s1), tail (s2)))))
            end ;
/* Une définition non récursive aurait aussi été possible. La définition de "trié" est alors : */
    assertion
        trié =
            lambda r in ENREG ⇒
            |       forall v in r then
            |           ordalph (nom_enr (v), nom_enr (next (v))
            |       end
            end ;
/* next (v) désigne l'élément qui suit v dans le tuple. Définisssons maintenant la sortie désirée : */
    assertion
        forall e in ENTREPRISE then
        |   given
        |       ℓ = liste_salaire (e)
        |   then
        |       trié (ℓ) and
        |       forall p in PERSONNEL (ℓ) then
        |           p in ℓ
        |       end
        |   end
        end ;
```

V.5. Portée, utilisation et problèmes de Z

Z a été utilisé jusqu'ici sur une échelle relativement modeste. Il a été appliqué à la spécification de systèmes informatiques de gestion ; il est employé comme outil dans un travail en cours dont le but est d'améliorer la compréhension et la mise en œuvre des algorithmes numériques, et a été appliqué à la spécification d'autres programmes, comme des traducteurs de macros. L'expérience a montré aussi que Z était applicable de manière tout à fait satisfaisante à la modélisation des systèmes parallèles, comme les processus "producteur-consommateur", les systèmes décrits par les résaux de Pétri, etc. Nous avons aussi trouvé Z intéressant comme outil théorique pour l'étude des processus de calcul.

Formalismes / Critères	Langages de programmation	Tables de decision	Diagrammes de transition	Réseaux de Pétri	HIPO	Warnier Jackson	SREM-SREP	ISDOS	SADT	Types abstraits de données	Z
Statique ou dynamique	Dynamique	Habituellement dynamique, mais peut-être statique	Dynamique	Dynamique	Dynamique	Dynamique	?	Plutôt dynamique	Statique	Statique	Statique
Niveau d'abstraction	Des facilités d'abstraction existent, mais tous les détails doivent finalement être donnés	Bas	Bas	Bas	Bas (dépend de l'utilisation)	Des facilités d'abstraction existent, mais tous les détails doivent finalement être donnés	Haut (?)	Haut	Haut	Haut	Haut
Généralité (c'est-à-dire portée des domaines d'application possibles)	Dépend du langage	Général	Général	Systèmes parallèles et temps-réel	Plutôt orienté vers les problèmes de gestion	Réputé pour être général mais très orienté vers les problèmes de gestion	Général plutôt orienté vers les problèmes temps-réel	Orienté vers les problèmes de gestion	Général ?	Général	Général
Aides dans le processus de décomposition ?	Oui, au moins dans les bons langages	Non	Non	Non	Non	Oui	Oui	Oui	Oui	Oui	Oui
Existence d'une (bonne) méthodologie de spécification et de conception ?	Normalement non (des efforts récents vont dans cette direction)	Non	Non	Non	Mieux que rien	Oui	Oui	Oui	Oui	Oui	Oui
Base théorique saine et rigoureuse ?	En partie pour quelques langages	Oui	Peut être définie	Oui	Non	Non	?	Non	Non	Oui (algèbres initiales)	Oui
Cadre linguistique		Tables	Diagrammes (ou automates)	Diagrammes (ou automates)	Langage naturel	Arbres (plus quelques constructions des langages de programmation)	Diagrammes plus quelques constructions des langages de programmation	Peu clair	Diagrammes	Théorie des ensembles, calcul des prédicats	Théorie des ensembles calcul des prédicats, syntaxe type-ALGOL
Description de la "syntaxe" (a "sémantique", ou les deux ?	Les deux	Les deux	Syntaxe	Syntaxe	Les deux	Les deux	Les deux (?)	Syntaxe	Syntaxe	Les deux	Les deux
Support graphique ?	Non	Non	Oui	Oui	Non	Arbres	Oui	Non (?)	Oui	Non	Non
Utilisateurs	Programmeurs	Utilisateurs, analystes, programmeurs	Utilisateurs, analystes, programmeurs	Analystes	Analystes	Analystes, programmeurs	?	Analystes	Analystes	Analystes, programmeurs	Utilisateurs, (?!) analystes, programmeurs.
Facilité d'enseignement	Facile	Facile	Facile	Modérément difficile	Facile	Plutôt facile	?	?	?	Difficile	Modérément difficile

Figure 8

Z semble applicable de façon fructueuse à tous les domaines d'application possibles en informatique. Toutefois, un plus large champ d'expérience nous manque manifestement pour justifier cette affirmation. Une telle expérience apporterait sans aucun doute des changements et des améliorations au langage.

Un des aspects de Z qui pourrait devenir fondamental mais demande beaucoup de travail est celui des transformations systématiques, qui permettent d'affiner une spécification en en préservant la sémantique et d'arriver petit à petit à un programme réaliste. Un catalogue d'environ 400 transformations a été écrit [2], qui se réfère à une ancienne version du langage ; toutefois, leur nombre même suggère qu'une meilleure approche reste à trouver – à moins que l'on considère que les transformations ne représentent que des équivalents bien exprimés des bons vieux "trucs" du programmeur, qui sont certainement aussi nombreux.

VI. RESUME ET CONCLUSIONS

VI.1. Résumé de l'étude comparative

Le tableau de la figure 8 montre le résultat obtenu en appliquant les critères définis dans la section II.2 aux formalismes étudiés. Le signe "–" signifie que le critère est sans signification pour le formalisme en question ; le signe "?" que nous manquons d'information sur ce formalisme pour ce critère.

L'étude montre de façon évidente que le problème des langages de spécification est loin d'être résolu. Nous pensons toutefois que quelques conditions nécessaires fondamentales ont été mises en évidence pour ces langages et nous allons les résumer.

VI.2. Conditions pour un bon langage de spécifications

Tout d'abord, pour être tout à fait digne du nom de "langage de spécification", un formalisme doit être statique et non-algorithmique, c'est-à-dire laisser de côté tous les aspects procéduraux.

Ensuite, il doit permettre d'exprimer à la fois la structure "syntaxique" d'un système et sa sémantique. Mais ce but ne doit pas être atteint aux dépens du précédent, c'est-à-dire que la description sémantique doit être non-procédurale. De tous les formalismes étudiés, les types abstraits de données et Z nous semblent les mieux à même de remplir ces objectifs.

Un autre problème est celui des supports graphiques, qui sont des éléments inclus à la base dans les diagrammes de transition, les réseaux de Pétri, SREM-SREP et SADT, mais non pour les tables de décision, les types abstraits de données ou Z. L'utilisation de tels supports met en avant un problème très général. Bien qu'ils soient très agréables au premier abord ("mieux vaut un bon dessin. . .") ils peuvent aussi être trompeurs, car la quantité d'information qu'on peut inclure dans un dessin est très limitée, même si c'est l'information la plus importante et si elle apparaît très clairement. Nous pensons par exemple que le problème principal de SADT est dû au fait que l'information sémantique ne peut faire partie des diagrammes eux-mêmes, mais doit être écrite en phrases du langage naturel, ce qui détruit l'intégrité conceptuelle du modèle.

D'un autre côté, les vertus pédagogiques des dessins sont telles qu'il serait dommage de ne pas essayer de les utiliser à des fins d'éclaircissement. Aussi, pouvons-nous hasarder la proposition suivante : Utiliser quelque chose comme SADT comme support graphique pour vulgariser et enseigner des spécifications écrites, transformées et contrôlées dans quelque chose comme Z.

Un autre problème est celui de la rigueur. Nous pensons qu'un langage de spécification doit avoir une base théorique bien définie et de taille restreinte – ce qui peut aussi impliquer que le langage sera trop mathématique, et difficile à comprendre pour des praticiens. Le problème débouche sur un autre, que nous n'avons pas étudié : qui écrira les spécifications et qui les utilisera ? Dans l'approche Z, comme dans la plupart des autres, sauf les plus simples (tables de décision, diagrammes de transition), la spécification est rédigée par un spécialiste, et peu d'utilisateurs finaux seront assez "sophistiqués" pour comprendre une spécification écrite dans ce langage. Ainsi, l'organisation devra comprendre un "médiateur" capable de retranscrire la spécification dans les propres termes de l'utilisateur final.

Parmi toutes les approches étudiées, Z est manifestement celle qui nous semble la plus prometteuse, et ce pour de nombreuses raisons : Z est applicable à tous les domaines de l'informatique ; il est statique et non-procédural, mais prépare cependant le terrain pour des transforma-

tions qui aident à parvenir aux programmes ; il a une base théorique saine ; il est complet, c'est-à-dire que les spécifications peuvent être entièrement exprimées à l'intérieur du cadre du langage ; et il combine la précision des mathématiques avec l'élégance d'expression des langages de programmation actuels.

Il est clair, cependant, que Z ne bénéficie pas d'une aussi large expérience que tous les autres systèmes étudiés. Il ne serait pas réaliste, en outre, d'ignorer les problèmes psychologiques et techniques qui peuvent se produire lors de l'introduction d'un tel formalisme. Nous pensons toutefois que si un langage de cette espèce se répand, la fiabilité et la souplesse des systèmes informatiques ainsi spécifiés seront bien supérieures aux critères aujourd'hui courants.

Remerciements

Nous remercions C. Baudoin pour son aide dans la préparation de cet article.

BIBLIOGRAPHIE

[1] ABRIAL Jean-Raymond. – *Z : A Specification Language,* IFIP, Kyoto, Août 1978.

[2] ABRIAL Jean-Raymond. – *Les transformations du langage Z* ; Rapport 1977.

[3] ALFORD MACK W. – A Requirements Engineering Methodology for Real-time Processing Environments, *IEEE Transactions on Software Engineering,* vol. SE-3, pages 60-69, Janvier 1977.

[4] BACKUS John. – Can Programming be liberated from the von Neumann Style ? A Functional Style and its Algebra of Programs ; *Communications of the ACM,* vol. 21, n° 8, pages 613-357, Août 1978.

[5] DEMUYNCK Michel et MEYER Bertrand. – *Les langages de spécification : vers une meilleure analyse des problèmes informatiques et des programmes plus fiables ;* Actes de la Convention Informatique, Paris (Palais des Congrès), Septembre 1978.

[6] GUTTAG John. – *The Specification and Application to Programming of Abstract Data Types;* rapport CSRG-59, Université de Toronto, Septembre 1975.

[7] JACKSON M.O. – *Principles of Program Design;* Academic Press, London, 1975.

[8] JENSEN Kathleen et WIRTH Niklaus. – *PASCAL User Manual and Report;* Springer-Verlag (Berlin), 2e édition, 1975.

[9] KIRK H.W. – Use of Decision Tables in Computer Programming; *Communications of the ACM,* vol. 8, n° 1, pages 41-43, Janvier 1965.

[10] LISKOV Barbara H. et ZILLES, Stephen N. – Specifications Techniques for Data Abstractions; *IEEE Transactions of Software Engineering,* vol. SE-1, n° 1, pages 7-18, Mars 1975.

[11] MEYER Bertrand. – Description des Structures de Données ; *Bulletin de la Direction des Etudes et Recherches EDF, série Mathématiques Informatiques,* n° 2, pages 81-90, décembre 1976. Egalement dans : *Bulletin du Groupe Programmation et Langages de l'AFCET* (GROPLAN), n° 2, janvier 1978.

[12] MEYER Bertrand et BAUDOIN Claude. – *Méthodes de programmation;* Eyrolles, Paris, 1978.

[13] PARNAS David L. – A Technique for Software Module Specification with Examples; *Communications of the ACM,* vol. 15, n° 12, pages 1053-1058, Décembre 1972.

[14] PETRI H. – *Kommunikation mit Automaten;* Thèse, Frankfurt-oder-Main, 1962.

[15] ROSS, Douglas T. et al. – Special Section on Requirements Analysis; *IEEE Transactions on Software Engineering,* vol. SE-3, n° 1, pages 2-34, janvier 1977.

[16] STAY J.F. – HIPO and Integrated Program Design; *IBM Systems Journal,* vol. 15, n° 2, pages 143-154, 1976.

[17] TEICHROEW D. et SAYANI H. – *Automation of System Building;* Datamation, pages 25-30, Août 1971.

[18] WARNIER J.D. – Ensemble de manuels : *Entraînement à la Construction des programmes d'Informatique; Entraînement à la Programmatin; L'Organisation des Données d'un Système; Guide LCS; Les Procédures de Traitement et leurs Données,* etc. Editions d'Organisation, Paris.

Service Informatique et Mathématiques Appliquées

Quelques concepts importants des langages de programmation modernes, et leur expression en SIMULA 67

[79b]

B. MEYER *

(*) Ingénieur-Chercheur au Département Méthodes et Moyens de l'Informatique

Cet article a fait l'objet d'une communication aux journées de travail sur le thème "Panorama des Langages d'aujourd'hui" organisées par le groupe GROPLAN (Programmation et Langages) de l'AFCET, à Cargèse (Corse), du 14 au 22 mai 1979. Actes dans le Bulletin GROPLAN, numéros 8 et 9.

E.D.F. – BULLETIN DE LA DIRECTION DES ETUDES ET RECHERCHES
SERIE C – MATHEMATIQUES, INFORMATIQUE n° 1, 1980, pp. 89-150.

I - INTRODUCTION

Il est devenu presque de règle, lorsqu'on discute quelques-unes des principales idées actuelles sur la conception des langages de programmation, et en particulier tout ce qui tourne autour du concept d'"abstraction de données", de citer au passage un langage précurseur, SIMULA 67, et d'inclure en bibliographie le document de base [4]. SIMULA mérite mieux qu'une mention incantatoire : il s'agit d'un langage de programmation actif, pratiqué dans le monde par de nombreux utilisateurs, mis en oeuvre sur un grand nombre de systèmes(1) et dans certains cas très efficacement, convenablement normalisé, et pouvant prétendre à la qualité d'un langage "industriel".

Le but du présent article est d'étudier de façon critique les techniques introduites par SIMULA pour résoudre certaines des questions que se posent aujourd'hui les concepteurs de langages, en particulier dans cinq domaines : modularité et types abstraits ; composition descendante ; conception de modules d'application (et de langages) spécialisés ; généricité ; quasi-parallélisme et coprogrammes.

Les spécialistes de SIMULA pourront trouver que certaines caractéristiques du langage ne sont pas présentées ici dans toute leur richesse : précisément, le but poursuivi ici n'est pas d'explorer toutes les finesses de SIMULA mais, au contraire, de chercher à en retenir les constructions simples qui peuvent aider à mettre en pratique quelques concepts méthodologiques importants. Il y a en fait deux écueils - entre autres - dans la discussion d'un langage de programmation. L'un consiste à prêter une attention exagérée aux finesses ; l'autre, opposé, à négliger l'importance des constructions linguistiques en remarquant qu'on peut tout faire dans n'importe quelle notation théoriquement assez puissante : FORTRAN, BASIC, la machine de Turing, etc.

(1) CDC 6600/7600, CYBER 70, IBM 360/370, UNIVAC 1108/1110, DEC-10, CII 10070/IRIS 80,....

Pour éviter de tomber dans le "fossé de Turing", il convient de garder présent à l'esprit que le choix d'un langage de programmation, ou plutôt d'un système de programmation résulte d'un compromis entre :

- d'une part, la facilité d'expression, c'est-à-dire (l'inverse de) la quantité de contorsions nécessaire pour faire passer dans le langage les structures conceptuelles qui déterminent l'organisation d'un programme ;

- et d'autre part, tous les aspects pratiques qui font la différence entre les systèmes de qualité "industrielle" et les autres : compilation séparée ; possibilité d'appeler des sous-programmes écrits dans d'autres langages ; possibilité de constituer des bibliothèques de modules d'application ; outils d'aide à la mise au point ; entrées et sorties réalistes ; possibilités de manipulation de fichiers, accès direct.

Nous reviendrons en conclusion sur ces critères. Il nous paraît cependant important de garder présent à l'esprit, dans la discussion ci-après, où l'on considère l'application à SIMULA de concepts qui font leur entrée en force dans une pléiade de propositions récentes (GREEN-ADA [9], CLU [14], ALPHARD [21], MEFIA [20], EUCLID [13], MESA [7], CSP [8] etc.), que SIMULA correspond à un ensemble de systèmes solides et utilisables dès aujourd'hui.

II - DONNEES DE BASE

SIMULA 67, faisant suite à un langage de simulation appelé SIMULA tout court [3], a été conçu en 1967 par Dahl et Nygaard au Centre de Calcul Norvégien (NCC) d'Oslo. Il s'agit en fait d'un langage de programmation tout à fait général, dans lequel la simulation n'est qu'une application possible ; plus précisément, la simulation est traitée par un "module" prédéfini, mais de même nature que tous ceux que l'on peut créer à volonté, grâce aux propriétés du langage en ce domaine, que nous aborderons au paragraphe IV.4. L'erreur historique des promoteurs du langage a sans doute été de ne pas changer son nom en conséquence.

Le document de référence officiel est [4], complété par des guides propres aux versions mises en oeuvre sur différentes machine. L'introduction la plus lisible est l'ouvrage "SIMULA BEGIN" [1] .

SIMULA 67 est une extension, presqu'entièrement "compatible vers le haut"(1), d'ALGOL 60. Nous donnons ici rapidement, pour le lecteur ne connaissant pas ALGOL 60, les quelques concepts algoliques de base nécessaires à la compréhension de la suite.

Structure des programmes

La notion fondamentale d'ALGOL 60 est celle de bloc. Un bloc a la forme

begin
déclaration 1;
déclaration 2;
.....
déclaration m;
} peuvent être absentes
instruction 1;
instruction 2;
⋮
instruction n
end

(1) différences essentielles : pas de déclarations *own* (*rémanent*) ; le passage par nom n'est pas le mode par défaut.

où les *"instructions"* peuvent utiliser les variables ou tableaux définis dans les *"déclarations"*, s'il y en a.

Un programme est un bloc de cette forme. Une instruction peut être une instruction de base (affectation par exemple), mais aussi un bloc, c'est-à-dire que les blocs peuvent être imbriqués les uns dans les autres. Une instruction d'un bloc interne peut utiliser des objets déclarés dans un bloc englobant.

Instructions de base et structures de contrôle

L'affectation est notée

variable := expression

(utilisateurs de FORTRAN, notez l'emploi de := plutôt que = utilisé comme opérateur d'égalité, *.EQ.* en FORTRAN).

L'instruction conditionnelle s'écrit

if condition then
instruction 1
else
instruction 2

où l'un des deux *instructions* sera exécutée selon que la *condition* est vraie ou fausse. Notez que grâce au mécanisme des blocs ces instructions peuvent être aussi complexes qu'on le désire. La partie *else instruction 2* peut être absente.

Les boucles s'écrivent

for i:= a step b until c do
instruction

(boucle avec compteur), ou

while condition do
instruction

(exécution de *instruction* tant que *condition* est vraie, donc peut-être jamais).

Procédures

Les sous-programmes s'appellent en ALGOL procédures. Une procédure sans résultat (*subroutine* en FORTRAN) est déclarée comme

type des arguments

procedure $p\ (arg_1,\ldots.arg_n)$; *integer* $arg_1;\ldots..;$
instruction (en général un bloc)

Une procédure à résultat, par exemple réel (*fonction* en FORTRAN), est déclarée comme :

real procedure $q\ (arg_1,\ldots arg_n)$;.....;
instruction, où l'on affecte à q
(en général un bloc)

Dans les deux cas, il s'agit de déclarations pouvant être imbriquées comme les autres : une procédure est donc appelable dans le bloc où elle est déclarée et dans les blocs internes ; elle peut être déclarée locale à une autre procédure.

Pour appeler une procédure, on écrit simplement son nom suivi d'une liste d'arguments réels, s'il y a lieu (une procédure peut ne pas avoir d'arguments) :

$p(a_1, \ldots..a_n);$
$r := q(b_1,\ldots..b_n)$

Les procédures peuvent être récursives :

procédure s(.....) ; .....
begin
.....
.....;s(.....);..
end

Déclarations et allocation dynamique

Tout objet utilisé dans une instruction doit être déclaré dans le même bloc ou un bloc englobant. Une déclaration de variable a la forme

```
integer x,y,z ;
real t ;
etc.
```

Une déclaration de tableau a la forme

```
boolean array b(0:5)
real array a(1:m, 1:n);
etc.
```

Dans le second cas, m et n doivent avoir été déclarées et initialisées dans un bloc englobant. A chaque nouvelle exécution du bloc dans lequel a est déclaré, il aura une nouvelle taille dépendant de m et n. Les valeurs précédentes seront perdues.

Manipulation de textes, entrées, sorties

SIMULA ajoute à ALGOL des mécanismes puissants, et particulièrement agréables à utiliser, pour :

- la manipulation de chaînes de caractères,
- les entrées et sorties.

Nous renvoyons à la note Atelier logiciel n°16 pour la description de ces propriétés.

III - MODULARITE ET REPRESENTATION DE TYPES ABSTRAITS

III.1 - Une vue systémique de la programmation

Nous commencerons notre revue de quelques-uns des grands concepts actuels par deux problèmes aujourd'hui reconnus comme étroitement liés : celui de la modularité et celui des types de données.

La difficulté majeure de la conception d'un programme d'une certaine taille tient à la méthode choisie pour le diviser en entités homogènes ou "modules". C'est elle qui conditionne en effet l'organisation du projet, la répartition éventuelle du travail, le succès ou l'échec de la phase d'intégration des différents éléments, et la faculté d'adaptation du produit final à des modifications de ses spécifications.

La méthode traditionnelle de division d'un programme en "sous-programmes", fondée sous une décomposition du traitement à effectuer, ne répond pas entièrement à ces exigences. Plusieurs travaux, en particulier ceux de Parnas [17], [18], [19] ont montré qu'on obtenait une décomposition plus solide en se fondant plutôt sur le critère dual du précédent, celui des structures de données, ou types, intervenant dans le système. On peut brièvement justifier ce principe en remarquant qu'un programme est un modèle d'un certain système physique, et qu'une structure satisfaisante pour un tel modèle est celle qui décrit le système en termes d'objets et de relations entre ces objets ; les objets en question sont, soit primitifs (types de base), soit eux-mêmes complexes (sous-systèmes). Dans l'évolution d'un tel modèle, on peut penser que les objets sont un élément plus stable que les relations, et qu'il est donc naturel de fonder sur eux la structure du programme. C'est ce qu'on peut appeler une conception "systémique" de la programmation.

Un "objet" intervenant dans une telle décomposition est un ensemble de données qui doit pouvoir être caractérisé par des propriétés purement externes, et indépendamment des problèmes de représentation. L'exemple le plus fréquemment cité est celui d'une pile d'entiers, qui peut être caractérisée par un ensemble d'opérateurs possédant certaines propriétés formelles (exemple 1). La classe des objets vérifiant ces propriétés est appelée un type abstrait dont l'exemple 1 donne la spécification fonctionnelle [15].

Fonctions

pilevide : *PILE* → *BOOLEEN*
empiler : *ENTIER* x *PILE* → *PILE*
dépiler : *PILE* → *ENTIER* x *PILE*
— *fonction partielle*

Assertions

pour tout p : PILE, e : ENTIER alors

| ∿ *pilevide (empiler (e,p))* et
| *dépiler (empiler (e,p)) = (e,p)* et
| [∿ *pilevide (p) => empiler (dépiler (p)) = p*]

fin

Exemple 1 : Spécification d'une pile d'entiers

Un grand nombre de langages de programmation récents cherchent à fournir ce mode de décomposition en proposant une structure linguistique permettant de regrouper la description d'une structure de données et de tous les opérateurs permettant de la manipuler, c'est-à-dire la mise en oeuvre d'un type abstrait. Ce mécanisme est fourni sous le nom de "grappe" en CLU, "forme" en ALPHARD, "module" en EUCLID,"package" en ADA, "enveloppe" en PASCAL PLUS, etc.

En reprenant l'exemple 1, ainsi, un module de gestion de pile conforme à la spécification précédente devra offrir aux autres modules la possibilité de manipuler une ou plusieurs piles grâce aux opérateurs *dépiler, empiler, pilevide*.

Lorsqu'on inclut une telle possibilité dans un langage de programmation, plusieurs choix doivent être faits :

- Les types peuvent être "statiques" ou "dynamiques". Dans le premier cas, un module décrit un exemplaire de la structure de données associée ; dans le second, il ne représente aucun objet concret, mais seulement un modèle de la structure de données, dont les utilisateurs peuvent alors créer à volonté de nouveaux exemplaires (par des opérateurs de création, *créerpile* dans notre exemple, qui doivent être ajoutés à la spécification abstraite).

- Certains langages permettent d'accéder de l'extérieur aux données internes d'un module. On peut au contraire n'autoriser que l'emploi des opérations de la spécification externe, en restreignant les "droits d'accès" à certains sous-programmes et à certaines données désignées comme "exportables". L'intérêt d'une telle restriction est de limiter la propagation des erreurs entre modules, et de faire en sorte qu'une modification de représentation interne dans un module (par exemple, le passage d'une représentation contiguë à une représentation chaînée pour une pile) soit sans effet sur les autres.

- On peut poursuivre plus loin encore l'objectif de protection, en permettant comme GIPSY [2] de définir des "droits d'accès" différents à un même module ; ou l'objectif de séparation entre spécification et représentation, en offrant comme ADA la compilation séparée de ces deux parties d'un module.

III.2 - Les classes en SIMULA

SIMULA fut le premier langage important à offrir une représentation de module correspondant aux définitions précédentes : la classe. Pour replacer cette notion par référence à la discussion ci-dessus, on peut définir une classe comme une représentation de type abstrait, dynamique, sans protection des données internes ni séparation physique entre la partie "spécification" et la partie "représentation".(1)

(1) Note : Les programmes présentés dans cet article n'ont pas été testés. Une version corrigée sera adressée sur demande.

Déclaration d'une classe

Une classe est définie en SIMULA par une déclaration de la forme

class nom_de_classe ; corps_de_classe

ou *class nom_de_classe (liste_de_paramètres_formels) ; corps_de_classe*

où le *corps_de_classe* est un bloc de la forme

begin

déclaration d'attributs { *variables* / *procédures* } ;

actions d'initialisation

end

ou, dans les cas dégénérés, une seule instruction d'initialisation (il n'y a pas alors d'attributs). Les "attributs" sont la représentation concrète des propriétés communes à tous les objets d'un type abstrait associé à la classe. Les "actions d'initialisation" sont destinées à être appliquées à tout exemplaire de la classe à sa création.

A titre d'illustration, on trouvera à l'exemple 2 la déclaration d'une classe réalisant une représentation concrète restreinte du type abstrait "pile d'entiers" du paragraphe précédent. On a choisi une représentation contiguë par un tableau, et ajouté à la spécification une restriction de taille représentée par le paramètre formel *n*. Les "attributs" sont le tableau *p*, l'indicateur *sommet*, et les procédures *dépiler, empiler, pilevide*. On a omis les procédures *erreur_pile_pleine* et *erreur_pile_vide*.

Objet dont le type est défini par une classe

Dans un programme incluant une telle déclaration de classe, on pourra déclarer des variables du type correspondant en écrivant :

ref (nom_de_classe) $o_1, o_2, \ldots, o_n$;

L'emploi du mot-clé *ref* est lié à la représentation physique d'une telle variable (pointeur), mais il est souvent utile conceptuellement

```
class pilentier (n) ; integer n ;
    begin
    comment attributs ;
      comment variables ;
          integer array p (1 : n) ;
          integer sommet ;
      comment procédures ;
          procedure empiler (x) ; integer x ;
                if sommet = n then erreur_pile_pleine
                else
                  begin sommet := sommet + 1 ;
                        p (sommet) := x
                  end empiler ;
          integer procedure dépiler ;
                if pilevide then erreur_pile_vide
                else
                  begin   dépiler := p (sommet) ;
                  sommet := sommet - 1 ;
                  end dépiler ;
          boolean procedure pilevide ;
                pilevide := (sommet = 0) ;
    comment action d'initialisation (à la création d'une pile
                                     initialement vide) ;
      sommet := 0
    end pilentier
```

EXEMPLE 2 : PILE D'ENTIERS CONTIGUË BORNEE EN SIMULA

de considérer qu'il s'agit d'une simple règle syntaxique et qu'on manipule en fait des objets du nouveau type *nom_de_classe* de la même façon que des objets *integer*, *text* etc. Un tel type,défini par une classe, peut servir à déclarer non seulement des variables mais aussi, bien entendu, des paramètres formels ou des résultats de procédure :

```
procedure p (x,y,z) ; integer x ; ref (c) y,z ;

    begin... corps de p... end p ;

ref (c) procedure q(...) ; ... ;
```

Un objet d'un tel type est à tout instant du déroulement d'un programme dans l'un des deux états suivants :

- vide. C'est l'état initial. On considère que sa valeur est alors celle de la constante spéciale (générique) *none*.

- créé. L'objet possède alors chacun des attributs (variables, procédures) définis dans la déclaration de classe.

Un objet o_1 de type *ref (nom_de_classe)* passe dans l'état "créé" par l'exécution d'une instruction

```
o1 :- new nom_de_classe
```

ou

```
o1 :- new nom_de_classe (liste_de_paramètres_réels)
```

selon que la déclaration de classe contenait ou non des paramètres. Par exemple :

```
ref (pilentier) X ;....;

X :- new pilentier (5000) ; ....
```

Comme nous l'avons annoncé, les types "classes" sont dynamiques en SIMULA, c'est-à-dire que l'évaluation d'une expression commençant

par *new* entraîne la création d'un nouvel exemplaire du type, et l'allocation de mémoire correspondante pour les attributs définis dans la déclaration de la classe ; ainsi, dans notre exemple, l'entier *sommet* et le tableau *p*, ici de 5000 éléments.

L'évaluation d'une "expression" *new*.... entraîne aussi l'exécution des actions d'intitialisation sur l'exemplaire nouvellement créé de la classe. Ici l'attribut *sommet* de *X* sera mis à zéro. On notera que les variables locales à une classe sont rémanentes.

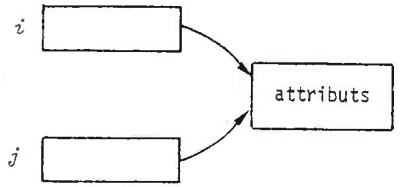
Opérations sur des objets de types définis par des classes

On aura noté le *:-* utilisé pour l'affectation à o_1 et *X* au lieu du *:=* traditionnel d'ALGOL qui est utilisé pour des objets de types classiques (ex. *i := 3*) mais serait illégal ici. De même, on emploiera comme opérateur de test de l'égalité == pour les objets de type *ref (nom_de_classe)* et = pour les autres. Ces différences de traitement sont dues à l'influence déjà mentionnée de la représentation physique des *ref* par des pointeurs : = et *:=* opèrent sur des valeurs, == et *:-* sur des adresses. On notera en particulier que la sémantique de ces opérations n'est pas la même, comme le montre la comparaison des exemples suivants :

```
integer i, j ;                          ref (c) i, j ;
.....;                                  .....;
comment ici prop (i) est vraie ;        comment ici prop (i) est vraie ;
j := i ;                                j :- i ;
mod (j) ;                               mod (j) ;
comment ici prop (i) est vraie ;        comment ici prop (i) peut être
                                                 fausse ;
```

où *mod (j)* est une opération ne pouvant modifier que son argument. Dans le premier cas toute propriété *prop (i)* portant sur *i* et sur des variables autres que *j* est invariante pour les deux instructions indiquées ; dans le

second, elle peut être rendue fausse si *mod (j)* modifie un attribut de *j*.



Cette différence de sémantique oblige à une certaine prudence lorsqu'on s'efforce de considérer en principe une déclaration de classe comme une définition d'un nouveau type(1).

Attributs et actions d'initialisation

Une fois créé par un *new*..., un objet de type *ref(...)* possède tous les attributs apparaissant dans la définition de classe ; au moment de l'évaluation du *new*..., les actions d'initialisation sont exécutées. Ainsi, dans

```
ref (pilentier) pil1 ;

pil1 :- new pilentier (100)
```

l'allocation de mémoire nécessaire au nouvel exemplaire *pil1* (en particulier pour le tableau *p*) est effectuée, et l'attribut *sommet* est mis à zéro.

Par la suite, on peut accéder à tout attribut, variable ou procédure, de l'objet grâce à une notation pointée

```
objet.attribut
```

semblable à celle de PL/1 ou PASCAL. A titre d'exemple, l'extrait de programme suivant créée et utilise une pile d'entiers :

(1) Pour le type *text*, mais pour celui-là seulement, les deux sémantiques coexistent : := est une recopie de caractères, :- une recopie de pointeur.

```
ref (pilentier) X ;
integer somme, i, j
integer array a[1 : 100] ;
.... initialisation de a ... ;
X :- new pilentier (1000) ; i := 1 ;
while i < 50 and a(i) > 10000 do
        X.empiler (a(i)) ;
..... ;
somme := 0 ;
while not X.pilevide do
      somme := somme + X.dépiler ;
.....
```

On notera qu'à l'intérieur d'une déclaration de classe, on désigne les attributs de l'"exemplaire courant" par leur simple nom : ainsi *sommet* dans la déclaration de *pilentier*. Par contre, on désignera un autre exemplaire de la même classe ou d'une autre par la notation pointée. Nous en aurons des exemples dans le paragraphe suivant, consacré à un exemple moins simple.

III.3 - Un exemple : les nombres complexes

Considérons le type abstrait "nombre complexe". Il est défini par la spécification de l'exemple 3 (page suivante) ; on a nommé les opérations par leur symbole mathématique, entouré d'un cercle ($\oplus$, $\ominus$ etc.) pour les distinguer des opérations sur les réels utilisées plus bas ; les fonctions d'accès x, y, ρ, θ sont la partie réelle, la partie imaginaire, le module et l'argument.

Si l'on cherche à réaliser un "module" représentant ce type, deux représentations s'offrent naturellement : la représentation cartésienne, où l'on conserve les parties réelles et imaginaires; la représentation polaire, où l'on conserve le module et l'argument. La première est appropriée aux opérations x, y, $\oplus$, $\ominus$, $\bar{}$, *cartésien* et rend les autres malaisées ;

fonctions

création

cartésien : REEL x REEL → COMPLEXE
(création d'un complexe à partir de ses parties réelle et imaginaire)

polaire : REEL x REEL → COMPLEXE
(à partir de son module et de son argument)

accès

x, y, ρ, θ : COMPLEXE → REEL x REEL

modification

$\oplus$, $\ominus$, $\otimes$, $\oslash$: COMPLEXE x COMPLEXE → COMPLEXE

$\bar{}$: COMPLEXE → COMPLEXE (conjugué)

$=$: COMPLEXE COMPLEXE BOOLEEN

assertions

pour tout a, b, r, t : REEL, c, c' : COMPLEXE *alors*

cartésien (x(c), y(c)) = c et *polaire* (ρ(c), θ(c)) = c et

x (*cartésien* (a,b)) = a et y (*cartésien* (a,b)) = b et

ρ (*polaire* (r,t)) = r et θ(*polaire* (r,t)) = t et

$\rho(c) = \sqrt{x(c)^2 + y(c)^2}$ et $\theta(c) =$ si $x(c) = 0$ alors $\pi/2$ sinon arc tg $(y(c)/x(c))$ et

$x(c) = \rho(c) \cos(\theta(c))$ et $y(c) = \rho(c) \sin(\theta(c))$ et

$x(c \oplus c') = x(c) + x(c')$ et $y(c \oplus c') = y(c) + y(c')$ et

$x(c \ominus c') = x(c) - x(c')$ et $y(c \ominus c') = y(c) - y(c')$ et

$\rho(c \otimes c') = \rho(c) \times \rho(c')$ et $\theta(c \otimes c') = \theta(c) + \theta(c')$ et

$\rho(c \oslash c') = \rho(c) / \rho(c')$ et $\theta(c \oslash c') = \theta(c) - \theta(c')$ et

$x(\bar{c}) = x(c)$ et $y(\bar{c}) = - y(c)$ et

$\rho(\bar{c}) = \rho(c)$ et $\theta(\bar{c}) = - \theta(c)$ et

$c = c' \Leftrightarrow (x(c) = x(c')$ et $y(c) = y(c'))$

fin

EXEMPLE 3 : LE TYPE ABSTRAIT *COMPLEXE* - SPECIFICATION FONCTIONNELLE

la seconde est appropriée aux opérations ρ, θ, x, $\oslash$, $\bar{}$, *polaire*.

Face à ce dilemme, le mieux est de ne pas choisir a priori, mais de passer d'une représentation à l'autre au grê des opérations demandées. Nous utiliserons deux variables locales à la classe, soit *cart_rep* et *pol_rep*, indiquant respectivement si les représentations cartésienne et polaire sont disponibles, et des variables *x_rep*, *y_rep*, *ro_rep*, *theta_rep* susceptibles de représenter partie réelle, partie imaginaire, module et argument. La classe doit être écrite de telle façon que la création d'un nouvel objet établisse et que chaque chaque appel de procédure maintienne l'invariant de représentation suivant :

(cart_rep ou pol_rep) (l'une des des deux au moins est bonne)

et *(cart_rep ⇒ x_rep et y_rep significatifs)*

et *(pol_rep ⇒ ro_rep et theta_rep significatifs)*.

La condition sine qua non de validité de ce module est qu'on l'emploie seulement à travers les procédures *cartésien*, *polaire*, *x*, *y*, *ro*, *theta*, *conjugué*, et jamais en modifiant les variables de représentation *x_rep*, *y_rep*, *ro_rep*, *theta_rep*, *cart_rep* et *pol_rep* ni en utilisant d'autres procédures internes (*cart_calcul* et *pol_calcul* ci-après).

Si cette condition est respectée, le module décrit à l'exemple 4, donné aux pages suivantes, répondra à la question.

```
class complexe ;
    comment représentation ;
        boolean cart_rep, pol_rep ;
        real x_rep, y_rep, ro_rep, theta_rep ;
        procedure cart_calcul ;
                comment procédure à usage exclusivement interne calculant s'il y a lieu la représentation cartésienne, l'autre étant supposée connue ;
                if not cart_rep then
                   begin
                   x_rep  := ro_rep * cos(theta_rep) ;
                   y_rep  := ro_rep * sin(theta_rep) ;
                   cart_rep := true
                   end cart_rep ;
        procedure pol_calcul ;
                comment procédure à usage exclusivement interne calculant s'il y a lieu la représentation polaire, l'autre étant supposée connue ;
                if not pol_rep then
                   begin
                   ro_rep := sqrt (x_rep ** 2 + y_rep ** 2) ;
                   theta_rep := if x_rep = 0 then pi/2
                                else arctg (y_rep/x_rep) ;
                   pol_rep := true
                   end pol_calcul ;
    comment procédures (exportées) de création ;
        procedure cartésien (a,b) ; real a,b ;
                begin
                x_rep := a ; y_rep := b ;
                cart_rep := true ; pol_rep := false
                end cartésien ;
        procedure polaire (r,t) ; real r,t ;
                begin
                ro_rep := r ; theta_rep := t ;
                pol_rep := true ; cart_rep := false
                end polaire ;
```

EXEMPLE 4 (DEBUT) : MODULE "COMPLEXE" EN SIMULA


```
    comment procédures (exportées) d'accès ;
        real procedure x ;
                begin
                cart_calcul ; x :=  x_rep
                end x ;
        real procedure y ;
                begin
                cart_calcul ; y  := y_rep
                end y ;
        real procedure ro ;
                begin
                pol_calcul ; ro :=  ro_rep
                end ro ;
        real procedure theta ;
                begin
                pol_calcul ; theta  := theta_rep
                end theta :

        boolean procedure égal_complexe (c) ; ref (complexe) c ;
                égal_complexe := if  cart_rep  then
                                    (x_rep = c.x  and  y_rep = c.y)
                                 else (ro = c.ro  and  theta = c.theta) ;
```

EXEMPLE 4 (SUITE) : MODULE "COMPLEXE" EN SIMULA

```
comment procédures (exportées) de modification ;

    ref (complexe) procedure plus (c) ; ref (complexé) c ;
            begin ref (complexe) p ;
            p :- new complexe ; p.polaire (x+c.x, y+c.y) ;
            plus :- p
            end plus ;

    ref (complexe) procedure moins (c) ; ref (complexe) c ;
            begin ref (complexe) m ;
            m :- new complexe ; m.cartésien (x-c.x, y-c.y) ;
            moins :- m
            end moins ;

    ref (complexe) procedure mult (c) ; ref (complexe) c ;
            begin ref (complexe) m ;
            m :- new complexe ; m.polaire (ro*c.ro, theta+c.theta) ;
            mult :- m
            end mult ;

    ref (complexe) procedure div (c) ; ref (complexe) c ;
            begin ref (complexe) d ;
            d :- new complexe ; d.polaire (ro/c.ro, theta-c.theta) ;
            div :- d
            end div ;

    ref (complexe) procedure conj ;
            begin ref (complexe) cj ;
            cj :- new complexe ;
            if cart_rep then cj.cartésien (x_rep, y_rep)
            else cj.polaire (ro_rep, theta_rep) ;
            conj :- cj
            end conj ;

comment actions d'initialisation ;

    x_rep := y_rep := ro_rep :- theta_rep := 0 ;
    cart_rep := pol_rep := true

end classe complexe
```

EXEMPLE 4 (FIN) : MODULE "COMPLEXE" EN SIMULA

Un certain nombre de remarques s'imposent sur cet exemple.

Remarque 4.1 On aurait pu éviter la nécessité pour l'utilisateur d'une double initialisation par *c :- new complexe*, puis *c.cartésien (...)* ou *c.polaire (...)* en donnant trois paramètres à la classe (représentation, réel 1, réel 2).

Remarque 4.2 On a pris le parti de calculer systématiquement la représentation cartésienne (resp. polaire) dès que x ou y (resp. ρ ou θ) est demandé.

Remarque 4.3 Une des contraintes des types représentés par des classes en SIMULA apparaît bien sur cet exemple : dans une déclaration de classe, on "parle" toujours d'un certain exemplaire courant de la classe. C'est pourquoi les procédures *plus, moins, mult, div* n'ont qu'un seul paramètre (et *conjugué* aucun) : l'autre est implicite ; c'est à ses attributs que se réfère une écriture comme x dans $x + c.x$ (procédure *plus*), $c.x$ désignant le même attribut propre à l'argument c de la procédure. Cette dissymétrie artificielle est assez gênante. On voit bien, en comparant par exemple avec ADA où le "package" *complexe* comprendrait une procédure *plus* à deux paramètres, qu'il y a eu dans la conception de la "classe" en SIMULA fusion de deux notions bien distinctes : une structure syntaxique pour regrouper ("encapsuler") toutes les propriétés d'un type abstrait ; une structure syntaxique décrivant la composition (attributs) et la vie (initialisation) d'un exemplaire de ce type.

Remarque 4.4 On peut éviter l'allocation de mémoire entraînée par l'exécution de chacune des procédures de modification *plus, moins, mult, div, conjugué* en les remplaçant par des procédures d'affectation du résultat de l'opération correspondante, par exemple :

```
procedure affplus (c,c') ; ref (complexe) c,c' ;
        comment affecte à c, supposé créé, la somme
                de c' et de l'"exemplaire courant" ;
        if c == none then erreur
        else c.cartésien (x + c'.x, y + c'.y)
```

et de façon correspondante pour les autres.

Remarque 4.5 Nous avons vu que la règle d'emploi de ce module était de passer par les procédures exportées, et elles seules. Il n'existe en SIMULA aucun moyen de faire respecter cette règle. On notera cependant que l'adjonction au langage de clauses restreignant la visibilité ne poserait aucune difficulté ; voir une proposition en ce sens dans [16].

Remarque 4.6 Dans les procédures calculant de nouveaux objets de type *complexe* (procédures *plus, moins, mult, div, conj*), l'emploi de variables locales (respectivement *p, m, m, d, cj*) est nécessaire pour éviter des récursivités non désirées. En effet, si l'on écrit dans le corps de la procédure *plus*

```
plus :- new complexe ;
plus.cartésien (x+c.x, y+c.y)
```

le *plus* de la première instruction, apparaissant à gauche du signe d'affectation, désigne bien le résultat de la procédure *plus*, mais celui de la seconde instruction représente un appel récursif (ici syntaxiquement illégal) de la même procédure. La règle (qui est une source bien connue de difficultés dans l'enseignement de la programmation avec les langages comme ALGOL 60) est qu'une apparition du nom d'une fonction dans le corps de cette fonction désigne un appel récursif sauf si elle se trouve à gauche d'un symbole d'affectation.

IV - COMPOSITION DESCENDANTE ET PREFIXATION DE CLASSES

IV.1 - Principe

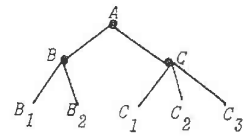
La composition descendante par affinages successifs [22] permet de développer un programme par niveaux d'abstraction en utilisant à chaque étape des éléments de niveau inférieur, non encore détaillés. Il est naturel de chercher à appliquer cette méthode non seulement aux programmes, mais aussi aux données, en particulier si la décomposition en modules est guidée par celles-ci.

Outre les modes d'abstraction de programmes (procédures) et de données (classes) partagés aujourd'hui avec d'autres langages, SIMULA offre un support linguistique original à la composition descendante dirigée par les données : la notion de préfixation de classe. Le principe est le suivant : étant donné une classe *A*, définissant un certain nombre d'attributs, variables et procédures, communs à tous les objets de type *ref (A)*, il est possible de définir de nouvelles classes *B, C,...* comme sous-classes de *A*, en préfixant leur déclaration par le nom de *A* :

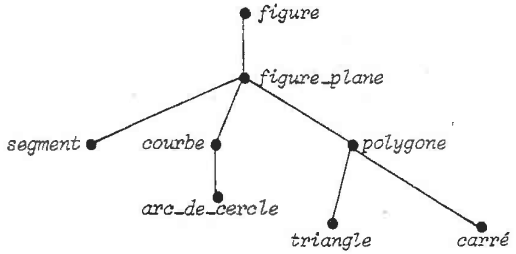
A class B ; begin ... attributs des objets B ... end ;

A class C ; begin ... attributs des objets C ... end ;

Les objets de types *B, C, ...* posséderont alors, outre les attributs intervenant dans la déclaration de *B, C, ...*, tous les attributs définis dans la déclaration de *A*. *B, C, ...* peuvent à leur tour préfixer de nouvelles déclarations de classes ; on obtient ainsi une structure hiérarchisée, arborescente, des déclarations de classes.

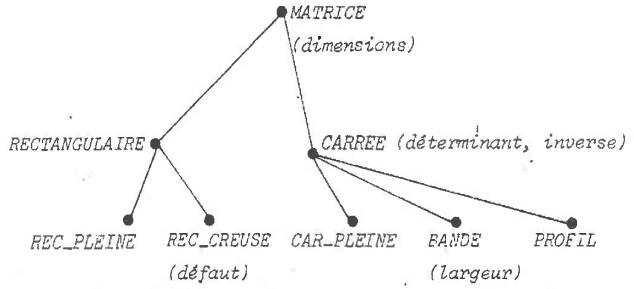


Le mécanisme de la préfixation permet de développer des structures de données progressivement, en spécialisant à chaque étape les objets décrits. Un exemple souvent choisi est celui des structures de données d'un système de manipulation d'images, pour lequel la hiérarchie des classes pourrait reproduire le schéma suivant :



Remarque : On notera que la préfixation d'une classe par une autre exprime la relation "est un exemple de", et non "utilise". Cette dernière relation est simplement représentée par des déclarations d'objets d'une autre classe. Ici, par exemple, les attributs de la classe *polygone* pourront comprendre des objets de type *ref (segment)*.

Un autre exemple est celui d'un logiciel numérique, offrant des opérations variables selon les caractéristiques des matrices manipulées : carrées ou non, pleines ou creuses, etc. Une hiérarchie possible pour décrire la structure du type *MATRICE*, et celle des classes SIMULA correspondantes, est la suivante :



On a indiqué au niveau de chaque noeud quelques-uns des attributs caractéristiques de ce noeud (et partagés par ses descendants).

Un autre exemple de la même démarche est, dans /6/, la spécification en SIMULA d'un système relationnel de gestion de bases de données, où les choix de représentation des relations sont cachés dans des sous-classes.

Remarque : On notera le caractère fondamental de la structure en arbre des préfixations de classes. Cette limitation explique pourquoi nous n'avons pas utilisé la préfixation pour définir la classe *COMPLEXE* en III.3 : une telle classe pourrait bien admettre des sous-classes *COMPLEXE_CARTESIEN* et *COMPLEXE_POLAIRE*, mais celles-ci seraient disjointes et l'on ne pourrait pas tirer parti du fait que les deux représentations sont disponibles pour certains nombres complexes. De la même façon, le mécanisme de la préfixation, transposé aux mathématiques, permettrait de définir la hiérarchie monoïde-groupe-espace vectoriel, et la théorie des espaces topologiques, mais non de les réunir pour introduire la notion d'espace vectoriel topologique. Une structure telle que celle qui est induite en ADA par la relation *use* permet au contraire de définir des graphes acycliques, et non pas seulement des arbres.

IV.2 - Les objets virtuels

En recensant les attributs d'une classe possédant des sous-classes, comme *figure* ou *MATRICE*, on trouve fréquemment, comme conséquence du principe même de la conception descendante, des objets qui sont véritablement caractéristiques de cette classe-"souche", mais dont la réalisation précise dépend des sous-classes. Ainsi, on peut imaginer que toute *figure* possède parmi ses attributs des procédures *translation* et *rotation*, qui seront détaillées différemment selon qu'il s'agit d'un arc de cercle, d'un carré, etc. De même, une *MATRICE* sera sujette à des procédures *valeur (i, j)* et *changer_valeur (r, i, j)*, dont la mise en oeuvre dépend de la représentation physique.

De telles procédures sont dites virtuelles ; elles sont regroupées en SIMULA dans un paragraphe spécial en tête de la déclaration de la classe-souche, où l'on indique leur nom et, s'il y a lieu, le type de leur résultat, mais sans mentionner les arguments [1] :

(1) Cette dernière clause, assez désagréable en pratique, semble liée à l'interdiction de faire référence dans une classe au nom de toute sous-classe de cette classe, pour ne pas rendre impossible la compilation en un seul passage.

```
class MATRICE ;
    virtual : real procedure valeur ;
              procedure changer_valeur ;
    begin integer m, n ;  comment les dimensions ;
    boolean procedure indices_corrects (i, j) ; integer i, j ;
          comment cette procédure n'est pas virtuelle ;
          indices_corrects := (i >= 1) and (i <= m) and
                              (j >= 1) and (j <= n)
    end MATRICE
```

Bien entendu, chaque procédure virtuelle devra être définie effectivement pour toute sous-classe susceptible de l'utiliser ; par exemple :

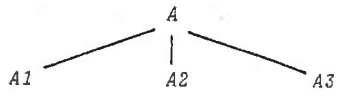
```
MATRICE class CARREE (ordre) ; integer ordre ;
    virtual : real procedure déterminant ;
              ref (CARREE) procedure inverse ;
    begin
    m := n := ordre
    end CARREE ;

CARREE class CAR_PLEINE ;
    begin
    real array représentation (1:ordre, 1:ordre) ;
    real procedure valeur (i, j) ; integer i, j;
              valeur := représentation (i, j) ;
    procedure changer_valeur (r, i, j) ; real r ; integer i, j ;
              représentation (i, j) := r ;
    real procedure déterminant ; ..... ;
    ref (CAR_PLEINE) procedure inverse ; ..... ;
    .....
    end CAR_PLEINE
```

On notera que seuls les attributs "procédures" peuvent être virtuels, non les attributs "données".

IV.3 - Discrimination entre sous-classes

Soit la hiérarchie suivante :

```
class A ; ..... ;
A class A1 ; ..... ;
A class A2 ; ..... ;
A class A3 ; ..... ;
```



Un objet *X* de type *ref (A)* est susceptible de recevoir des valeurs de type *ref (A1)*, *ref (A2)*, *ref (A3)* :

X :- new A1 , etc.

Pour savoir ce qu'il en est à un moment donné de l'exécution, on peut utiliser l'instruction *inspect* :

```
inspect X
      when A1 do   action_1
      when A2 do   action_2
      when A3 do   action_3
```

On notera que cette instruction est à la fois une discrimination entre sous-types (*case* de PASCAL) et un mécanisme d'importation de noms externes (*with* en PASCAL, cf. *use* en ADA) car *action_1*, *action_2*, *action_3* peuvent utiliser directement les attributs de *X* relatifs à *A1*, *A2*, *A3* respectivement, sans passer par la notation pointée habituelle.

Il faut mentionner une autre méthode de discrimination utilisant l'opérateur *is* :

```
if X is A1 then .....
else if .....
```

La méthode utilisant *inspect* est de toute évidence préférable : outre qu'elle est plus rigoureuse, elle permet un contrôle statique de la validité des références. Notons cependant qu'un contrôle est toujours effectué (en partie à l'exécution dans le pire des cas) : le problème de la "référence folle", présent dans de nombreux langages, n'existe pas en SIMULA.

IV.4 - Constitution de modules d'application

Un aspect intéressant du mécanisme de préfixation est qu'il permet de constituer de façon commode des modules d'application ("packages"). On peut en effet préfixer par un nom de classe non seulement de nouvelles classes, mais aussi des programmes ou plus généralement des blocs, qui héritent alors des attributs de la classe préfixante. Il suffit donc d'"encapsuler" convenablement dans celle-ci les données et les opérations, éventuellement fort complexes, relatives à un domaine d'application spécialisé, et d'en fournir le mode d'emploi à l'utilisateur qui, dans le cas le plus simple, n'aura à connaître de SIMULA que les déclarations d'objets, l'affectation et l'appel de procédure. Des applications à l'enseignement de l'informatique se présentent naturellement : on peut créer un "univers" permettant à un projet d'étudiant d'aborder d'emblée des problèmes fins d'un système. SIMULA peut de ce point de vue être considéré comme un générateur de langages spécialisés - langages à qui, selon une interview récente [23], l'avenir appartiendrait.

V - GENERICITE

V.1 - Principe et exemple simple

Une étape supplémentaire dans l'effort d'abstraction caractéristique de l'évolution actuelle de la programmation, et qui a donné naissance aux concepts introduits dans les deux paragraphes précédents (abstraction par rapport aux modalités de représentation des données : types abstraits, abstraction par rapport à la suite du développement du système : conception descendante), consiste à suivre le chemin tracé par les mathématiciens, qui cherchent à appliquer les mêmes résultats à des objets concrètement différents, mais caractérisés par des structures identiques. Deux catégories importantes d'objets susceptibles d'être ainsi traités sont :

- les opérations : il est loisible d'utiliser le même symbole + pour l'addition d'entiers, de réels, de matrices, ou le ou booléen, les propriétés externes étant les mêmes (associativité, élément neutre, etc.) ;

- les ensembles : les mêmes raisonnements, fondés sur la théorie des groupes, peuvent être appliqués à des ensembles de nombres, de transformations etc. munis de cette structure.

En programmation, un objet tel que l'opérateur + ou le "type" GROUPE, caractérisé par des propriétés fixes de structure mais susceptible de plusieurs interprétations concrètes, est dit générique. Certains langages de programmation récents permettent de définir de tels objets. Les deux grandes catégories d'objets génériques correspondent aux deux cas définis ci-dessus ; selon qu'on considère l'ordre des traitements ou celui des données, on distingue, selon la terminologie de [11] :

- La généricité "incrémentale", permettant d'interpréter différemment des opérateurs selon le type des données auxquels ils sont appliqués. C'est ce qui se passe dans les langages de programmation courants avec des opérateurs comme "+" et "*" qui représentent des calculs entiers ou flottants selon le type de leurs arguments ; de tels opérateurs sont dits "surchargés". ALGOL 68 permet de définir de nouvelles surcharges d'opérateurs.

- La généricité "structurale", intéressante surtout si on l'étend à la notion de "type abstrait générique", permettant de définir des types de données paramétrés par d'autres types. En remarquant ainsi dans l'exemple 1 (pile d'entiers) que le type des éléments empilés, *ENTIER*, ne joue aucun rôle dans la compréhension de la structure de pile, on est naturellement amené à définir le type générique *PILE (t)*, dont les opérations associées *(empiler, dépiler, pilevide)* peuvent être étudiées indépendamment du type *t* des objets empilés ; des "cas" particuliers du type *PILE (t)* sont par exemple *PILE (ENTIER)*, *PILE (TEXTE)*, voire *PILE (PILE (ENTIER))* etc.

On notera que la seconde espèce de généricité inclut la première : un type de données étant défini, comme nous l'avons vu au paragraphe III, par les opérations associées, ces opérations devront être génériques si le type l'est. *Empiler, dépiler,* etc., seront ainsi génériques dans notre exemple. La généricité incrémentale étant, prise isolément, d'un intérêt limité, nous étudierons ici la généricité de la seconde espèce.

V.2 - Types génériques en SIMULA

Une façon simple de représenter en SIMULA un type générique *typgen*, paramétré par un type *t*, est de définir d'abord une classe correspondant au paramètre *t* ;

class t ; ..... ;

et, dans la définition de la classe associée à *typgen*, de représenter tout objet du type *t* par un objet SIMULA de type *ref (t)*.

Les types correspondant à des arguments réels d'un cas particulier de *typgen* seront alors définis par des sous-classes de *t* :

t class t_1 ; ..... ;
t class t_2 ; ..... ; etc.

Cette méthode est appliquée à la définition d'une pile générique avec cette fois une représentation chaînée (exemple 5), et donc plus de restriction de taille.

```
class pile ; comment générique en t ;
    begin
    comment partie représentation ;
        ref (t) tête ;
        ref (pile) corps ;
    comment opérateurs de la spécification ;
        ref (t) procedure dépiler ;
            begin
            dépiler :- tête ;
            tête :- corps.tête ; corps :- corps.corps
            end dépiler ;
        procedure empiler (x) ; ref (t) x ;
            begin ref (pile) c ;
            c :- new pile ;
            c.tête :- tête ; c.corps :- corps ;
            tête :- x ; corps :- c
            end empiler ;
        comment une pile vide a été créée par un new mais a un
            corps vide (none) ;
        boolean procedure pilevide ;
            pilevide := (corps == null)
end classe pile
```

EXEMPLE 5 : PILE GENERIQUE CHAINEE

On peut faire sur cet exemple les commentaires suivants :

Remarque 5.1 On ne peut pas avec cette méthode définir des piles d'entiers, de textes, de réels, de tableaux etc. ; il faut passer par des types intermédiaires préfixés par *t* :

```
t class tentier ;
        begin integer x end tentier ;

t class tréel ;
        begin real x end tréel ;

t class ttabréel ;
        begin real array x[1 : 100] end ttabréel ;
```

Remarque 5.2 Cette méthode ne permet pas de garantir l'intégrité d'une pile : rien n'empêche l'utilisateur de constituer une pile contenant à la fois des *tentier*, des *tréel* etc.

Remarque 5.3 Il n'est pas très satisfaisant d'avoir à déclarer une nouvelle classe, *t*, sans lien syntaxique avec la classe *pile*, pour rendre cette dernière générique. Le lecteur aura peut-être fait le rapprochement avec la remarque de la fin du paragraphe IV.2 : seuls les attributs "procédures" peuvent être virtuels.

V.3 - L'arbre binaire

L'exemple précédent ne permet pas de saisir tous les problèmes de la généricité, car il n'existe dans l'absolu aucune contrainte particulière sur le type des objets "empilables"(1). Il est au contraire assez fréquent qu'un type générique *typgen (t)* impose certaines conditions sur son paramètre formel, le type *t*.

(1) Cette affirmation n'est pas tout à fait exacte : le lecteur aura peut-être remarqué dans l'exemple 5 qu'on utilisait dans *empiler* l'affectation de pointeurs :- sur le type *t*, ce qui peut entraîner le partage de structures de données et des effets de bord non désirés. En fait, il aurait fallu même ici inclure une opération d'affectation dans la spécification de *t*.

Si l'on cherche par exemple à définir le type *arbin (t)*, pour arbre binaire de recherche contenant des objets de type *t*, il faut pouvoir disposer sur *t* d'une relation d'ordre (permettant les comparaisons), d'une procédure permettant l'affectation (pour les insertions) et du test d'égalité.

Le mécanisme des procédures virtuelles permet d'exprimer élégamment ces contraintes :

```
class  scalaire ; comment un objet "insérable" dans un arbre binaire ;
       begin
       virtual : boolean procedure inférieur ;
                 procedure affectation ; boolean procedure égal ;
       comment on peut à ce niveau compléter par les procédures suivantes : ;
       boolean procedure infouégal (s) ; ref (scalaire) s ;
               infouégal := inférieur (s) or égal (s) ;
       boolean procedure différent (s) ; ref (scalaire) s ;
               différent := not égal (s) ;
       boolean procedure supérieur (s) ; ref (scalaire) s ;
               supérieur := not infouégal (s) ;
       boolean procedure supouégal (s) ; ref (scalaire) s ;
               supouégal := not inférieur (s)
       end scalaire.
```

Exemple 6 : scalaire (à mettre dans un arbre binaire de recherche)

Des exemples de types "scalaires" sont alors :

```
scalaire class scalent ;

        begin scalaire entier ;

        integer i ;

        boolean procedure inférieur (s) ; ref (scalent) s ;
                inférieur := (i < s.i) ;

        procedure affectation (s) ; ref (scalent) s ;
                i := s.i ;

        boolean procedure égal (s) ; ref (scalent) s ;
                égal := (i = s.i)

        end scalent ;

scalaire class scal_chaîne_car ; ..... ;

scalaire class entrée_table_symbole ; ..... ;
```

On notera que le mécanisme permettant de contraindre le type *t* (ici *scalaire*) est purement syntaxique : des objets d'une classe préfixée par *scalaire* et à laquelle manquerait une des procédures *inférieur*, *affectation* ou *égal* pourraient être insérés dans un arbre binaire (erreur à l'exécution) ; mais aucun moyen n'existe pour spécifier dans la déclaration de *scalaire* que la réalisation de ces procédures dans des sous-classes devra répondre à une sémantique compatible avec la notion habituelle de relation d'ordre, d'affectation et d'égalité.

Nous donnons ci-dessous une réalisation du type générique *arbin*, utilisant la classe *scalaire* précédente (exemple 7).

```
class arbin ; comment arbre binaire de recherche ;
    ref (scalaire) racine ; ref (arbin) gauche, droite ;

    procedure insérer (s) ; ref (scalaire) s ;
        if racine == null then racine.affectation (s)
        else
          begin ref (arbin) père, fils ;
            fils :- this arbin ;
            while fils =/= null do
                    begin
                    père :- fils
                    fils :- if s.inférieur (père.racine) then
                            père.gauche
                            else père.droite
                    end ;
            fils :- new arbin ; fils.affectation (s) ;
            if s.inférieur (père.racine) then père.gauche :- fils
            else père.droite :- fils
            end insérer ;
    boolean procedure recherche (s) ; ref (scalaire) s ;
        if racine == null then recherche := false
        else
          begin ref (arbin) noeud ; boolean b ;
            noeud :- this arbin ; b := true ;
            while b do
                    begin

                    if noeud == null then b := false
                    else if s.égal (noeud.racine) then b := false
                    else if s.inférieur (noeud.racine) then
                                        noeud :- noeud.gauche
                    else noeud :- noeud.droite
                    end boucle ;
            recherche := (noeud =/= null)
            end cas non vide ;

..... suite de la classe .....
```

EXEMPLE 7 : ARBRE BINAIRE GENERIQUE

Cet exemple appelle quelques remarques :

Remarque 7.1 =/= représente l'inégalité pour les objets de type *ref(...)*.

Remarque 7.2 On utilise dans les procédures *recherche* et *insérer* la construction *this c* qui, dans une déclaration d'une classe *c*, désigne l'objet de type *ref (c)* qui est précisément l'exemplaire courant de *c*, celui qu'on "est en train de décrire" (cf. la remarque 4.3).

Remarque 7.3 La programmation pour le moins maladroite de la boucle dans *recherche* vient de ce que la norme de SIMULA ne garantit pas (comme le fait celle d'ALGOL W par exemple) que le *and* est un "et conditionnel", c'est-à-dire qu'on peut écrire *a and b* si *a* est faux et *b* non défini.

VI - PROGRAMMATION QUASI-PARALLELE : LES COPROGRAMMES

VI.1 - Généralités

En décrivant les deux composantes d'une classe, introduites au paragraphe II :

- attributs (variables et procédures)
- actions (d'initialisation)

nous avons jusqu'ici mis l'accent sur la première.

L'éclairage inverse permet de considérer une classe non pas seulement comme une "capsule" de données assurant la représentation d'un type abstrait, mais comme un modèle de processus exécutant répétitivement des actions, qui ne sont plus simplement d'"initialisation".

Cette possiblité ouvre la voie à une méthode de programmation intéressante, qui considère le système physique modélisé comme un ensemble de sous-systèmes coopérants dont l'activité se poursuit en parallèle, et qui doivent de temps à autre se synchroniser et échanger de l'information.

Pour pouvoir modéliser convenablement une telle situation, il faut disposer d'opérations représentant la synchronisation et l'échange. Bien entendu, dans un contexte d'exécution classique, le modèle lui-même est strictement séquentiel, et non pas parallèle.

La "synchronisation" est assez bien représentée en SIMULA par la primitive *resume*, dont l'effet peut être défini (au moins pour les utilisations simples) de la façon suivante : si *X* et *Y* sont des variables de type *ref (...)* désignant des processus, telles que les processus associées aient été créés (par *X :- new...*, *Y :-new...*), alors l'exécution par le processus associé à *X* de *resume Y* a pour effet de suspendre l'exécution de ce processus et de reprendre celle du processus désigné par *Y*,au dernier endroit où elle avait précédemment cessé. L'exécution du processus désigné par *X* pourra se poursuivre ultérieurement, à l'instruction devant suivre le *resume Y*, si un processus quelconque, ou le programme principal, exécute une instruction *resume X*.

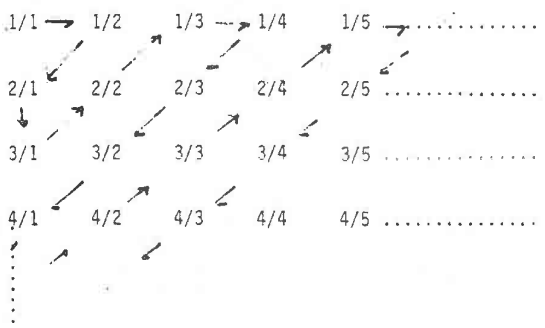
L'aspect "communication" est moins bien traité ; en fait SIMULA n'offre pas d'instruction particulière permettant l'échange de messages entre deux processus représentés par des exemplaires de classes ; on notera que l'instruction *resume X* ne permet pas de transmettre de paramètres à *X*. La communication d'informations entre unités de programme quasi-parallèles se fait généralement par le moyen de variables globales, ce qui n'est pas une méthode de programmation très sûre, ni très élégante.

Au-delà des détails de mise en oeuvre qui en obscurcissent souvent la présentation, la programmation quasi-parallèle à la SIMULA permet, par le simple jeu de la notion de classe et de l'instruction *resumé*, de représenter sans les trahir des situations réelles où intervient un parallélisme véritable ; c'est sur cette base que sont en particulier définies les primitives de simulation du langage. La programmation quasi-parallèle est cependant également intéressante lorsque la réalité modélisée n'est pas fondamentalement parallèle, mais simplement constituée d'un certain nombre de sous-systèmes qui se déterminent les uns par rapport aux autres sur un pied d'égalité, plutôt que selon une dépendance hiérarchique correspondant à la décomposition classique en sous-programmes. Le programme SIMULA sera alors formé de classes de type "processus" se comportant comme des coprogrammes.

Nous étudierons l'utilisation des coprogrammes à l'aide de deux exemples, l'un simple, l'autre plus compliqué. Le but de ces exemples n'est pas de prôner systématiquement cette approche, mais d'illustrer un style peu classique de programmation.

VI.2 - Coprogrammes : Un exemple simple

Nous appliquerons d'abord cette démarche à un exemple simple. Il s'agit d'un problème tiré de Jackson [11]: imprimer les rationnels selon un ordre diagonal inspiré de la méthode de Cantor :



On impose de n'imprimer que les 1000 premiers rationnels de cette énumération (ou une autre condition d'arrêt *c(i,j)* dépendant du rationnel *(i/j)*).

En ignorant d'abord cette condition d'arrêt, on peut écrire le programme d'impression comme une boucle infinie

```
integer i, j ;
i := 1 ; j := 1 ;
while true do
    begin
    while  i >= 1  do
        begin
        imprimer_fraction (i, j) ;
        i := i-1 ; j := j+1
        end descente ;
    i := 1 ;
    while  j >= 1  do
        begin
        imprimer_fraction (j, i) ;
        j := j-1 ; i := i+1
        end montée ;
    j := 1
    end
```

La difficulté, si l'on ajoute une condition d'arrêt, est qu'elle peut intervenir après le calcul d'un rationnel quelconque, c'est-à-dire dans l'une des boucles internes. Une solution classique amènerait à mélanger le calcul des fractions successives avec le contrôle de la condition d'arrêt (et il faut imaginer, au-delà de cet exemple précis, un programme comprenant beaucoup de tels calculs dans des boucles profondément imbriquées).

Dans une solution quasi-parallèle, il suffit d'avoir deux coprogrammes : l'un, *cont*, contrôlera l'arrêt ; l'autre, *calc*, calculera et imprimera les rationnels successifs.

Le programme prend alors la forme suivante :

```
begin
class calcul ;  begin ..... cf. ci-après ..... end calcul  ;
class contrôle ; begin ..... cf. ci-après ..... end contrôle ;

ref (calcul) calc ; ref (contrôle) cont ;
calc :- new calcul ; cont :- new contrôle ;
resume calc
end programme principal
```

A l'occasion de *calcul* et *contrôle*, nous introduisons ci-dessous la primitive *detach* qui joue un rôle important, à côté de *resume*, dans l'expression des coprogrammes ; elle permet de rendre le contrôle à l'élément de programme qui a créé l'exemplaire de classe dans lequel elle intervient. Elle sera indispensable pour qu'un objet de type *ref (calcul)* ou *ref (contrôle)* (ici *calc* ou *cont*), après sa création par un *new* ..., rende temporairement le contrôle au programme principal ci-dessus.

Le programme de contrôle est simplement :

```
class contrôle ;
    begin  integer n ;
    detach ;
    for n := 1 step 1 until 1000 do
        resume calc ;
    detach
    end contrôle ;
```

Le programme de calcul devient :

```
class calcul ;
    begin  integer i, j ;
    detach ;
    ..... le programme initial, où l'on a fait précéder
          chaque appel à "imprimer_fraction" de
          "resume cont ;" .....
    end calcul ;
```

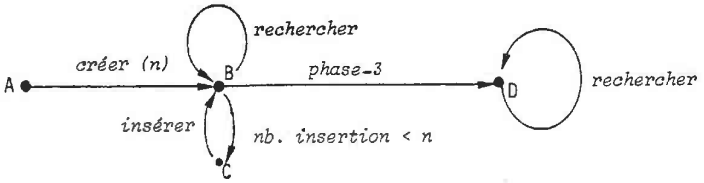
En séparant ainsi la partie "calcul" et la partie "contrôle", nous avons obtenu une décomposition en deux coprogrammes faiblement couplés. C'est l'un des avantages de ce style de programmation que de produire des unités de programme pouvant dans une large mesure être lues et comprises indépendamment les unes des autres. On notera en particulier que *n* est une variable locale à *contrôle*, *i* et *j* à *calcul*. Cependant, une condition d'arrêt plus complexe aurait obligé à introduire des variables globales, qui seront également nécessaires dans l'exemple suivant où les différents coprogrammes se transmettent plus d'informations.

VI.3 - Types abstraits avec scénario

Nous introduirons, pour illustrer la démarche quasi-parallèle, un exemple de ce qu'on peut appeler un "type abstrait avec scénario". Il s'agit d'un module défini, comme à la section II, par son aptitude à répondre d'une façon bien spécifiée à certaines opérations, mais avec la contrainte supplémentaire que seules certaines successions de ces opérations doivent pouvoir être acceptées. En d'autres termes, on impose un "scénario" réglant les vies possibles du module.

L'exemple choisi est celui d'un module de gestion de table, dont la spécification inclut comme opérations de base *créer (n)* (résultat : une table pouvant abriter jusqu'à *n* éléments), *rechercher (c, tab)* (résultat : un élément de clé *c* dans *tab*, s'il y en a), *insérer (x, tab)*, (la nouvelle table résultant de l'ajout de *x* à *tab*). Les propriétés formelles du type abstrait correspondant (de la forme *rechercher (clé(x), insérer (x, tab)) = x* etc.) sont faciles à écrire.

Nous ajoutons ici les restrictions suivantes. La vie d'une table se déroule en trois phases, symbolisées par le diagramme de transition ci-dessous. Au cours de la première phase, l'utilisateur doit exécuter l'opération *créer (n)*. Au cours de la seconde, il peut effectuer librement des recherches et des insertions, sans toutefois exécuter plus de n insertions. Au cours de la troisième, il doit effectuer une fois une opération sans argument notée *phase_3* (un signal de changement de phase), et ne peut plus ensuite demander que des recherches. (On peut interpréter cet exemple comme celui d'un compilateur en plusieurs "passages" qui, après un certain temps, recherche mais n'insère plus dans la table des symboles).



On désire que le module de gestion de la table soit programmé de façon à n'accepter que les suites de transitions correspondant au diagramme ci-dessus, où les transitions non figurées sont des cas d'erreur.

VI.4 - Rappel sur CSP

Il sera particulièrement commode d'illustrer d'abord la programmation quasi-parallèle de cet exemple dans la notation CSP de Hoare [8] [12], qui nous fournira un modèle pour la programmation en SIMULA.

L'intérêt de CSP ("Communicating Sequential Processes") pour traiter ce problème est double. Tout d'abord, il s'agit d'un formalisme conçu pour l'expression de programmes parallèles, mais prévu pour que le programme décrivant le fonctionnement d'un système avec parallélisme puisse être réinterprété, dans un contexte séquentiel, comme le programme de simulation quasi-parallèle du même système.

Une autre caractéristique de CSP qui rend cette notation intéressante comme modèle simple de systèmes parallèles ou quasi-parallèles est que ses opérations primitives, l'entrée et la sortie, réalisent à la fois la synchronisation et la communication. L'écriture est la suivante : un processus P exécute une sortie à l'intention d'un processus Q, notée :

$$Q!sig\ (x_1, \ldots x_n)$$

où *sig* est un nom de signal, et $x_1, \ldots x_n$ des expressions définies dans P. A cette sortie doit correspondre dans Q une entrée notée :

$$P?sig\ (y_1, \ldots y_n)$$

où *sig* est le même nom de signal, et $y_1, \ldots y_n$ sont des variables de Q. A l'exécution de l'une de ces instructions dans l'un des deux processus, ce processus s'arrête jusqu'à ce que l'autre ait effectué l'instruction symétrique, compatible par le nom du signal et le nombre des arguments. Lorsque la "rencontre" se produit, il y a affectation simultanée des valeurs de $x_1, \ldots x_n$ à $y_1, \ldots y_n$, et les deux processus poursuivent leur chemin. Bien entendu, elle peut ne jamais se produire, auquel cas l'un des processus, ou les deux, resteront bloqués indéfiniment. Une synchronisation simple sans échange d'information correspond à un signal sans paramètres *sig( )*.

Les structures de contrôle de CSP sont reprises des structures de Dijkstra [5], avec une notation plus synthétique. L'écriture :

$$[C_1 \rightarrow A_1 \,[]\, C_2 \rightarrow A_2 \,[]\, \ldots\ldots \,[]\, C_n \rightarrow A_n]$$

désigne une instruction conditionnelle exécutant l'une des instructions A_i telle que la condition C_i correspondante est vraie, échouant si aucune ne l'est ; l'écriture :

$$*[C_1 \rightarrow A_1 \,[]\, C_2 \rightarrow A_2 \,[]\, \ldots\ldots \,[]\, C_n \rightarrow A_n]$$

est une boucle exécutant répétitivement l'instruction conditionnelle précédente, à cette différence près que le cas où toutes les C_i sont fausses devient ici la sortie normale de boucle.

CSP ajoute à ce formalisme des "commandes gardées" une extension aux "conditions", leur permettant d'inclure des instructions d'entrée *P?sig* $(y_1, \ldots y_n)$. L'évaluation d'une telle "condition" est suspendue jusqu'à ce que P envoie un signal compatible ; si cela se produit, l'instruction d'entrée s'effectue et produit la valeur *vrai*. Si P est terminé, la valeur *faux* est produite (on peut ainsi programmer un test de fin de fichier).

Les conditions peuvent être construites à partir de l'opérateur ; qui représente le "et conditionnel" : C_1 ; C_2 ; ... C_n signifie *si* C_1 *alors si* C_2 *alors* ... *si* C_{n-1} *alors* C_n ; une telle condition complexe peut inclure une instruction d'entrée unique, qui en est alors obligatoirement le dernier élément C_n ; la raison évidente de cette restriction est que l'évaluation d'une instruction d'entrée est irréversible. Notons que la présence d'instructions d'entrée dans plusieurs branches d'une instruction conditionnelle ou d'une boucle correspond à l'attente du premier parmi plusieurs événements possibles.

VI.5 - Un modèle en CSP

Ce rappel des bases de CSP permet d'écrire le programme, fort simple, traitant le "type abstrait avec scénario" décrit précédemment.

```
n : integer ;
   [utilisateur?créer (n) →
            i : integer ; phase_2 : boolean ;
            i := 0 ; phase_2 := true ;
           *[phase_2 ; utilisateur?rechercher (x) →
                                ...recherche (linéaire) de x... ;
                                utilisateur!résultat (...)
            [phase_2 ; i < n ; utilisateur?insérer (x) →
                                ...insertion de x...
            [phase_2 ; utilisateur?phase_3( ) → phase_2 := false
           ] ; réorganiser la table ;
           *[utilisateur?rechercher (x) →
                                ...recherche (associative) de x... ;
                                utilisateur!résultat (...)
```

EXEMPLE 7 : TYPE AVEC SCENARIO EN CSP

Une caractéristique intéressante de ce modèle est qu'on spécifie seulement le comportement légal du module, décrit par le schéma précédent ; toute tentative illégale (plus de n insertions, recherche ou insertion non précédées par une création, plusieurs créations) entraîne un blocage , etc.

VI.6 - Traitement de l'exemple en SIMULA

On peut utiliser le programme CSP précédent comme modèle pour l'expression de la même solution en SIMULA. Le résultat ne sera pas tout à fait aussi élégant, mais il est possible, on va le voir, de conserver l'essentiel de la structure.

La différence principale tient à l'absence de primitives de communication. L'information entre le module *U* (utilisateur) et le module *G* (gestionnaire de la table) passera par des variables globales qui serviront de boîtes aux lettres. Plus précisément :

- une boîte *taille* (variable entière) servira à déposer n pour la création ;
- une boîte *dépôt* (variable de type "*clé*") servira pour déposer une clé à rechercher ;
- une boîte *retrait* (variable de type "*info*") servira à abriter l'objet résultat d'une recherche ;
- une boîte *entrée* ("*info*") servira à déposer un objet à insérer ;
- une boîte *demande* (variable entière) contiendra le type de demande (*créer*, *insérer*, *rechercher*, ou *phase_3*).

On notera sur ce dernier point que le réflexe immédiat est d'écrire à la PASCAL :

type demande = (créer, insérer, rechercher, phase_3)

et qu'il faut malheureusement revenir ici à des codes numériques choisis par le programmeur. L'absence des types par énumération est de toute évidence un handicap sérieux dans les langages pré-pascaliens.

On suppose donc que le programme principal a la forme de l'exemple 8-1.

```
begin
integer demande ;
integer créer, insérer, rechercher, phase_3 ;
integer taille ;
class info ; ..... ; ref (info) retrait, entrée ;
class clé  ; ..... ; ref (clé) dépôt ;
class utilisateur ; .... cf. exemple 9 .... ;
class gestionnaire ; .... cf. exemple 10 .... ;
ref (utilisateur) U ; ref (gestionnaire) G ;
créer := 0 ; insérer := 1 ; rechercher := 2 ; phase_3 := 3 ;
U :- new utilisateur ; G :- new gestionnaire ; resume U
end programme principal
```

EXEMPLE 8-1 - PROGRAMME PRINCIPAL

L'"utilisateur" passe alors par des procédures données à l'exemple 8-2. Le module de gestion de table utilise les procédures de l'exemple 8-3, et sa partie algorithmique est donnée en 8-4.

On note dans cet exemple une certaine lourdeur dans le mécanisme de communication entre coprogrammes. Mais, comme le programme CSP, le coeur du processus de gestion de table (exemple 8-4) est remarquable en ce qu'il décrit le déroulement du scénario et rien d'autre. Toute suite de demandes incorrecte entraînera une erreur (bouclage, ou tentative de reprise d'un objet terminé) ; bien entendu, on aurait pu traiter explicitement ces erreurs (ce qui revient à ajouter au diagramme de transition des arcs supplémentaires). Mais on a bien en 8-4 une description du module *gestionnaire* en tant que processus homogène et autonome, ce qui était le but recherché.

```
class utilisateur ;
    begin
    procedure demander_création (n) ; integer n ;
            begin
            demande := créer ; taille := n ; resume G
            end demander_création ;
    procedure demander_insertion (x) ; ref (info) x ;
            begin
            demande := insérer ; entrée := x ; resume G
            end demander_insertion ;
    ref (info) procedure demander_recherche (c) ; ref (clé) c ;
            begin
            demande := rechercher ; dépôt :- c ; resume G ;
            demander_recherche :- retrait
            end demander_recherche ;
    procedure demander_phase_3 ;
            begin
            demande := phase_3 ; resume G
            end demander_phase_3 ;
    detach ;
    end utilisateur ;
```

EXEMPLE 8-2 : UTILISATEUR

```
class gestionnaire ;
    begin
    integer tailletab, nombrélem ;
    procedure création ;
            begin
            tailletab := taille ; ... créer la table... ;
            resume U
            end création ;
    procedure insertion ;
            begin ref (info) x ;
            x :- entrée ; ... insérer x dans la table ... ;
            nombrélem := nombrélem + 1 ; resume U
            end insertion ;
    procedure recherche_linéaire ;
            begin ref (clé) c ; ref (info) resul ;
            c :- dépôt ; ... rechercher, dans la table gérée
                             linéairement, un élément résul de clé c... ;
            retrait :- résul ; resume U
            end recherche_linéaire ;
    procedure recherche_associative ;
            begin ref (clé) c ; ref (info) résul ;
            c :- dépôt ; ... rechercher, dans la table gérée
                             associativement, un élément résul de clé
                             c ... ;
            retrait :- résul ; resume U
            end recherche_associative ;
    procedure changer ;
            begin
            ... réorganiser la table linéaire en table associative...
            end changer ;
```

EXEMPLE 8-3 : GESTIONNAIRE (PROCEDURES NON EXPORTEES)


```
    detach ;
    if demande = créer then
        begin
        nombrélem := 0 ;
        while (demande ≠ phase_3) do
            if demande = recherche then
                recherche_linéaire
            else if demande = insertion and nombrélem < tailletab then
                insertion ;
        changer ;
        while demande = recherche do
            recherche_associative
        end vie de la table
end classe gestionnaire
```

EXEMPLE 8-4 : GESTIONNAIRE (INSTRUCTIONS DE LA CLASSE)

VII - CONCLUSION

Nous nous permettrons, pour conclure, d'abandonner le ton de l'analyse académique pour ouvrir une réflexion plus subjective sur le problème de l'évaluation et du choix d'un langage de programmation.

En examinant la manière dont SIMULA résiste à un examen selon les critères étudiés, qui correspondent à quelques-unes des préoccupations principales dans la réflexion actuelle sur les langages de programmation, on peut suggérer le bilan suivant :

- SIMULA propose des constructions répondant au moins en partie à tous les concepts examinés, à l'exception de celui de la protection, dont l'application reste essentiellement à la charge du programmeur - tout au moins tant que la proposition de Palme [16] ne fait pas partie de la norme.

- Les concepteurs de SIMULA se sont heurtés en 1967 à des difficultés que devaient connaître à leur suite les créateurs de toute la série de langages plus récents cherchant à répondre aux critères cités. Sans qu'une comparaison sérieuse ait été ébauchée dans cet article (on pourra cependant se reporter aux autres communications du même volume), il semble bien que les solutions retenues en SIMULA 67, si elles ne sont pas meilleures, ne sont pas non plus tellement pires que ce qui a été choisi dans ces propositions nouvelles.

La présente étude de SIMULA a été suscitée par une réflexion sur un problème tout à fait concret : quel langage, ou, plutôt, quel système de programmation, peut-on envisager d'introduire dans un environnement industriel de programmation, pour aider à améliorer la manière dont sont écrits les programmes (un des sous-produits d'un tel effort, s'il réussit, étant de permettre aux utilisateurs de faire *moins* de programmation et plus de physique, d'automatique, de comptabilité, ou d'autre chose) ?

Considérons plus précisément le cas de l'informatique scientifique, et de projets d'une certaine taille exigeant des ressources (temps, mémoire centrale, fichiers) importantes. Nous écartons donc la micro- et la mini-informatique. Essayons d'analyser les difficultés qu'éprouvent dans un tel cadre les utilisateurs de calcul avec leur outil principal, en général FORTRAN. Il semble qu'elles se ramènent pour une grande part (sans que les intéressés en aient en général conscience) aux problèmes cités ci-dessous par ordre d'importance décroissante, le premier se détachant nettement comme crucial :

1. La modularité, la conception descendante : comment développer et conserver un grand programme sous forme de "morceaux" cohérents, l'ensemble restant maîtrisable ? Notons en particulier que l'impossibilité dans les langages classiques de construire un module autour d'une structure de données (une classe) entraîne des pratiques conduisant par réaction à une visilité totale ("communs poubelles") qui sont une cause d'anarchie particulièrement répandue.

2. L'allocation dynamique de tableaux. Ce problème peut sembler peu sérieux en théorie, mais joue un rôle important en programmation scientifique. Chaque programmeur a ses techniques propres pour pallier l'absence d'allocation dynamique, de la gestion d'un commun spécial à l'utilisation d'un "préprocesseur" ad hoc. Il s'agit là aussi d'un facteur important d'incohérence dans les programmes (et d'inefficacité).

3. La récursion. On rencontre régulièrement des programmes qui effectuent (souvent à l'insu du programmeur) des opérations fondamentalement récursives, comme des parcours d'arbres. L'absence de récursivité dans le langage entraîne un mélange douteux entre les opérations de l'algorithme et le contrôle de la récursion (gestion de pile), qui sape la structure du programme.

4. (Pour les utilisateurs de FORTRAN).La manipulation de caractères.

5. Les structures de contrôle.

Bien entendu, on ne peut proposer de remplacement aux solutions actuelles qui détruirait leurs qualités les plus appréciables :

A - L'efficacité. On peut dans la plupart des cas accepter une certaine dégradation de performances comme prix d'une programmation plus rapide et plus fiable, mais pas au-delà d'une certaine limite, de 20% à 50%.

B - La compilation séparée, la possibilité de réutiliser des programmes écrits dans d'autres langages et de constituer des bibliothèques.

C - Des entrées et sorties riches, l'accès direct.

D - La qualité "industrielle" de la documentation, la facilité d'emploi du compilateur, l'accès à des opérations du système d'exploitation, etc.

E - La normalisation du langage et la transportabilité des programmes (exigeant en général un certain compromis avec C et D).

Notons que toutes ces qualités s'appliquent non à des langages seuls, mais à des systèmes de programmation - qui sont ce que l'utilisateur "voit"".

Si l'on écarte les langages récents encore à l'état d'expérimentation, pour ne considérer que les langages de large diffusion, on peut faire les constatations suivants :

- FORTRAN ne répond pas aux critères 1,2,3,4 ni 5 ; la nouvelle norme n'améliorera sensiblement les choses que pour 4 (caractères) et un peu 5 (structures de contrôle).

- COBOL satisfait 4 et à peu près 5, mais non 1, 2 ni 3.

- PL/1 ne répond pas au critère 1 (modularité). Il est encore mal normalisé (E).

- PASCAL n'est pas satisfaisant pour 1 ni pour 2 (tableaux alloués à l'exécution). La compilation séparée (B) n'est pas définie de façon normalisée, bien que certains systèmes la permettent.

- SIMULA offre une solution satisfaisante à 1, excellente à 2,3,4 (la gestion du type *text*, à la façon d'un type abstrait, est remarquablement élégante), et bonne à 5, malgré l'absence regrettable d'une instruction *case...of* ... . Les qualités A à D dépendent évidemment à la fois de E (normalisation) et des divers systèmes. On peut noter que le langage est théoriquement bien normalisé (document [7]), mais que l'épaisseur des "guides du programmeur" propres à chaque système montre qu'il reste des problèmes. Pour A (efficacité), les systèmes "officiels" diffusé par le NCC sont satisfaisants. Le point B (compilation séparée) est fixé par la norme de façon également satisfaisante, mais comme une facilité optionnelle que n'offrent donc pas tous les systèmes, à l'origine en tout cas. Les entrées et sorties et l'accès direct (C) sont bien définis, grâce à la classe système *BASICIO*, et SIMULA est agréable à utiliser pour ce type d'opérations ; noter cependant que pour des raisons d'ailleurs évidentes la norme est assez discrète sur les détails de l'accès direct. Enfin, le critère D (qualité industrielle) est convenablement rempli par les systèmes NCC.

Choisir un langage de programmation et, encore plus, un langage destiné à en supplanter un autre, est une lourde décision, et les critères à considérer ne se limitent pas à ceux que nous venons de voir. Nous nous garderons donc de toute affirmation péremptoire. Nous espérons cependant avoir permis au lecteur de noter que dans le paysage changeant des modes en programmation, et en attendant le langage du futur, UTOPIA 84, ou 94, ou 2004, SIMULA est plus qu'un monument antique et respectable.

REMERCIEMENTS

Je remercie les organisateurs de GROPLAN et de la réunion de Cargèse pour avoir suscité une réflexion comparative sur les langages de programmation actuels, l'auditoire pour ses critiques lors de mon exposé à cette réunion, J. André pour plusieurs remarques importantes, ainsi qu'A. Bossavit, M. Demuynck, B. Logez et N. Penot pour leurs commentaires sur la première version de cet article.

BIBLIOGRAPHIE

[1] Birtwistle, Graham M., Ole Jordan Dahl, Bjorn Myhrhaug, et Kristen Nygaard :
SIMULA BEGIN ; Studentliteratur, Lund (Suède) et Auerbach Publishers, Philadelphie (Pennsylvannie), 1973

[2] Carrez, C. :
La Protection dans les systèmes et son expression dans les langages de programmation ; Réunion AFCET-GROPLAN, Cargèse, 14-22 mai 1979.

[3] Dahl, Ole Jordan :
SIMULA, An Algol-Based Simulation Language
Communications of the ACM, 9, 9, pages 671-681, juillet 1966.

[4] Dahl, Ole Jordan, Bjorn Myhrhaug et Kristen Nygaard :
Common Base Language ; Norwegian Computing Center, Publication S22, Octobre 1970

[5] Dijkstra, Edsger W. :
A Discipline of Programming
Prentice-Hall, Englewood Cliffs (N.J.), 1976

[6] Dufourd, J. F. :
Types abstraits, modèles relationnels, et langage SIMULA
Colloque "Les bases de Données, modèles, mise en oeuvre, évaluation" ; Chapitre français de l'ACM, Université Paris VI, 14-15 juin 1979.

[7] Geschke, Charles M., James H. Morris Jr., et Edwin H. Satterthwaite
Early Experience with Mesa ; Communications of the ACM, 20, 8, pages 540 - 553, août 1977

[8] Hoare C.A.R. :
Communicating Sequential Processes ; Communications of the ACM, 21, 8, pages 162-180, Août 1978

[9] Honeywell, Inc. et Cii Honeywell Bull : *Reference Manual for the GREEN Programming Language* ; 15 Mars 1979.

[10] Jackson, Michael A. :
Principles of Program Design ; Academic Press, Londres, 1975.

[11] Jacquet, Paul :
La Généricité comme outil d'abstraction dans les langages de programmation ; Théories et Techniques de l'informatique, Congrès AFCET-TTI, Gif-sur-Yvette, 13-15 Novembre 1978

[12] Kaubisch W.H. et C.A.R. Hoare :
Discrete Event Simulation Based on Communicating Sequential Processes ; Department of Computer Science, The Queen's University, Belfast (Irlande du Nord), 1978

[13] Lampson B.W., J.J. Horning, Ralph L. London et G.J. Popek :
Report on the Programming Language Euclid ; SIGPLAN Notices 12, 2, pages 1-79 (n° spécial), Février 1977

[14] Liskov, Barbara H., Alan Snyder, Russel Atkinson et Craig Schaffert :
Abstractions Mechanisms in CLU ; Communications of the ACM, 20, .8, Août 1977, pages 564-576.

[15] Meyer, Bertrand, et Claude Baudoin :
Méthodes de Programmation ; Eyrolles, Paris, Collection de la Direction des Etudes et Recherches EDF, 1978.

[16] Palme, Jacob :
New Feature for Module Protection in Simula ; SIGPLAN Notices, 11, 5, 1976.

[17] Parnas, David L. :
A Technique for Software Module Specification with Examples ; Communications of the ACM, 15, 5, pages 330-336, Mai 1972.

[18] Parnas, David L. :
On the Criteria to be used in Decomposing Systems into Modules ; Communications of the ACM, 15, 12, pages 1053-1058, Décembre 1972

[19] Parnas, David L. :
Designing software for Ease of Extension and Contraction ;
IEEE Transactions on Software Engineering, SE-5, 2, pages 128-138, Mars 1979

[20] Scholl, Pierre-Claude, Pierre-Yves Cunin et Michael Griffiths :
Aspects fondamentaux du Langage MEFIA ; Journées d'Etude sur la Fiabilité des Programmes dans les Applications Industrielles, Chapitre français de l'ACM, EDF (Clamart), 26-27 avril 1978

[21] Shaw, Mary L., William A. Wulf et Ralph L. London :
Abstraction and Verification in Alphard : Defining and Specifying Iteration and Generators ; Communications of the ACM, 20, 8, pages 553-564, Août 1977.

[22] Wirth, Niklaus :
Program Development by Stepwise Refinement ; Communications of the ACM, 14, 4, pages 221-227, Juillet 1971.

[23] (Wirth, Niklaus) : *Libres Questions à Niklaus Wirth* ; Interview dans 01-Informatique, 130, pages 106-112, mai 1979.

ELECTRICITE DE FRANCE
DIRECTION DES ÉTUDES ET RECHERCHES

Service Informatique
et Mathématiques Appliquées

Le 7 Novembre 1979

[80a]

DEPARTEMENT METHODES ET MOYENS
DE L'INFORMATIQUE

1, Avenue du Général de Gaulle
92141 CLAMART

Tél. 765.43.21

MEYER B.

A BASIS FOR THE CONSTRUCTIVE APPROACH TO PROGRAMMING

HI.3291/01 13 pages

Version 2 : le 22 janvier 1980
Version 3 : le 15 avril 1980
Version 4 : le 16 Mai 1980
Article paru dans les actes du congrès IFIP 80
(Tokyo, 6-9 octobre 1980).

Résumé : Although programming is a difficult and creative activity, useful strategies and heuristics exist for solving programming problems. We analyse some of the most fundamental and productive among them ; their knowledge and conscious application should help the programmers in constructing programs, both by stimulating their thinking and by helping them to recognize classical situations. The precise framework for the analysis is provided by the specification language Z, informally described in the paper.

Several examples are provided, drawn from various fields (in particular numerical analysis), and ranging from trivial problems to new algorithms.

A shortened version of this paper appeared in Information Processing 80 (proceedings of the IFIP World Computer Congress, TOKYO, October 6-9, 1980), S.H. Lavington (Ed.), NORTH-HOLLAND, 1980 (pp 293-298).

A BASIS FOR THE CONSTRUCTIVE APPROACH TO PROGRAMMING

Bertrand MEYER
Electricité de France, Direction des Etudes et Recherches (service IMA)
1 avenue du Général de Gaulle 92141 Clamart (France)

Although programming is a difficult and creative activity, useful strategies and heuristics exist for solving programming problems. We analyse some of the most fundamental and productive among them; their knowledge and conscious application should help the programmers in constructing programs, both by stimulating their thinking and by helping them to recognize classical situations. The precise framework for the analysis is provided by the specification language Z, informally described in the paper.

Several examples are provided, drawn from various fields (in particular numerical analysis), and ranging from trivial problems to new algorithms.

1. THE NEED FOR A CONSTRUCTIVE APPROACH TO PROGRAM ANALYSIS

1.1 Introduction

The evolution of the various domains of computer science has led to the development of powerful program analysis methods. They make it possible to study many properties of problems and programs; e.g., to determine whether a problem can be solved at all, and, if so, whether there exist realistic algorithms; to evaluate the abstract and concrete complexity of a program; and to prove it correct relative to some specification.

Useful as these techniques may be, they do not provide a completely satisfactory answer to the practicing programmer, whose immediate concern is to *build* programs which will solve given problems. Programming is a difficult intellectual activity, and it can hardly be expected that straightforward "methods" will ever be discovered, let alone "algorithms", to deduce programs from problems. To anyone seriously concerned with programming, however, it is obvious that certain fruitful thought patterns do recur with a remarkable frequency, and it is quite a temptation to try to analyze and formalize them with the hope that their knowledge will be of some help for those who construct programs. Such is the aim of this paper.

Several authors have remarked that program proving techniques are of less use for proving the correctness of existing programs than as tools to help the programmers write programs which will be correct in the first place. An important work in this direction is that of Dijkstra [5]. We will try to elaborate on these methods and give a precise basis for their application.

For any precise reasoning about programming, the use of a formal notation is unavoidable. We will rely on such a notation, the "Z" specification language, the essentials of which are summed up in section 2. Section 3 is devoted to the presentation of our framework for the constructive study of programs, and a brief comparison with other approaches. Section 4 contains several significant examples. In section 5, we analyze the scope and implications of the concepts and techniques discussed.

Before going into detailed analysis, it may be useful to set the general tone of the work by informally introducing some of the ideas on a toy example. More serious examples are treated in section 4.

1.2 A toy example

Assume we are looking for an algorithm to compute square roots. The problem is to find a method which, given any $a \geq 0$, will yield x such that

$$x \geq 0 \ \underline{and}\ x * x = a \qquad (1)$$

Looking at the form of eq. (1), we decide to try a heuristic called "uncoupling" (see 4.1), which roughly suggests that "one variable appearing twice may be replaced by two equal variables", i.e. here eq. (1) may be replaced by

$$x \geq 0 \ \underline{and}\ y \geq 0 \ \underline{and}\ x * y = a \ \underline{and}\ x = y \qquad (2)$$

The heuristics used also suggests that we call "invariant" the conjunction of the first three clauses in eq. (2), and "goal" the last one $(x = y)$, and look for an algorithm of the form:

```
establish invariant;
while not goal do
    begin
    get x and y closer to each other;
    restore invariant
    end
{goal and invariant} {i.e., the desired conclusion}
```

establish invariant is readily implemented by the assignments

$x := 1 ; y := a$

get x and y closer to each other may be chosen (among many other possibilities : this is probably the central design decision) as :

$$x := (x + y)/2 .$$

restore invariant may then be :

$$y := a/x$$

What we have obtained is the classical algorithm known as Newton's method; it is indeed easily seen that the sequence of values taken by x satisfies $x_{n+1} = (x_n + a/x_n)/2$. Of course, one must show that the algorithm terminates in a finite number of steps (adding to *"x = y"* the mention "within a prescribed margin of accuracy"), i.e. that $|y_n - x_n|$ is a converging sequence.

After this example which shows that few "creative" decisions may be needed in order to discover a good algorithm, we come to the description of the notation used in the rest of this paper.

2. A NOTATIONAL BASIS : THE Z SPECIFICATION LANGUAGE

2.1 Basic concepts

In order to precisely define the constructive meaning of programming structures, we need a notation which is both formal and readable. Moreover, it should be purely static, i.e. involve only well-known mathematical constructs.

The "Z" specification language satisfies these requirements. This language has been used [1] to model all kinds of information processing problems, ranging from text editing to "on-the-fly" garbage collection, system problems, programming language semantics, business data processing problems etc. Z is based on formal set theory and logic; it uses a notation similar to that of programming languages, and has in particular been influenced by the syntax of ADA [8].

We shall in no way attempt a complete description of Z, which is to be found in [1]. Instead, we will informally introduce the main constructs. Many of the examples will be notions used later.

A Z text is divided into "chapters" which play for specifications a role comparable to that of libraries for programs. A chapter is a sequence of definitions of the form

<name> ≙ <expression>

where ≙ stands for "equals by definition", and/or theorems of the form :

<theorem_name> ≙ theorem <predicate> end

Names defined in a chapter may be made visible in another one through a "use" clause similar to that of ADA. A chapter thus has the following form :

<chapter_name> ≙ chapter $chap_1,\ldots,chap_n$ *def*
<list of definitions and/or theorems>
end

where the list of chapters, which may be empty, allows all the objects defined in the corresponding chapters to be freely used in the sequence of definitions enclosed by *def* ... *end*.

2.2 Basic constructs

<expression>'s denote either sets in the usual mathematical sense or classes, i.e. structures (such as the group or topological structures in mathematics). Syntactically, expressions may be names, functional or relational expressions of the form $r(e_1,\ldots,e_n)$, or quantified expressions which have the following general "comb-like" structure* :

<quantifier> $id_1,\ldots, id_n$ *for*
id_1 : SET_1;
.....
id_n : SET_n
where
<boolean expression>
then
<definitions>
proof
<justification>
given
<definitions>
end

$id_1,\ldots, id_n$ are quantified variables ranging over the given sets. Depending on the quantifier, one of the *where* ... and *then* ... clauses, or both, may be present. The *given* ... clause is an optional notational facility which allows for giving local names to expressions used in a definition. The *proof* clause will be used when it is deemed necessary to give a formal or informal justification of the correctness of an expression.

The following simple example is a boolean expression using the set *NAT* of natural numbers (rigorously defined in one of the basic chapters), the set difference operator - and the quantifier *forall* :

*forall x for x : NAT - {0,1} then x**2 > x end*

The following *<definition>* uses the quantifiers *set* and *exist* and defines a set :

* For readability, set and class names are usually written in capital letters, whereas elements, relations and functions are denoted by small-case characters. This is mere notational convenience, since there is no theoretical difference in Z between sets and elements.

ODD_EVEN_COUPLE ≙
set x, y for x,y : NAT where
exist i for i : NAT where
*x = 2*i and y = x+1*
end
end

ODD_EVEN_COUPLE is a subset of the cartesian product *NAT* × *NAT*. To conclude with these basic elements of Z, note that :

—Comments in Z begin with — and end with the — line.

2.3 Relations and functions

Besides set, one may also define relations and functions. The latter are single-valued. *X* and *Y* being sets, the following sets are defined : *subset(X)* is the set of subsets of *X*, including *null*, the empty subset; $X \leftrightarrow Y$ is the set of relations between *X* and *Y*, formally defined as

subset$(X \times Y)$;

$X \to Y$ is the set of total functions from *X* to *Y* (partial functions will not be used in this paper).

The quantifiers *rel* and *func* are used for defining relations and functions :

divides ≙ rel d ↔ n for d, n : NAT where
*exist q for q : NAT where n = q*d end*
end ;

*square = func n → s for n, s : NAT then s ≙ n**2*
end

Note the difference between the syntax for *rel*, which uses a *where <boolean expression>* clause, and that for *func* which uses

then <result> ≙ <expression>

with the "equals by definition" operator rather than the equality predicate : it should be clearly deducible from the text of a function definition that there is one and only one result for every argument which belongs to the allowable domain. If on the other hand a relation *r* is shown to be functional, i.e. total and single-valued, the associated function may be denoted by *function(r)*.

2.4 Generic definitions

Unlike the above examples, most definitions in Z (as well as theorems) have one or more generic parameters, i.e. dummy names representing arbitrary sets. They appear in square brackets. For example, the following generic definition introduces the identity relation *id[X]* relative to any set *X* :

id[X] ≙ rel x, y for x, y : X where x = y end

We get an instance of the relation *id* by providing an actual (set) parameter for *X*. For example *id[NAT]* is the identity relation between natural numbers.

Another generic definition is that of the constant function, introduced as :

constant_func[X,Y] ≙
func y → f for y : Y; f : X → Y then
f ≙ func x → y' then y' ≙y end
end

Note that *constant_func* is a function whose result is a function; i.e., for *y* in *Y*, *constant_func[X,Y](y)*, or simply *constant_func(y)* (the generic parameters may be omitted if no ambiguity results), is the constant function *f* such that *f(x) = y* for all *x*. We will also use the relation *constant_rel[X,Y]* which is identical to *constant_func* except that *constant_rel(y)* is a relation.

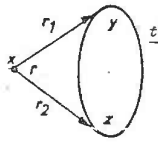
An important feature of Z is the ability to perform "relational calculus" on relations and functions. To this end, relational operators are provided, such as ∘ (the composition operator) which may be defined as follows :

op(∘)[X,Y,Z] ≙ func g, f → h for
f : X ↔ Y ;
g : Y ↔ Z ;
h : X ↔ Z
then
h ≙ rel x↔z for x : X; z : Z
where
z ≙ g(f(x))
end
end

The notation *op(f)* denotes a function which is to be used as a binary operator *f* (such as ∘ here or +, etc.) in infixed form.

Another important relational operator is &. r_1 & r_2 associates with *a* all couples *(b,c)* such that *a* is related to *b* through r_1 and to *c* through r_2.

op(&) [X,Y,Z] ≙ func r_1, r_2 → *r for*
$r_1 : X \leftrightarrow Y$;
$r_2 : X \leftrightarrow Z$;
$r : X \leftrightarrow Y \times Z$



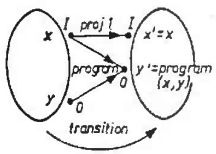
then
r ≙ rel x ↔ y, z for x : X;
y : Y; z : Z where
$r_1(x \leftrightarrow y)$ *and*
$r_2(x \leftrightarrow z)$
end
end

The notation $r_1(x \leftrightarrow y)$ expresses that *(x,y)* belongs to the relation r_1.

As an example, let *program* be a function from $I \times O$ into *O*. The standard generic function *proj1[I,O]* is the first projection from $I \times O$ (i.e. into *I*). Then

transition = proj1[I,O]& program

is a function from $I \times O$ into itself such that *transition* (x,y) is



$(x, program\ (x,y))$; i.e. *transition* conserves its first argument, and applies *program* to get the second component.

Since relations and functions are defined as sets, set operators apply to them. For example, *program* being a function from I to O and *problem* a relation in $I \leftrightarrow O$, the notation *program* $\subset$ *problem* means exactly the same as

```
forall x for x : I then
    problem (x ↔ program(x))
end
```

i.e., any couple of the form $(x, program(x))$ is in the relation *problem*.

A last thing to note about relations is that, r being a relation, $r(x)$ denotes the set of y such that $r(x \leftrightarrow y)$, and $dom(r)$ is the domain of r, i.e. the set of all x such that $r(x) \neq null$.

2.5 Classes

To end this overview of Z, we introduce the notion of class. Classes correspond to structures in mathematics. They are characterized by a certain number of named components satisfying certain constraints. For example, the structure of monoid over a set X is defined by :

```
MONOID[X] ≙
    class with
        oper : X × X → X;
        neutral : X
    where
        forall x, y, z for x, y, z : X then
            assoc : oper(x, oper(y,z))=
                oper(oper(x,y), z);
            lneutr : oper(x, neutral) = x;
            rneutr : oper(neutral, x) = x
        end
    end
```

Note that successive elements in a list of conditions are labeled (see here *assoc*, *lneutr*) for ease of reference, e.g. in *proof* clauses.

Another, more computer-related, example, defines the properties one might expect from a character set :

```
CHARACTER[C] ≙ class with
                  blank, newline : C;
                  ALPHABETIC : subset (C);
                  NUMERIC    : subset (C)
                  SPECIAL_CHAR : subset (C)
                where
                  {ALPHABETIC, NUMERIC, SPECIAL_
                  CHAR, SEPAR} ∈ PARTITION [C]and
                  finite(C)
                given
                  SEPAR = {blank, newline}
                end
```

Note that *PARTITION[X]*, whose definition the reader may wish to write in Z, is the set of all partitions of X; that *finite(X)* is a boolean expression denoting the finiteness of X; and that the notation $\{a, b, \ldots\}$defines a set by enumeration of its elements.

Given a class definition, objects belonging to this class may be defined through the use of the *cons* operator which gives actual values to the class components, provided they meet the requirements of the *where* clause. Note that *cons* will often have a *proof* clause to make it clear that the defined objets do indeed comply with the class requirements.

We may for example define the following objects:

```
NAT_MON ≙ cons MONOID[NAT] with
              oper ≙ op(+); neutral ≙ 0
          proof
              op(+) ∈ ASSOCIATIVE[NAT] ∩
                  INVERSIBLE[NAT];
              0 = NEUTRAL_ELEMENT(+)
          end;

— 0..n for n in NAT, is the integer interval
— {0,1,2,... n}

EBCDIC ≙ cons CHARACTER[0..255]with
             blank ≙ 64; newline ≙ 21;
             ALPHABETIC ≙ {81;82,...}
             NUMERIC    ≙ {240;241,...}
             SPECIAL_CHAR ≙ {...}
         end;

ASCII ≙ cons CHARACTER[0..63] with
            blank ≙ 45; newline ≙ 41;
            ALPHABETIC ≙ {1,2,... 26};
            NUMERIC    ≙ {27,...  36};
            SPECIAL_CHAR ≙ {...}
        end
```

The names of components of a class may be used as functions defined on the set of objects belonging to this class. For instance, for any X satisfying the requirements of class *CHARACTER*, *blank* is a function from *CHARACTER[X]* into X. Thus *blank(ASCII)* is equal to 45.

Another property of classes, which we shall not use here, is the ability to combine the features of one or more classes (named between *class* and *with*) into a new one, either conjunctively or disjunctively. This is very useful for the stepwise specification of non-trivial systems.

2.6 Order relations

As an example which will introduce notions used below, we write a small chapter describing order relations, and in particular "well-founded" order relations, also called "noetherian" [3].

```
STRICT_ORDER ≙

    chapter <some basic chapters; see [1]> def
    — transitive, irreflexive and order
    — relations :
    trans[X] ≙ set r for r : X ↔ X where
        (r ∘ r) ⊂ r end;
    irreflex[X] ≙ set r for r : X ↔ X where
        r ∩ id[X] = null end;
    strict_order[X] ≙ trans[X] ∩ irreflex[X];

    integer_order ≙ theorem op(<) ∈ strict_
        order[NAT] end;
    minimum[X] ≙ rel A, r ↔ m for
                    A : subset(X);
                    r : strict_order (X);
                    m : X
                where
                    A ⊂ r(m) — i.e. r(m↦a)
                      —for all a in A
                end;
    strict_well_founded[X] ≙
                set r for r : strict-order[X]
                where
                    forall A for A : subset(X)
                      then minimum (A,r) ≠
                      null
                    end
                end;

-- Let f be a function from X into X and n a
-- NAT. Then iter(n) (f), defined in another
— chapter, is the n-th iterate of f.
— A being a subset of X, f ⌈A is the restric-
-- tion of f to A.
converge[X] ≙ theorem — see reference[3],
                     — III.51, prop. 6.
            forall A, f, a for
                    A : subset(X) - {null};
                    f : (X → X);
                    a : A
            where — f on A is the inverse
              --of a strict well founded
              —relation :
                    f ⌈A ∈ inv(strict_well_
                      founded[X]);
            then
                    exist n for n : NAT where
                          iter(n) (f) (a) ∉ A
                    end
            end
        end;

limit[X] ≙ func A, f → g for
                A : subset(X) - {null};
                f : X → X;
                g : A → X - A (— A's comple-
                              — ment)
           where
                f ⌈A ∈ inv(strict_well_
                  founded[X]);
           then
                g ≙ func a → b for a : A;
                                   b : X-A then
                    b ≙ iter(n) (f) (a)
                given
                    N ≙ set m for m : NAT
                        where iter(m) (f)
                        (a) ∉ A
                        end;
                    n ≙least (N)
                      — smallest elt.
                proof
                    N ≠ null from
                    theorem "converge"
                end
           end
end
```

3. A FORMAL BASIS FOR THE CONSTRUCTION OF PROGRAMS

3.1. Guidelines

The framework for our representation of programs is the concept of a solution to a programming problem, which will be characterized by :

- two(generic) sets : the input set I and the output set O;
- a relation on $I \times O$, which is the problem to be solved;
- a function from I to O, which represents a program "implementing" the relation. Programming is the search for explicit functions compatible with relations given in implicit form.

This is readily expressed in Z by a class which we call *solution* :

```
solution[I, O] ≙ class with
                     problem : X ↔ Y;
                     program : X → Y
                 where
                     program ⊂ problem
                 end
```

In this approach, any constructive theory of programming may be considered as a set of rules for constructing solutions for certain programs, and deducing solutions to new problems from solutions to simpler ones. As examples from the first class, we find the following cases :

```
identity[X] — the "no_op" program for the
            — trivial problem.
          ≙ cons solution[X,X] with
                problem ≙ id[X];
                program ≙ function (id[X])
            end;

assignment[I,O] ≙ — the assignment rule
    func o → s for o : O; s : solution[I,O]then
        s ≙ cons solution[I,O] with
                problem ≙ constant_rel[I,O](o);
                program ≙ constant_func[I,O](o)
            end
    end ;

abort[I,O] = — the impossible problem has no
           — solution.
           theorem
               X = null
           given
               X ≙ set s for s : solution[I,O]
                   where problem(s) = null
                   end
           end
```

In the first two examples, the relations *problem* are functional (and, in fact, equal to their programs). Of course, this is not the case in general; many problems are not single-valued, as exemplified by the simple case of a sorting problem with no stability constraint, or most problems in numerical analysis which ask for o such that $|f(i,o)| < \varepsilon$ for some ε. Indeed, the rules given below will be of the form

```
programming_method[...] ≙
    func s1, s2, ... sn → s for
        s1, s2, ... sn, s : solution [...]
    where some_condition (s1, s2, ..., sn)
    then
        s ≙ cons solution [...] with
                problem ≙ f(problem(s1), ...
                                problem(sn));
                program ≙ g(program(s1), ...
                                program (sn))
            end
    end
```

where the function *programming_method* is not inversible. If it were, then programming would be an algorithmic activity, which it certainly isn't. The rules given below are thus to be used as heuristics which help the programmer recognize some situations wherein certain constructive rules may be applied to build the solution.

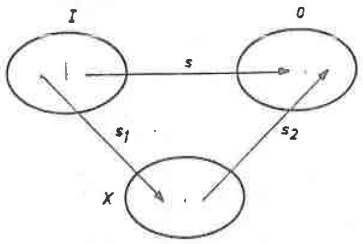
The rules we give correspond to classical control structures : sequence, choice and loop. They can be extended to others, such as recursion, along the same lines.

3.2 Sequence

The rule for program composition is as follows :

```
sequence [I,X,O] ≙
    func s1, s2 → s for
        s1 : solution[I,X];
        s2 : solution[X,O];
        s  : solution[I,O]
    then
        s ≙ cons solution [I,O] with
              problem ≙ problem (s2)
                o problem(s1);
              program ≙ program (s2)
                o program(s1)
            proof
              program is a function and
                program ⊂ problem
            end
    end
```

The constructive interpretation of this rule could be phrased as : if you can't go there directly, then go indirectly. The rule is a functional equivalent of Hoare's and Dijkstra's axioms for composition [5][7].
Note the *proof* clause used to justify the *cons*.

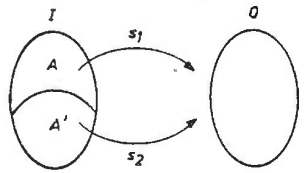


3.3 Choice

The two-way choice, or alternative, may be described as follows :

```
alternative[I,O] =
    func s1, s2, A → s for
        A : subset (I);
        s1 : solution [A,O]; s2 : solution [A',O];
        s  : solution [I,O]
    then
        s ≙ cons solution [I,O] with
              problem ≙ problem (s1)
                ∪ problem (s2);
              program ≙ program (s1)
                ∪ program (s2)
            proof
              program ∈(I → O) and program
                ⊂ problem
            end
    given
        A' ≙ I - A
    end
```

The constructive interpretation of this rule is that when looking for a solution to a programming problem with I as input domain it may be useful to look for partial solutions defined on disjoint subsets of I. The rule could of course be generalized to a partition involving more than two subsets. It does not, however, gracefully generalizes to Dijkstra's non-deterministic *if ... fi* construct. This is quite natural since we stated from the beginning that we were looking for functions.



The PASCAL construct corresponding to the above functional expression is :

```
if i ∈ A then
    o := f1 (i) {where f1 = program(s1)}
else {i ∈ A'}
    o := f2 (i) {where f2 = program(s2)}
```

3.4. Loops

We choose to model the so-called *while* loop. It is well-known from the work of Floyd [6] and Hoare [7] that one of the basic concepts in connection with loops is that of "invariant" or "inductive assertion". However, invariants are often presented after the construction of a loop; on the other hand, in [5], although invariants are used throughout as a constructive technique, the corresponding rule is not an axiom of the proposed semantics, but a consequence of the axioms for loops. It seems to us that the concept of invariant is so important that it should be part of the definition of loops. As we shall see, this can be done in a simple way; loop initialization and loop invariant turn out to be one concept.

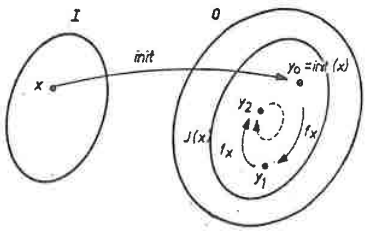
Before we turn to the formal description, it is useful to present a closely related mathematical analogy which gives much insight into the essence of loops. Let $p(x,y) = 0$ be an equation where x is given and y is the unknown. x and y are usually vectors. Under certain conditions, this equation may be transformed into the fixpoint equation $y = f(x,y)$ where $f(x,y)$ is defined as $y + Ap(x,y)$ A being some linear application. This equation is solved by taking the limit of the sequence y_n, defined by :

$$y_0 = init(x);\ y_{i+1} = f(x, y_i)$$

For the sequence to converge in a certain domain $J(x)$, f must satisfy a certain condition, called the Lipschitz condition, under which for all y, y' in $J(x)$:

$$|f(x,y') - f(x,y)| < k_x\ |y' - y|$$

for a value $k_x < 1$ independent of y. The choice of A helps meet this requirement. Moreover, the initialization function *init* must ensure that *init(x)* belongs to the convergence domain $J(x)$.



In programming a loop is a fixpoint computation method which generalizes the above scheme. It has an initialization part *init* which ensures the initial validity of the invariant assertion; the latter expresses membership in the "convergence domain" $J(x)$. Then the loop has a body which is a transition function f_x; for all x, the transition function has the property that $f_x(y)$ belongs to $J(x)$ if y does (except perhaps when y is equal to the limit of the sequence), i.e. that the invariant is indeed invariant under f_x; and that f_x "converges" i.e. satisfies a kind of "Lipschitz" condition usually expressed by an associated variant function v_x such that $v_x(f_x(y)) < v_x(y)$ and the range of v_x is a strict well-founded set; of course, programming, unlike classical analysis, usually requires that sequences reach their limits.

With this suggestive analogy in mind, we are ready to express the loop as a constructive problem-solving method.

```
loop[I,O] ≙
    func start, body, exit → ℓ for
        start, ℓ: solution[I,O];
        body    : solution[I × O, O];
        exit    : I ↔ O
    where
        inv : transition (invariant-exit)
                ⊂ invariant;
        dec : transition ∈ inv(strict well_
                founded[invariant-exit]);
    then
        ℓ ≙ cons solution[I,O] with
              problem ≙ invariant ∩ exit;
              program ≙ proj2
                o limit (transition) o init2
            proof
              theorem "converge", section 2.6
            end
    given
        invariant ≙ problem (start);
        transition ≙ proj1[I,O] & program (body);
        initialization ≙ program (start);
        init2 ≙ id[I] & initialization
    end
```

The apparent complexity of the last three lines (relational operators in the definitions of *transition* and *init2*) stems from the fact that *transition* cannot just be *program(body)* i.e. a function from $I \times O$ into O, since we need its limit : it must be made into a function $I \times O \to I \times O$ which does not modify I component, the input data. In like manner, *init2* is the $I \times O \to I \times O$ version of *initialization*.

The constructive interpretation of this rule is that it may be useful to try and express the goal as the conjunction of two relations *invariant* and *exit*, and solve the new problems thus obtained in a quite dissymetric way : no strategy is implied for obtaining *invariant* (i.e. the initialization), whereas *exit* is to be reached by a fixpoint method keeping *invariant* true right up to the end.

3.5 A few remarks

Remark 1

The above functional definition corresponds to a programming language construct of the following form (in PASCAL notation) :

```
var i : I {input},
    o : O {output};
.....................
o := initialization(i);{invariant (i,o) is
      true}
while not exit (i,o) do
  o := transition (i,o);
{invariant (i,o) and exit(i,o)}
```

In view of the key role played by the initialization in any loop, as evidenced by the above Z

model, it seems regrettable that initialization is not syntactically part of the loop in common programming languages. It is well-known that omission of the initialization part is both a frequent and a serious programming error. It may thus seem advisable to include it in the syntax for loops, giving something like :

```
from
    <initialization part>
until
    <exit condition>
keeping
    <invariant assertion>
loop
    <transition>
end;
```

The "*keeping* ..." clause should be optional since one cannot expect all programmers to use formal methods.

Remark 2

It should be noted that a given transition function is usually shown to be a strict well-founded order not by working on the function itself, but by introducing an auxiliary function, say v, from $I \times O$ into V, where V is strictly well founded by a relation $op(<)$, v being such that

```
forall io for io : invariant-exit then
        v (transition (io)) < v(io)
end
```

v is called the variant function [5]. V is often the set *NAT* of natural numbers, with the usual order relation; however, other choices may be more practical in some cases, e.g. for programs which manipulate date structures. Note that existence of a computable variant function v ranging over *NAT* is not only a sufficient, but also necessary condition for convergence : if a loop converges, then $v(io)$ can be constructed as the number of iterations until the limit is reached, which is computable since it can be determined by running the loop with a counter, a process which terminates by hypothesis. Existence of a variant is of course undecidable.

Remark 3

There is another way of describing loops, namely to use well-founded order relations (antisymmetric and reflexive) instead of strict well-founded ones. If f is a function $X \to X$ whose inverse is well-founded, then a theorem equivalent to *converge* (section 2.6) asserts that f has a fixpoint in X and, more precisely, that any sequence $x_0 \in X, \ldots, x_{n+1} = f(x_n)$, converges after a finite number of steps. In the new model, *exit* is not needed any more; the hypothesis is that *transition* is a well-founded order over *invariant*, and the value computed by the loop is the projection on O of the fixpoint of *transition* obtained for any input i from the sequence beginning with $x_0 = (i, start(i))$.

This second model precisely corresponds to the structure advocated by Schwartz [11]:

```
(while) body
```

where it is understood that execution of the loop terminates as soon as *body* has ceased to change any component of the program state.

It is easily seen that the two formulations are indeed equivalent. In our first model, if we define function *trans2* such that

```
trans2 (io) = if exit (io) then io
              else transition (io)
```

then, *transition* being a strict well-founded order over *invariant-exit*, *trans2* will be a well-founded order, and the "fixpoints" it computes are the "limits" computed by *transition*. Conversely, *trans2* being a well-founded order over *invariant*, if we define *exit* as its set of fixpoints, then *transition* $\triangleq$ *trans2*$\lceil$(*invariant-exit*) is the required strict well-founded order.

Both points of view are useful. The "strict" one shows the decomposition of the goal into two parts, *invariant* $\cap$ *exit*, which is in close relationship with the usual concept of *while* loop, and useful in our constructive approach. The other one, which can of course be formalized in Z along the same lines, has the advantage of embodying the important and suggestive concept of fixpoint.

3.6 Comparison with other approaches

The "weakest precondition" of Dijkstra [5] may be described as follows :

```
wp[I,O] ≙ func program, problem → A, for
                program : I → O;
                problem : I ↔ O;
                A       : subset (I)
          then
              A ≙ set i for i : I where
                      problem (i ↔ program(i))
              end
          end
```

The emphasis is thus put on finding the largest possible domain where the object (*problem*, *program*) will be a "solution" in our sense. Indeed, since it is well known from computation theory that many relations of interest are partial, such an approach is quite justified. In a practical constructive approach, however, we feel that it is reasonable to be satisfied with total relations; i.e. knowing *problem* $\in I \leftrightarrow O$ such that *domain*(*problem*) = I, we set out to find (total) *program* such that *program* $\in I \to O$ and *program* $\subset$ *problem*.

Another well-known approach is Scott's (e.g.[12]), which also uses fixpoints as a basic concept. These are fixpoints of functionals, however, not of functions (i.e. the fixpoints are functions instead of elements in the domains of discourse). The whole theory is thus one level of abstraction higher than the one presented here.

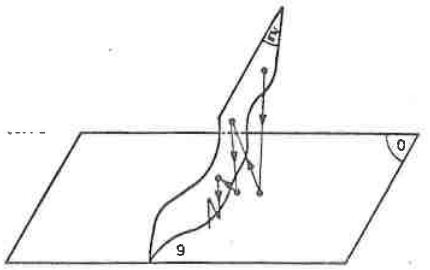
4. STRATEGIES FOR PROGRAM CONSTRUCTION AND EXAMPLES

4.1 Embedding, constant relaxation and uncoupling

One of the lessons we draw from the previous section is that loops may be considered as a program construction strategy whereby the goal is expressed as the conjunction of an "invariant" which is easier to establish than the goal itself, and an "exit" condition, in such a way that a "transition" function can be found, which will not destroy the validity of the invariant while decreasing a "variant" function so long as the exit condition is not satisfied.

It is easy to find many prototypical examples which fit nicely into this framework. For instance the "simplex" algorithm is nothing else [4] than keeping a certain point on the edges of a convex polyhedron (the invariant relation) while minimizing a cost function (the variant). Note that the latter may not be decreasing everywhere, which is a well-known theoretical problem in linear programming. On the other hand, so-called "relaxation methods" in numerical analysis vary the shape of a certain surface which is kept isomorphic to itself (invariant) and gradually decrease its energy function (variant).

A basic problem of program construction is : how do we weaken the goals to get the invariants ? The general method may be called embedding : use a larger domain D, a subset of which is isomorphic to the output set O. Find a relation *inv* defined on D which is easier to establish on D than the goal g is on O, but such that *inv* implies g on O. Then use *inv* as invariant, and membership in O as the exit condition. The method can be visualized a succession of "frog leaps" between the "surface" *inv* and the "hyperplane" O in D (see figure).



Many algorithms are direct applications of embedding. For example, *for* loops, such as operations on matrices, usually set out to solve a problem on the set of (n,n) matrices by embedding it in the set of (i,i) matrices for $0 \le i \le n$. Initialization is usually trivial; transition adds one to the dimension.

A particular case is "constant relaxation" : replace the goal $P(n)$, where n is a constant belonging to some set X, by $P(i)$ *and* $i = n$, where i is a variable constrained to range over X . This is also typical of *for* loops : to compute $s = \sum_{k=1}^{n} a[k]$, we replace this goal by

$s = \sum_{k=1}^{i} a[k]$ *and* $i = n$ and let i range from 0 to n.

Another closely related heuristics in uncoupling, which applies to a goal of the form $P(\ldots, i, \ldots, i, \ldots)$ where i appears twice, replacing it by $P(\ldots, i, \ldots, j, \ldots)$ *and* $i = j$. We saw a simple example of this method in section 1.

It is surprising to see how many loop algorithms may be recognized as instances of the latter two variants of embedding. In fact, it proves quite hard to find loop algorithms which escape these categories - which is rather disappointing when one has embarked upon a tentative classification of useful heuristics. However, these two are useful without any doubt, and we shall now conclude by analysing the way they apply to four examples : binary search, array partitioning, the QR algorithm for computing matrix eigenvalues, and an original algorithm, selective Cholesky factorization.

4.2 A Searching Example

Our first example, binary search, is a classical algorithm. It is well known, however, that writing it without errors is not that easy; as Knuth puts it [9, p. 407] , "although the basic idea of binary search is comparatively straightforward, the details can be somewhat tricky, and many good programmers have done it wrong the first few times they tried". It may thus be worthwile to construct the algorithm systematically, emphasizing the few places where creative intuition is actually needed.

Assume we have an array t of type X and size n (arrays may be defined in Z as functions from the subset $(\{1,2,\ldots n\}$ of *NAT* into X) and an element x also of type X. t is sorted (this is also easy to define) for a total order relation $\le$ on X. The first thing to do is to define the relation which the desired output i(a *NAT*) is required to satisfy. This relation, an instance of *problem*, may be expressed as follows :

```
search[X] ≙ rel t, x ↔ i for
              t : array[X]; x : X; i : NAT
            where
              s : sorted (t, op(≤));
              v : 0 ≤ i ≤ n;
              p : forall k for k : 1..n then
                    l: k ≤ i => t(i) ≤ x;
                    r: i < k => t(i) > x
                  end
            given
              n ≙ size(t)
            end
```

Note how careful one has to be when defining this relation if *search* is to be total (i should always exist) and single-valued. Change of a relational operator will destroy the specification. One may suspect that the main source of trouble with this problem lies therein.

Quite naturally, the problem lends itself to "uncoupling". We may replace *search* by the new relation

```
rel t,x ↔ i,j for
        t : array[X];
        x : X;
        i,j : NAT
where
        s : sorted(t, op(≤));                    } inv :
        v : 0 ≤ i ≤ j ≤ n;                       } s and
        q : forall k for k : 1..n then           } v and
              ℓ : k ≤ i => t(k) ≤ x;             } q
              r : j ≤ k => t(k) > x
    end;
        e : i = j
given
        n ≙ size(t)
end
```

and the corresponding strategy is, if we call *inv* everything above but $i = j$:

```
establish inv;
while i ≠ j do
        get j closer to i while maintaining inv;
{inv and i = j}
```

establish inv may be $i := 0$; $j := n$. To *get j closer to i*, many choices are possible; the idea of binary search is to look at the mid-point m of i and j to see whether it may be assigned to i or j (note the relationship with Newton's algorithm in 1.2). So we try the following :

```
{inv is satisfied and i ≠ j}
m := ⌈(i+j)/2⌉; {note the ceiling function :
                 1 ≤ m ≤ n}
"change i and j so that inv remains true"
```

Since t is sorted, the following conserves *inv*:

```
if t(m) ≤ x then
        {forall k where 1 ≤ k ≤ m then t(k) ≤ x}
        i := m
else
        {t(m) > x, thus forall k where m < k ≤ n
             then t(k) > x}
        j := m
```

However, this alternative assignment is not a correct solution since it fails to actually *get j closer to i* when $j = i+1$ *and* $t(j) > x$; in other words the variant $j-i$ does not decrease and the algorithm would not terminate. We note that the pessimistic assignment $j := m$ may be replaced by $j := m-1$, since actually $t(m) > x$ also implies

```
{forall k where m-1 < k ≤ n then t(k) > x}
```

and we get the correct binary search algorithm:

```
i := 0 ; j := n ;
while i ≠ j do {invariant : inv; variant :j-i}
    begin
    m := ⌈(i+j)/2⌉;
    if t(m) ≤ x then i := m else j := m-1
    end
{((i > 0 and t(i) = x) or x does not appear in t}
```

If the reader likes floors better than ceilings, he may build a symmetric algorithm along the same lines (see e.g. [10]).

4.3 Array partitioning

A straightforward case of uncoupling is Hoare's method for partitioning arrays, as used in Quicksort. If we use the first element as the pivot [13], the problem is to establish for some s in $i..j$:

```
forall k for k : i + 1 .. j then
        k ≤ s => a(k) ≤ a(i)
        s+1 ≤ k => a(k) ≥ a(i)
end
```

Uncoupling the two clauses with respect to s, i.e. replacing s by t in the second one to get the invariant, will yield an algorithm schema of the form

```
s := i ; t := j ; {invariant is true}
while s ≠ t do
        "get s closer to t, maintaining the
           invariant"
```

To represent the quoted statement, the partitioning method moves s and t towards each other, then restores the invariant :

```
begin
while s ≠ t and a(s) ≤ a(i) do
        s := s+1 ;
while t ≠ s and a(t) ≥ a(i) do
        t := t-1 ;
{t = s or (a(s) > a(i) and a(t) < a(i))}
exchange elements a(s) and a(t)
end
```

A variant is the algorithm for the "Dutch National flag" problem [5].

4.4 The QR algorithm for computing matrix eigenvalues

We turn now to a quite difficult numerical algorithm. Assume we wish to compute the eigenvalues of a matrix a. A possible course of action is based on the following two properties of eigenvalues :

1. The eigenvalues of a and b are the same if a and b are similar matrices.
2. Eigenvalues are particularly easy to compute for some classes of "good" matrices, e.g. orthogonal and triangular ones.

Algorithms which for given a compute a "good" b similar to a will thus yield the solution. The subproblem may be expressed as finding b and s (the similarity matrix) such that b is "good", s is regular, and

$$b = s^{-1}\, a\, s$$

If we write this as $sb = as$, uncoupling with respect to s is once again very tempting. Knowing that a relatively simple algorithm is known for factoring (i.e. for given m find regular s, and "good" r, such that $sr = m$), we are led to an algorithm of the following form :

```
s := 1 ; t := 1 ; b := 1 ;
while sb ≠ at do
    begin
    (s,b) := factoring (at); *;
    t := s
    end
```

(Note that here $s = t$ is the invariant and $sb = at$ the goal. The reverse choice would also work). Now if we define, in location marked *, q as $t^{-1}s$, we recognize an efficient algorithm known as QR or LR depending on the class of "good" matrices chosen (resp. orthogonal or triangular) [14]. This algorithm computes :

$$q_0 r_0 \triangleq a$$
$$q_1 r_1 \triangleq r_0 q_0$$
$$\cdots$$
$$q_i r_i \triangleq r_{i-1} q_{i-1}$$

which converges towards a pair (q_i, r_i) where r_i and q_i are "good" matrices and $r_i q_i$ is similar to a.

Of course the method shown only yields an algorithm schema; a proof of convergence, which is mathematically far from trivial, is required. It looks remarkable, however, that such a "technical" algorithm may be obtained through the application of very general rules.

4.5 Selective Cholesky Factorization

The algorithms presented above are not new. The question may thus be asked : may the techniques introduced in this paper be used not only to provide a better understanding of known algorithms, but to discover original ones ? A numerical algorithm which was actually discovered and developed using these techniques has been presented elsewhere [2]; it is applied to some kinds of finite-element problems. It is not possible to describe the whole program construction process here ; let us just mention that [2] contains a series of "constant relaxations" and "uncouplings" which yield a FORTRAN program, starting from the following problem specification : given matrix a of order n, such that indices are split into "external" (E) and "internal" ones, find s of order n such that

```
forall j,i for j,i : NAT where 1 ≤ j ≤ i ≤ n then
        j ∉ E => Σ'_{k≤j} s_ik s_jk = a_ij
        j ∈ E and i ∈ E => Σ'_{k≤i} s_ik s_jk = a_ij
        j ∈ E and i ∈ E => Σ'_{k≤n} s_ik s_jk + s_ij = a_ik
end
```

where Σ' denotes a sum where external indices are omitted.

5. CONCLUSION

We hope to have shown that basic programming concepts such as control structures may be described in a simple way, using no particular mathematical apparatus other than well-known notions such as sets, relations, functions, partitions, orders, etc.; that this can be done in a clear and persuasive way thanks to the use of a rigorous yet readable formalism, namely Z; and that such a description paves the way for expression of powerful mechanisms which are basic in the design of algorithms. As was mentioned before, we do not mean to imply in any way that programs can be invented through application of recipes of any kind. The rules presented here do however provide much insight, as it seems to us, into the structure of programs; they should be part of any set of rules used in work toward program synthesis. These methods, as well as the general formalizing approach presented here, have proved helpful both in teaching programming, and in looking for new algorithms.

ACKNOWLEDGEMENT

Many ideas come from numerous discussions with J.R. Abrial and A. Bossavit.

REFERENCES

[1] J.R. Abrial, S.A. Schuman and B. Meyer, *Specification Language;* Proceedings of School on Program Construction, Belfast; Cambridge : Cambridge University Press, 1980.

[2] A. Bossavit and B. Meyer, *On the Constructive Approach to Programming : The Case for Partial Cholesky Fastorization (A Tool for Static Condensation)*; in Advances in Computer Methods for Partial differential Equations III, Vichnevetsky and Stepleman (Eds.), IMACS, 1979.

[3] N. Bourbaki, *Eléments de Mathématiques - Théorie des Ensembles;* Paris : Hermann, 1970.

[4] G.B. Dantzig, *Linear Programming and Extensions;* Princeton (N.J.) : Princeton University Press, 1963.

[5] E.W. Dijkstra, *A Discipline of Programming ;* Englewood Cliffs (N.J.) : Prentice-Hall, 1976.

[6] R.W. Floyd, *Assigning Meaning to Programs;* Proc. Sym. in Applied Mathematics 19, Schwartz J.T. (Ed.), American Mathematical Society, pp. 19-32.

[7] C.A.R. Hoare, *An Axiomatic Basis for Computer Programming;* Communications ACM, 12, 10, pp. 576-583.

[8] J.D. Ichbiah et al, *Preliminary ADA Reference Manual; Rationale for the Design of the ADA Programming Language;* SIGPLAN Notices, 14, 6, Parts A and B.

[9] D.E. Knuth, *The Art of Computer Programming,* Vol. 3, *Sorting and Searching;* Reading (Mass.) : Addison-Wesley, 1973.

[10] B. Meyer and C. Baudoin, *Méthodes de programmation*; Paris : Eyrolles, 1978.

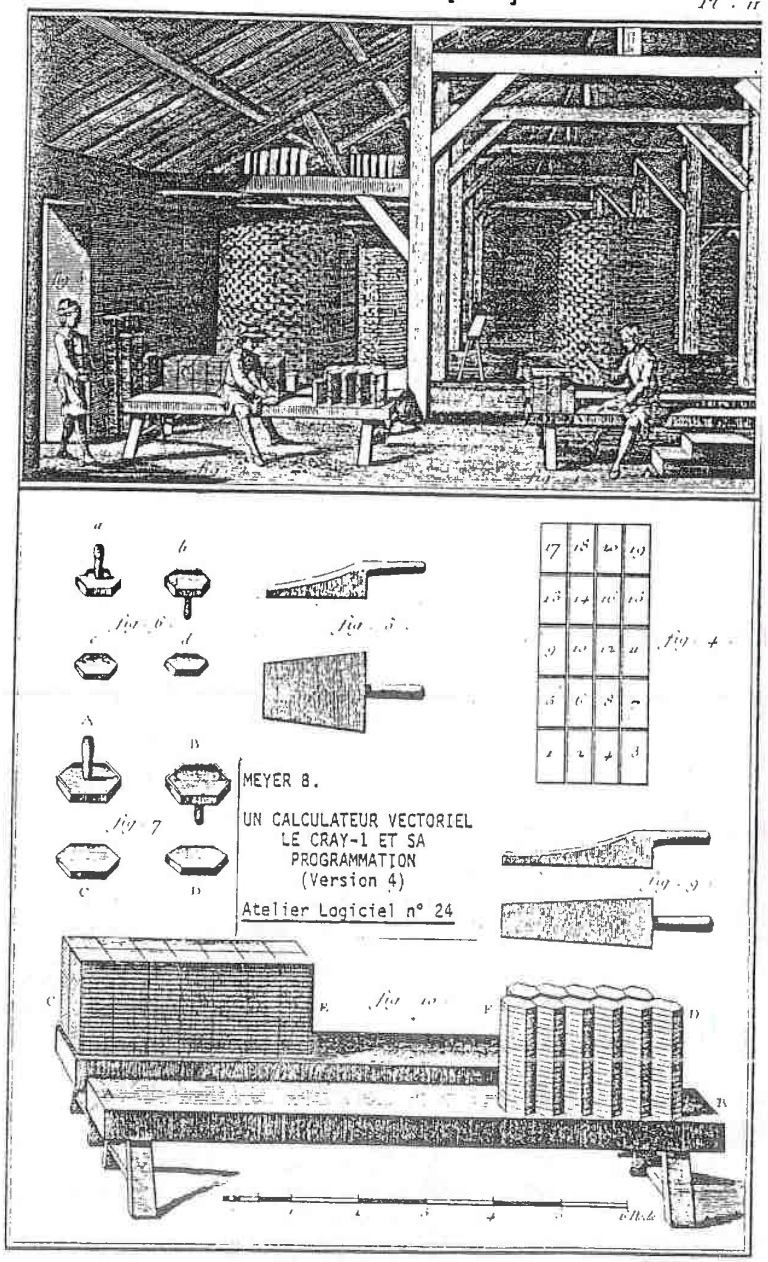
[11] J.T. Schwartz, *Program Genesis and the Design of Programming Languages;* in Current Trends in Programming Methodology, Vol. IV, *Data Structuring*, Yeh (Ed.), pp. 185-215; Englewood Cliffs (N.J.) : Prentice-Hall, 1978.

[12] D.S. Scott, *The Lattice of Flow Diagrams;* in Symposium on Semantics of Programming Languages, Engeler (Ed.), Lecture Notes in Mathematics, pp. 311-366; Berlin : Heidelberg : New-York : Springer-Verlag, 1971.

[13] R.S. Sedgewick, *Quicksort;* Ph. D. Thesis, Stanford University, 1975.

[14] J.H. Wilkinson and C. Reinsch, *Linear Algebra (Handbook for Numerical Computation, vol. 2)*; Berlin : Heidelberg : New York : Springer-Verlag, 1971.

ATELIER LOGICIEL



DER
IMA

NOTE SUR LA VERSION 4

La préparation des versions 3 et 4 de cette note a permis :

- de corriger quelques erreurs (en particulier, la définition des "éléments réguliers" au § V.4 était inexacte);

- de prendre en compte l'évolution du logiciel et des publications Cray, en particulier l'arrivée de la version 1.09 du compilateur Fortran avec les possibilités de pseudo-vectorisation des opérations de réduction (cf. V.5.3).

Elle a bénéficié de nombreuses conversations avec des responsables de la société Cray à Mendota Heights, Chippewa Falls et Clamart. Je remercie en particulier L. Higbie, R. Nelson, I. Qualters et D. Robbe.

Signalons qu'un document audiovisuel (bande vidéo) d'une demi-heure a été réalisé à partir de cette note et sous le même titre. Il présente la machine et sa programmation sous une forme imagée. Pour tout renseignement, s'adresser à E. de Drouas, ou à l'auteur.

Depuis sa première parution, cette note a été complétée par de nombreux autres documents EDF sur le Cray (langage de commande, vectorisation, etc.). On se reportera aux références signalées par un astérisque dans la bibliographie.

I INTRODUCTION

Le Cray-1, dont un exemplaire, exploité en commun par EDF et CISI, est disponible au Centre de Calcul des Etudes et Recherches d'EDF à Clamart, est un ordinateur destiné au calcul scientifique. Conçu grâce à des techniques de pointe en matière d'architecture et de construction de machines, il permet en particulier le traitement efficace de séries de données ou *vecteurs*. Selon les représentants de la firme qui l'a conçu, il s'agit, même en mode scalaire, du "plus rapide calculateur aujourd'hui disponible" [Dungworth 79].

Dans cette note de présentation, nous décrivons la place d'un calculateur de ce type dans un centre de calcul (section II); nous introduisons ensuite brièvement les concepts les plus originaux de l'architecture du Cray-1, pour autant qu'ils importent aux programmeurs (section III); enfin, après avoir indiqué les précautions méthodologiques nécessaires (section IV), nous discutons les méthodes permettant de programmer en Fortran de façon à tirer au mieux parti des possibilités de *vectorisation* offertes par cette machine (section V). En annexe, sont fournies une bibliographie, la récapitulation des caractéristiques de base de la machine, de celles du Fortran proposé, les cartes de contrôle nécessaires, des définitions mathématiques de quelques concepts délicats introduits à la section V, et la liste des règles proposées dans le texte.

Un mot de précaution : la présente note tente une synthèse entre les aspects matériels et logiciels du Cray-1; on a tenté de relier les règles de la programmation efficace aux contraintes du calcul vectoriel, bien qu'aucun document synthétique n'existât chez le constructeur. Il a donc fallu interpréter, au risque de se tromper.

La section IV (vectorisation) a bénéficié de nombreuses suggestions d'E. de Drouas, qui est par ailleurs l'auteur de l'annexe D (cartes de contrôle).

II PLACE D'UN CRAY-1 DANS UN CENTRE DE CALCUL

Un ordinateur très puissant (on parle souvent de "super-ordinateur") tel que le Cray-1 vise un type de traitement bien précis : le calcul scientifique lourd, vectoriel en particulier; il est très efficace pour cette application, mais non pour l'ensemble des tâches d'"intendance" auxquelles les calculateurs consacrent ordinairement une bonne partie de leur puissance (dialogue, communication, gestion de ressources, manipulations complexes de fichiers, etc.).

Aussi le Cray-1 n'est-il pas destiné à constituer le "noyau" d'un grand centre de calcul, en remplacement d'un ordinateur plus classique comme un IBM 3081. Sa place normale (figure 1) est celle d'un calculateur arrière *(back-end processor)* venant "épauler" un tel ordinateur, déjà présent auparavant, et qui sera désormais considéré comme frontal *(front-end)*.

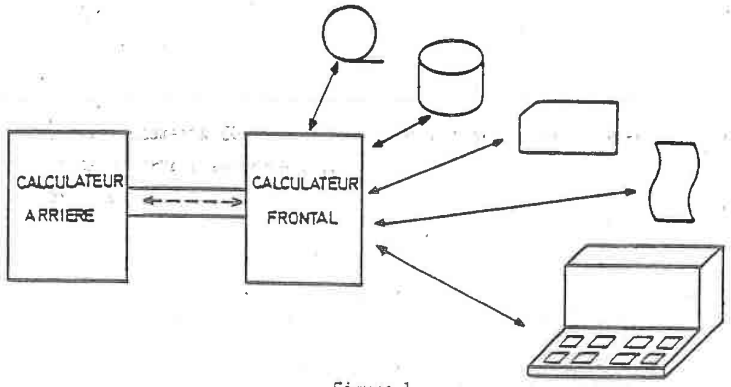


Figure 1
Place d'un super-ordinateur dans un centre de calcul

Le calculateur frontal continuera comme par le passé d'assurer l'interface avec les utilisateurs, l'entrée et la sortie des travaux, les contrôles de routine, la gestion des périphériques, etc. A ces tâches s'ajoutera désormais celle de filtrer les travaux : le calculateur frontal

en gardera une partie par-devers lui, pour les traiter comme par le passé; les autres seront transmis pour exécution au calculateur arrière.

Trois séries de conditions sont nécessaires pour qu'un travail puisse être transmis au calculateur arrière.

1. Le travail doit en comporter la *demande* explicite, grâce à un code compris par le frontal.

2. Les caractéristiques du travail doivent être telles que les *règles d'exploitation* du centre de calcul permettent de l'affecter au calculateur arrière.

3. Les *ressources* logicielles et matérielles demandées pour l'exécution du travail doivent être disponibles sur le calculateur arrière; elles incluent en particulier :

- les périphériques;
- les sous-programmes de bibliothèques;
- les langages de programmation et leurs compilateurs.

Au Centre de Calcul des Etudes et Recherches, le Cray-1 est accessible à partir de deux machines frontales : l'IBM 3081 et le concentrateur CII-HB 66, qui sert d'interface avec le réseau RETINA (cf. [Glaziou 81]).

III LE MATERIEL : CARACTERISTIQUES ORIGINALES DU CRAY-1

III.1. - RIEN NE SERT DE COURIR : IL FAUT PARTIR ENSEMBLE

Pour construire des ordinateurs permettant d'exécuter les programmes de plus en plus vite, deux voies sont possibles :

1. Augmenter la vitesse de base avec laquelle les circuits électroniques exécutent les instructions.

2. Augmenter le degré de parallélisme, c'est-à-dire chercher à exécuter de plus en plus d'opérations simultanément.

La voie 1 a conduit à des résultats spectaculaires (c'est peu dire) dans le passé. Le Cray-1 peut effectuer une série d'opérations arithmétiques en 12,5 nanosecondes chacune (1 ns = 10^{-9} s), c'est-à-dire *deux millions de fois plus vite* que l'ENIAC de 1946 (cf. [Moreau 81]). Les techniques de miniaturisation et d'intégration (LSI, VLSI) permettent d'aller toujours plus loin dans cette voie. Il faut reconnaître cependant que l'on approche aujourd'hui des limites absolues, liées à la vitesse de la lumière; c'est d'ailleurs cette constatation qui a amené les concepteurs du Cray-1 à donner à leur machine des dimensions si étonnamment réduites (6,6 m^2 au sol, pour un poids d'environ 5 tonnes, cf. figure 2), le but poursuivi étant de minimiser la longueur des connexions, donc la vitesse de transmission. La simplicité de la construction (trois types de "puces" électroniques seulement étaient utilisés à l'origine, complétés ultérieurement par deux autres) joue également un rôle important.

Au point où en est aujourd'hui la technique, chaque nanoseconde épargnée par la voie 1 est chèrement gagnée. Il reste donc la voie 2, qu'on peut décrire par le principe suivant :

> PRINCIPE DE PARALLELISME
>
> *Pour faire plus de choses en moins de temps, faisons plusieurs choses en même temps.*

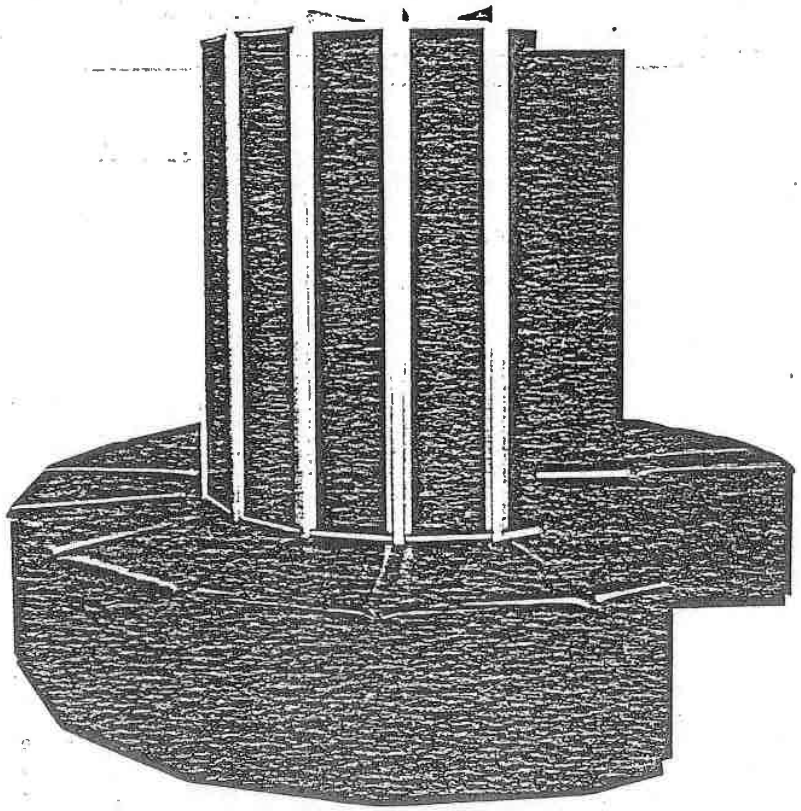


Figure 2 : Un Cray-1

Cette méthode est appliquée depuis longtemps, par exemple, aux systèmes d'exploitation, qui permettent d'affecter l'unité centrale à un programme pendant qu'un autre effectue des entrées ou des sorties, afin de ne pas perdre de temps en attente. Une autre application du principe de parallélisme est constituée par les réseaux et les systèmes distribués, par exemple les réseaux de micro-processeurs, permettant de répartir une tâche entre plusieurs calculateurs. Ce type de structures, où des processus autonomes se synchronisent à des instants initialement inconnus, est étudié, entre autres, dans [Brinch Hansen 73] et [Hoare 78], avec une bonne introduction dans [Brinch Hansen 79].

Cette variété de parallélisme n'est pas celle qui fait l'originalité des "super-ordinateurs" comme le Cray-1 - encore qu'ils tirent parti, comme tous les calculateurs, des techniques classiques de désynchronisation du calcul et des échanges. Sur ces machines, tout au moins au niveau auquel elles se présentent à leurs utilisateurs, la commande du processus de calcul reste centralisée, mais le matériel offre la possibilité de *commencer une opération avant que la précédente soit terminée*. Nous constaterons deux variantes de cette technique de "parallélisme contrôlé" (figure 3) :

- le *découplage* : cas où les opérations considérées sont différentes, et peuvent être confiées à des dispositifs disjoints;

- la *segmentation* : cas où les opérations sont les mêmes, et confiées au même dispositif, mais s'effectuent en n étapes indépendantes, le matériel pouvant exécuter simultanément des étapes distinctes d'opérations successives. On obtient alors, à condition que l'alimentation en opérations soit continue, un effet de parallélisme apparent - mais seulement en régime stationnaire, après un *temps d'amorçage* nécessaire à la production du premier résultat (n étapes)[1].

(1) Le terme de "segmentation" se trouve dans les publications Cray (à propos de la segmentation des unités fonctionnelles). "Découplage" est introduit par l'auteur.

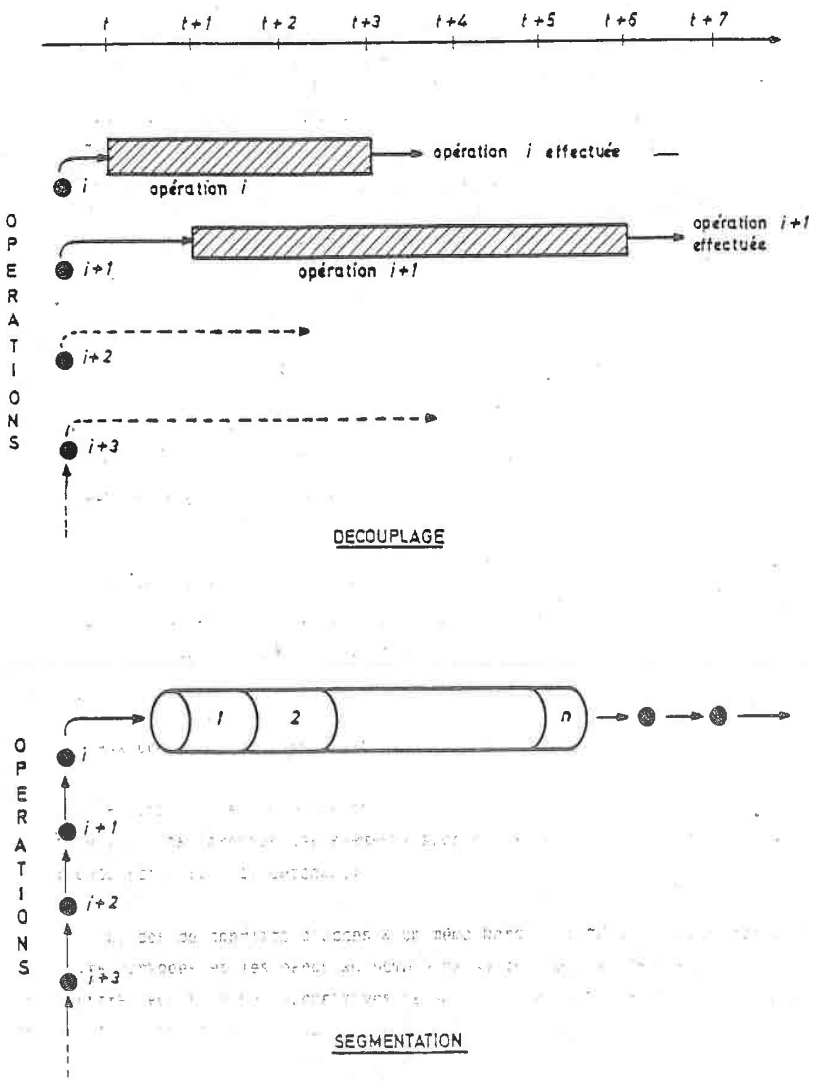


Figure 3

Découplage et segmentation

Un cas particulier de segmentation privilégié par les "super-ordinateurs" est l'application d'une *même instruction* à des données successives différentes. C'est ce qu'on nomme le calcul vectoriel, étudié ci-après en III.7. Cette forme de parallélisme est souvent appelée SIMD (*Single Instruction, Multiple Data* : instruction unique, données multiples), par opposition à des structures plus complexes (MIMD).

Nous étudions dans la suite de cette section comment le principe de parallélisme s'applique aux différents éléments de la conception du Cray-1 : la mémoire (III.2); les unités fonctionnelles, qui effectuent les calculs (III.3); les tampons d'instruction, qui abritent les instructions prêtes à être exécutées (III.4); les registres, qui reçoivent opérandes et résultats (III.5); les entrées et les sorties (III.6); les techniques de calcul vectoriel (III.7). On récapitulera en III.8 la signification du principe de parallélisme pour le programmeur.

PERIODE DE L'HORLOGE

Dans tous les cas, le *temps de référence* auquel le Cray-1 cherche, par application du principe de parallélisme, à ramener des opérations plus longues si le matériel les effectue séquentiellement, est la période de l'horloge, temps de base de l'ordinateur.

Une période d'horloge = 12,5 nanosecondes.

III.2. - LA MEMOIRE

La mémoire centrale du Cray-1 comprend, en configuration maximale, quatre mégamots (4 194 304 mots de 64 bits)*. Le temps d'accès à un mot est normalement de quatre périodes d'horloge (50 ns); le temps de transfert de mémoire à registre est d'onze périodes (137,5 ns).

L'influence de ce temps de transfert est, en régime stationnaire, tempérée par la segmentation : on peut lancer un accès à la mémoire avant la fin du transfert précédent.

Pour diminuer le temps d'accès observé, on fait naturellement appel au découplage : la mémoire est divisée en seize sections indépendantes appelées *bancs*; les adresses, prises consécutivement, recouvrent cycliquement les seize bancs** (figure 4). Le temps d'accès de 4 périodes s'applique à des accès successifs à un même banc, mais il n'y a pas de contrainte pour des accès à des bancs différents (par exemple pour des adresses consécutives).

En situation optimale, donc, la mémoire pourra transmettre aux registres, ou en recevoir, *une donnée par période d'horloge* en régime stationnaire, après un temps d'amorçage de quinze périodes (4 + 11).

On notera que cette situation optimale ne pourra être obtenue que si les deux conditions suivantes sont réunies :

a) accès à des adresses prévisibles : on devra donc être en mode vectoriel, où les adresses des éléments successifs peuvent être calculées à l'avance (cf. III.7 ci-dessous);

b) pas de conflits d'accès à un même banc : le temps d'accès étant de quatre périodes et les bancs au nombre de seize, ceci exige que la différence entre deux adresses successives ne soit pas un multiple de 16 (accès se faisant alors toutes les quatre périodes)** ni un multiple impair de 8 (un accès toutes les deux périodes).

Ces points devront être pris en compte, dans la mesure du possible, par le programmeur soucieux d'utiliser au mieux les possibilités de calcul vectoriel (section IV).

---

* La mémoire disponible sur la machine de Clamart est actuellement (janvier 82) d'un mégamot (1 048 576 mots).

** La machine de Clamart ne possède actuellement (janvier 82) que huit bancs.

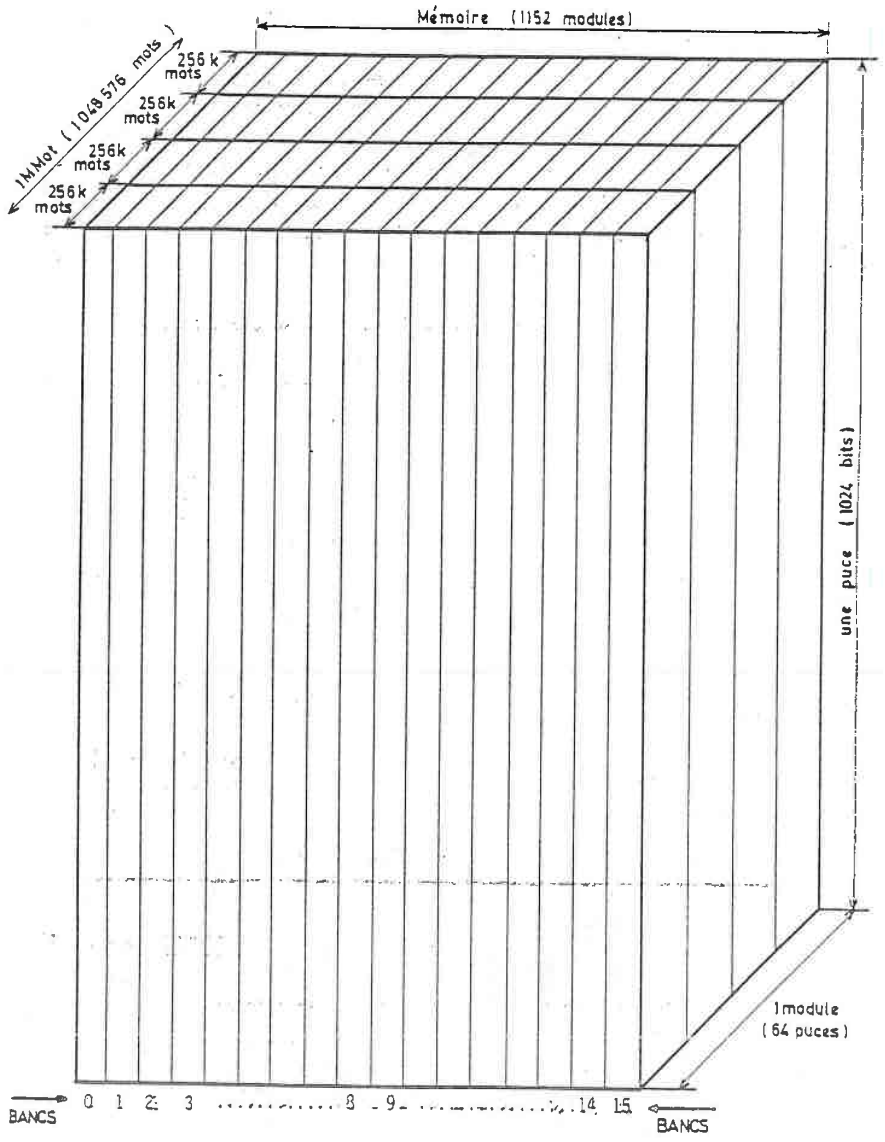


Figure 4

Organisation physique de la mémoire du Cray-1

III.3. - LES UNITES FONCTIONNELLES

Le calcul proprement dit est effectué sur le Cray-1 par douze *unités fonctionnelles*, spécialisées chacune en vue d'un type d'opérations (par exemple : opérations arithmétiques, logiques, etc.).

Chaque unité fonctionnelle requiert, pour exécuter une opération, un certain nombre, en général supérieur à un, de périodes d'horloge; il faut ainsi trois périodes pour une addition entière, sept pour une multiplication flottante.

Pour améliorer artificiellement ces temps, on fait là encore appel aux deux formes de parallélisme :

- *découplage*, c'est-à-dire fonctionnement simultané d'unités fonctionnelles différentes;

- *segmentation* de chaque unité fonctionnelle.

La première technique, appliquée par exemple aux deux opérations successives

$a \leftarrow b * c$; {*multiplication flottante*}

$m \leftarrow i + j$ {*addition entière*},

permet de lancer la seconde une période d'horloge après la première, et de laisser fonctionner concurremment les deux unités fonctionnelles utilisées. Ceci ne serait pas possible pour deux opérations s'adressant à la même unité spécialisée, ou partageant une variable.

La segmentation des unités fonctionnelles permet à chacune d'entre elles d'exécuter les instructions en un certain nombre n d'étapes, chaque étape durant exactement une période d'horloge, de telle façon que des étapes différentes d'instructions successives puissent se dérouler simultanément pendant une même période d'horloge (figure 5). Rappelons que, selon la définition de la segmentation, ces "instructions successives" sont en fait des exemplaires successifs de la même instruction, appliquée à une suite de données. Si ces données sont envoyées à l'unité fonctionnelle à la vitesse d'une par période

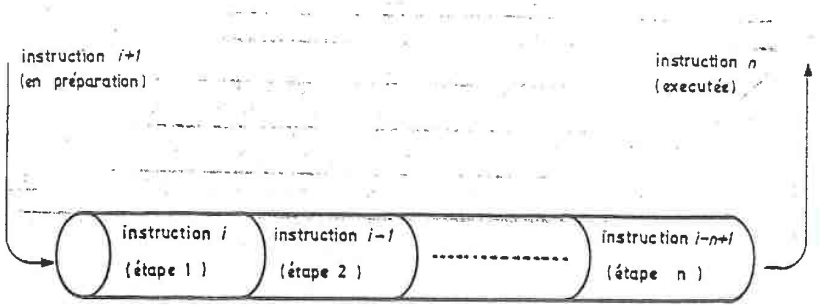


Figure 5

Segmentation d'une unité fonctionnelle

(cf. III.4 ci-dessous), l'unité produira donc également en régime stationnaire un résultat d'instruction par période, comme si son temps d'exécution avait été divisé par n.

Toutes les unités fonctionnelles du Cray-1 sont entièrement segmentées.

On trouvera à la figure 6 la liste des unités fonctionnelles, des types d'instructions qu'elles exécutent, et des temps correspondants. On notera que le Cray-1 ne possède pas d'instruction câblée de division flottante, mais seulement l'inversion approchée. Une division sera calculée par une inversion suivie d'une multiplication.

On notera la présence d'une unité vectorielle booléenne permettant de constituer des masques (vecteurs binaires), et d'extraire grâce à eux les éléments d'un vecteur vérifiant une certaine propriété. Un registre de masque *VM* est présent à cet effet (III.5 ci-après).

III.4. - LES TAMPONS D'INSTRUCTIONS

L'utilisation efficace du découplage et de la segmentation exige que l'arrivée des instructions ne soit retardée en aucune façon.

UNITES FONCTIONNELLES	TEMPS D'EXECUTION (en périodes d'horloge)
Unités fonctionnelles de calcul d'adresse (24 bits)	
Addition - soustraction	2
Multiplication	6
Unités fonctionnelles scalaires (mots de 64 bits)	
Opérations booléennes	1
Décalage	2 ou 3
Addition ou soustraction entière	3
Comptage des bits à 0 (ou 1)	4 ou 3
Unités fonctionnelles flottantes (mots de 64 bits)	
Addition ou soustraction	6
Multiplication	7
Inversion approchée	14
Unités fonctionnelles vectorielles (mots de 64 bits)	
Opérations booléennes (masquage et extraction)	2
Décalage	4
Addition ou soustraction entière	3

Figure 6

Unités fonctionnelles du Cray-1

Le chargement d'une instruction à partir de la mémoire requiert quatorze périodes d'horloge (il est vrai que les instructions placées aux adresses suivantes sont ensuite chargées au rythme de quatre mots par période). Pour éviter ce retard, quatre *tampons d'instruction* se trouvent dans l'unité de calcul; chacun d'eux a une capacité de 64 mots; le format des instructions étant de 16 ou 32 bits, c'est-à-dire un quart ou une moitié de mot, les quatre tampons peuvent donc abriter de 128 à 256 instructions.

Lorsque des instructions se trouvent dans les tampons d'instructions, leur exécution séquentielle est lancée au rythme d'une "parcelle" de 16 bits (instruction ou demi-instruction) par période. Un branchement à une autre instruction présente dans le tampon requiert deux périodes. Une référence à une série d'instructions non présente dans le tampon entraîne un retard initial de quatorze périodes puisqu'elle implique un accès à la mémoire.

Ce mode d'exécution favorise de toute évidence les *boucles courtes*, dont le corps, comportant peu d'instructions, tiendra dans 256 parcelles (les termes "boucle courte" et "boucle longue" font ici référence au nombre d'instructions contenues dans la boucle et non, bien sûr, au nombre d'itérations de la boucle à l'exécution). C'est en vertu de ce principe qu'on est amené à remplacer les boucles longues par plusieurs boucles courtes lorsque c'est possible (voir la *règle d'ouverture des boucles* en III.7 ci-après).

On notera l'analogie avec la notion d'espace de travail dans les systèmes d'exploitation à mémoire virtuelle (hiérarchie de ressources dont les plus rapides sont de capacité limitée).

III.5. - LES REGISTRES

Parmi les opérandes et le résultat d'une instruction, deux éléments au moins (sur trois en général) doivent se trouver dans l'un des *registres* de la machine. Les registres jouent en outre pour les données le même rôle que les tampons pour les instructions : accueillir des éléments à l'avance, de façon à éviter des accès abusifs à la mémoire.

Le Cray possède sept types de registres désignés par les lettres *A, B, S, T, V, VL* et *VM* (figure 7).

Type de registre	Longueur d'un registre de ce type	Nombre de registres de ce type	Rôle
A	24 bits	8	Adresses Index (pour l'adressage), compteurs de boucles, compteurs de décalages, contrôle des canaux (entrées et sorties).
B	24 bits	64	"Réservoir" pour les registres *A* transfert *B* ↔ *A* : une période; transfert *B* ↔ mémoire, par blocs : une période par mot.
S	64 bits	8	Source et destination des opérations scalaires (c'est-à-dire les opérations entières ou flottantes portant sur des mots).
T	64 bits	64	"Réservoir" pour les registres transfert *S* ↔ *T* : une période; transfert *B* ↔ mémoire, par blocs : une période par mot.
V	64 mots de 64 bits	8	Traitement de vecteurs (cf. III.6)
VL	7 bits	1	Longueur de vecteur
VM	64 bits	1	"Masque" binaire pour la sélection d'éléments d'un vecteur.

Figure 7
Registres du Cray-1

III.6. - ENTREES ET SORTIES

Les échanges du Cray sont effectués grâce à 24 calculateurs périphériques ou *canaux*, spécialisés par moitié en entrée ou en sortie, et répartis en quatre "groupes" de six, également spécialisés.

A chaque période d'horloge, on examine l'un des groupes, cycliquement, pour détecter une demande d'accès à la mémoire; s'il s'en trouve, elle n'est servie que quatre périodes plus tard (sauf si elle entre en conflit avec un accès à la mémoire demandé par l'unité de calcul; elle sera alors soumise à nouveau huit périodes plus tard). Les six canaux d'un groupe ont des priorités différentes.

Chaque canal peut donc émettre des demandes d'accès à la fréquence maximale d'une toutes les huit périodes d'horloge. Quatre périodes après une demande non satisfaite, cependant, on peut satisfaire une demande d'un canal moins prioritaire du même groupe, et servir les autres groupes dans les trois périodes suivantes.

On peut donc atteindre, pour les échanges, la vitesse de référence du Cray : un traitement par période d'horloge.

III.7. - LE CALCUL VECTORIEL

L'aptitude du Cray à faire du "calcul vectoriel" résulte pour l'essentiel de la combinaison de trois caractéristiques :

- la présence des huit registres vectoriels *V* (figure 7), de 64 mots chacun, pouvant donc contenir des séries de 64 entiers ou réels;

- l'accès rapide à la mémoire pour des adresses en progression régulière (III.2);

- la segmentation des unités fonctionnelles (III.3).

Le Cray-1 permet, grâce à ces propriétés, d'obtenir en régime stationnaire, dans les cas favorables, un traitement continu de données fournies à la vitesse d'une par période d'horloge. Ces données seront consommées par tranches de 64 éléments au plus.

Le fonctionnement continu permis par ces propriétés (figure 8) est connu sous le nom de mode "pipeline"; on peut proposer *bitoduc* en français [Bossavit 79]. La figure 8 représente le cas où les deux entrées *E1* et *E2* et la sortie *SO* sont vectorielles, c'est-à-dire effectuées sur des registres *V*. *E1* pourrait être aussi un registre scalaire du type *S* (exemple : multiplication d'un vecteur par un scalaire).

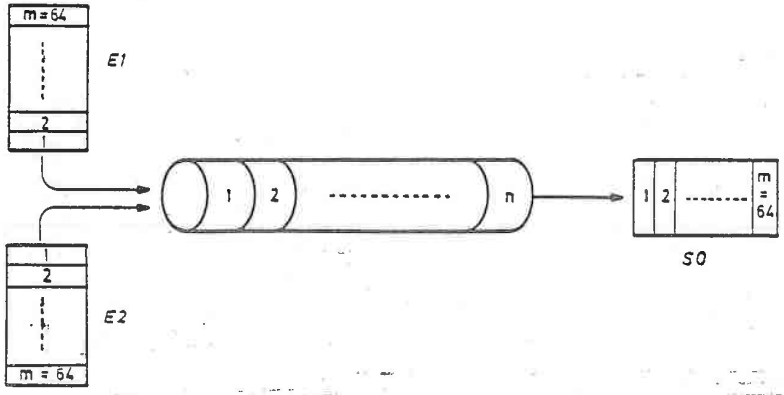


Figure 8

Fonctionnement d'un bitoduc

Le traitement en mode "bitoduc" permet, en régime stationnaire et dans les bons cas, la production d'un résultat par période d'horloge.

Le résultat ainsi obtenu peut à son tour alimenter, comme opérande, une autre unité fonctionnelle; c'est ce qu'on nomme le *chaînage* des opérations vectorielles (figure 9). On peut même créer des boucles en réinjectant la sortie d'un bitoduc à son entrée. Cette technique de chaînage récursif est utilisée pour améliorer l'efficacité de certaines opérations qui ne sont pas vectorisables au sens strict; c'est ce qu'on appelle la *pseudo-vectorisation*, appliquée en particulier par le compilateur Fortran à partir de sa version 1.09 (cf. ci-après V.5.3).

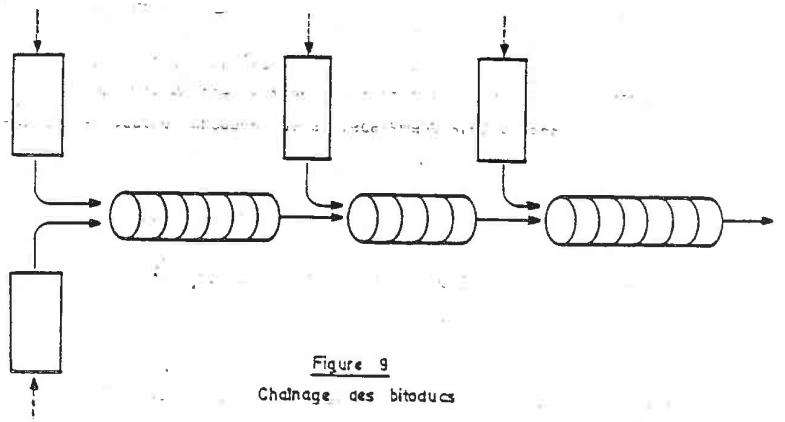


Figure 9

Chaînage des bitoducs

Le temps d'amorçage, plaie de tous les calculateurs vectoriels, donne la limite de taille des vecteurs au-dessous de laquelle le traitement vectoriel n'est pas avantageux. Il est clair que sur le Cray-1 cette limite est directement liée au nombre de périodes requis par chaque unité fonctionnelle (figure 6).

Six unités fonctionnelles sur douze peuvent être utilisées en liaison avec un registre *V*; il s'agit des six dernières mentionnées sur la figure 6 : les trois unités dites "flottantes" (accessibles également au traitement en mode scalaire), et les trois dites "vectorielles" (réservées au calcul vectoriel).

Lorsqu'une opération vectorielle est lancée sur une unité fonctionnelle, celle-ci est réservée jusqu'à la fin de l'opération, ce qui interdit jusque là toute autre opération sur la même unité.

Cette contrainte, jointe à celle qui exige qu'en mode vectoriel les instructions à exécuter soient en petit nombre, pour pouvoir rester dans les tampons d'instructions (III.4), entraîne le principe suivant :

> REGLE D'OUVERTURE DES BOUCLES
>
> Le calcul vectoriel préfère aux boucles longues les suites de boucles courtes.

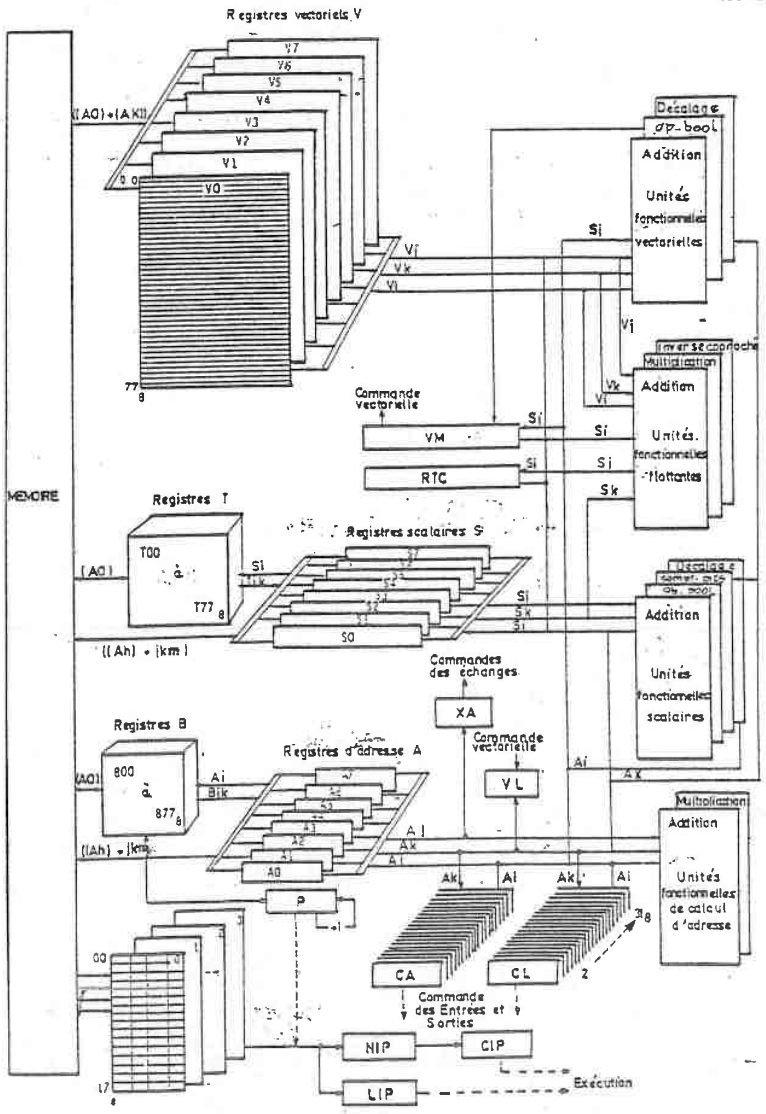


Figure 10

Schéma général de l'unité de calcul d'un Cray 1
(unité de calcul)

Le remplacement pourra le plus souvent être effectué par les compilateurs si l'on programme dans un langage de haut niveau. Sa possibilité entraîne cependant une contrainte assez rigoureuse pour la programmation vectorielle : la règle de non-dépendance croisée (V.6).

III.8. - EN GUISE DE SYNTHESE

Délaissant (enfin) les détails techniques, rassemblés sur le schéma d'organisation générale de l'unité de calcul du Cray-1 (figure 10), nous pouvons essayer de résumer ce que signifient pour le programmeur les structures qui viennent d'être décrites.

L'image mentale qui est apparue peu à peu à travers la description des différents éléments du Cray-1 est celle d'un réseau mythique de tuyauteries complexes, par lesquelles s'écoule un fluide étrange, transportant des instructions, des données et des résultats. L'équilibre idéal, toujours recherché, jamais atteint, est l'état où ce fluide parcourt tous les conduits avec une grâce égale et un débit constant - la période de l'horloge.

Les rouages de ce système parfait répètent à l'unisson, depuis l'aube des temps, la mécanique inhumaine rythmée par la période de l'horloge : avaler une donnée - cracher un résultat. Avaler-cracher. Avaler-cracher. Avaler - ...

La règle de diamant, pour qui veut s'approcher de ce Nirvaña, est de *prévoir* : pour alimenter continûment valves, soupapes et embouchures, il faut savoir à l'avance où chercher les éléments à venir. Ceci se traduit de deux façons :

- Pour les instructions, on cherchera des combinaisons aussi simples que possible : séries contiguës enchaînées, boucles courtes (on préférera deux boucles simples à une boucle complexe); éviter les branchements lointains, les appels de sous-programmes effectués dans le feu de l'action.

- Pour les données, on traitera des séries munies de structures très rigides - essentiellement des suites de données dont les adresses forment des progressions arithmétiques -, pour pouvoir appliquer les techniques de calcul vectoriel.

C'est à ce dernier aspect qu'est consacrée la suite de cette note, après une présentation d'ordre méthodologique.

IV AVANT DE PROGRAMMER VECTORIELLEMENT

IV.1. - QU'EST-CE QU'UN PROGRAMME VECTORIEL ?

Un élément de programme écrit dans un langage de haut niveau sera dit vectorisable si le compilateur peut le traduire sous une forme utilisant les dispositifs de calcul vectoriel offerts par la machine (III.7.). Les éléments non vectorisés seront traités en mode non vectoriel, dit "scalaire".

Nous étudierons à la section suivante (V) les conditions nécessaires et suffisantes pour qu'un élément de programme soit vectorisable. La poursuite de cet objectif peut entraîner, par ordre de difficulté croissante :

- l'adjonction de directives à l'intention du compilateur;
- des modifications locales des programmes;
- des changements d'algorithme.

Avant d'examiner en détail les techniques vectorielles, telles qu'elles découlent de la description précédente du matériel, il est important de bien comprendre la place de la vectorisation par rapport aux autres contraintes de la programmation.

IV.2. - POLITIQUE DE VECTORISATION

La présence d'un calculateur vectoriel ne fait pas disparaître les difficultés classiques de la programmation, et en soulève de nouvelles. A la première catégorie se rattachent la question de la validité des programmes et celle du choix des algorithmes; à la seconde, celle de la portabilité des programmes. Nous nous limiterons à les traiter par deux "règles" (règle de relativité, règle de portabilité). Ceci ne doit pas conduire à sous-estimer leur importance, qui ne peut que croître avec l'arrivée de calculateurs de plus en plus puissants.

REGLE DE RELATIVITE

a) L'aptitude au traitement vectoriel n'est qu'*un des éléments de l'efficacité* d'un programme. En particulier :

i) de nombreux programmes sont limités non par le calcul, mais par les échanges;

ii) la vectorisation n'a de sens que si l'algorithme utilisé est bon. Toute vectorisation produit par exemple un effet négligeable par rapport au remplacement d'un algorithme de tri en temps n^2 par un algorithme en temps $n \log n$.

b) L'amélioration de l'efficacité n'est intéressante que *relativement aux boucles les plus internes* des programmes, représentant souvent une faible proportion du code. On considère couramment que les programmes de type scientifique pur passent de 80 à 90% de leur temps dans quelques pourcent de leur texte.

c) L'efficacité n'est qu'*un des éléments de la qualité* d'un programme. La qualité la plus importante est la validité.

Voir aussi le chapitre VIII de [Meyer 78]. La compréhension de la règle de relativité devrait permettre d'éviter toute frénésie de vectorisation.

Lorsqu'un centre de calcul reçoit un calculateur vectoriel, qui coexistera avec un "frontal" (section II), il cesse d'être totalement homogène. Les programmeurs ne peuvent donc plus se permettre d'ignorer les problèmes de la portabilité. Aussi proposons-nous la règle suivante, au demeurant modérée.

REGLE DE PORTABILITE

Tous les programmes écrits en vue d'être vectorisés doivent pouvoir s'exécuter sur le calculteur frontal (non vectoriel), éventuellement au prix de modifications minimes.

Sur la programmation portable en Fortran, on consultera [APCOL 82].

Nous verrons que le Cray-1 permet de respecter cette règle au prix d'une certaine discipline.

Les deux règles méthodologiques qui précèdent resteront à l'arrière-plan de toute la discussion sur la vectorisation.

IV.3. - PROBLEMES DE LANGAGE

Les langages actuellement diffusés sur Cray-1 de façon officielle sont un langage d'assemblage (CAL) et Fortran.

Un compilateur Pascal existe, développé au laboratoire atomique de Los Alamos mais non soutenu par Cray. Un compilateur Pascal "officiel", développé pour Cray par l'Université de Manchester (G.B.) est annoncé pour la fin de 1982.

Les règles de vectorisation qui vont suivre, et les documents existants, sont relatifs à Fortran. On notera que Fortran est probablement le langage le plus mal adapté au type de manipulation entraîné par la recherche de vectorisation (comme à bien d'autres critères), du fait de la pauvreté de ses structures de contrôle et de données. On peut en fait caractériser une bonne partie des techniques appliquées par le compilateur comme consistant à reconstituer à grand-peine une structure globale des manipulations de données détruite par l'emploi de Fortran, et qui se serait exprimée tout naturellement à travers APL, Algol 68 ou PL/I.

Le langage Fortran disponible sur le Cray-1 (cf. Annexe) est un dialecte original, plus étoffé que celui de la norme 1966 (Fortran IV), et comprenant un certain nombre d'éléments de Fortran 77 [Meyer 79], mais encore en deçà de cette nouvelle norme, en particulier à l'article des manipulations de caractères. Il s'en rapproche peu à peu, plus lentement hélas que nous ne l'espérions dans la première édition de cette note; le point le plus gênant est la manipulation de caractères à la mode Fortran 77 qui, disponible aujourd'hui (janvier 82) sur IBM (compilateur Fortran VS), n'est toujours pas prête sur Cray.

Dans la situation actuelle, dont on souhaite qu'elle soit transitoire, les différents cas sont les suivants :

- un programme écrit en Fortran IV (Fortran H Extended sur IBM) sera assez facilement adapté au Cray-1 (voir cependant à l'annexe B quelques sources d'ennuis possibles);

- un programme écrit en Fortran 77 (Fortran VS sur IBM) soulèvera des difficultés tant que le Fortran Cray n'inclura pas l'ensemble de la norme Fortran 77;

- un programme écrit en Fortran Cray sera facile à adapter à Fortran VS ou à une autre version de langage conforme à Fortran 77 (en revanche, le retour à Fortran IV n'est pas un exercice recommandé).

IV.4. - LE COMPILATEUR FORTRAN ET LA VECTORISATION

Le compilateur Fortran du Cray-1, appelé CFT (Cray Fortran Translator) dans la documentation technique, produit du code vectorisé lorsqu'il reconnaît des structures de programme se prêtant à cette vectorisation. Les programmeurs écrivent donc dans un Fortran conforme aux règles normales du langage, non sans adapter toutefois leur style de programmation afin de faciliter cette reconnaissance. C'est l'objet des "techniques vectorielles" vues ci-après.

La politique de vectorisation adoptée par le Cray est, de ce point de vue, plus favorable à la portabilité des programmes que les solutions consistant à concevoir des langages spécialisés pour le traitement vectoriel [Perrot 79a, 79b, 79c], à proposer des extensions à un langage [Flanders 79], ou à vectoriser seulement les appels à des sous-programmes spécialisés.

Cet avantage, considérable au vu de notre "règle de portabilité", doit cependant être tempéré par deux remarques :

a) La vectorisation extrême peut exiger une réécriture considérable du code, rendant inefficace l'exécution sur une machine non vectorielle, et peu compréhensible le programme résultant (voir en V.5 le programme calculant la somme des éléments d'un vecteur).

b) Pour des traitements vectoriels d'emploi fréquent, il est tentant de faire appel aux sous-programmes très efficaces de la bibliothèque du super-ordinateur.

L'inconvénient *a* doit être limité à de petites portions de programme (cf. ci-après, règle des boucles internes). Pour éviter des surprises désagréables avec *b*, nous préconiserons la règle d'exploitation suivante, qui découle tout naturellement de la règle de portabilité :

> REGLE DE COMPATIBILITE
>
> Ne diffuser un programme de la bibliothèque du super-ordinateur qu'après avoir inclus dans la bibliothèque du calculateur frontal un sous-programme de même nom et de même spécification externe, écrit de préférence dans un langage normalisé.

La règle de relativité mettait l'accent sur la petitesse des portions de programmes accélérables de façon significative (b), et sur la validité des programmes (c). Les deux règles qui suivent témoignent de l'attitude conservatrice du compilateur, bien conforme à ces principes.

> REGLE DES BOUCLES INTERNES
>
> Seules les boucles les plus internes sont susceptibles d'être vectorisées par le compilateur.

Il est donc inutile de changer quoi que ce soit aux autres. Le volume de code à réécrire restera ainsi proportionnellement faible dans les cas normaux.

> REGLE DE SECURITE
>
> Le compilateur ne vectorise de lui-même que les boucles répondant de façon démontrée aux conditions de vectorisation.

La nécessité de cette règle vient de ce que les résultats d'un même programme peuvent être différents selon qu'il est exécuté en mode scalaire ou vectoriel. L'interprétation "par défaut" définie par les normes des langages étant celle des ordinateurs classiques, c'est-à-dire l'interprétation scalaire, le compilateur CFT ne vectorisera jamais sans être sûr de pouvoir le faire, de peur de produire des résultats erronés. Les programmeurs disposent donc d'une bonne sécurité à cet égard.

On notera que des cas peuvent se produire où un élément de programme est vectorisable, mais le compilateur ne dispose pas des éléments pour le démontrer; ceci se produit en particulier en liaison avec le problème de la dépendance arrière (V.5) et celui de la dépendance croisée (V.6). Les programmeurs peuvent le signaler au compilateur grâce à des directives de vectorisation qui vont lever ses hésitations (s'il n'y a pas d'impossibilité détectée par ailleurs). Ceci, bien sûr, est alors aux risques et périls du programmeur.

Les directives de vectorisation adressées au compilateur Fortran du Cray-1 ne soulèvent pas de difficultés de portabilité car elles revêtent la forme de commentaires (on parlerait de "pragmas" en Algol 68 ou en Ada).

V LES GRANDES TECHNIQUES VECTORIELLES

V.1. - LES CONDITIONS DE LA VECTORISATION

Les contraintes que doivent respecter les programmes pour que le compilateur puisse les vectoriser peuvent se ramener aux cinq conditions nécessaires et suffisantes de la règle ci-dessous[1].

> REGLE DE VECTORISATION
>
> Un calcul est vectorisable si et seulement s'il respecte les cinq conditions suivantes :
>
> [C] . c'est une *série continue* d'opérations (V.2);
>
> [P] . chacune de ces opérations est *primitive* (V.3);
>
> [R] . les données traitées sont *régulières* (V.4);
>
> [DC] . elles ne présentent pas de *dépendance croisée* (V.5);
>
> [DA] . elles ne présentent pas de *dépendance arrière* (V.6).

Ces différentes conditions sont détaillées dans les cinq paragraphes qui suivent. Pour donner une vue d'ensemble, on peut les expliquer grossièrement ainsi :

- une "série continue" d'opérations est, en pratique, une boucle;

- une opération "primitive" est une opération n'entraînant aucune rupture de contrôle (en FORTRAN, ceci ne laisse guère apparemment que

(1) Cette règle est une interprétation, par l'auteur, des nombreux exemples épars dans les publications Cray. La terminologie de cette section est celle de l'auteur, non celle des documents Cray.

l'affectation est le *CONTINUE*; nous verrons qu'on peut élargir un peu cette classe);

- un ensemble "régulier" de données est, grosso modo, une constante, un entier variant en progression arithmétique ou une sous-suite d'un tableau dont l'adresse varie en progression arithmétique (exemple : la sous-suite *A(M), A(M+L), A(M+2*L), A(M+3*L)* ... du tableau *A*);

- une relation de "dépendance"[1] lie un élément apparaissant à gauche d'une affectation (ou la suite à laquelle il appartient) aux éléments de la partie droite (exemple : dans *A(I) = C*B(I+1)* *A* dépend de *C* et de *B*);

- une dépendance est "croisée" si elle lie des éléments de deux suites différentes modifiées l'une et l'autre dans les opérations d'une série (exemple : si une boucle d'indice *I* contient les affectations *A(I) = ...* et *B(I) = A(I+K)*, alors *B* dépend de *A* qui est modifié).

- une dépendance est une dépendance "arrière" lorsqu'elle fait dépendre un élément d'une suite d'un élément antérieur de la même suite (exemple : *A(I+7) = C*A(I-1)*).

Examinons maintenant ces cinq conditions.

V.2. - SERIES CONTINUES.

Le fonctionnement continu en mode vectoriel exige que les instructions nécessaires restent en permanence disponibles dans les tampons d'instructions (III.4). Ces tampons ne peuvent abriter que 128 à 256 instructions, qui doivent donc contenir une répétition pour que le calcul vectoriel présente un intérêt.

Le compilateur vectorisera donc des boucles. La règle appliquée par le compilateur Cray (CFT) est plus précise encore :

(1) Le terme de "récursion" est employé dans les publications Cray.

REGLE DU *DO*

Seules les boucles *DO* sont vectorisables.

Combinée à la "règle des boucles internes" (IV.4), cette règle indique donc que les seuls éléments susceptibles d'être vectorisés sont les boucles *DO* au niveau le plus interne.

Les boucles par *IF* et *GOTO* ne le sont donc pas. Ceci serait particulièrement gênant en FORTRAN IBM (où les boucles *DO* sont de type répéter ... jusqu'à et non tant que); l'instruction *DO* du Cray-1, conforme à Fortran 77, équivaut à une instruction vide si *(borne_sup - borne_inf)*pas < 0*.

Il vaut mieux donc sortir d'une boucle *DO* que d'hésiter à y rentrer. Méthodologiquement, ceci n'est pas très sain.

On peut espérer que les versions ultérieures du compilateur lèveront cette restriction.

V.3. - OPERATIONS PRIMITIVES

Le principe d'alimentation continue exige, nous l'avons dit, que les séries continues d'instructions qui forment les boucles vectorielles soient prévisibles, c'est-à-dire n'entraînent aucune rupture de séquence. Elles doivent donc ne contenir aucune opération non "primitive" au sens de la définition suivante.

REGLE DES OPERATIONS PRIMITIVES

Pour qu'une boucle soit vectorisable, il faut qu'elle ne contienne aucune des opérations non primitives suivantes :

a) les entrées et les sorties;

b) les tests et les branchements;

c) les appels de sous-programmes.

Le cas a) semble irrémédiable. b) et c) justifient une discussion plus fine.

Tests dans les boucles

Le cas *b* est ennuyeux dans des exemples même très simples, comme celui de l'affectation conditionnelle; ainsi la boucle suivante, non vectorisable, qui affecte une valeur ou une autre aux éléments de A selon le signe de leur valeur initiale :

```
      DO  100  I = 1, N
           IF (A(I).GT.0) A(I) = COS (A(I))
           IF (A(I).LE.0) A(I) = SIN (A(I))
100        CONTINUE
```

(N.B. On peut préférer un style de programmation utilisant des branchements plutôt qu'un test redondant).

La structure matérielle du Cray-1 permet ici en fait de vectoriser une boucle équivalente. Le principe est de calculer toutes les valeurs éventuellement requises, et d'extraire ensuite celles qui sont véritablement nécessaires grâce aux instructions de masquage (cf. III.7). Cette opération équivaut en termes de Fortran à remplacer l'extrait ci-dessus par :


```
      REAL A(N), P(N), Q(N)
      LOGICAL MASQUE(N)
      ------------------
      DO  101  I = 1,N
           P(I) = COS (A(I))
           Q(I) = SIN (A(I))
           MASQUE(I) = A(I).GT.0
101        CONTINUE
C     -- EXTRACTION --
      DO  102  I = 1,N
           IF (MASQUE(I))     B(I) = P(I)
           IF (NOT.MASQUE(I)) B(I) = Q(I)
102        CONTINUE
```

La première boucle correspond au (double) calcul de toutes les valeurs possibles, la seconde à une extraction, opérée par des instructions vectorielles de masquage. Bien entendu, ce texte Fortran représente symboliquement le programme équivalent produit par le compilateur, et ne correspond à rien qui existe concrètement sous la forme donnée.

Pour atteindre l'effet de ces boucles vectorielles, la structure logicielle proposée en Fortran est un jeu de deux sous-programmes spéciaux *CVMGT* (comparer pour supérieur strictement) et *CVMGZ* (comparer à zéro). On peut ici remplacer la boucle par :

```
      DO  103  I = 1,N
           B(I) = CVMGT (A(I).GT.0, COS (A(I)), SIN (A(I)))
103        CONTINUE
```

On aura noté que cette méthode va tout à fait à l'encontre du critère de portabilité énoncé en IV.2., et constitue une exception à l'emploi exclusif par le Cray-1 de constructions Fortran normales.

La reconnaissance par le compilateur des constructions de boucle telles que *100* (ou *102*), sans exiger l'emploi de fonctions spécialisées, fait partie des extensions annoncées par Cray. On peut donc espérer que la difficulté liée à la portabilité sera bientôt résolue.

Il reste cependant, fonctions spécialisées ou non, deux points délicats. Le premier est celui des branches d'alternative exigeant plus d'une instruc-

tion Fortran. L'autre est celui des quantités non définies. L'appel à *CVMGT* ou *CVMGZ*, produit par un programmeur ou dans l'avenir par le compilateur, évalue des vecteurs complets pour n'en garder que certains éléments. Les autres ont pu être écartés volontairement, comme dans l'exemple ci-dessous :

```
      DO  150  I = 1, N
          IF (A(I).GE.0) B(I) = SQRT (A(I))
          IF (A(I).LT.0) B(I) = SQRT (-2*A(I))
150   CONTINUE
```

Il est clair qu'on ne veut pas ici évaluer complètement les deux vecteurs $\sqrt{A}$ et $\sqrt{-2\,A}$. Cette boucle ne peut donc pas, en tout état de cause, être vectorisée.

Appels de sous-programmes

Pour résoudre le cas c) (appels dans une boucle), on peut appliquer la règle suivante :

> REGLE DES APPELS
>
> Mettre le sous-programme dans la boucle, ou la boucle dans le sous-programme.

La première solution consiste à remplacer l'appel

```
CALL SP( .... )
```

dans le corps d'une boucle, par le texte même du corps du sous-programme après remplacement des arguments.

La seconde consiste à remplacer toute entière une boucle de la forme :

```
      DO  200  I = 1, N
          ..........
          CALL SP (A(I), B(I), ...)
          ..........
200       CONTINUE
```


par un seul appel de sous-programme

```
CALL SPVEC (A, B, ....)
```

où *SPVEC* est un sous-programme "vectoriel", opérant répétitivement, par une boucle, sur les éléments de *A* et *B*.

Lorsqu'elle est applicable, c'est cette seconde solution (mettre la boucle dans le sous-programme) qui doit être recommandée. La première est en effet contraire à tout principe de modularité et de structuration des programmes (voir la règle de validité); la seconde permet au contraire de conserver, voire d'améliorer la structure de contrôle, en l'associant à celle des données : on aboutit à une architecture dans laquelle les sous-programmes manipulent des tableaux entiers; en d'autres termes, on écrit des algorithmes véritablement vectoriels, manipulant des objets de type "tableau".

On notera que cette seconde solution entre dans le cadre des techniques visant à ramener un programme Fortran à son équivalent dans un langage permettant la manipulation de tableaux, comme APL, PL/I ou Algol 68 (cf. IV.3).

V.4. - ENSEMBLES REGULIERS DE DONNEES

Pour qu'une boucle soit vectorisable, toutes les références qu'elle effectue à des données doivent être prévisibles. En pratique, ceci signifie que la variation des valeurs de ces données (pour des entiers) ou, plus généralement, de leurs adresses, doit s'effectuer en progression arithmétique (ce qui comprend en particulier les constantes). De tels éléments seront dits "réguliers".

> REGLE DE REGULARITE
>
> Un élément *E* (constante, variable, élément de tableau, expression) intervenant dans une boucle est dit régulier si et seulement s'il vérifie l'une des conditions suivantes :
>
> a) *E* n'est pas modifié dans le corps de la boucle; nous dirons alors qu'*E* est constant relativement à la boucle;
>
> b) *E* est l'indice de boucle;

c) E est une expression de la forme $\pm C*E'$, où C, E, E' sont entiers, C est constant relativement à la boucle et E' régulier;

d) E est une expression de la forme $E'\pm C$, où C, E, E' sont entiers, C est constant relativement à la boucle et E' régulier;

e) E est un élément de tableau dont tous les indices sont constants relativement à la boucle, sauf éventuellement un qui est alors une variable régulière.

Pour qu'une boucle soit vectorisable, il faut que tous les objets qui y apparaissent (variables, éléments de tableaux) soient réguliers.

La règle de régularité est récursive, ce qui ne devrait pas poser de problème particulier de compréhension.

Exemple

Soit une boucle d'indice I. Soient $A(20)$, $B(10,50)$, $C(20,30,10)$ des tableaux réels, M et N des entiers n'apparaissant pas dans la boucle à gauche d'un signe =.

Les éléments suivants sont réguliers :

I
M
N
$I + N$
$I - N$
$M * I$
$M * I - 3 * M * N$
$A(I)$
$B(I, M)$
$C(2*N - M, M * I - 3*M*N, 72 + M)$

Les éléments suivants ne sont pas réguliers :

$B(I, I + 1)$	(deux indices non constants relativement à la boucle)
$M/(N*I)$	(division)
$N*C(I)$	($C(I)$ n'est pas entier)

Nota : Le compilateur CFT impose des restrictions supplémentaires (relatives par exemple à l'imbrication des parenthèses), qui semblent évoluer rapidement. En l'absence d'une documentation précise, et en particulier d'une grammaire en BNF, il nous a paru préférable de ne pas les détailler. On trouvera des indications plus concrètes dans [de Orouas 81].

V.5. - LA DEPENDANCE ARRIERE

V.5.1. - Définition

Le fonctionnement en mode "bitoduc" entraîne qu'à tout instant de l'exécution d'une boucle *DO* une itération de rang i peut être lancée alors qu'une itération précédente, de rang $j \leq i$, n'est pas encore terminée. Plus précisément, ceci ne peut se produire que si :

$$i - j < cb$$

où cb est la capacité du bitoduc (64 pour le Cray-1). Les calculs seront donc erronés si la seconde utilise, dans le programme, des résultats produits par la première. C'est un cas de dépendance arrière.

Exemple :

La boucle

```
      DO  300  I = 2, N
          .............
          A(I) = 3*A(I-1)
300       CONTINUE
```

n'est pas vectorisable du fait de la dépendance arrière directe. Il existe aussi des dépendances arrière indirectes :

```
      DO  350  I = 1, N
          B(I) = 3/I
          A(I) = I**B(I-8)
350       CONTINUE
```

A utilise ici un élément de *B* calculé 8 itérations plus tôt. Cette dépendance arrière est en fait ici un cas particulier de la "dépendance croisée" vue plus loin.

On peut résumer cette situation par la règle suivante.

> REGLE DE NON-DEPENDANCE ARRIERE
>
> Pour qu'une boucle soit vectorisable, il faut qu'aucune des affectations qu'elle contient ne fasse dépendre un élément d'un vecteur, à une itération i quelconque, d'un élément du même vecteur qui a pu être modifié à une itération j, avec $i - cb < j < i$ ($cb = 64$).

On trouvera à l'annexe D une définition formelle de la dépendance arrière[1].

V.5.2. - Exemples de dépendances arrière; solutions

Un cas trivial de dépendance arrière ne faisant pas nécessairement intervenir d'indice est celui d'une affectation de la forme $X = f(X)$ (où f n'est pas l'identité). Il y a dépendance arrière entre la variable X (considérée comme vecteur à un élément) et elle-même. Deux cas importants en pratique sont la somme d'un vecteur

```
      S = 0
      DO  400 I = 1, N
          S = S + A(I)
400       CONTINUE
```

(1) Dans son état actuel, le compilateur CFT refuse toujours la vectorisation en cas de dépendance arrière, même s'il peut vérifier que $j \leq i - cb$. Il faut donc, si l'on sait que $j \leq i - cb$, utiliser une directive *IVDEP* (cf. V.7). Cette situation est regrettable.

où l'on remarque d'ailleurs que *S* n'est pas régulière, et le produit matriciel :

```
      DO  600 I = 1, M
          DO  550 J = 1, N
              DO  500 K = 1, P
                  C (I,K) = C(I,K) + A(I,K) * B(K,J)
500           CONTINUE
550       CONTINUE
600   CONTINUE
```

On a supposé ici qu'une première boucle (vectorisable), non écrite, avait initialisé *C* à zéro.

La dépendance arrière n'est gênante que si elle ne dépasse pas la capacité cb du bitoduc ($cb = 64$).

La dépendance arrière est de toute évidence une propriété de l'algorithme utilisé; on ne peut guère s'attendre à la voir supprimée par une modification superficielle du style de programmation. Il serait intéressant d'examiner les grands algorithmes numériques à la lumière de ce critère; il est clair par exemple que des algorithmes où l'approximation se déplace parallèlement sur un "front" (Jacobi) sont préférables de ce point de vue à ceux (Gauss-Seidel) pour lesquels l'approximation de la solution au [*point p, instant t*] dépend d'un certain nombre de valeurs approchées [p', t] pour d'autres points p', et non pas seulement de valeurs [p', $t - 1$]. Gauss-Seidel possède cependant une version vectorielle ; sur la vectorisation de ces deux algorithmes, voir [Bossavit 81a]

Dans certains cas, une modification relativement simple d'un algorithme à dépendance arrière peut le rendre vectorisable. C'est le cas du calcul matriciel : si au lieu de penser "élément"

$$c_i^j = \Sigma\, a_i^k\, b_k^j$$

on pense "vecteur (ligne)"

$$c_i = \Sigma\, a_i^k\, b_k$$

on obtient la version vectorisable ci-après :

```
      DO  800 I = 1, M
C         — LIGNE I —
          DO  750  K = 1, P
              DO  700  J = 1, N
                  C(I,J) = C(I,J) + A(I,K) * B(K,J)
700           CONTINUE
750       CONTINUE
800   CONTINUE
```

(Note importante : Il devient ici fondamental d'avoir effectué l'initialisation à zéro de la matrice *C* dans une première boucle disjointe).

La version vectorisable du calcul de la somme d'un vecteur est beaucoup moins convaincante. Nous la donnons ici à titre d'illustration; avant de s'inquiéter, on lira le paragraphe V.5.3 ci-après. Elle utilise un sous-tableau auxiliaire *B* permettant de se ramener à 64 sommes de sous-vecteurs :

```
C     CALCUL DE LA SOMME DE N ELEMENTS DU VECTEUR A
C     — BOUCLE VECTORISABLE —
      DO  900  I = 1, N
          B(I) = A(I)
900       CONTINUE
C     — BOUCLE VECTORISABLE (EVENTUELLEMENT NULLE) —
      DO  930  I = 65, N
          B(I) = B(I) + B(I - 64)
930       CONTINUE
C     — BOUCLE NON VECTORISABLE (64 ITERATIONS AU PLUS) —
      S = 0.
      M = MAXO.(N - 63, 1)
      DO  960  I = M, N
          S = S + B(I)
960       CONTINUE
```

L'utilisation d'une constante "magique" telle que 64 est évidemment exécrable d'un point de vue méthodologique. C'est ce genre de pratique qui donne, après quelques années, des programmes mutilés, portant les stigmates des systèmes successifs. On préfèrera utiliser la constante symbolique *CB*, après avoir déclaré (cf. ci-après V.7) :


```
C     — TAILLE DU BITODUC DU CRAY-1
      PARAMETER (CB = 64)
      INTEGER CB
```

V.5.3. - Les opérations de réduction et la pseudo-vectorisation

A partir de la version 1.09 du compilateur (janvier 1981), une amélioration importante est venue remédier aux difficultés soulevées par les problèmes tels que le calcul d'un produit scalaire ou celui de la somme des éléments d'un vecteur. C'est la pseudo-vectorisation des opérations de réduction. On appelle opération de réduction sur des vecteurs *A*, *B* ... le calcul d'une valeur x par une boucle de la forme

```
      x = valeur initiale
      DO 970  I = 1, N
970       x = x ⊕ g (A(I), B(I), ...)
```

où $\oplus$ et g sont des opérations arithmétiques. Le produit scalaire et la somme d'un vecteur sont typiquement des opérations de réduction. D'après ce qui précède, une réduction n'est pas vectorisable du fait de la dépendance arrière entre x et lui-même. Mais elle est pseudo-vectorisable; la technique de pseudo-vectorisation consiste à utiliser un bitoduc par étapes successives, en réinjectant la sortie dans l'entrée (voir [de Drouas 81] pour les détails de cette technique). Le résultat sera un temps de calcul plus grand que celui d'une opération vectorielle au sens propre, mais bien inférieur à celui d'une opération non vectorielle.

V.6. - LA DEPENDANCE CROISEE

La règle de non-dépendance arrière s'applique à des éléments d'une même suite. Pour des suites différentes, la règle est plus stricte :

> REGLE DE NON-DEPENDANCE CROISEE
>
> Pour qu'une boucle soit vectorisable, il faut qu'aucune des affectations qu'elle contient ne fasse dépendre un élément d'un vecteur, à une itération i quelconque, d'un élément de vecteur qui peut être modifié par une autre affectation à une itération j, avec $|j-i| < cb$ ($cb = 64$).

On trouvera une définition formelle de la dépendance croisée à l'annexe D[1]. Ainsi, la boucle :

```
      DO  1000  I = 1, N
          A(I) = 2*A (I+2)
          B(I) = 3*A (I+1)
1000  CONTINUE
```

n'est pas vectorisable du fait de la dépendance croisée entre *B* et *A* (et non pas de la dépendance directe entre *A* et lui-même, qui est une dépendance avant).

A quoi est due cette règle, draconienne puisqu'elle peut faire référence à des éléments postérieurs des suites considérées (comparer $|i-j| < cb$ dans cette règle à $i - cb < j < i$ dans la précédente) ? Elle découle en fait de la règle d'ouverture des boucles (III.7). Nous avons vu que, pour vectoriser une boucle "longue", on peut être obligé de la scinder en plusieurs boucles. Ici le résultat d'une telle ouverture de boucle serait donc le même que si le programmeur, au lieu de la boucle *1000*, avait écrit :

```
      DO  1010  I = 1, N
1010      A(I) = A (I+1) + 3
      DO  1011  I = 1, N
1020      B(I) = A (I+1) - 1
```

Le résultat est ici différent : en supposant le vecteur *A* initialisé à zéro, la boucle *1000* donne un vecteur *B* tout à zéro (puisque la seconde affectation de la boucle utilise les anciennes valeurs de *A (I+1)*, alors que les boucles *1010-1020* affectent la valeur 1 à tous les éléments de *B* (dans la boucle *1020*, on utilise les nouvelles valeurs de *A*).

(1) Dans son état actuel, le compilateur CFT refuse toujours la vectorisation en cas de dépendance croisée, même s'il peut vérifier que $|j-i| \geq cb$. Il faut donc, si l'on sait que $|j-i| \geq cb$, utiliser une directive *IVDEP* (cf. V.7). Cette situation est regrettable.

Bien entendu, l'interprétation classique de Fortran est l'interprétation séquentielle (*1000*). De peur d'induire une erreur par l'interprétation vectorielle en cas d'ouverture (*1010-1020*), le compilateur ne vectorisera donc pas une boucle présentant une telle dépendance croisée.

Aucune règle simple ne permet d'éviter la dépendance croisée, qui est en général, comme la dépendance arrière, une propriété de l'algorithme.

Ce type de manipulation de boucles rejoint les problèmes classiques de la théorie des transformations de programme. On trouvera une étude en ce sens dans [Bossavit 82].

V.7. - DIRECTIVES AU COMPILATEUR - PRINCIPE DE PARALLELISME

Dans certains cas, une connaissance approfondie du programme permet d'affirmer qu'il n'y a pas de dépendance (arrière ou croisée), alors que la lettre du texte ne le garantit pas.

Dans

```
      DO  1200  I = 1, N
          A(I) = A(I+P)
          B(I) = A(I+Q)
1200  CONTINUE
```

si l'on sait que $(P \geq 0 \text{ ou } P \leq -64) \geq (|Q| \geq 64)$, alors la boucle est vectorisable. On peut alors inciter le compilateur à larguer ses scrupules (cf. IV.4, règle de sécurité) et à la vectoriser, en la faisant précéder de la directive spéciale suivante, qui apparaît comme un commentaire :

```
CDIR$  IVDEP
```

Il est clair qu'une telle opération est aux risques et périls du programmeur.

Les documents Cray indiquent que, même avec une telle directive, la vectorisation ne sera pas effectuée si le compilateur la détecte comme impossible. En d'autres termes la clause *IVDEP* sert uniquement à lever les soupçons de dépendance (arrière ou croisée) que le compilateur nourrit à l'égard des variables qui sont constantes relativement à la boucle (*P* et *Q* ci-dessus).

On peut regretter que le mode d'expression offert au programmeur ne soit pas plus précis : ALGOL W ou Ada auraient proposé des assertions permettant de dire exactement ce que l'on sait, ou croit savoir, en écrivant quelque chose comme :

```
CDIR$ ASSERT  (P.GE.0.OR.P.LE. - CB) .AND. (ABS(Q) . GE. CB)
```

[ATTENTION, CECI EST UNE SUGGESTION DE L'AUTEUR ET NON UNE DIRECTIVE LEGALE DU FORTRAN CRAY !]

Une situation assez fréquente est celle où les éléments comme P, Q ne sont pas en fait des variables, mais des paramètres du problème, qui restent constants pendant une exécution du programme; on les désigne par leurs noms symboliques plutôt que par leurs valeurs numériques en prévision d'un changement possible. Si leur valeur est fournie par un ordre *DATA*

```
DATA P/7/,Q/2250/
```

le compilateur n'en tire aucune assurance particulière, car des variables initialisées ainsi peuvent être modifiées. Par contre, l'ordre *PARAMETER* présent dans le Fortran du Cray-1 et en Fortran 77 permet de définir des constantes symboliques, non susceptibles d'affectation :

```
PARAMETER (P = 7, Q = 2250)
```

Si les valeurs des éléments soupçonnés d'interdire la vectorisation pour fait de dépendance sont indiquées de cette façon, le compilateur CFT vectorisera les boucles du type *1200* sans qu'il soit besoin d'inclure une directive *IVDEP*.

On notera enfin, pour terminer sur les problèmes de dépendance, qu'un cas particulièrement agréable est celui où il n'y a aucune dépendance entre les itérations d'une boucle, c'est-à-dire où elles pourraient toutes s'exécuter en même temps. Il s'agit d'une condition suffisante, mais non nécessaire (sinon nous n'aurions pas autant développé la dépendance arrière et la dépendance croisée). Un exemple est la version vectorielle du produit matriciel (V.5, boucle *700*).

V.8. - L'ADRESSAGE INDIRECT

Introduisons pour conclure une technique utile de vectorisation. Une boucle de la forme :

```
      DO  1300 I = 1, N
        J = f(I)
        A(I) = g(B(J))
1300  CONTINUE
```

où f et g sont des expressions régulières (cf. V.4; par exemple, f pourrait être un nom de tableau, et g une expression plus compliquée) n'est pas vectorisable car J n'est pas une variable régulière.

Le dédoublement de cette boucle, avec utilisation d'un vecteur auxiliaire *IND*, permet de rendre "à-demi" vectorisable :

```
C     -- BOUCLE NON VECTORISABLE --
      DO  1330  I = 1, N
        J = f(I)
        IND (I) = B(J)
1330    CONTINUE
C     -- BOUCLE VECTORISABLE --
      DO  1360  I = 1, N
        A(I) = g(IND(I))
1360    CONTINUE
```

Le gain pourra être appréciable si g est une expression compliquée.

ANNEXE A

PRINCIPALES CARACTERISTIQUES DU CRAY-1

UNITE DE CALCUL

Structure

modes de traitement : scalaire, vectoriel
instructions : 128 codes différents
arithmétique : entière et flottante (N.B. pas de division flottante câblée)
unités fonctionnelles : 12, segmentées
interruptions : système à priorités

Constantes de base

période de l'horloge : 12,5 nanosecondes (1 ns = 10^{-9} s)
taille des mots : 64 bits
adresses : sur 24 bits

Registres

4 tampons d'instructions (64 éléments de 16 bits chacun)
8 registres *A* (adresses : 24 bits)
64 registres *B* (adresses intermédiaires : 24 bits)
8 registres *S* (scalaires : 64 bits)
64 registres *T* (scalaires intermédiaires : 64 bits)
8 registres *V* (vecteurs : 64 éléments de 64 bits chacun)
1 registre *VL* de longueur de vecteur
1 registre *VM* de masque de vecteur
1 registre d'horloge temps réel (64 bits)

ANNEXE D

DEFINITION FORMELLE DE LA DEPENDANCE

Considérons une boucle exécutée ni fois. Elle calcule ni éléments d'un certain nombre de suites. Soit ns le nombre de ces suites. Chaque exécution de la boucle opère ns affectations à des variables ou éléments de tableaux ; nous ne considérons que des éléments de tableaux en traitant les variables simples comme des tableaux à un element.

Pour i compris entre 1 et ni, l'exécution de la i-ieme itération de la boucle modifie la valeur de nt éléments de tableaux qui sont donnés par

$$v_t\,(g_t(i)) \qquad 1 <= t <= nt$$

g pour "partie gauche" (d'une affectation)

La valeur affectée à chacun de ces éléments peut être notée :

$$f_t\,(v_{h_{t,1}}(d_{t,1}(i)),\ v_{h_{t,2}}(d_{t,2}(i)),\ldots\ldots,v_{h_{t,p(t)}}(d_{t,p(t)}(i)))$$

(d pour "partie droite" ; p(t) est une borne dépendant de t)

Exprimons l'existence de ces différentes fonctions par une "classe" en langage Z, qui nous permettra d'énoncer les conditions de non-dépendance :

```
non_dépendance ==
    class with
        cb : NAT
        ni, ns : NAT ;
        INDICE, TABLEAU : subset (NAT) ;
        g : TABLEAU -> (INDICE -> INDICE) ;
        p : TABLEAU -> NAT ;
        d : TABLEAU * NAT -/-> (INDICE -> INDICE) ;
        h : TABLEAU * NAT -/-> TABLEAU ;
    def
        cb == 64 ; INDICE == 1..ni ; TABLEAU == 1..ns
    where
        dom (d) = set t, n where n (- 1..p(t)) end ;
        dom (h) = dom (d) ;
        forall i, t1, t2 , k1 for
            i : 1..ni ; t1, t2 : 1..ns ; k1 : 1..p(t1)
        then
            arrière :
                h(t1, k1) = t1 ==>
                    d (t1, k1) (i) (/- g (t1)(i-cb+1..i) ;
            croisée :
                h(t1, k1) = t2 ==>
                    d (t1, k1)  (i) (/- g (t2) (i-cb+1..i+cb-1)
        end
    end
```

BIBLIOGRAPHIE

Pour les documents Cray, la date et le numero donnes sont ceux de la version disponible au moment de la publication de cette note. Ces renseignements sont evidemment susceptibles de modification.

[AFCOL 82] Division AFCOL : Vers une norme : 101 Conseils pour la programmation en Fortran ; Note EDF, Atelier Logiciel no. 35, mars 1982.

[Beauchamp 81] Anne Beauchamp : LECIBM et ECRIBM ; Note EDF Cray no. 5, HI-3953/01, octobre 1981.

[Beauchamp 81a] Anne Beauchamp : Sous-Programme d'Allocation dynamique de Memoire ; Note Cray no. 7, HI-3984/01, novembre 1981.

[Bossavit 79] Alain Bossavit : Bientot, la Vectorisation (Chronique d'Analyse numerique) ; BUCCER Numero 75, septembre 1979.

[Bossavit 81] Alain Bossavit, Bertrand Meyer : The Design of Vector Programs ; Note EDF Cray no. 9, HI-3694/01, octobre 1981 (Communication au Congres "International Symposium on Algorithmic Languages", Amsterdam, 26-29 octobre 1981 ; J. W. de Bakker (Ed.), North-Holland, 1982.

[Bossavit 81a] Alain Bossavit : Vectorisation de quelques Algorithmes numeriques ; Note EDF Cray no. 6, HI-4001/01, decembre 1981.

[Bossavit 82] Alain Bossavit, Bertrand Meyer : Methods for Vector Programming ; a paraitre.

[Brinch Hansen 73] Per Brinch Hansen : Operating Systems Principles ; Prentice-Hall, 1973.

[Brinch Hansen 79] Per Brinch Hansen : A Keynote Address on Concurrent Programming ; Computer (IEEE), 12, 5, pages 50-56, mai 1979.

[Butel 81] Remy Butel : Evaluation Theorique des Performances du Cray ; Note EDF Cray no. 10, HI-3947/00, octobre 1981.

[Cray Assemblage] Cray-1 Computer System - CAL Assembler Version 1 Reference Manual, Document Cray numero SR-0000, version I-01, juin 1981.

[Cray Bibliotheque] Cray-1 Computer System - Library Reference Manual, Document Cray numero SR-0014, version E-01, juin 1981.

[Cray Fortran] Cray-1 Computer System - FORTRAN (CFT) Reference Manual, Document Cray numero SR-0009, version H, aout 1981.

[Cray Machine] Cray-1 S Series Hardware Reference Manual, Document Cray numero HR-0808, juin 1980.

[Cray Presentation] The Cray-1 Computer System - Document Cray de presentation, sans numero ni date.

[Cray Progiciels] Scientific Applications Package Handbook, Document Cray sans numero, version C, aout 1981.

[Cray Systeme] Cray-1 Computer System - Cray OS Reference Manual ; Manual, Document Cray numero SR-0011, version I-02, juillet 1981.

[De Drouas 81] Eric de Drouas : Vectoriser sur le Cray-1 ; Note EDF Cray no. 8, Atelier Logiciel no. 31, HI-3919/01, juin 1981.

[De Drouas 81a] Eric de Drouas, Marie Farges : Performances du Cray-1 en Fortran ; Note EDF Cray no. 4, HI-3919/01, septembre 1981.

[Dungworth 79] : M. Dungworth : The Cray 1 Computer System ; dans [Infotech 79], tome 2, pages 51-76, numero de reference 80H324843 a la Documentation.

[Flanders 79] P.M. Flanders : Fortran Extensions for a Highly Parallel Processor ; dans [Infotech 79], tome 2, pages 117-134, numero de reference 80H324846 a la Documentation.

[Glaziou 81] Y. Glaziou : Acces au Cray des Utilisateurs du Reseau Retina ; Bucser (Bulletin du Centre de Calcul des Etudes et Recherches), no. 98, decembre 1981.

[Higbie 79] Lee Higbie : Vectorization and Conversion of FORTRAN Programs for the Cray-1 (CFT) Compiler - Document Cray numero 2240207, juin 1979.

[Hoare 78] C.A.R. Hoare : Communicating Sequential Processes ; Communications of the ACM, volume 21, numero 8, pages 666-677, aout 1978.

[Infotech 79] Infotech State of the Art Report on Supercomputers ; tome 1 (Total Systems Issues) ; tome 2 (Invited Papers) ; Maidenhead, 1979. Numeros de reference 80H324840 et 80H324841 a la Documentation (certains articles ont en outre des numeros individuels).

[Magnier 81] M. Magnier : Langage de Commande du Cray-1, Manuel de Reference ; Note EDF Cray no. 2, HI-3569/01, avril 1981 (version 2 a paraitre en janvier 82)

[Meyer 78] Bertrand Meyer et Claude Baudoin : Methodes de Programmation - Eyrolles, Paris, 1978.

[Meyer 79] Bertrand Meyer : La nouvelle Norme Fortran 77 ; Note EDF Atelier Logiciel numero 10, 1979.

[Meyer 80] Bertrand Meyer, Eric de Drouas : Un Calculateur vectoriel - le Cray-1 et sa programmation ; Film video, septembre 1980.

[Moreau 1981] Rene Moreau : Ainsi naquit l'informatique ; Dunod, Paris, 1981.

[Perrot 79a] R.H. Perrot : Parallel Languages, dans [Infotech 79], tome 1, pages 117-149, numero de reference 80H324836 a la Documentation.

[Perrot 79b] R.H. Perrot : A Standard for Supercomputer Languages, dans [Infotech 79] , tome 2, pages 291-308, numero de reference 80H324848 a la Documentation.

[Perrot 79c] R.H. Perrot : A Language for Array and Vector Processors ; TOPLAS (Transactions on Programming Languages and Systems, ACM), volume 1, numero 2, pages 177-195, 1979.

[Williams 79] S.A. Williams : The Portability of Programs and Languages for Vector and Array Processors ; dans [Infotech 79], tome 2, pages 381-394, numero de reference 80H324849 a la Documentation.

Chronique reguliere "Cray" au Bucser (Bulletin du Centre de Calcul des Etudes et Recherches).

VOS COMMENTAIRES S'IL VOUS PLAIT !

Votre opinion nous est précieuse. Aidez-nous à améliorer nos produits en renvoyant cette feuille à Mme Fumadelles, Division APCOL, Clamart, après y avoir porté vos remarques.

NOTE : Atelier logiciel n° 24

TITRE : UN CALCULATEUR VECTORIEL ; LE CRAY-1 ET SA PROGRAMMATION.

VERSION : 2, le 4 Juin 1980

Votre nom :

Votre adresse précise :

Désirez-vous

- recevoir à l'avenir les nouvelles notes de la série Atelier logiciel ?

☐ oui ☐ non

- recevoir dès maintenant les notes AL n° :

Vos commentaires sur cette note et/ou le produit qu'elle décrit :

*** La Documentation est disponible au Bureau C.121 ***
à Clamart - Tél. : 765 3746
****

		n.Version - Date	- Auteurs
n.1	- L'Atelier Logiciel	2 - Oct. 1978	- B.Meyer
n.2	- MASQUE : Récupération des erreurs à l'exécution	2 - Avr. 1981	- E.Audin
n.3	- FORTRACE : Aide à la mise au point (trace et profil)	2 - Avr. 1981	- B.Logez
n.4	- ENSORCELE 1 : Entrées et sorties sans format (1ère partie : sorties)	4 - Avr. 1981	- B.Meyer
n.6	- CHRONOS : Mesures précises de temps d'exécution	4 - Mars 1984	- G.Brisson B.Meyer
n.7	- ENSORCELE 2 : Entrées et sorties sans format	4 - Déc. 1982	- G.Brisson B.Meyer F.Vapné
n.8	- Un ramasse-miettes par tri	2 - Aout 1978	- B.Meyer
n.9	- APOTHECE : Gestion et documentation automatiques des bibliothèques de programmes	9 - Déc. 1982	- E.Audin
n.10	- La nouvelle norme Fortran 77	1 - Juin 1980	- B.Meyer
n.11	- MORTRAN : Un processeur de macros et un sur-langage de Fortran	1 - Avril 1978	N.Andriantsizafy
n.12	- ANAGIC : Analyse syntaxique	2 - Déc. 1982	- B.Logez
n.13	- TRI : Tri interne rapide	3 - Fev. 1984	- G.Brisson B.Meyer
n.14	- Les langages de spécification	4 - Avr. 1981	- M.Denuynck B.Meyer
n.15	- SIMULA 67 . Concepts	1 - Juin 1979	- B.Meyer
n.16	- SIMULA 67 : Manuel de références Manuel d'utilisation	4 - Mars 1982	- N.Penot - P.Guesnet - B.Meyer
n.17	- Résolution d'un problème de simulation en SIMULA 67	1 - Janv 1979	- N.Penot
n.18	- Procédures TSO conversationnelles d'un intéret gén.:la biblioth. AL	2 - Avr. 1981	- G.Brisson
n.19	- AXEDIR : Accès direct	4 - Aout 1982	- E.Audin
n.20	- REDUCE : Calcul symbolique	1 - Juil.1979	- P.Gaudron B.Lalande-B.Meyer
N.21	- LE LANGAGE ADA : Introduction ET Bibliographie	1 - DEC. 1983	- E.DE DROUAS B.Meyer
N.22	- GESCRAN : Gestion d'écrans en mode page	3 - NOV. 1983	- E.AUDIN G.BRISSON F.FICHEUX-VAPNE B.MEYER
n.23	- Sur le formalisme dans les specifications	1 - Aout 1979	- B.Meyer
n.24	- Un calculateur vectoriel : 4 - le Cray-1 et sa programmation	4 - Janv.1982	- B.Meyer
n.25	- INDEX : constitution de l'index d'un document	2 - Aout 1980	- E.de Drouas B.Meyer
n.26	- CONTREX : Controle conversationnel de l'exécution d'un programme	2 - Aout 1983	- B.Logez
n.27	- Panorama des langages de programmat.	2 - Avr. 1981	- B.Meyer
n.28	- La spécification	2 - Aout 1982	- B.Meyer
n.29	- Principles of Package Design (Principes de conception de Progiciels)	2 - Juil.1982	- B.Meyer
n.30	- ERRARE : Outils de traitement des erreurs	1 - Mars 1984	- G.Brisson B.Meyer
n.31	- Vectoriser sur le Cray-1	1 - Juin 1981	- E.de Drouas
n.32	- SVP : Assistance en différé	1 - Oct. 1981	- E.de Drouas
n.33	- Un environnement conversationnel à deux dimensions	1 - Janv.1982	- B.Meyer
n.34	- J'ai meme rencontré des programmeurs heureux	1 - Janv.1982	- B.Meyer
n.35	- Vers une norme : cent-un conseils pour la programmation en Fortran	1 - Mars 1982	- F.Vapné et coll.
n.36	- CONSCRAN : Construction intéractive d'écrans alphanumériques	(provisoire)	- B.Logez P.Nardy
n.37	- Entrées/sorties conversationnelles en Pascal	1 - Sept.1982	- E.de Drouas
n.38	- Régles de présentation des programmes Fortran	1 - Janv.1983	- G.Brisson F.Vapné
n.39	- TADYFOR : Tableaux dynamiques en Fortran	- à paraitre-	E.Audin
n.40	- Panorama des outils d'aide à l'amélioration de la qualité des logiciels	1 - Avr. 1983	- JM.Nerson
n.41	- Les ateliers logiciels intégrés : Développements français actuels	1 - Avr. 1983	- E.de Drouas JM.Nerson
n.42	- Quatre outils d'analyse statique	1 - Avr. 1983	- JM.Nerson B.Meyer
n.43	- VALPRO : Analyse, Documentation et tests de programmes Fortran	1 - Janv.1984	- F.Guy

EXTRAIT DE LA COLLECTION

TECHNIQUES DE L'INGÉNIEUR

TECHNIQUES DE L'INGÉNIEUR
21 RUE CASSETTE 21
75006 PARIS

Langages de programmation

Présentation

par **Bertrand MEYER**

Ancien Élève de l'École Polytechnique
Ingénieur-civil de l'École Nationale Supérieure des Télécommunications
Master of Science (Stanford)
Chef de Division (Direction des Études et Recherches) à l'Électricité de France

1 Pourquoi des langages de programmation?

Pour décrire à l'attention d'un calculateur une méthode de résolution d'un problème, il faut disposer d'un formalisme adéquat. Une telle notation, passerelle entre le logiciel et le matériel, est appelée langage de programmation. Il existe aujourd'hui des milliers de tels langages, qui jouent un rôle fondamental dans l'emploi et l'étude de l'informatique. Avant de passer en revue leurs principales caractéristiques, il convient de bien délimiter le sens de cette notion.

1,1 DÉFINITION

Les langages de programmation sont des *codes* utilisés pour décrire des *algorithmes* et leurs *données* sous une forme permettant à la fois leur *exécution* par une machine et leur *compréhension* par des lecteurs humains.

Cette définition, dont chaque terme est important, mérite quelques commentaires.

• Le mot **code** est plus neutre que *langage*; de fait, il convient de ne pas se leurrer quant aux analogies suggérées par l'emploi d'un même terme pour des langues humaines et pour les notations qui servent de support à la programmation. On décèle certes aisément des points communs: il s'agit dans l'un et l'autre cas de systèmes de signes codifiés; le rôle de véhicule pour la communication, évident dans le cas des langues humaines, intervient également dans les langages de programmation; l'étude de ceux-ci, enfin, emprunte tout naturellement des termes et des concepts à la linguistique (*syntaxe* et *sémantique* d'un langage, *grammaire*, *style* de programmation, etc.). En dépit de ces analogies, cependant, la distance est grande des langages naturels aux langages artificiels utilisés en programmation. Les premiers sont d'une richesse inépuisable et d'une infinie diversité; les seconds offrent un maigre catalogue de moyens de construction de programmes. Les premiers privilégient le sous-entendu, la litote, le raccourci, le métaphorique et le métonymique; les seconds exigent une rigueur et une précision supérieures à celles qui sont de mise en mathématique, et le caractère implacable de l'automate qui les interprète est, pour tous les débutants, source de quelques surprises désagréables. On peut aussi rappeler le rôle dominant joué par les signes auditifs dans le langage humain, et dont on ne voit guère l'équivalent en Fortran ou en APL.

Une délimitation plus précise des différences entre langages de programmation et langages humains peut être obtenue par référence à la classification, due à C.S. Peirce (1867), des signes linguistiques en trois catégories:

– *index*, qui désignent directement le signifié (comme le doigt qui montre l'avion dans le ciel);

– *icônes*, qui l'évoquent physiquement (comme le tableau qui ressemble au paysage);

– *symboles*, qui n'ont avec lui aucun lien concret (comme le mot *rouge*, qui n'a pas en lui-même de couleur).

Les langues humaines utilisent ces trois sortes de signes, le dernier majoritairement; les langages de programmation, en revanche, n'ont que peu de véritables symboles: presque tous leurs signes évoquent immédiatement, du fait qu'ils sont empruntés au langage commun (*GO TO*, =, *article*, etc.), les signifiés correspondants. Les langages qui font exception à cette règle sont critiqués pour leur obscurité.

• Un **algorithme** est un procédé de calcul associé à une certaine fonction, permettant d'obtenir la valeur de cette fonction pour toutes les valeurs possibles de ses arguments (*données d'entrée*), et répondant aux trois conditions suivantes:

– il est décrit sous une forme telle que son interprétation ne souffre aucune ambiguïté;

– il est exprimé comme une combinaison d'opérations élémentaires;

– il est assuré de fournir, pour toute donnée d'entrée, un résultat après l'exécution d'un nombre fini d'opérations élémentaires.

Nous n'avons pas défini le terme *élémentaire*; en pratique, on pourra considérer qu'une opération élémentaire est une opération immédiatement traductible en une suite d'ordres pour une machine existante.

Pour illustrer les trois conditions de cette définition, on notera que les recettes de cuisine ne répondent pas au premier critère (*ajouter une pincée de sel* est insuffisamment précis); que l'énoncé *établir la comptabilité de l'entreprise X* n'est pas exprimé en termes d'opérations élémentaires; et qu'un procédé de calcul qui chercherait, par énumération, des entiers x, y, z et n tels que $n > 2$ et $x^n + y^n = z^n$ n'est pas assuré de se terminer, l'existence d'une solution ne pouvant être garantie (hypothèse de Fermat).

• Les **données** sont les informations manipulées par les algorithmes (pour une définition plus précise, cf § 2,2). Il convient de distinguer les *données d'entrée*, qui sont les objets fournis à un algorithme (tels des arguments à une fonction) pour qu'il calcule les résultats correspondants, et les *données internes*, utilisées dans le cours du calcul pour représenter les informations structurées nécessaires à son déroulement. La notion de **programme** est excellemment décrite par l'« équation » utilisée (en sens inverse) par Wirth comme titre de l'un des ses ouvrages [I.b. 39]:

Programmes = Algorithmes + Structures de données

qui introduit la dualité fondamentale de la programmation. La description des données a été historiquement négligée par les langages de programmation, au profit de celle, complémentaire, des calculs. Elle n'en est pas moins fondamentale en pratique, en particulier lorsque les programmes deviennent complexes et ambitieux. Nous verrons au paragraphe 5,1 que le programmeur peut jouer sur la dualité algorithme–données en faisant passer certains éléments d'une catégorie dans une autre.

• La possibilité d'**exécuter** les programmes sur un ordinateur est une exigence très forte, qui influe de façon déterminante sur les langages de programmation et oblige leurs concepteurs à un constant réalisme. La structure des ordinateurs actuels est en effet finalement très fruste, et les opérations de base relèvent d'un niveau d'abstraction fort éloigné des termes dans lesquels on se pose ordinairement un problème. Les langages de programmation jouent le rôle d'intermédiaire entre ces niveaux extrêmes, mais restent souvent prisonniers du plus bas. Nous verrons au paragraphe 5.3 les tentatives menées pour s'en abstraire dans les *langages de très haut niveau*.

• Il convient, inversement, de ne pas négliger la fonction de **compréhension** humaine mentionnée dans notre définition. Les langages de programmation servent à écrire des programmes, mais aussi – mais surtout – à les *lire*. Cet aspect, souvent mal apprécié dans les premiers langages de programmation, est aujourd'hui mieux pris en compte, en liaison avec la profonde remise en cause des idées sur la programmation intervenue depuis le début des années soixante-dix (cf article *La programmation structurée* et [I.b.16]). L'une des difficultés essentielles de la conception de langage (§ 3) est de concilier cet objectif avec le précédent, qui privilégie, parmi les différents modes d'expression possibles, ceux auxquels on peut immédiatement associer une représentation efficace sur machine.

1,2 NIVEAUX

On programme toujours dans un certain langage de programmation. La technique la plus sommaire, de plus en plus rare aujourd'hui, consiste à utiliser directement le *code machine*, en général binaire, directement interprétable par les circuits de l'ordinateur. En général, l'accès le plus direct à la machine se fait à un niveau déjà supérieur: on programme non la machine réelle que définissent ces circuits, mais une machine déjà plus évoluée, dite **machine virtuelle**, construite sur la machine réelle par l'intermédiaire d'un *microprogramme* (cf article *Conception des ordinateurs* dans ce traité). La microprogrammation permet d'adapter des circuits physiquement inchangés à des buts différents (sur la structure hiérarchisée des ordinateurs actuels, on consultera avec profit [I.b. 37]).

Le niveau de langage le plus bas parmi ceux qui sont largement utilisés en pratique est celui des **langages d'assemblage**, dont chaque ordre correspond directement soit à *une instruction* du code machine (plus généralement du code de la machine virtuelle microprogrammée), soit à la description d'*une donnée* élémentaire telle qu'elle est représentée en mémoire: un octet, un mot, etc.

Le terme *assembleur pour langage d'assemblage* est une impropriété à proscrire: l'assembleur est le programme de traduction, décrit au paragraphe 1.4.

Les langages d'assemblage sont encore couramment employés, de façon d'ailleurs sans doute exagérée. Leurs points d'ancrage principaux sont: l'écriture des systèmes d'exploitation (cf article *Écriture des systèmes informatiques. Problèmes de base et concepts fondamentaux* dans ce traité), pour laquelle il est nécessaire d'avoir accès à des caractéristiques directement liées au matériel; la programmation des applications de temps réel, pour lesquelles il est parfois important de pouvoir contrôler, à une unité près, le nombre d'instructions exécutées; et la programmation des micro-ordinateurs (cf article *Microprocesseurs* dans ce traité), en particulier lorsque, sur les modèles les plus récents, des langages de plus haut niveau ne sont pas encore disponibles.

L'emploi des langages d'assemblage souffre de plusieurs inconvénients: leurs liens étroits avec un seul type de machine (leur absence de *portabilité*, § 3,211), leur faible niveau d'abstraction, qui fait de la programmation une tâche longue, répétitive et fastidieuse, et le manque de fiabilité des programmes résultants, peu de contrôles automatiques étant possibles (§ 3,208).

Les langages dits couramment de *haut niveau* tentent de remédier à ces défauts en offrant des notations plus proches du mode de pensée humain. Leur «niveau» est en fait très variable; certains, comme Fortran ou Basic, n'offrent par rapport aux langages d'assemblage qu'une amélioration marginale, en se libérant essentiellement du premier défaut (l'association à une machine unique); d'autres, plus ambitieux, méritent mieux l'appellation traditionnelle.

1,3 LANGAGES GÉNÉRAUX. LANGAGES SPÉCIALISÉS

En prenant pour critère non plus le niveau linguistique, mais le champ d'application du langage, on distingue deux catégories:

– les **langages généraux**, permettant de résoudre en principe n'importe quel problème traitable mécaniquement;

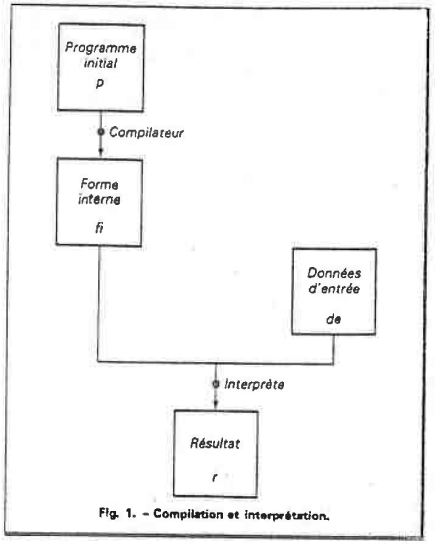


Fig. 1. – Compilation et interprétation.

– les **langages spécialisés**, destinés à un type particulier d'applications (exemples: commande de machines-outils, conception assistée par ordinateur, visualisation graphique, etc.).

Pour bien comprendre cette distinction, on remarquera que tout langage peut être considéré comme définissant une *machine virtuelle* mise à la disposition de l'utilisateur, et dont les *commandes* sont les constructions du langage. Les langages généraux correspondent à un ordinateur universel programmable dans une notation de niveau adéquat, les langages spécialisés à des machines spécifiques. La notion de langage spécialisé présente de nombreux liens avec celle de progiciel ou package (cf article *spécialisé* dans ce traité et § 5,1).

Cet article s'intéresse essentiellement aux langages généraux, cela pour trois raisons: d'une part, certains langages spécialisés sont traités dans la rubrique *Langages de programmation* de ce traité; d'autre part, les problèmes linguistiques de base sont les mêmes dans les deux cas; enfin, la tendance se fait jour, de plus en plus nettement, de concevoir les langages spécialisés comme des restrictions et/ou des extensions de langages généraux existants. Certains langages généraux, dont les plus notables sont Simula 67, APL et Ada, ont été explicitement conçus à cet effet.

1,4 MISE EN ŒUVRE

Dès qu'un programme n'est pas exprimé dans le code directement interprétable par la machine dont on dispose, il faut fournir un intermédiaire lui permettant de s'exécuter sur cette machine. La méthode de mise en œuvre universellement employée comprend deux étapes: compilation et interprétation (fig. 1).

On appelle **compilation** la traduction d'un programme, écrit dans un langage quelconque, en une forme interne plus proche du *code machine*. Elle est effectuée par un programme spécial appelé **compilateur** (ou **assembleur** dans le cas d'un langage d'assemblage).

L'**interprétation** est le calcul des résultats de ce programme, mis sous forme interne, lorsqu'on l'applique à un certain jeu de données d'entrée. Elle est effectuée par un mécanisme (logiciel, matériel ou mixte) appelé **interprète**.

Mathématiquement, ces deux étapes peuvent s'exprimer par:

$$\begin{cases} fi = comp\ (p) \\ r = int\ (fi,\ de) \end{cases}$$

avec *p* programme initial,
comp traduction effectuée par le compilateur,
fi forme interne de *p*,
de données d'entrée,
int calcul effectué par l'interprète,
r résultat.

Deux cas extrêmes sont:

– le mode **purement compilé**, où la forme interne est identique au code machine: l'interprète, purement matériel, est alors constitué par l'unité centrale de l'ordinateur;

– le mode **purement interprété**, où la forme interne est identique au langage de programmation, l'interprète étant purement logiciel: c'est à son tour un programme, mis en œuvre d'une façon ou d'une autre.

Cependant, dans presque tous les cas, les langages de programmation sont exécutés en mode mixte: compilation puis interprétation. Les poids respectifs de chacune de ces phases et le type de forme interne utilisé sont déterminés en fonction de plusieurs critères: caractéristiques du langage, efficacité recherchée, mode d'utilisation envisagé (interactif, incrémental, à distance).

Les compilateurs et interprètes sont étudiés dans l'article *Techniques de traduction* de ce traité; un ouvrage de référence, quelque peu vieilli mais non surpassé, est [l.b.19].

2 Ossature d'un langage

2,1 SYNTAXE ET SÉMANTIQUE

Les contrastes entre les langages de programmation sont aveuglants. Une étude un peu approfondie montre cependant que la plupart des langages se ressemblent en fait plus qu'ils ne diffèrent; le nombre d'ingrédients à combiner est assez faible. Ce sont ces ingrédients qui sont étudiés ci-après.

Un langage de programmation est défini dans deux champs duaux:

– la **syntaxe**, ensemble des règles gouvernant la formation des programmes;

– la **sémantique**, définition du sens associé aux programmes.

Il est impossible de dégager totalement l'étude de la sémantique de celle de la syntaxe. Cette dernière ne détermine pas seulement l'apparence externe d'un langage (et, par voie de conséquence, le goût ou le dégoût de l'utilisateur potentiel); elle est aussi le support concret de toutes ses caractéristiques sémantiques: c'est ainsi qu'une syntaxe inadaptée peut bloquer la compréhension, la diffusion et l'évolution d'un langage en rendant malaisée l'expression de certains concepts.

Si nous reprenons l'*«équation» de Wirth* (§ 1,1):

Programmes = Algorithmes + Structures de données

en la complétant par:

Système logiciel = Programmes + Modularisation

et en ajoutant qu'un langage de programmation permet d'écrire des systèmes logiciels dans un *formalisme* adéquat, nous obtenons les quatre éléments qui constituent l'ossature d'un langage:

– les **données** (objets manipulés par les programmes),
– les **calculs** (unités d'écriture des algorithmes),
– la **modularité** (mode de division des systèmes),
– la **syntaxe** (support visible de ces propriétés).

Nous examinerons successivement comment ces quatre catégories sont traitées par les langages de programmation.

2,2 DONNÉES

Par données nous entendons ici l'ensemble des **informations**, internes et externes, que les programmes peuvent **créer, utiliser, combiner, échanger et détruire.**

Un sens plus restrictif du mot donnée est celui d'information fournie *en entrée* à un programme; cette notion est désignée dans cet article par l'expression *donnée d'entrée*. L'une et l'autre acception du mot donnée sont couramment employées, non sans ambiguïté.

Toute donnée d'un programme est caractérisée par un certain nombre d'**attributs**, dont les plus importants sont:

– les *droits* des programmes sur cette donnée;

– le *type* de la donnée, c'est-à-dire l'ensemble de valeurs auquel elle se rattache et les opérations possibles correspondantes.

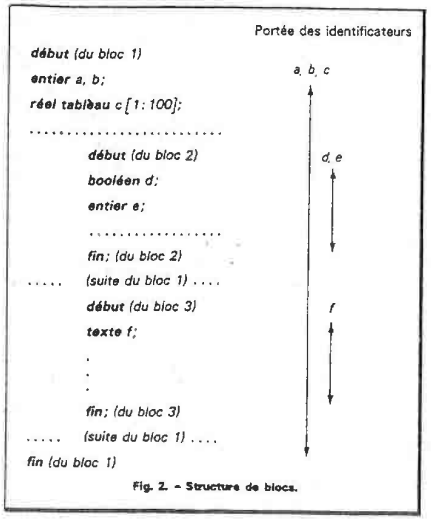


Fig. 2. – Structure de blocs.

2,21 Droits d'accès: constantes et variables.

Les **droits d'accès** définissent le *pouvoir* d'un programme sur une donnée, en particulier le pouvoir de la modifier. Deux cas simples sont:

– la **constante**, dont le programme peut utiliser la valeur, mais non la modifier; une constante peut être désignée par sa valeur même (exemple: *3,141 592*) ou par un nom symbolique dit *identificateur* (exemple: *pi*);

– la **variable**, dont la valeur peut être modifiée au cours du déroulement du programme grâce à une instruction dite *affectation*; une variable est désignée par un identificateur.

Ces deux catégories ne représentent cependant pas toute la palette des droits que l'on peut décider d'accorder ou non à un programme sur une donnée.

• La liste des **opérations permises** et des opérations interdites peut être plus complexe. Dans le cas d'une donnée composée comportant plusieurs attributs (exemple: une donnée représentant un compte bancaire, dont les attributs sont le numéro du compte, le nom de son titulaire, le solde courant, etc.), on peut autoriser un certain programme à modifier certains attributs (exemple: le taux de découvert autorisé) mais non d'autres (exemple: le solde courant).

• Une donnée peut être rendue accessible à **plusieurs unités de programmes.** Cet effet est par exemple obtenu en Fortran en indiquant qu'elle appartient à une zone commune (*COMMON*); dans les langages de la série Algol, en PL/I, etc., un programme peut contenir un ou plusieurs autres éléments de programme qui accèdent alors à ses données (*structure de blocs*: fig. 2). Certains langages permettent d'attribuer des droits différents sur une même donnée aux programmes qui la partagent.

• Le problème se complique à nouveau lorsque les programmes partageant une donnée sont susceptibles de s'**exécuter simultanément** (*cf* article *Écriture des systèmes informatiques. Problèmes de base et concepts fondamentaux* dans ce traité et [l.b. 13]). Ainsi, une donnée pourra représenter une ressource physique qui doit être affectée pendant chaque phase du calcul à un seul programme, telle une imprimante sur laquelle on ne désire certainement pas mélanger les lignes envoyées en sortie par plusieurs travaux; on devra associer à des données de ce type une politique particulière de *réservation* et de *libération*.

2,22 Notion de type.

Outre les droits que les programmes exercent sur elle, la caractéristique principale d'une donnée est son **type**.

On définit un type comme la conjonction de deux éléments:

– un ensemble de *valeurs* possibles;

– un ensemble fini d'*opérations* s'appliquant aux éléments du type et possédant des propriétés connues.

Exemple: le type *entier* se caractérise par un ensemble de valeurs possibles (cet ensemble est fini dans une représentation sur machine), par les opérations *addition, comparaison*, etc., et par les propriétés de ces opérations (associativité, relation d'ordre, etc.).

Le type *liste linéaire* a pour valeurs des suites finies et pour opérations l'insertion d'un nouvel élément, le test déterminant si une liste est vide, etc. (Pour de plus amples développements sur ce mode de caractérisation des types, consulter [l.b. 20 et 30 (chapitre V)].)

Tous les langages offrent des types de **base**, prédéfinis, et des **procédés de construction**, simples ou évolués, permettant d'obtenir des objets de types plus complexes.

2,23 Types prédéfinis.

Parmi les types prédéfinis offerts par les différents langages, les plus courants sont:

– les **types numériques**, qui fournissent une approximation finie des ensembles mathématiques correspondants; entiers, réels (en fait un sous-ensemble fini des rationnels), complexes;

– le **type logique** ou **booléen**, à deux valeurs notées *vrai* et *faux*, servant à représenter les critères de décision dans le déroulement des algorithmes;

– le **type chaîne de caractères**, permettant de manipuler des caractères ou des suites de caractères;

– le **type programme**, dont les éléments désignent des sous-programmes, des fonctions, des instructions (étiquette); les opérations permises sur les objets de ce type, lorsqu'il est présent, sont en général restreintes (passage comme argument à un sous-programme, affectation).

2,24 Données composées.

2,240. Un programme est un modèle d'un certain système physique, qui peut être fort complexe; dans presque tous les cas, des objets appartenant aux types élémentaires ci-dessus ne suffiraient pas à décrire un tel système de façon satisfaisante. Il faut pouvoir

Pour trouver un renseignement, consulter la table alphabétique H 12.

introduire des données composées, de structure plus élaborée. Les langages offrent à cet effet divers mécanismes de construction.

Ces mécanismes sont de deux sortes.

• Certains permettent la définition d'*objets individuels composés*. Ainsi, en Pascal, la déclaration

[Pascal] ***var z: article re: REEL; im: REEL fin***

définit une variable *z* qui désignera un objet (*article* ou *record* en anglais) à deux composantes, toutes deux de type réel, appelées respectivement *re* et *im*.

• D'autres mécanismes, plus généraux, n'introduisent eux-mêmes aucun nouvel objet mais décrivent un nouveau *type*, c'est-à-dire un *modèle* auquel se conformeront tous les objets d'une certaine classe. Ainsi, toujours en Pascal, la déclaration

[Pascal] ***type COMPLEXE = article re: REEL; im: REEL fin***

définit un nouveau type dont chaque élément aura la même structure que *z* précédemment. *z* lui-même sera alors plutôt déclaré par

[Pascal] ***var z: COMPLEXE***

Parmi les langages de programmation, certains n'offrent que des mécanismes de construction applicables à la déclaration d'objets individuels (Fortran, Cobol, PL/I); d'autres permettent exclusivement de définir de nouveaux types (Algol W et Simula 67, en écartant les tableaux); d'autres encore admettent les deux possibilités (Algol 68, Pascal, Ada).

Les principaux mécanismes de construction de structures de données composées sont cités ci-après; pour plus de détails, on pourra se reporter aux références [l.b. 20, 23, 30 (chapitre V) et 39].

2,241 Tableaux. – Ce sont des groupements finis de données de même type, en nombre fixe pour la durée d'une exécution de la section de programme à laquelle elles appartiennent. Cette structure de données correspond étroitement à la structure linéaire des mémoires couramment employées; elle est présente dans presque tous les langages (une exception notable est Lisp dans sa version dite *pure*). Pour certains, comme Fortran et Basic, elle constitue le seul mécanisme de description de données complexes.

Exemple: la notation suivante définit en Fortran un tableau d'entiers à deux dimensions (matrice), indexé de 1 à 10 pour la première et de 1 à 1 000 pour la seconde:

[Fortran] *INTEGER T (10,1 000)*

Ce tableau contient 10 000 éléments qui peuvent être désignés individuellement par la notation *T (i,j)* avec $1 \leqslant i \leqslant 10$ et $1 \leqslant j \leqslant 1000$.

2,242 Fichiers séquentiels. – Ce sont des suites finies d'objets de même type, en nombre en général inconnu du programme à priori.

Exemple: l'écriture suivante en Cobol indique que le programme utilisera un fichier *standard* appelé *FACTURES*.

[Cobol]
```
FD  FACTURES
    LABEL RECORDS ARE STANDARD
    DATA RECORDS ARE LG1, LG2, LG3, LG4.
```

La structure des éléments du fichier doit être fournie par ailleurs à des lignes étiquetées *LG1, LG2, LG3, LG4*.

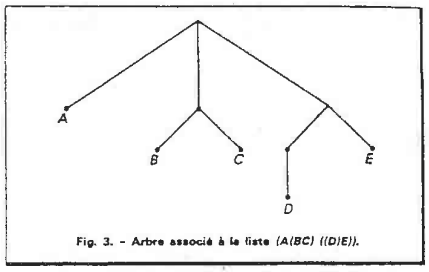


Fig. 3. – Arbre associé à la liste *(A(BC) ((D)E))*.

2,243 Enregistrements ou articles. – Ce sont des groupements finis de données en nombre généralement fixe, de types éventuellement différents, désignées chacune par un nom. Dans certains langages (Cobol, PL/I, Pascal, Algol 68), il s'agit d'une structure hiérarchique, chaque élément pouvant à son tour être un enregistrement.

Exemple: la *structure* PL/I suivante est un enregistrement à trois niveaux:

[PL/I]
```
1 ADRESSE,
  2 RUE CHARACTER (30),
  2 VILLE,
    3 CODE_POSTAL BINARY FIXED (5),
    3 NOM_DE_VILLE CHARACTER (20);
```

Elle définit une adresse comme se composant d'une rue (chaîne de caractères) et d'une ville, laquelle comprend elle-même un code postal (entier sur cinq chiffres) et un nom de ville (chaîne de caractères).

2,244 Types énumérés. – Introduits par Pascal, ils ont pour valeurs des objets appartenant à un ensemble fini, donné par l'énumération des noms symboliques de ses éléments.

Exemple: on peut définir en Pascal l'ensemble des sujets des traités des Techniques de l'Ingénieur en écrivant:

[Pascal]
***type** SUJET = (Généralités, Plastiques, Mécanique_et_Chaleur, Construction, Électrotechnique, Électronique, Informatique, Génie_chimique, Constantes, Métallurgie, Analyse_chimique_et_Caractérisation, Mesures_et_Contrôle, Conception_des_produits_industriels)*

2,245 Types ensembles. – Ils ont pour valeurs les sous-ensembles d'un ensemble fini donné.

Exemple: étant donné le type *SUJET* précédent, on peut définir en Pascal le type de ses sous-ensembles par:

[Pascal] ***type** ABONNEMENT = **ensemble de** SUJET*

2,246 Type liste. – Il admet pour éléments des suites finies d'objets dont chacun est soit un *atome*, soit à son tour une liste.

Exemple: la liste suivante de Lisp

[Lisp] *(A (B C) ((D)E))*

possède trois éléments: un *atome A*; une liste *(B C)* à deux éléments atomiques; une liste *((D)E)* à deux éléments dont l'un est une liste à un élément et l'autre un atome. Cette structure correspond à celle d'un *arbre* (fig. 3).

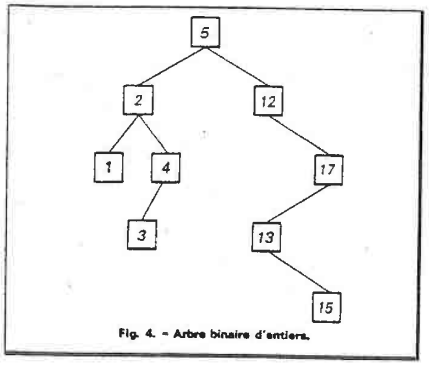


Fig. 4. – Arbre binaire d'entiers.

De telles structures jouent un rôle fondamental en calcul symbolique (cf article *Langages de manipulation de symboles* de ce traité), en intelligence artificielle, etc.

2,247 Type pointeur. – Il admet des éléments qui prennent pour valeur des *noms d'éléments* d'un type donné. Cette notion correspond à celle d'adresse au niveau de la mémoire. Elle permet de définir des structures chaînées (graphes, listes linéaires, arbres, etc.).

Exemple: l'écriture Algol 68 suivante définit un type d'enregistrements contenant des pointeurs, désignés par *ref*, vers des objets de même type:

[Algol 68] ***mode** ARBRE_BINAIRE =*
***struct (integer** info, **ref** ARBRE_BINAIRE gauche, droit)*

Les objets de ce type sont connus en programmation comme des *arbres binaires* d'entiers. Il s'agit d'une structure de données (telle que celle représentée sur la figure 4) permettant de ranger commodément des informations. A chaque élément est associée une information (ici une valeur entière); il peut en outre désigner deux autres éléments: son *fils gauche* et son *fils droit* (l'un et l'autre peuvent être absents); se référer par exemple à [l.b. 27, chapitres V et VII].

2,3 CALCULS

Sur les objets que nous venons d'évoquer, les programmes effectuent, pour réaliser un certain algorithme, des **calculs**, ce terme devant être pris dans un sens très large: calculs numériques, formels, textuels, etc. Pour analyser les mécanismes offerts par les langages de programmation pour exprimer ces calculs, on est amené à considérer deux oppositions orthogonales:

– les possibilités offertes se partagent entre la description des calculs élémentaires (*opérations*) et des mécanismes de combinaison d'actions (*structures de contrôle*);

– les calculs peuvent être décrits sous une forme prescriptive (*instructions*) ou implicite (*expressions*).

Nous examinerons successivement ces deux divisions.

2,31 Opérations. Structures de contrôle.

• **Les opérations élémentaires** s'appliquent à un ou plusieurs objets et permettent d'en examiner et/ou d'en modifier certains attributs.

Exemples.

[Cobol] *MULTIPLY A BY B GIVING C*
Effet: multiplie *A* par *B* pour donner *C*

[Lisp] *(CAR L)*
Effet: fournit le premier élément de la *liste L* (où *L* est une liste de forme indiquée au paragraphe 2,246). Pour la liste de la figure 3, le résultat est *A*.

[Algol, Simula, Pascal, etc.] $X := Y$
[Fortran, Basic, PL/I] $X = Y$
Effet: donne à la variable *X* la valeur de la variable *Y* (instruction d'affectation)

[APL] $X \leftarrow \Gamma T$
Effet: fournit l'élément maximal du tableau *T*.

• **Les structures de contrôle** permettent de combiner des opérations – élémentaires ou non – pour constituer de véritables algorithmes, c'est-à-dire des modèles décrivant des suites d'actions avec une structure de décision éventuellement complexe.

Les structures de contrôle définissent le déroulement des calculs élémentaires indépendamment de la nature de ceux-ci. Wilkes [l.b. 38] parle de syntaxe *externe* des langages de programmation, par opposition à leur syntaxe *interne*. Dans

***répéter** 20 000 **fois** A*

la partie en gras appartient à la syntaxe externe; c'est une structure de contrôle dont le sens est indépendant du fait que *A* est *ajouter 1 à la variable I*, *lancer une fusée* ou *lire un élément du fichier F*.

Les linguistes (Sapir, Jakobson) distinguent de même, dans le langage humain, les *concepts matériels* et les *concepts relationnels* [l.b. 25].

Exemple: soient *c* une condition (un objet prenant des valeurs booléennes), *I1*, *I2*, *I3* des instructions. Alors la plupart des langages de programmation offrent des notations permettant de décrire les trois structures de contrôle suivantes:

• *I1; I2; I3*
(décrit un algorithme dont l'exécution est celle de *I1*, suivie de celle de *I2*, suivie de celle de *I3*);

• ***si** c **alors** I1 **sinon** I2*
(décrit un algorithme dont l'exécution est celle de *I1* si la condition *c* est vérifiée, celle de *I2* dans le cas contraire);

• ***tant que** c **répéter** I1*
(décrit un algorithme dont l'exécution est celle de *I1*, répétée aussi longtemps que *c* est vraie, peut-être jamais).

Les trois structures de contrôle de cet exemple sont appelées **enchaînement**, **choix** et **boucle**. Dans de nombreux langages, elles sont complétées, ou remplacées, par l'instruction de branchement prescrivant de poursuivre l'exécution du programme à un emplacement déterminé. L'un des principes les plus connus de la méthode de programmation structurée (cf article spécialisé dans ce traité) est d'éviter, pour des raisons de lisibilité et de fiabilité des programmes, l'emploi de l'instruction de branchement, au profit de structures fondées sur les trois précédentes.

Il convient de noter que la distinction entre opérations et structures de contrôle n'est pas absolue, mais dépend en partie du niveau d'abstraction du langage. Ainsi, pour examiner une donnée complexe d'un certain type, il faudra une boucle dans certains langages; d'au-

[l.b. *n*] renvoie à la référence *n* de l'index bibliographique placé en fin d'article.

tres permettront d'obtenir le même résultat au moyen d'une opération considérée comme élémentaire.

Exemple: la somme de deux matrices V et W est notée $V + W$ en APL ou en PL/I. Dans la plupart des autres langages, il faut la calculer terme à terme par une boucle (on peut cependant *cacher* cette boucle dans un sous-programme : § 2,41).

Parmi les structures de contrôle importantes, il convient encore de mentionner :

– la **récursion**, qui permet de décrire un algorithme en termes du même algorithme appliqué à des données plus simples, comme dans la définition suivante du coefficient C_n^m du triangle de Pascal :

$$C_n^m = \textit{si } m = 0 \textit{ alors } 1$$
$$\textit{sinon si } n = 0 \textit{ alors } 0$$
$$\textit{sinon } C_{n-1}^m + C_{n-1}^{m-1}$$

– les **structures de synchronisation**, qui permettent de décrire la coopération de processus qui se déroulent en parallèle (cf article *Écriture des systèmes informatiques. Problèmes de base et concepts fondamentaux* de ce traité).

2,32 Instructions. Expressions.

Pour exprimer les manipulations effectuées sur les données, les langages de programmation hésitent entre deux philosophies. L'une, que l'on peut appeler **prescriptive**, considère l'algorithme comme une suite d'ordres, ou *instructions*, que doit exécuter un automate. L'autre, plus **implicite**, le présente comme la description statique, sous forme d'*expressions*, du résultat cherché.

Exemples : l'écriture Cobol

[Cobol] *MULTIPLY A BY B GIVING C*

est plus prescriptive que l'expression $A * B$ dans l'instruction d'affectation suivante :

[Fortran] $C = A * B$

La récursion est plus implicite, l'itération (emploi de boucles) plus prescriptive. On comparera ainsi dans le tableau I deux modes de calcul d'un même résultat en Algol 68 (cf article spécialisé dans ce traité; la syntaxe devrait cependant être claire sans explication particulière : ***sinsi*** signifie ***sinon si***).

La forme implicite est en général plus proche du problème initial, la forme prescriptive de la mise en œuvre finale sur l'ordinateur. Dans le cas d'un problème de nature mathématique, la forme implicite a l'avantage d'être statique, c'est-à-dire indépendante de tout ordre d'exécution, et donc de relever des raisonnements mathématiques classiques sur les fonctions et les formules. Par contre, la forme prescriptive est plus directement orientée vers la machine; elle pose donc souvent moins de problèmes de compilation, mais sa compréhension fait intervenir la notion de temps, et la démonstration de validité des programmes (§ 3,34) y est moins aisée.

Presque tous les langages incluent à la fois la notion d'expression et celle d'instruction. Certains favorisent nettement la forme prescriptive (Fortran, Cobol, Pascal, etc.); d'autres, au contraire, la forme implicite (Lisp et dérivés, langages dits *fonctionnels*). D'autres enfin maintiennent l'équilibre entre les deux approches : c'est le cas des langages (Algol W et surtout Algol 68) qui confondent, en théorie, instructions et expressions. L'unité de base du langage, appelée **clause** en Algol 68, donne alors à l'exécution à la fois un *effet* et une *valeur*. Ainsi, l'affectation

$$x := 3$$

aura pour effet de modifier la valeur associée à la variable x, et pour valeur celle qui est affectée à x, soit 3. On notera que la version itérative de *combinaison*, dans l'exemple du tableau I, fait suivre une boucle (instruction) d'une expression, *c[n,m]*, dont la valeur est celle que la procédure renvoie.

2,4 MODULARITÉ

2,40. Les éléments de base esquissés précédemment – données, calculs – permettent, en théorie, de construire des programmes quelconques. En pratique, ils ne suffisent pas : un programme est une construction humaine complexe et, pour être compris et maîtrisé, il doit être découpé en éléments plus simples – comme un livre en chapitres, sections et paragraphes, ou une ville en quartiers, îlots et immeubles.

Tableau I. – Récursion et itération en Algol 68.

Récursion

```
[Algol 68]
proc combinaison =
     (ent n, m) ent:
          si m = 0 alors 1
          sinsi n = 0 alors 0
          sinon combinaison (n–1, m–1)
               + combinaison (n–1, m)
          fsi
```

Itération

```
[Algol 68]
proc combinaison =
     (ent n,m) ent:
          début
          [0:n,0:m] ent c; # tableau #
          pour j depuis 1 jqà m faire c [0,j] := 0 fait;
          pour i depuis 1 jqà n faire
               c[i,0] := 1;
               pour j depuis 1 jqà m faire
                    c[i,j] := c[i–1,j–1] + c[i–1,j]
               fait
          fait;
          c [n,m] # valeur renvoyée #
          fin
```


L'unité de découpage d'un système logiciel est appelée **module**. On distingue deux sortes de modules :

– un module qui correspond à une étape du calcul est appelé **sous-programme** ;

– un module qui correspond à un sous-ensemble des données est (de façon moins universelle) appelé **classe**.

Le premier mode de découpage est le plus généralement présent. Nous verrons cependant qu'il n'est pas entièrement satisfaisant.

2,41 Sous-programmes.

Un sous-programme est un élément de programme destiné à être utilisé par d'autres éléments de programme, qui lui confieront une tâche à effectuer ou des valeurs à calculer.

Un sous-programme utilise des **données d'entrée** fournies par les éléments de programme qui l'utilisent (qui l'**appellent**), et leur renvoie en sortie des **résultats**. Données d'entrée et résultats constituent l'ensemble des **arguments** du sous-programme ; certains arguments peuvent figurer à la fois en entrée et en sortie (**données modifiées**). Dans la plupart des langages, les arguments d'un sous-programme sont en nombre fixe ; d'autres (Ada, Pop2, JCL) permettent au contraire l'omission de certains arguments au cours d'un appel si des valeurs par défaut ont été prévues dans le sous-programme.

Un sous-programme est défini dans une **déclaration de sous-programme**, comprenant une *tête* qui fournit la liste des arguments, désignés par des noms locaux (dits *arguments formels*), et un *corps*, suite d'instructions et/ou d'expressions définissant les calculs à effectuer.

Un appel de sous-programme désigne celui-ci par son nom suivi, s'il y a lieu, d'une liste d'objets (variables, constantes, expressions) dits *arguments réels*, qui correspondent élément par élément aux arguments formels.

Cette correspondance est en général assurée par l'énumération des arguments réels dans l'ordre des arguments formels. Une autre méthode possible est d'associer à chaque argument formel un *mot-clé* distinctif, qui devra être énoncé avec l'argument réel correspondant. L'ordre des arguments réels est alors quelconque ; il est plus facile avec cette méthode d'offrir la possibilité de ne pas inclure certains arguments réels, en ayant recours à des valeurs par défaut.

Un appel de sous-programme peut être exprimé à l'aide d'un verbe *appeler* explicite :

[Fortran, PL/I] *CALL LIRFIC (A,B,3,X+5)*

[Cobol] *CALL'LIRE_FICHIER' USING A B 3 X+5*

ou par la simple mention du nom du sous-programme :

[Algol, Pascal, Simula, etc.] *lire_fichier (a,b,3 x+5)*

Ce dernier mode est utilisé dans tous les langages pour les sous-programmes dits de type *expression* ou *fonction*, qui calculent en fonction de leurs arguments une valeur unique (éventuellement composée) pouvant être utilisée dans une expression :

distance (p1, p2)

premier_caractère_alphabétique (TEXTE1)

occupé (IMPRIMANTE1)

Notons que, malgré leur nom, ces «fonctions» ne sont pas assurées de répondre, sauf dans quelques langages particulièrement sourcilleux de ce point de vue (Ada), aux caractéristiques bien connues de leurs homologues mathématiques. Les sous-programmes qui leur sont associés peuvent en effet, avant de calculer la valeur demandée, effectuer diverses instructions ; il ne sera donc pas toujours vrai en programmation que

$$a=b \Rightarrow f(a)=f(b)$$

Exemple : la *fonction* Fortran suivante calcule une valeur entière égale à celle de son argument augmentée de 1 :

```
[Fortran]
          INTEGER FUNCTION F (I)
          INTEGER I
          I = I + 1
          F = I
          RETURN
          END
```

Mais elle modifie en outre la valeur de son argument. Soit A une variable entière de valeur 0. Alors les expressions :

$$F(A) + F(A)$$
$$\text{et } 2 * F(A)$$

seront respectivement évaluées, sur beaucoup de systèmes, à 3 et 2 ; elles sont donc différentes, contrairement à toutes les règles mathématiques.

De tels *effets de bord* sont évidemment préjudiciables à la fiabilité des programmes (§ 3,208).

2,42 Classes.

Le sous-programme, unité de division de modules la plus couramment offerte par les langages de programmation, ne fournit pas toujours un mode de découpage satisfaisant. Un programme complexe peut être considéré comme un système (au sens de la théorie des systèmes), formé à la base d'éléments (les données) et de relations entre ces éléments (les calculs). Le but de la modularisation est de le diviser en sous-systèmes aussi cohérents que possible, c'est-à-dire communiquant par un petit nombre de liens explicites. Un découpage en sous-programmes, organisé autour des étapes du calcul, n'est pas nécessairement le meilleur à cet effet ; il souffre en effet de plusieurs défauts.

- Si l'on considère l'évolution d'un programme, correspondant à celle de ses spécifications, il est fréquent que la nature des objets de base reste plus stable que celle des relations qui les lient. Or, l'un des critères d'une programmation modulaire est précisément de faire en sorte qu'à une petite modification des spécifications corresponde une modification restreinte dans le programme.

- Outre la communication explicite par les appels, les sous-programmes interagissent implicitement par l'accès à des données partagées. En Fortran, par exemple, deux sous-programmes *A* et *B*, dont aucun n'appelle l'autre directement ni indirectement, pourront être en réalité fortement couplés du fait que *A* modifie une donnée placée dans une zone commune et utilisée ultérieurement par *B*. Ce couplage caché est un obstacle important à la cohérence et à l'intégrité des modules.

- Un grand nombre de programmes (systèmes de gestion de base de données, systèmes d'exploitation, programmes en temps réel, programmes interactifs comme les éditeurs de textes ou les programmes d'interrogation de fichiers) n'effectuent pas à proprement parler un traitement déterministe unique dont le début et la fin sont clairement définis ; ils remplissent plutôt, en réponse à des requêtes dont

Pour la table analytique, se reporter à la première page de cet article.

l'ordre est a priori inconnu, un ensemble de fonctions d'accès à certaines données. Regrouper les fonctions autour des données auxquelles elles se rapportent est, dans de tels cas, plus naturel que l'inverse.

• Une même fonction du programme requiert souvent, pour pouvoir être menée à bien répétitivement, une phase d'initialisation. Dans un programme de gestion, par exemple, les sous-programmes d'accès à un fichier (lecture et écriture) ne fonctionneront que si le fichier a été *ouvert* au début de l'exécution du programme. De même, dans un compilateur, l'*analyseur lexical*, qui décode les mots successifs, doit lire un caractère d'avance, et l'initialisation doit donc inclure la lecture du premier caractère. Dans un découpage en sous-programmes, l'initialisation et le cas général seront traités par des sous-programmes différents, alors qu'ils sont logiquement liés à deux aspects d'une même fonction.

Un mode de découpage des programmes qui permet de pallier ces défauts a été introduit par le langage de programmation Simula 67 [I.b. 9 et 31], et repris sous des dénominations diverses, avec des changements d'éclairage, par de nombreux langages expérimentaux (Alphard, Clu, Euclid, etc.). Simula reste, en attendant Ada (§ 4,3), le seul langage de large diffusion incluant une telle structure, et nous lui empruntons le terme de *classe*.

Une **classe** est une structure de programme permettant de regrouper dans un même module:
- des données (variables, etc.);
- des sous-programmes;
- des instructions d'initialisation.

Des exemples d'application de la notion de classe sont:

– la représentation d'un type (§ 2,22): les données sont la représentation des éléments du type, et les sous-programmes correspondent aux opérations intervenant dans sa définition (la figure 5 donne une représentation d'une pile conforme à ce principe en Simula);

– le regroupement dans un même module d'un ensemble de données (zone *COMMON* en Fortran, descriptif d'un fichier en Cobol), des opérations d'accès à ces données, et de leur initialisation;

– le regroupement des éléments relatifs à une certaine ressource physique, comme un terminal graphique: données qui caractérisent son état à un instant donné, et sous-programmes associés à chaque fonction offerte;

– l'écriture d'un élément de programme assurant une fonction qui requiert une initialisation, et doit conserver les valeurs de certaines variables entre les appels successifs; tel est le cas d'un analyseur lexical, ou encore d'un module calculant élément après élément une suite de valeurs pseudo-aléatoires, dont le premier élément est déterminé à l'initialisation du module, en fonction d'une valeur fournie à celui-ci.

Les différents langages offrant une structure de type *classe* varient sur les détails (caractère statique ou dynamique d'une définition de classe, problèmes de visibilité et de protection, possibilité de séparer spécification et représentation, lien avec la compilation séparée, etc.). Le principe général reste cependant le même, et la classe fournit dans tous les cas une méthode de modularisation efficace, destinée sans nul doute à jouer un rôle de premier plan, aux côtés du sous-programme, dans les langages futurs.

On notera que d'autres langages (Fortran dans certaines de ses versions, PL/I, langages d'assemblage) permettent d'obtenir sous une forme très restreinte des structures se rapprochant quelque peu de la classe, par l'emploi de sous-programmes ayant plusieurs *entrées*.

```
class pilentier (n); integer n;
    begin
    comment attributs;
        comment variables;
                integer array p (1 : n);
                integer sommet;
        comment procédures;
                procédure empiler (x); integer x;
                    if sommet = n then
                    erreur _ pile _ pleine
                    else
                        begin sommet :=
                              sommet + 1;
                              p (sommet) := x
                        end empiler;
                integer procedure dépiler;
                    if pilevide then erreur_pile_vide
                    else
                        begin dépiler := p (sommet);
                        sommet := sommet - 1;
                        end dépiler;
                boolean procedure pilevide;
                    pilevide := (sommet = 0);
    comment action d'initialisation (à la création d'une pile
                                     initialement vide);
        sommet := 0
    end pilentier
```

Fig. 5. – Module de gestion de pile en Simula 67.

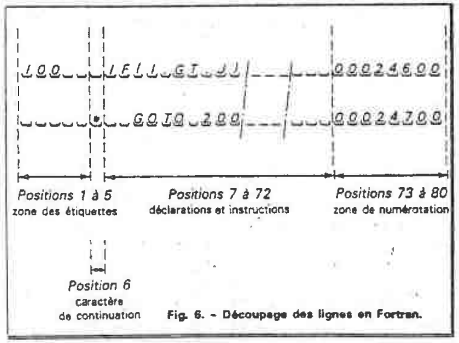


Fig. 6. – Découpage des lignes en Fortran.

2,5 SYNTAXE

La très grande diversité des notations employées dans les langages de programmation est, pour une part, due à des raisons historiques ou circonstancielles. Ainsi, le format rigide de Fortran, relatif à des lignes qui sont des images de cartes perforées (fig. 6), est un vestige du temps de la mécanographie, qui détonne à l'heure de la télématique et des terminaux conversationnels.

Plus profondément, cependant, les différences de forme traduisent des attitudes divergentes dans la conception des langages, et des philosophies différentes de la programmation. Six **grandes tendances** nous paraissent se dégager en ce domaine:

– l'**école mathématique** cherche à ramener la syntaxe des langages de programmation à des notations respectant l'esprit, sinon la lettre, de l'écriture mathématique classique, telle qu'elle s'applique en particulier aux expressions;

– l'**école naturaliste** fuit au contraire la mathématique comme la peste, et veut fournir des notations «naturelles», proches du langage courant;

[Fortran]

```
      SUBROUTINE IMPSC (U,V,M)
      INTEGER M
      REAL U(M), V(M)
C     IMPRIMER LA SOMME DES TERMES
C     DE U APPARAISSANT DANS V
      INTEGER I,J
      REAL SOM
      SOM=0.
      I=0
100   I=I+1
      IF(I.GT.M)GOTO 1000
      J=0
200   J=J+1
      IF(J.GT.M)GOTO 100
      IF(U(I).NE.V(J)) GOTO 200
      SOM=SOM+U(I)
      GOTO 100
1000  PRINT 90000,SOM
      RETURN
90000 FORMAT (1X,E12.5)
      END
```

Mode d'appel pour deux tableaux *A* et *B* de taille *N*:

```
CALL IMPSC(A,B,N)
```

[Algol W]

```
PROCEDURE IMPRIMER_SOMME_COMMUNS
    (REAL ARRAY U,V(•); INTEGER VALUE M);
    BEGIN COMMENT: IMPRIMER LA SOMME DES
                   TERMES DE U APPARAISSANT
                   DANS V;
    LOGICAL PROCEDURE APPARTIENT
            (INTEGER VALUE X);
        BEGIN COMMENT: U (K) APPARAIT-IL DANS V?;
        INTEGER J; J := 1
        WHILE J <= M AND U (K)¬= V(J) DO
              J := J+1;
        J <= M
        END;
    REAL SOMME; SOMME := 0;
    FOR I := 1 UNTIL M DO
        IF APPARTIENT (I) THEN
            SOMME := SOMME + U(I);
    WRITE (SOMME)
    END
```

Mode d'appel pour deux tableaux *A* et *B* de taille *N*:

```
IMPRIMER_SOMME_COMMUNS(A,B,N)
```

[Lisp] (version 1.6)

```
(DE IMPSOMCOM (U V)
   (PRINT (SOMME (COMMUNS U V)))
)
(DE SOMME (L)
    (COND ((NULL L) 0)
          (T (PLUS (CAR L) (SOMME (CDR L))))
    )
)
(DE COMMUNS (P Q)
    (COND ((NULL P) NIL)
          ((APPARTIENT (CAR P) Q)
               (CONS (CAR P)
                     (COMMUNS (CDR P) Q)))
          (T (COMMUNS (CDR P) Q))
    )
)
(DE APPARTIENT (X L)
    (AND  (NOT (NULL L))
          (OR (EQ X (CAR L))
              (APPARTIENT X (CDR L))
          )
    )
)
```

Mode d'appel pour deux listes *A* et *B*:

```
(IMPSOMCOM A B)
```

Note: *APPARTIENT* peut être remplacé par la fonction prédéfinie *MEMBERP*

[APL]

```
∇ U ISC V
  □← + /U/U ∈ V
∇
```

Mode d'appel pour deux tableaux *A* et *B*:

```
A ISC B
```

Problème: soient deux suites de réels *a* et *b* de même longueur. Ecrire un sous-programme qui imprime la somme de tous les éléments de *a* qui apparaissent aussi dans *b* (comptés autant de fois qu'ils apparaissent dans *a*).

Fig. 7. – Diversité syntaxique des langages.

– l'**école malthusienne** tient qu'un langage doit être engendré par un nombre aussi restreint que possible de notations de base; le but principal est l'économie des moyens;

– l'**école libérale** (ou *physiocratique*, ou du *laisser-faire*) se situe à l'opposé du malthusianisme: elle vise à offrir à l'utilisateur potentiel du langage des notations abondantes et diverses, en lui laissant la plus grande liberté;

– l'**école laconique** veut avant tout permettre une expression concise, jusqu'à la sécheresse, de programmes même complexes;

– l'**école mécaniste**, enfin, calque la structure des langages sur celle des ordinateurs (et, plus précisément, de leurs codes d'instruction), le but suprême étant l'implantation efficace sur les machines existantes.

Les langages de la série Algol, ainsi qu'APL qui possède également de fortes tendances physiocratiques et laconiques, sont de bons représentants de l'école mathématique. Fortran hésite entre le mathématique et le mécaniste. Lisp est un langage très malthusien à hérédité mathématique. Snobol, qui dans sa version la plus épurée n'a qu'un type d'instruction, est particulièrement malthusien. Basic et les langages d'assemblage sont tout à fait mécanistes, mais Basic est bien plus laconique. Cobol et les *langages pour non-informaticiens* (§ 5,2) se veulent naturalistes. PL/I et les langages extensibles poursuivent des buts libéraux, et fort naturalistes dans le premier cas.

L'exemple de la figure 7 illustre quelques-uns des types de notation offerts par les différents langages. Le problème commun traité par les quatre programmes proposés est le suivant: étant donné deux suites finies de réels $a = (a_1, a_2, \ldots, a_n)$ et $b = (b_1, b_2, \ldots, b_m)$ de même longueur, imprimer la somme de tous les éléments a_i de a qui apparaissent aussi dans b.

3 Conception et choix d'un langage

3,1 ENJEU

La conception des langages de programmation est le type même d'activité qui démultiplie le travail d'un individu, ou d'un petit groupe, par un facteur considérable en cas de succès: les utilisateurs pourront à terme se compter par dizaines de milliers.

L'examen des principaux problèmes que soulève cette activité présente cependant un grand intérêt pour un cercle bien plus large que le petit groupe des concepteurs de langages généraux nourrissant l'ambition de survivre durablement. Il est en effet précieux pour tout technicien ayant à réaliser des **mises en œuvre** (compilateur, interprète) de langages existants, pour tout utilisateur amené à choisir un langage en vue d'une classe d'applications, et, plus généralement encore, pour presque tous les programmeurs, si l'on veut bien reconnaître qu'une part déterminante, quoique trop souvent négligée, de la construction d'un programme consiste à concevoir un **langage d'entrée** dans lequel seront codées les données d'entrée nécessaires à son utilisation.

Concevoir un langage de programmation est chose difficile; l'histoire des langages généraux montre que l'échec est la règle, le succès l'exception. L'une des difficultés principales de cette tâche est qu'elle exige un compromis constant entre des impératifs techniques, économiques et humains, mettant en jeu des interlocuteurs très dissemblables – techniciens, commanditaires, utilisateurs finals – dans un dessein d'ensemble qui doit rester homogène. Une analogie qui vient à l'esprit est celle du métier d'architecte (cf article *Rôle de l'architecte* dans le traité Construction): l'une et l'autre profession exigent à la fois la maîtrise de techniques difficiles, un solide réalisme et des qualités fort difficiles à enseigner qui détermineront en dernier lieu l'acceptabilité du produit final: l'élégance, la simplicité, la cohérence.

Il conviendra donc d'apprécier que les *critères* énumérés ci-après sont dans une large mesure contradictoires, toute solution pratique étant un compromis, et que les *méthodes* indiquées ensuite ne sont qu'un support technique permettant de poser correctement certains problèmes, non d'obtenir des solutions – tout au plus de les décrire.

3,2 GRANDS CRITÈRES

Un certain nombre de critères peuvent être dégagés pour guider la conception ou le choix d'un langage. En voici onze parmi les principaux.

3,201 Homogénéité. Régularité.

L'homogénéité est sans doute la qualité la plus importante d'un langage. Elle implique que les choix de conception effectués aux diverses étapes soient réguliers et cohérents, s'inscrivant dans un dessein d'ensemble qui évite à l'utilisateur les surprises désagréables et l'apprentissage de cas particuliers multiples.

Exemple: Fortran IV fournit, pour des raisons historiques, de nombreux contre-exemples au principe d'homogénéité. Ainsi, le langage permet d'utiliser des expressions quelconques dans une affectation ou un appel de sous-programme; mais les seules expressions permises comme indices de tableaux sont de la forme:

- *constante*
- *variable*
- *variable* ± *constante*
- *constante* * *variable* ± *constante*

et les seules expressions permises comme bornes d'une itération (boucle *DO*) sont les constantes et les variables. La version plus récente du langage (Fortran 77) persiste dans la même direction; on y trouve ainsi la règle selon laquelle une expression comprenant une élévation à une puissance, comme *B**3, est légale comme taille de tableau, mais non comme taille d'une chaîne de caractères.

3,202 Orthogonalité.

L'orthogonalité (le terme a été introduit à propos de la conception d'Algol 68) prolonge l'homogénéité. C'est la règle selon laquelle deux caractéristiques quelconques du langage peuvent être combinées sans restriction si elles ne présentent pas d'incompatibilité logique.

Exemple: Algol W offre à la fois des *tableaux* et des types *enregistrements* (§ 2,241 et 2,243). Ces deux possibilités ne sont pas conçues orthogonalement, puisque les tableaux d'enregistrement sont autorisés mais non les enregistrements comprenant des tableaux parmi leurs composants. L'une et l'autre possibilité sont en revanche offertes en Algol 68, Pascal, Simula, Ada ou PL/I.

3,203 Simplicité.

La simplicité d'un langage détermine sa facilité d'apprentissage, la fiabilité des compilateurs, et plus généralement la fiabilité des programmes (on programme mieux, et avec un plus grand sentiment de sécurité, dans un langage que l'on maîtrise totalement).

La simplicité est l'un des buts les plus difficiles à atteindre, en particulier au cours de l'évolution d'un langage en des versions successives, qui traduisent les demandes toujours plus pressantes des utilisateurs, dont chacun ne voit que les extensions censément utiles à son application, au détriment de l'image d'ensemble.

Un contre-exemple souvent cité est PL/I qui, en voulant réunir les vertus de Fortran, Algol 60 et Cobol, a abouti à une construction particulièrement complexe. Fortran 77, nouvelle version de Fortran, témoigne bien de la difficulté de faire évoluer un langage; le document de définition est six fois plus long que celui de Fortran IV (norme 66), décrit en vingt-six pages. On notera à ce propos que, si la simplicité d'un langage est un facteur éminemment subjectif et difficile à mesurer, la longueur de sa définition (formelle ou non) fournit un indice assez significatif.

3,204 Généralité.

Tous les langages de programmation *généraux* ne permettent pas de traiter commodément tous les types de problèmes. Fortran et Algol 60 sont par exemple acceptables pour le calcul scientifique, mais inadaptés aux exigences des programmes d'*informatique système* comme les compilateurs ou les systèmes d'exploitation (structures de données complexes), des programmes de *traitement de textes* (manipulation de caractères), et des applications de *gestion* (manipulations de fichiers). Cobol est tout à fait impropre au calcul scientifique et, dans une large mesure, à l'*informatique système*. Pascal n'est pas adapté à l'écriture de bibliothèques de programmes scientifiques (problèmes de la compilation séparée, cas des tableaux à bornes fixées dès la compilation, etc.). Parmi les langages de haut niveau largement diffusés, seuls PL/I, Ada et Algol 68 offrent des possibilités pour gérer plusieurs tâches concurrentes.

L'objectif de généralité dépend étroitement du créneau visé par le langage. Mais il faut tempérer cette évidence par la remarque selon laquelle un langage qui réussit la percée se retrouve souvent employé à des fins très différentes de celles qu'avaient envisagées ses concepteurs initiaux. Fortran et Cobol sont ainsi, du fait de leur disponibilité quasi universelle, utilisés dans des applications auxquelles ils sont en principe tout à fait inadaptés, et que leurs créateurs n'auraient jamais imaginées.

3,205 Extensibilité.

Les critères de généralité et de simplicité sont en apparence totalement contradictoires, la généralité paraissant impliquer l'inclusion dans le langage de provisions pour tous les types de problèmes imaginables. Une technique qui permet de résoudre en partie ce dilemme est celle de l'extensibilité. Un langage est dit extensible [l.b. 36] s'il contient des mécanismes permettant au programmeur de définir sémantiquement et/ou syntaxiquement de nouvelles constructions dans le langage lui-même. Le langage *noyau* initialement fourni consiste en quelques constructions de base et quelques mécanismes d'extensibilité, et peut donc rester simple.

Les **sous-programmes** fournissent un moyen d'extensibilité immédiat, puisqu'ils permettent d'étendre le jeu des instructions et des expressions du langage. L'extensibilité duale est offerte dans les langages successeurs d'Algol de la «seconde génération» (§ 4,2) par les facilités de construction de nouveaux **types de données**. Dans certains cas, on peut même étendre la *syntaxe* du langage; le point délicat est de délimiter la frontière au-delà de laquelle l'homogénéité et la simplicité sont mises en cause.

Exemple: une extensibilité syntaxique restreinte est offerte en Algol 68, qui permet d'associer à un sous-programme une syntaxe d'opérateurs. Par exemple, un sous-programme calculant la somme de deux matrices pourra être défini de façon à être invoqué sous la forme mathématique habituelle *A + B* dans une expression.

3,206 Compilabilité.

Tout langage de programmation doit être mis en œuvre sur un ordinateur, et les contraintes techniques correspondantes sont déterminantes.

Exemples.

- Algol 68 propose des tableaux *flexibles* (à nombre variable d'éléments), dont la mise en œuvre se révèle fort difficile sur les machines actuelles et fait partie des problèmes qui ont retardé le développement des compilateurs et la diffusion du langage.
- L'introduction de structures de données pouvant être créées dynamiquement, au fur et à mesure de l'exécution (Algol W, Algol 68, PL/I, Simula 67, Ada, etc.) entraîne, pour une gestion efficace de l'espace de mémoire, la nécessité d'inclure un processus *ramasse-miettes* dont l'effet sur le temps d'exécution est difficile à délimiter.

3,207 Clarté.

A l'autre extrémité de la chaîne, les programmes sont destinés aux humains; un programme est *lu* beaucoup plus souvent qu'il n'est *écrit*. La clarté des notations offertes par le langage est donc fondamentale. Son appréciation est très largement subjective, et dépend beaucoup de la *culture* du lecteur et de ses habitudes; on peut juger, du moins si l'on est anglophone, la forme suivante:

[Algol W] ***write (if x > = y then x else y)***

plus claire que:

[APL] $\square \leftarrow X \lceil Y$

mais un praticien expérimenté d'APL ne sera pas de cet avis.

Les éléments fondamentaux de la clarté des programmes peuvent cependant être ramenés à deux critères: d'abord, les possibilités de **structuration** (des algorithmes, des données); en second lieu, les moyens d'**explication** (commentaires, assertions) offerts par le langage.

3,208 Sécurité et fiabilité.

L'évolution de la science de la programmation a conduit à mettre l'accent sur les difficultés de produire des programmes corrects. Le langage peut être un atout considérable en imposant des règles précises qui permettront de détecter de nombreuses erreurs dès la compilation. Il est reconnu aujourd'hui que l'une des précautions les plus fructueuses consiste à imposer un **typage** strict aux données: tout objet du programme – variable ou constante – doit être explicitement muni d'un type, et les *conversions* entre objets de types différents doivent être elles aussi exprimées explicitement.

Exemple: un exemple célèbre d'erreur de programmation ayant causé un échec retentissant (celui de la sonde américaine *Mariner-1* vers Vénus) est le suivant:

dans l'instruction Fortran de début de boucle, écrite

[Fortran] *DO 10 I = 1,100*

le remplacement fortuit de la virgule par un point fait que l'instruction est comprise comme

[Fortran] *DO10I = 1.100*

car les blancs ne sont pas significatifs dans ce langage. Les variables n'ayant pas à être déclarées, et les identificateurs ne commençant pas par *I, J, K, L, M* ni *N* étant pris par défaut comme désignant des variables réelles, cette instruction est comprise comme l'affectation de la valeur décimale 1,1 à la variable *DO10I*, et l'erreur passe donc inaperçue.

Dans les langages de la série Algol, toutes les variables doivent au contraire être déclarées comme possédant un certain type, et l'emploi d'une variable non déclarée déclenche une erreur dès la compilation.

Pour trouver un renseignement, consulter la table alphabétique H 12.

Les travaux relatifs à la méthodologie de la programmation ont mis en évidence l'intérêt de démontrer la validité des programmes, c'est-à-dire leur conformité aux buts qu'ils s'assignent, par des techniques formelles.

Ces techniques n'ont pas atteint un développement permettant leur application courante à des programmes importants; leur connaissance permet cependant de mieux programmer, et, pour le concepteur de langages, elles fournissent des indications intéressantes. Certaines constructions se prêtent bien, en effet, aux démonstrations de validité, alors que d'autres les rendent impraticables. Ainsi, les trois structures de contrôle dites de la *programmation structurée*, décrites dans l'exemple du paragraphe 2,31, permettent des démonstrations qu'interdisent en pratique les instructions de branchement (*GOTO*) employées de façon incontrôlée.

Certains langages de programmation proposent des instructions de vérification (*ASSERT* en Algol W) s'insérant bien dans une méthodologie rigoureuse de programmation.

Les méthodes de démonstration de programmes se rapprochent de celles qui sont employées pour décrire la sémantique des langages (§ 3,34); l'une des plus couramment employées est la sémantique *axiomatique* (§ 3,343). Sur la démonstration des programmes, on pourra consulter [l.b. 28]; sur ses applications à la programmation, [l.b. 14 et 30].

3,209 Souplesse et commodité d'emploi.

Pour attirer vers un langage le plus d'utilisateurs possible, il est naturel de chercher à le rendre commode d'emploi, à lever les restrictions, à permettre des abréviations, etc.

On notera que, pris à la lettre, cet objectif s'oppose directement au précédent.

3,210 Puissance expressive.

Lors de manipulations longues et répétitives, il est naturel de chercher à utiliser des notations aussi chargées de sens que possible. Des langages comme PL/I et APL offrent un grand nombre de notations variées permettant d'exprimer en peu de termes des opérations complexes. Ici encore, ce critère doit être mis en balance avec la simplicité et l'homogénéité.

3,211 Complétude de la définition et portabilité.

Il est important pour l'utilisateur, comme pour celui qui écrit un compilateur, que le langage soit complètement défini indépendamment de toute réalisation sur un ordinateur; de cette définition dépendent en effet la possibilité de maîtriser le langage et surtout la portabilité des programmes, c'est-à-dire la possibilité de les adapter à différents systèmes informatiques.

Exemples.

- Le document de définition d'Algol 60 ne soufflait mot des entrées ni des sorties: de nombreuses versions incompatibles ont donc vu le jour.
- Fortran IV n'inclut pratiquement pas de possibilités de manipulations de caractères. Chaque compilateur permet de résoudre en partie le problème, d'une façon qui lui est propre.

On trouvera dans [l.b. 10] des notions importantes sur les techniques de la portabilité.

3,3 MÉTHODES: DÉFINITIONS FORMELLES

3,31 Intérêt d'une étude formelle.

Le créateur d'un langage dispose aujourd'hui de techniques qui permettent de poser clairement les choix de conception – à défaut de les résoudre. Elles servent à définir rigoureusement la **syntaxe** et la **sémantique** d'un langage de programmation, c'est-à-dire la forme et l'effet d'un programme légal.

- L'étude formelle de la syntaxe vise deux objectifs:

– fournir un cadre concret pour la description claire et non ambiguë de la forme que doit revêtir un programme dans le langage considéré;

– servir de support à la partie syntaxique des compilateurs (cf article *Techniques de traduction* dans ce traité); ce but est particulièrement important, car les compilateurs actuels sont le plus souvent **dirigés par la syntaxe**, c'est-à-dire que l'*analyseur syntaxique* forme l'ossature autour de laquelle s'articule le compilateur tout entier; grâce à la formalisation de la syntaxe, cet analyseur peut aujourd'hui être créé *automatiquement* à partir de la donnée de la syntaxe du langage.

- L'étude formelle de la sémantique vise deux objectifs:

– fournir un cadre concret pour la vérification approfondie de la cohérence des langages (les *déverminer* comme on le ferait d'un programme);

– servir de support à la partie sémantique des compilateurs (synthèse, production du code objet; cf article *Techniques de traduction* dans ce traité).

Il est important de noter que la formalisation ne répond pas seulement à des nécessités techniques: elle permet en retour de guider le concepteur dans certains choix, les contraintes techniques s'avérant en définitive fructueuses.

Exemple: certains systèmes de production automatique d'analyseurs syntaxiques exigent des grammaires traitées qu'elles possèdent la propriété dite *LL(1)*, selon laquelle, grossièrement parlant, le premier élément de toute *phrase* du langage – instruction, expression, etc. – détermine sans ambiguïté la nature de cette phrase (un contre-exemple de cette propriété est l'instruction *DO* de Fortran, du fait de la confusion possible avec une affectation; § 3,208). Cette contrainte, de nature essentiellement technique à l'origine, se révèle en fait profitable à la clarté et à la lisibilité des langages.

3,32 Syntaxe.

La syntaxe est, depuis Algol 60, couramment définie sous une forme appelée BNF (Forme Normale de Backus, ou Forme de Backus-Naur). Cette technique, issue des travaux de Chomsky dans les années cinquante (travaux qui avaient pour objet initial les langues humaines, non les langages artificiels; cf par exemple [l.b. 12]), définit un langage en terme de **terminaux** (les symboles de base qui entreront dans la confection d'un programme: lettres, chiffres, caractères spéciaux, mots réservés) et de **non-terminaux**, correspondant aux entités grammaticales (expression, instruction, programme, etc.). La *grammaire* d'un langage est alors décrite en terme d'**équations syntaxiques** qui définissent chaque non-terminal en fonction de terminaux et d'autres non-terminaux.

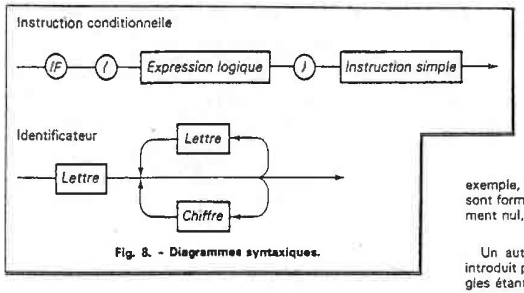


Fig. 8. – Diagrammes syntaxiques.

Exemple: notons, selon l'une des conventions fréquemment adoptées, les non-terminaux par des identificateurs (par exemple *instruction*) et les terminaux entre apostrophes (exemple: *'IF'*). L'*équation* ci-après, où *::=* signifie *est défini comme*, décrit la syntaxe des instructions conditionnelles (*IF logique*) en Fortran:

instruction-conditionnelle ::= 'IF' '(' expression-logique ')' 'instruction-simple'.

Elle indique en effet que l'on obtient une *instruction conditionnelle* en faisant suivre le mot (terminal) *IF* d'une parenthèse ouvrante, d'une *expression logique*, d'une parenthèse fermante, enfin d'une *instruction simple*. Les non-terminaux *expression-logique* et *instruction-simple* doivent être définis dans d'autres équations du même type, où ils apparaissent à gauche.

Il est fréquent qu'une équation de BNF soit **récursive**, c'est-à-dire contienne dans sa partie droite des occurrences du non-terminal situé à gauche. Cette possibilité permet de définir des structures *répétitives* ou *imbriquées*. Les équations contiennent alors deux ou plusieurs branches séparées par le symbole | correspondant à ou. Une équation de la forme:

$$a ::= X \mid Y \mid \ldots \mid Z$$

signifie: un *a* est défini comme un *X*, ou comme un *Y*, . . ., ou comme un *Z*.

Exemples.

- L'équation

identificateur ::= lettre | lettre identificateur

signifie qu'un *identificateur* est soit une *lettre*, soit une *lettre* suivie d'un *identificateur*: c'est-à-dire, en définitive, qu'un identificateur est une suite de une, deux ou plusieurs lettres.

- Les équations

expression ::= variable | variable opérateur variable | '(' expression ')'
opérateur ::= '+' | '' | '–' | '/'*

définissent la syntaxe des expressions mathématiques courantes.

Plusieurs notations équivalentes à la BNF sont également utilisées en pratique. Pour éviter un usage exagéré de la récursion, on emploie parfois le formalisme des **expressions régulières**; par exemple, l'*identificateur* de l'équation ci-avant peut être défini par:

lettre $^+$

où $^+$ signifie «une ou plusieurs occurrences», * signifie «zéro, une ou plusieurs occurrences». De véritables **expressions** peuvent être formées par ces opérateurs, le symbole | et des parenthèses; par exemple, les identificateurs des langages de programmation courants sont formés d'une lettre suivie d'un nombre quelconque, éventuellement nul, de lettres et de chiffres; ils seront décrits par:

lettre (lettre | chiffre) *

Un autre formalisme équivalent à la BNF est celui, graphique, introduit par N. Wirth à propos de la définition de Pascal. Les rectangles étant associés à des non-terminaux et les cercles à des terminaux, on pourra définir l'*instruction conditionnelle* de Fortran et les *identificateurs* précédents par les diagrammes de la figure 8.

3,33 Sémantique statique.

Les techniques de type BNF ne permettent de décrire qu'une partie de la syntaxe, dite "sans contexte"; des règles plus complexes, de la forme *tout identificateur employé dans une instruction doit avoir été déclaré au préalable dans une déclaration*, ne peuvent être exprimées dans un tel formalisme. On est donc conduit en pratique à traiter au niveau de la sémantique des propriétés qui relèvent à proprement parler de la syntaxe. C'est ce que l'on appelle la *sémantique statique* du langage.

3,34 Sémantique.

Les méthodes de formalisation de la sémantique ne font pas l'objet d'un accord aussi large que leurs homologues pour la syntaxe. Nous citerons ici les quatre formalismes les plus répandus: sémantique par attributs, sémantique opérationnelle, sémantique axiomatique, sémantique dénotationnelle. Pour plus de détails, on se reportera à un ouvrage de *théorie du calcul* [l.b. 28] et à une étude synthétique [l.b.18].

3,341 Sémantique par attributs. – La méthode des attributs consiste à **décorer** les équations syntaxiques en leur ajoutant des propriétés représentant la sémantique.

Exemple: soient les équations suivantes, qui définissent syntaxiquement la notion de constante entière positive ou nulle en notation décimale:

entier ::= chiffre | entier chiffre
chiffre ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

Les équations décorées correspondantes, qui comprennent la sémantique des constantes entières, c'est-à-dire la description abstraite du calcul de leurs valeurs, peuvent s'écrire:

entier ⌈v⌉ *::= chiffre* ⌈c⌉ *c | entier* ⌈ve *chiffre* ⌈vc⌉ *10 * ve + vc*
chiffre ⌈v⌉ *::= '0'*⌊0⌋ *| '1'*⌊1⌋ *| '2'*⌊2⌋ *| '3'*⌊3⌋ *| '4'*⌊4⌋ *| '5'*⌊5⌋ *| '6'*⌊6⌋ *| '7'*⌊7⌋ *| '8'*⌊8⌋ *| '9'*⌊9⌋

Dans ces équations, les noms précédés de ⌈ représentent les caractéristiques ou *attributs sémantiques*, associées à chaque non-terminal, à la façon d'arguments formels désignant les résultats d'un sous-programme (§ 2,41). Les expressions précédées de ⌊ donnent la valeur de l'attribut du non-terminal figurant en partie gauche de l'équation (nous nous sommes limités à des non-terminaux n'ayant qu'un attribut).

[l.b. *n*] renvoie à la référence *n* de l'index bibliographique placé en fin d'article.

Dans cet exemple, ainsi, une constante entière a un attribut noté *v* (sa valeur); si la constante est un chiffre, d'attribut *c*, alors $v = c$; la valeur de *c* est donnée par la seconde équation (c'est la valeur numérique déduite du chiffre). Si la constante entière est de la forme *entier chiffre*, l'entier ayant pour attribut *ve* et le chiffre *vc*, alors son attribut aura pour valeur $v = 10\ ve + vc$.

Par **exemple**, l'attribut associé à '*2385*' a pour valeur :
$10\ ve + vc$,
avec *vc* valeur entière du chiffre '*5*',
ve attribut de '*238*'.

Cette méthode de décoration des équations syntaxiques à l'aide d'attributs sémantiques est la plus utilisée en pratique pour la construction des compilateurs (cf article *Techniques de traduction* dans ce traité).

3,342 Sémantique opérationnelle. – La sémantique opérationnelle ou *interprétative* définit l'effet des instructions du langage en termes d'instructions sur une machine virtuelle bien définie, mais indépendante des contraintes techniques des machines réelles.

3,343 Sémantique axiomatique. – La sémantique axiomatique permet, plus que les deux méthodes précédentes, de se rattacher à un cadre mathématique classique en associant à chaque instruction un *transformateur de prédicats* qui met en correspondance les propriétés vérifiées avant et après son exécution.

Exemple : soit *A* l'instruction d'affectation suivante :
$x := E$ (x : variable – E : expression).
Il lui est associé pour toute propriété *P* la règle :
$$pfp\ (P, A) = P[E/x]$$
où $P[E/x]$ désigne la propriété obtenue en remplaçant dans *P* toutes les occurrences de *x* par *E*, et *pfp* (P, A) est la **plus faible précondition** qui, si elle est vérifiée avant l'exécution de *A*, entraîne que *P* sera vérifiée après cette exécution.

Ainsi,
si *A* est $x := y$ et *P* est $(x > 0)$, alors *pfp* (P, A) est $(y > 0)$;
si *A* est $x := x + 3$ et *P* est $(x \neq 12)$, alors *pfp* (P, A) est $(x + 3 \neq 12)$, c'est-à-dire $(x \neq 9)$.

3,344 Sémantique dénotationnelle. – La sémantique dénotationnelle vise à ramener encore plus la définition des langages de programmation à un formalisme mathématique ; elle considère tout programme comme un **point fixe** d'une équation récursive, à laquelle la théorie permet d'appliquer des notions de continuité, de monotonie, de convergence, etc. Très étudié sur le plan théorique, ce formalisme a été plus utilisé pour des études de cohérence de langages et pour des expériences de conception que pour la construction de compilateurs.

4 Un peu d'histoire

Agés de moins de trente ans, les langages de programmation possèdent déjà une histoire riche, foisonnante même, et pleine d'enseignements. Nous nous contenterons d'évoquer ici les grandes tendances. Pour plus de détails, nous renvoyons à l'œuvre de J.E. Sammet [I.b. 35], valable pour les langages essentiellement américains jusqu'en 1967-1968, et aux actes d'une conférence ACM (Association for Computing Machinery : société savante américaine d'informatique) de 1978 [I.b. 2], qui racontent par des exposés approfondis, subjectifs, et souvent fascinants, l'histoire de treize langages parmi les principaux. On consultera aussi, plus particulièrement sur Algol et ses rapports avec les langages contemporains, [I.b. 7].

A la manière de la chronologie souvent adoptée pour le matériel, nous distinguerons trois **générations**, correspondant approximativement aux décennies 1950, 1960 et 1970 (que ces générations aient près de dix ans de retard sur leurs homologues matérielles est tout à fait significatif du retard général des progrès du logiciel).

4,1 PREMIÈRE GÉNÉRATION : LES PIONNIERS

Entre 1954 et 1961 quatre langages ont été développés qui, ensemble, introduisaient presque tous les concepts importants qui devaient être repris par la suite : Fortran, Algol (58 et 60), Lisp et Cobol. Les idées d'APL remontent à la même époque, mais c'est la période suivante qui devait voir son développement comme langage de programmation.

4,11 Fortran.

Le lecteur se reportera utilement à l'article *Fortran* dans ce traité.

Bien que l'idée des langages de programmation fût sans doute apparue aux inventeurs des ordinateurs dès la fin des années quarante, c'est Fortran qui le premier montra de façon irréfutable son application pratique. Conçu à partir de 1954, par une équipe d'IBM autour de John Backus, comme un système permettant de coder à l'intention des ordinateurs des formules mathématiques en notation habituelle (*FORmula TRANslation*, traduction de formules), il allait rapidement être complété par des instructions qui en firent un véritable langage de programmation. Pour des raisons fort respectables d'efficacité – la partie était loin d'être gagnée d'avance contre les tenants de la programmation en langage d'assemblage –, Fortran était très marqué par la première machine-cible, l'*IBM 704* ; et cette influence s'est malheureusement transmise, hors de son contexte, dans toutes les versions ultérieures : en particulier Fortran IV (1962, normalisé officiellement en 1966), qui est aujourd'hui (1980) la version couramment employée, et Fortran 77 (normalisé officiellement en 1978) qui, annoncé comme devant être la version de l'avenir, corrige quelques-uns des défauts du langage (en fournissant en particulier des possibilités de manipulation de caractères et de fichiers), au prix d'une complexité et d'une hétérogénéité accrues. Pour une analyse approfondie, voir [I.b. 27].

4,12 Algol.

En réaction contre le caractère *ad hoc* et peu rigoureux de Fortran, deux groupes qui devaient par la suite fusionner prirent en 1957 l'initiative de créer un langage algorithmique universel : l'un américain (ACM), l'autre allemand (GAMM : Gesellschaft für Angewandte Mathematik ; maintenant GI : Gesellschaft für Informatik). Le premier résultat, produit en 1958, fut révisé en 1960 sous le nom d'Algol 60.

Il s'agissait d'un langage très élégant, bien défini, mais souffrant de lacunes évidentes : ainsi, en raison de la diversité des systèmes d'exploitation existants, les entrées et les sorties n'avaient pas été incluses. Algol devait perdre commercialement la partie face à Fortran pour ces raisons techniques et d'autres relevant plutôt de la sociopolitique (pour une étude détaillée, voir [I.b. 7]) ; mais son influence a été fondamentale dans toute la suite des développements des langages et des compilateurs, du fait de l'intérêt des concepts introduits (structure de blocs, récursion, structures de contrôle, procédures), des progrès en compilation suscités par le langage, et de la méthode qu'il inaugurait pour la définition formelle de la syntaxe.

4,13 Lisp.

Le lecteur se reportera utilement à l'article *Langages de manipulation de symboles* dans ce traité.

Développé par J. McCarthy au MIT (Massachusetts Institute of Technology) en 1957, Lisp [I.b. 33] est resté jusqu'à aujourd'hui de diffusion limitée aux spécialistes de l'intelligence artificielle et du calcul symbolique. Mais son influence réelle a, comme celle d'Algol, dépassé la communauté des utilisateurs directs du langage ; Lisp démontrait que l'on pouvait réaliser un langage extrêmement puissant à partir d'un nombre très faible de concepts (les listes, les expressions conditionnelles, les définitions récursives et cinq opérations de base), et que les ordinateurs pouvaient être utilisés avec profit pour des calculs entièrement symboliques et non numériques (démonstration à vrai dire commencée par le langage IPL-V dès 1955).

4,14 Cobol.

Le lecteur se reportera utilement à l'article *Cobol* dans ce traité.

En 1959, le Department of Defense (DoD) était préoccupé par la prolifération des langages pour un domaine dont l'importance, faible au temps des premiers ordinateurs, croissait rapidement : les applications de l'informatique à la gestion. Un groupe de représentants des constructeurs fut convoqué et ses membres enjoints de définir un langage commun orienté vers les problèmes de gestion (*COmmon Business-Oriented Langage*). Le résultat – Cobol –, diffusé en 1961, devait être soutenu par le DoD à l'aide d'arguments convaincants (compilateur Cobol exigé comme condition préalable à l'accréditation d'un constructeur pour tout contrat avec un organisme fédéral). Cette politique a réussi : Cobol est aujourd'hui le langage le plus employé.

Pour la table analytique, se reporter à la première page de cet article.

A travers ses versions successives, Cobol a maintenu ses caractéristiques principales :

– syntaxe orientée vers le langage naturel, c'est-à-dire emploi de mots anglais (noms, prépositions, verbes), de longues phrases – d'où une impression de verbosité ;

Exemple :

[Cobol]
WRITE ENREG-PAYE-MOIS FROM CALC-PAYE AFTER ADVANCING NLIGN LIGNES AT END-OF-PAGE GO TO EN-TETE

– grand nombre et variété des possibilités offertes (le langage contient plus de mille mots réservés) ;

– richesse des mécanismes de description des fichiers ;

– volonté d'indépendance par rapport aux machines cibles (l'exécution de programmes presque identiques sur des ordinateurs RCA et Remington-Rand-Univac, en 1960, fut un événement, et sans doute la première expérience réussie de portabilité) ;

– séparation entre la description des données (fichiers) et celle des programmes.

L'évolution de Cobol s'est effectuée pour l'essentiel en dehors des courants généraux qui ont affecté les autres langages. Ceci explique que certains concepts considérés ailleurs comme fondamentaux aient longtemps attendu avant d'atteindre Cobol : les *sous-programmes*, par exemple, n'appartiennent à la norme que depuis 1974. Cobol a par contre de son côté exercé une grande influence sur d'autres domaines de l'informatique comme la modélisation des bases de données (cf rubrique *Bases de données*).

4,2 DEUXIÈME GÉNÉRATION : L'AMBITION

De 1962 à 1970 apparaissent des centaines de nouveaux langages. Malgré leur très grande diversité, beaucoup d'entre eux présentent des caractéristiques communes :

– ils dérivent en général des langages de la génération précédente (*exceptions* : Snobol et APL sont fondés sur des concepts radicalement nouveaux) ;

– ils sont ambitieux et veulent offrir le maximum de possibilités (*exceptions* : Algol W et Pascal visent délibérément la simplicité) ;

– ils mettent très nettement l'accent sur la description et la structuration des données, fort négligées en Fortran et Algol, un peu mieux traitées en Cobol et en Lisp.

Nous nous intéresserons ci-après à quelques-uns des plus importants parmi les langages de cette période qui ont survécu : PL/I, Algol 68, Simula 67, Algol W et Pascal, Snobol, APL.

4,21 PL/I.

Le lecteur se reportera utilement à l'article *PL/I* dans ce traité.

Issu d'une réflexion commune d'IBM et de Share et Guide, associations d'utilisateurs d'ordinateurs de cette marque, PL/I [I.b. 8] fut annoncé en 1964 par IBM ; après de nombreuses modifications du langage, le premier compilateur (PL/I niveau F) fut diffusé en 1966. Initialement conçu comme devant déboucher sur une extension à Fortran, le projet PL/I avait produit un langage extrêmement ambitieux, intégrant les principaux concepts de Fortran, d'Algol 60 et de Cobol, et se voulant universel, c'est-à-dire propre à la résolution de tous les types de problèmes.

PL/I n'a pas réussi dans son ambition de remplacer ses trois géniteurs, du fait de problèmes techniques (inefficacité des compilateurs initiaux), de son image trop liée à IBM et de sa complexité. Il a atteint cependant une diffusion respectable, en particulier en informatique de gestion. Très critiqué dans les années soixante-dix à cause de son manque d'homogénéité et de rigueur, PL/I est une construction impressionnante qui montre sans doute la limite en matière de généralité, de puissance et de complexité.

4,22 Algol 68.

Le lecteur se reportera utilement à l'article *Algol 68* dans ce traité.

Ce qu'Algol 60 avait été à Fortran, Algol 68 [I.b. 3] a voulu le reproduire vis-à-vis de PL/I : avec le même champ d'application (là le calcul scientifique, ici les applications les plus générales), il s'agissait de concevoir un langage beaucoup plus homogène, régulier, rigoureux. Ce but a sans conteste été atteint par Algol 68 qui reprend en les systématisant les principes méthodologiques d'Algol 60, retrouve ses qualités, et approfondit ses concepts (ainsi la description syntaxique à deux niveaux permet de décrire la sémantique statique ; les structures de contrôle sont généralisées ; la distinction entre *nom* et *valeur* est clairement définie ; une syntaxe extensible est offerte ; etc.). Pourtant, Algol 68 s'est répandu de façon encore plus modeste qu'Algol 60 et, surtout, n'a pas été, contrairement à son ancêtre, la souche d'une nouvelle famille. Les raisons de cet échec pratique sont multiples : malgré son approbation officielle par l'IFIP (International Federation for Information Processing), Algol 68 a conservé un parfum universitaire et ésotérique ; l'aridité du document original de définition, destiné à des spécialistes, a fait croire que le langage lui-même était incompréhensible par des programmeurs ordinaires ; les compilateurs ont tardé à venir ; et les divergences qui sont apparues dans le comité de définition dès avant la publication du langage, entraînant une scission, ont empêché Algol 68 de bénéficier comme Algol 60 du soutien unanime de la communauté universitaire.

4,23 Algol W. Pascal.

Le lecteur se reportera utilement à l'article *Pascal. Langages d'écriture de systèmes* dans ce traité.

Algol W [I.b. 11 et 27] et Pascal [I.b. 39] représentent parmi les successeurs d'Algol 60 l'école rivale de celle d'Algol 68 ; leurs objectifs suprêmes sont la simplicité et la fiabilité, qui doivent être atteintes au détriment de la généralité et de la souplesse s'il le faut.

L'un et l'autre sont des langages destinés à l'origine à l'enseignement. Algol W (1966) est une version simplifiée d'Algol 60, qui en reprend sous une forme élaguée les principaux concepts, en leur ajoutant des possibilités de définition de données complexes (enregistrements et pointeurs). Pascal (1970) va plus loin encore dans le même sens : simplifications et restrictions (suppression des variables locales à un bloc qui n'est pas une procédure, tableaux à bornes nécessairement fixées à la compilation) ; types de données (ensembles, types définis par énumération, fichiers séquentiels). Au prix de sérieuses limitations, le langage résultant est petit et d'apprentissage facile.

Pascal n'a pas suivi le sort des nombreux dérivés d'Algol qui ont fleuri dans les années soixante et, pour la plupart, rapidement quitté la scène après avoir popularisé quelques concepts : il a au contraire été l'objet d'une expansion rapide, en particulier à partir de 1975. Les raisons de ce succès sont multiples : simplicité et atouts pédagogiques ; utilisation comme langage d'enseignement, transformant les anciens étudiants en autant de zélateurs ; propagande menée habilement ; existence dès l'origine d'un compilateur écrit en Pascal même et destiné à être adapté à de nouvelles machines-cibles ; et identité des buts affirmés du langage avec les objectifs de *fiabilité du logiciel* qui, dans les années soixante-dix, ont commencé à s'imposer comme déterminants. C'est dans le domaine de la mini-informatique et de la micro-informatique, ainsi que dans celui de l'informatique système, que les progrès de Pascal ont été les plus remarquables.

4,24 Simula 67.

Le lecteur se reportera utilement à l'article *Langages de simulation* dans ce traité.

Simula 67 [I.b. 9 et 28], développé à l'université d'Oslo, représente encore une autre branche dans la famille Algol, fondée sur la compatibilité avec Algol 60 et sur le développement de structures permettant pour la première fois une programmation véritablement modulaire.

Simula 67 – mal nommé puisqu'il s'agit d'un langage de programmation général, dont la simulation n'est qu'une application possible – est issu d'un langage appelé Simula tout court, et a subi l'influence d'Algol W. Il ajoute à Algol 60 la notion de classe, une structure de programme qui permet de représenter [I.b. 31] :

– la mise en œuvre de structures de données complexes, considérées comme associées à de nouveaux types, avec les opérations associées (§ 2,22) ;

– des processus quasi parallèles ou coprogrammes ;

– plus généralement, des modules de programmation homogènes (§ 2,42).

Diffusé assez largement en Europe du Nord, Simula 67 est resté pendant plusieurs années à l'écart du développement général des langages. Redécouvert aux États-Unis à partir de 1975 en liaison avec les recherches sur les types abstraits, Simula 67 est avec Pascal à la source de la plupart des langages de la troisième génération (§ 4,3).

4,25 Snobol.

Snobol, développé à partir de 1962 à Bell Telephone Laboratories, est un langage entièrement consacré à la manipulation de chaînes de caractères et permettant à l'aide d'un nombre extrêmement restreint d'éléments de base (l'essentiel est la notion de *filtrage*, ou remplacement conditionnel d'une chaîne de caractères par une autre, effectué seulement si une certaine propriété est vérifiée) d'effectuer des transformations complexes. La famille Snobol a donné plus récemment SL5 et Icon [I.b. 29].

4,26 APL.

Le lecteur se reportera utilement à l'article *APL* dans ce traité.

APL [I.b. 15] est une notation mathématique proposée en 1962 par K. Iverson, qui fut d'abord appliquée à la description de machines et d'algorithmes ; un sous-ensemble devint disponible comme langage de programmation à partir de 1966.

La notation d'APL est à la fois très concise et très puissante grâce à des opérateurs agissant sur des tableaux tout entiers ($A + B$ désigne la somme de deux matrices), à l'opérateur de généralisation $/$ ($+/X$ est la somme des éléments du vecteur X, $\times/X$ leur produit, etc.), et à de nombreuses autres primitives permettant d'exprimer des opérations complexes de façon très brève. Exigeant un grand nombre de caractères spéciaux, elle présente une apparence hiérophique assez contraire aux principes de lisibilité (§ 3,207). APL est très goûté par toute une communauté d'utilisateurs, souvent non professionnels de l'informatique, qui apprécient la possibilité qu'il offre d'écrire et de mettre au point rapidement des programmes, en particulier pour essayer et valider des méthodes, des idées d'algorithmes, construire des maquettes de systèmes, etc. Il est surtout adapté à des programmes dont la taille reste limitée et qui n'auront pas à être entretenus («maintenus») longtemps, combinés à d'autres éléments de logiciels, modifiés, ni transmis à d'autres programmeurs.

4,3 TROISIÈME GÉNÉRATION : L'INDUSTRIALISATION

Les années soixante-dix et le début des années quatre-vingt sont marqués par la poursuite des expériences et des recherches sur les langages, mais aussi par l'arrêt des créations ambitieuses des étapes précédentes – à une exception près, notable, celle d'Ada. Cette période se signale par la consolidation accrue des positions quasi inexpugnables des grands ancêtres – Fortran, Cobol – qui essayent d'accroître leur universalité par un fastidieux mais indispensable effort de normalisation ; et par une ébauche de rapprochement entre certaines branches de l'industrie (micro-informatique, commande de processus, temps réel, systèmes) et les langages de la tradition Algol issus de l'université, plus particulièrement Pascal.

De nombreux langages expérimentaux ont essayé d'opérer la synthèse entre les objectifs de simplicité et de sécurité mis en vedette par Pascal et la modularité offerte par Simula 67 autour de la structuration des données. Alphard, Clu, Mesa, Euclid entrent dans cette catégorie. Plus récemment (1977 à 1980), un effort considérable lancé par le Department of Defense (DoD), vingt ans après Cobol, a abouti à la suite d'une compétition internationale au choix d'un nouveau langage, Ada [I.b. 24], conçu par une équipe d'origine française. Ada cherche à concilier tous les objectifs des recherches qui l'ont précédé avec les contraintes d'efficacité, de réalisme et de fiabilité imposées par les très grosses applications intégrées. L'avenir dira si le pari peut être tenu.

5 Au-delà des langages de programmation

L'étude des langages de programmation débouche inévitablement sur des concepts frontaliers, dont les ambitions sont, selon les cas, en deçà ou au-delà de celles des langages couramment employés. C'est au bref examen de quelques-uns d'entre eux que nous consacrerons la conclusion de cette étude.

5,1 Langages et progiciels.

Le lecteur se reportera utilement à l'article *Progiciels* dans ce traité.

Les progiciels, encore appelés *packages*, sont, d'après l'article *Progiciels* de ce traité, «des programmes répondant à certains critères de généralité». Pour bien comprendre cette notion et son lien avec celle de langage de programmation, il convient de considérer l'un des dilemmes fondamentaux de la programmation : le compromis **spécificité / généralité** (fig. 9).

Plus un programme est *spécifique* (orienté vers un type de problème particulier) et plus ses données (son langage d'entrée) seront simples; à la limite, un programme résolvant un problème unique (par exemple : chercher le plus petit entier naturel n tel que $n = a^2 + b^2 = c^2 + d^2$ pour deux couples différents $\{a, b\}$ et $\{c, d\}$ d'entiers naturels) n'a pas de données d'entrée. Inversement, plus un programme est *général* et plus le codage de ses données d'entrée sera complexe pour résoudre un problème particulier. La limite dans cette direction est constituée par les langages de programmation généraux, qui définissent la structure des données d'entrée pour un système (compilateur et interprète + ordinateur) capable de résoudre tout problème traitable automatiquement, pour peu qu'on sache l'exprimer.

La programmation se prête à un jeu fréquent entre la complexité des programmes et celle des données : on peut, pour un même problème, privilégier l'une ou l'autre en se déplaçant sur l'une des «équipotentielles» symbolisées sur la figure 9.

Dans ce compromis (qu'il pourrait être intéressant d'analyser sous une forme plus mathématique, par exemple selon les méthodes quantitatives de la *science du logiciel* [I.b. 21]), les progiciels se situent à l'équilibre : leur but est de fournir un moyen de résoudre les problèmes d'une certaine classe, aussi large que possible (*généralité*), de façon aussi simple que possible, donc avec peu de données dans chaque cas (*spécificité*). Tout progiciel définit un langage d'entrée, spécialisé, qui devra tenir compte de ces objectifs contradictoires.

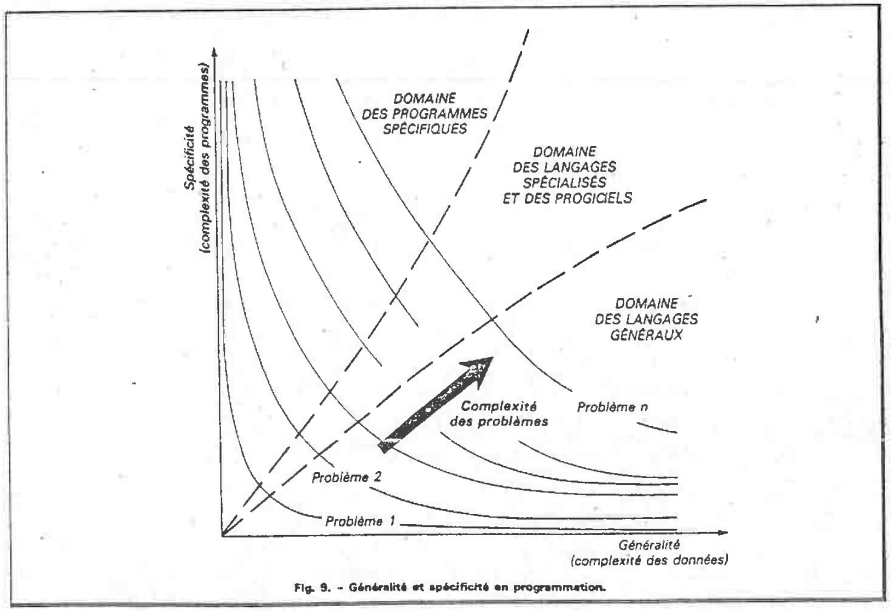


Fig. 9. - Généralité et spécificité en programmation.

5,2 Langages pour non-informaticiens.

La difficulté d'apprentissage et d'emploi des langages de programmation s'oppose à une volonté fréquemment affichée de démocratiser l'informatique. Bien des auteurs ont proposé des langages mettant essentiellement l'accent sur la commodité d'emploi (§ 3,209).

Il est clair que de tels langages sont nécessaires pour permettre à un grand nombre de personnes d'utiliser des systèmes existants; cette nécessité croît chaque jour, en liaison avec le fantastique développement des réseaux, des micro-ordinateurs, de l'informatique individuelle, etc.

Il convient cependant de se garder de toute illusion : privilégier à l'extrême la commodité d'emploi conduit à négliger certaines des autres caractéristiques vues au paragraphe 3 : sécurité, modularité, homogénéité, etc. En outre, à la lumière du paragraphe précédent, il est clair que les langages pour non-techniciens correspondent à la partie gauche du diagramme de la figure 9, c'est-à-dire à l'utilisation de programmes spécifiques. L'utilisation de l'informatique peut être très largement généralisée; la programmation proprement dite (cf article spécialisé dans ce traité) restera l'affaire de professionnels.

5,3 Langages de très haut niveau.

Un observateur objectif constate rapidement que, des deux pôles décrits au paragraphe 1 – l'homme, la machine –, c'est le second qui est très nettement privilégié dans les langages de programmation courants : malgré tous les efforts d'abstraction, on n'a pas beaucoup à gratter pour retrouver sous les concepts proposés – variables, tableaux, instructions, instructions conditionnelles, primitives de synchronisation, etc. – des éléments très matériels présents sur tous les ordinateurs classiques (adresses, registres d'index, ordres, tests et branchements, interruptions, etc.).

Des tentatives ont été menées depuis longtemps pour rapprocher les concepts offerts par les langages de programmation de ceux dans lesquels les problèmes sont normalement exprimés. Nous avons déjà observé des tendances en ce sens : les possibilités de manipulation globale de données offertes par APL, les techniques de définition de nouveaux concepts proposées par les langages extensibles (types en Algol 68 ou Pascal, classes en Simula 67) en sont des exemples. Plus ambitieux sont les langages dits **non procéduraux** (ou non algorithmiques), qui permettent, souvent dans la lignée de Lisp, la définition non prescriptive (§ 2,32) de traitements à effectuer, les **langages de description de données**, qui font abstraction des programmes utilisant ces données, les **langages ensemblistes**, dont le plus notable est Setl [I.b. 26], permettant de manipuler des objets complexes (ensembles, suites) avec des opérations associées (union, concaténation, boucles ensemblistes, etc.). La **programmation fonctionnelle** proposée par Backus [I.b. 5] vise à s'affranchir de la structure «von Neumann» des ordinateurs actuels en offrant des objets complexes, des fonctions en particulier, et le calcul associé.

Tant que la structure des ordinateurs restera proche de la norme actuelle – c'est-à-dire effectivement peu différente de ce qu'elle était à l'époque des pionniers –, et que leurs limitations en capacité et en vitesse continueront de privilégier le critère d'efficacité, les langages de très haut niveau auront peu de chances de supplanter les langages classiques. Une de leurs applications intéressantes est cependant, dès aujourd'hui, l'expérimentation de nouvelles méthodes de calcul, la construction de maquettes, la mise au point d'algorithmes, la comparaison de techniques de mise en œuvre.

5,4 Langages de spécification.

De plus haut niveau encore que les langages de très haut niveau sont les langages de *spécification* (ou d'*analyse fonctionnelle*) dont le but est d'exprimer les problèmes sans les résoudre. Ils sont donc **non exécutables**, ce qui les distingue tout à fait des langages de programmation, de quelque degré d'abstraction que soient ceux-ci.

Ces langages, développés depuis quelques années [I.b. 17 et 34], tirent leur justification du fait qu'en informatique comme dans les autres sciences les véritables difficultés sont souvent liées à la manière de poser les problèmes plus qu'à leur résolution proprement dite. Cela est particulièrement net dans le cas de grandes applications informatiques de temps réel ou de gestion par exemple, dont les difficultés proprement techniques sont souvent moindres que la difficulté de fournir une description des fonctions attendues du système (*cahier des charges*) qui soit à la fois complète, précise, claire et structurée. Le langage de spécification vise à fournir un support pour la rédaction d'un tel document, qui servira de guide constant dans les phases ultérieures de la construction du logiciel (conception, programmation, mise au point, entretien).

Parmi les langages de spécification existants, certains comme Sadt [I.b. 34] sont non formels et destinés plutôt à la communication avec les commanditaires du système; d'autres comme Special ou Z [I.b. 1] sont à base mathématique et privilégient l'objectif de fiabilité, en liaison avec les travaux sur la démonstration, la transformation et la synthèse de programmes.

Le développement des langages de spécification en est encore à ses débuts et leur utilisation reste marginale. Nul doute cependant que leur étude approfondie fournira en retour une meilleure compréhension des problèmes soulevés par les langages de programmation eux-mêmes, qui, indépendamment des caractéristiques liées à leur exécution sur telle ou telle machine, sont le témoignage de l'effort le plus sérieux jamais entrepris par l'humanité pour créer des systèmes de signes cohérents, puissants et rigoureux.

Pour trouver un renseignement, consulter la table alphabétique H 12.

INDEX BIBLIOGRAPHIQUE

Des milliers d'articles, de communications et de livres ont été consacrés aux langages de programmation, et il n'est pas question de fournir une liste même simplement représentative. On s'est borné à citer:

- les références mentionnées dans le texte de cet article;
- quelques ouvrages généraux sur les langages, indiqués par (*);
- quelques monographies sur des langages particuliers, non citées dans les autres articles de ce traité, et écrites en français (à l'exception de [I.b. 9 et 24]); ces documents sont signalés par (*).

1. ABRIAL (J.R.), SCHUMAN (S.A.) et MEYER (B.). – *Specification language.* Proceedings of Belfast Summer School on the Construction of Programs, sept. 1979 Cambridge University Press.
2. *ACM Sigplan history of programming languages conference.* ACM Preprints. Sigplan Notices (USA) 13 n° 8 juin 1978. (*).
3. *Manuel du langage algorithmique Algol 68.* AFCET. Groupe Algol. 1975. Hermann. (*).
4. *Panorama des langages d'aujourd'hui.* AFCET. Groupe Groplan (Programmation et Langages). Réunion de Cargese (Corse) 14-22 mai 1979 Bull. Groplan (F) n° 8 et 9 1979. (*).
5. BACKUS (J.). – *Can programming be liberated from the von Neumann style? A functional style and its algebra of programs.* Communications of the ACM 21 n° 8 août 1978 p. 613-41.
6. BARRON (D.W.). – *An introduction to the study of programming languages* 1977 Cambridge University Press (*).
7. BEMER (R.W.). – *A politico-social history of Algol* dans: HALPERN (M.I.) et SHAW (C.J.). – *Annual review on automatic programming.* 1969 Pergamon Press.
8. BERTHET (C.). – *Le langage PL/I* 1971 Dunod. (*).
9. BIRTWISTLE (G.M.), DAHL (O.J.), MYRHAUG (B.) et NYGAARD (K.). – *Simula BEGIN;* 1973 Studentlitteratur, Lund (Suède). Petrocelli-Charter (USA). (*).
10. BROWN (P.J.). – *Software portability, an advanced course.* 1977 Cambridge University Press.
11. CHION (J.S.) et CLEEMAN (E. F.). – *Le langage Algol W, initiation aux algorithmes.* 1973 Presses univ. Grenoble. (*).
12. CHOMSKY (N.). – *Syntactic structures;* 1957 Mouton traduction française: *Structures syntaxiques* 1969 Le Seuil.
13. CROCUS. – *Systèmes d'exploitation des ordinateurs. Principes de conception.* 1975 Dunod.
14. DAHL (O.J.), DIJKSTRA (E.W.) et HOARE (C.A.R.). – *Structured programming.* 1972 Academic Press.
15. DEMARS (G.), RAULT (J.-C.) et RUGGIU (C.). – *Les langages et les systèmes APL.* 1974 Masson. (*).
16. – DIJKSTRA (E.W.). – *Notes on structured programming.* Cf I.b. 14.
17. DEMUYNCK (M.) et MEYER (B.). – *Les langages de spécification.* Journées de Pont-à-Mousson sur le Génie Logiciel. IRIA. 1979. Également dans Bull. Direction des Études et Recherches EDF, Série C (informatique) n° 1 1979 p. 39-60.
18. DONAHUE (J.E.). – *Complementary definitions of programming languages.* 1976 Springer Verlag.
19. GRIES (D.). – *Compiler construction for digital computers.* 1971 Wiley.
20. GUTTAG (D.). – *Abstract data types and the development of Software.* Communications of the ACM 20 n° 6 juin 1977, p. 396-404.
21. HALSTEAD (M.). – *Elements of software science.* 1977 North-Holland/Elsevier. Voir aussi pour une introduction: CHAMPENOIS (M.). – *Physique du logiciel. Aspects théoriques et expérimentaux;* Rairo-Informatique (AFCET), Série bleue; *14* n° 1 1980, p. 3-23.
22. HIGMAN (B.). – *A comparative study of programming languages.* 1967 McDonald-Elsevier. Traduction française: *Étude comparative des langages de programmation.* 1973 Dunod.
23. HOARE (C.A.R.). – *Notes on data structuring.* Cf I.b. 14.
24. ICHBIAH (J.) et coll. – *Preliminary ADA reference manual. Rationale for the design of the ADA programming language.* Sigplan Notices (USA) *14* n° 6 parties A et B (numéro spécial, deux volumes) juin 1979. (*). *Ada reference manual-Proposed standard document.* US DoD juil. 1980.
25. – JAKOBSON (R.). – *Poésie de la Grammaire et Grammaire de la Poésie* dans: *Huit questions de poétique* 1977 Le Seuil; cf aussi SAPIR (E.). – Language 1921 Harcourt, Brace and World; traduction française: *Le langage* 1960 Payot.
26. – KENNEDY (K.) et SCHWARTZ (J.). – *An introduction to the set theoretical language Setl.* J. Computer and Mathematics with Applications *1* 1975 p. 97-119. (*).
27. LARMOUTH (J.). – *Serious Fortran.* Software, Practice and Experience (GB) *3* 1973 1re partie 87-107 et 2e partie 197-225. (*).
28. LIVERCY (C.). – *Théorie des programmes.* 1978 Dunod.
29. LECARME (O.). – *Une famille de langages de programmation: Snobol, SL5 et Icon,* Bull. Groplan n° 10 1980 p. 1-46. (*).
30. MEYER (B.) et BAUDOIN (C.). – *Méthodes de programmation* 1978 Eyrolles. (*).
31. MEYER (B.). – *Sur quelques concepts modernes des langages de programmation et leur représentation en Simula 67.* Cf I.b. 4 p. 421-832. (*).
32. NICHOLLS (J.E.). – *The structure and design of programming languages.* 1975 Addison-Wesley. (*).
33. RIBBENS (D.). – *Programmation non numérique. Lisp 1.5.* 1969 Dunod. (*).
34. ROSS (D.T.) et coll. – *Special section on requirements.* I.E.E.E. Trans. on Software Engng. (USA). *SE-3* n° 1 janv. 1977 pages 2-85.
35. – SAMMET (J.E.). – *Programming languages: history and fundamentals.* 1969 Prentice-Hall. (*). Ce recensement de langages fait l'objet de mises à jour annuelles dans les *Communications* de l'ACM.
36. SCHUMAN (S.A.). – *Proceedings of International Symposium on Extensible Languages Grenoble, septembre 1971.* ACM Sigplan Notices *6* n° 12 déc. 1971.
37. TANENBAUM (A.S.): *Structured computer organization.* 1976 Prentice-Hall.
38. WILKES (M.V.). – *The outer and inner syntax of a programming language.* Computer J. (GB) mars 1968.
39. WIRTH (N.). – *Algorithms + data structures = programs.* 1976 Prentice-Hall. (*).

[81a]

Algorithmic Languages, de Bakker/van Vliet (eds.)

, 99–114

The Design of Vector Programs

Alain Bossavit and Bertrand Meyer

Direction des Etudes et Recherches, Electricité de France, Clamart, France

Current vector computers such as the Cray-1, Cyber 205 S1, DAP or BSP pose a special challenge to the software designer as the available software tools and techniques are far behind the hardware developments, and the goals of efficient vector programming seem to conflict with some of the basic principles of good software engineering. After studying some properties of these computers, with particular emphasis on the Cray-1, we purport to show that a systematic approach to vector programming is possible and fruitful; the proposed methods are applied to the systematic, proof-oriented derivation of several vector algorithms. Language aspects are also considered.

1. Introduction

The advent of 'second-generation' vector processors [8] such as the Cray-1, CDC Cyber 205, Lawrence Livermore Laboratory S1, ICL DAP and Burroughs BSP, is one more piece of evidence for the fact that software lags far behind hardware as far as practical industrial usage is concerned. These computers, built with the latest LSI or VLSI technology in highly optimized architectures, are capable of achieving speeds which were unheard of before: for example, a Cray-1 computer will in good conditions carry out more than 100 million 'actual' operations, excluding control, per second. On the other hand, a look at the software provided with these 'super-computers' will show them to be what may be called **Fortran machines:** even though processors for other languages may exist, these computers are obviously tailored to a philosophy of programming which has the static array as its only data structure and the DO-loop as its main control structure. Recipes given for writing efficient programs in that

framework [6], seem at first glance to be very far from modern ideas about programming, if not incompatible with them.

Vector programming thus appears as a challenge for the software specialist. Areas where advances are needed include the following interrelated topics:

(1) algorithmics (algorithms for vector processing, and methods for finding such algorithms);

(2) program design (how to find program and data structures which will lead to efficient use of supercomputers while ensuring other program qualities such as reliability, clarity, portability, modularity, etc.);

(3) program transformation (methods for adapting existing programs to efficient execution on vector computers);

(4) languages for vector programming;

(5) proof methods.

The aim of this paper is to lay some foundations for a systematic treatment of vector programming. It is mostly concerned with (1) and (2), with a brief discussion of (4).

The particular machine which motivated this study is the Cray-1 computer, which seems to be the most widely available among the 'second generation' vector machines, and is quoted as the fastest currently available computer, even in scalar mode [4, 8]. Most of the discussion is, however, also valid for the other machines.

In Section 2, we give a software interpretation of the rules which must be obeyed by a computation in order to be able to use the vectorization capabilities of the hardware. In Section 3, we give a more abstract interpretation of these rules in terms of the data types involved. Section 4 discusses language problems. Section 5 is devoted to a study of systematic program construction techniques applied to vector programming; several algorithms, in particular a 'vector Cholesky', are derived.

2. Rules for Vectorization

Vector machines require that a program satisfy certain conditions in order to be vectorizable, i.e. amenable to processing in vector, as opposed to scalar, mode. The study of these conditions is particularly interesting in the case of vector computers such as the Cray-1 or BSP which accept standard FORTRAN, so that vectorization rests with the compiler rather

than the programmer. Abstracting from machine peculiarities, five basic conditions appear as necessary and sufficient:

- repetitive series of operations;
- primitive operations only;
- regularity;
- no backward dependency;
- no cross dependency.

These conditions are studied in [12] for the Cray case. We shall outline them here in general terms.

2.1. *Repetitive series of operations*

The only sequences amenable to vectorization are loops, and, more precisely, ***for*** loops, i.e. counter loops with a number of executions known at the outset. The ***for*** loop control structure, associated with the array data structure, is the software representative of the so-called SIMD (*Single Instruction stream, Multiple Data stream*) mode of restricted parallelism.

2.2. *Primitive operations only*

With some slight extensions, only assignments and numerical or boolean operations are allowed in a vector loop. This precludes in particular jumps, thence conditional statements other than conditional assignments. The Cray-1 Fortran compiler (CFT) will also inhibit vectorization of a loop containing a subprogram call (except the subprogram is known to CFT as having a vector version) or another loop (thus restricting vectorization to the innermost loops).

2.3. *Regularity*

For a loop to be vectorizable, it must involve only 'regular' array elements, i.e. elements whose indices follow a strictly defined pattern, so that they can be fetched in advance for vector operations. On the Cyber 205, the only regular elements are those which are stored contiguously; on the Cray-1, a sequence is regular iff the distance between successive elements is constant (but not necessarily 1). Thus only certain types of subarrays may be processed in vector mode.

2.4. *No backward dependency*

Let a loop with i as a counter contain the following array element assignment:

$$a[f_0(i)] := op(b_1[f_1(i)], b_2[f_2(i)], \ldots, b_m[f_m(i)])$$

where ALGOL-like brackets are used for array elements, op is some numerical or logical operation, the f_k's are linear functions (from the regularity rule), and all arrays are considered as one-dimensional (which is always possible on a machine with a linear store).

This assignment has a backward dependency, which will inhibit vectorization, iff for some k ($1 \le k \le m$) b_k is a, and for some pair of values p, q in the range of i, the following holds:

$$p < q \text{ and } f_k(p) = f_0(q).$$

In other words, the computation of $a[f_0(q)]$ will use the value of another element of a, which was fetched for updating in some previous iteration. For example, the assignment $a[i] := a[i-1] + 1$ introduces a backward dependency.

The reason for this rule is that the vector interpretation of such a computation would use the old value of the array element, not the new one as in the standard (sequential) interpretation of the loop.

Note that the vector interpretation makes perfect sense; it is only different from the sequential one.

On the Cray-1 the condition is less stringent; a backward dependency will actually arise only if the above condition holds together with

$$q - 64 < p$$

where 64 is the length of the vector registers, which on the Cray must be used for the operands and results of vector operations (in contrast, the Cyber 205 and BSP work directly on vectors stored in memory). Vector processing on the Cray-1 may be considered, for all practical purposes, as successive processing of 64-element vector slices, all elements in a slice being processed in parallel.

An important case of backward dependency occurs when the dependency affects a simple variable (which may be considered as a one-element array, whose index is constant through the loop), i.e. when the loop contains an assignment of the form

$$x := op(x, b_1[f_1(i)], b_2[f_2(i)], \ldots).$$

Such an operation is called a reduction; it is particularly unfortunate that it should not vectorize, since it corresponds to the very common case of accumulating a result into a variable, as in the computation of the sum of the elements of a vector, or of the scalar (inner) product of two vectors. In practice, techniques exist for reducing the loss of efficiency of reductions as compared to truly vectorizable operations; reductions may thus be thought of as 'pseudo-vectorizable' operations who execute more slowly than vectorizable operations but faster than scalar ones.

2.5. *No cross dependency*

Let a loop contain the following assignments:

$$a[f_0(i)] := op(\ldots);$$

$$c[g_0(i)] := op'(\ldots, a[g_1(i)], \ldots).$$

They induce a cross dependency, which will inhibit vectorization, iff for some pair of values p, q in the range of i, the following holds:

$$g_1(p) = f_0(q)$$

with $|q - p| < 64$ (on the Cray-1).

For example, the following statements in a loop on i will cause a cross dependency:

$$a[i] := 1; \qquad c[i] := a[i+1].$$

The rule stems from the fact that, due to the limited size of the instruction buffers, long loops may have to be split into several shorter ones in order to be vectorized (by slices of 64 on the Cray); thus the two assignments might end up in two different loops, giving a different semantics for the program. In our example, assuming a was initially all 0, then c would receive the previous null values in the sequential case and the new unity values in the vector case.

3. Basic Thoughts for a Vector Programming Methodology

Considering the preceding rules, even though they do not include many details which may be found in manufacturers' documentation, it is quite tempting to dismiss them as too low-level and machine-dependent, and

assert that vector programming is just programming with objects of data type 'vector'. Although we will use this definition as the basis for our approach to vector program construction, it should be pointed out that it is not quite sufficient and that the previous rules, especially the last ones on dependency, must also be taken into account for practical purposes.

Let us illustrate this point with an important vector algorithm: matrix multiplication. Assume c is initialized to zero; a, b, c have dimensions (m,n), (n,p) and (m,p) respectively. The ordinary algorithm will not vectorize (notations are mostly taken from [11]):

```
for i in 1, ..., m do
    for j in 1, ..., p do
        for k in 1, ..., n do
            c[i,j] := c[i,j] + a[i,k] * b[k,j]
```
(3.1)

In terms of the preceding rules, we may say that $c[i,j]$ has a backward dependency on itself (the last line is a reduction). Now if we reverse the loops on j and k, the program becomes vectorizable. This in fact means that instead of the 'element' formula which forms the basis for algorithm (3.1):

$$c[i,j] = \sum_{k=1}^{n} a[i,k] * b[k,j]$$

one relies on the 'vector' formula

$$c[i,*] = \sum_{k=1}^{n} a[i,k] * b[k,*]$$

(where $x[i,*]$ and $x[*,j]$ respectively denote the ith line and jth column of matrix x).

However, if we applied a purely functional view of vector programming, i.e. obtained a program directly from an 'abstract data type' specification of matrix multiplication, the initial version of our program, as deduced from the last formula, would require, for each line i, n vector variables:

$c_1[i,*] := a[i,1] * b[1,*];$
$c_2[i,*] := a[i,2] * b[2,*] + c_1[i,*];$
...
$c_m[i,*] := a[i,m] * b[m,*] + c_{m-1}[i,*];$
$c[i,*] := c_m[i,*].$

For practical reasons (storage) this is excluded; the same variable $c[i,*]$ has to be used all along. This programming simplification is correct because it does not conflict with the no backward dependency rule, as every operation of the form

$$c[i,*] := op(c[i,*])$$

will be implemented as a counter loop whose body is $c[i,j] := op(c[i,j])$ without any reference to $c[i,l]$ for $l \neq j$ (note that the loop counter here is j). This condition guarantees that the vectorized form of the new version (i.e. the standard program where loops on j and k have been interchanged) is indeed semantically equivalent to the standard program.

Such a condition, which is more restrictive but conceptually simpler than the no backward dependency rule, may be used as a replacement for it in a systematic approach. It can be formalized in the following way, inspired from the presentation of sequences in the specification language Z [1]. Let $VEC\,X[(n)]$, for $n \in \mathbb{N}$ (the set of n-vectors of elements of X) be defined as the set of all total functions from $1, \ldots, n$ to X. Let & be the functional binary operator such that, if f and g are two functions with the same domain Y, then $f \& g$ is the function h such that, for any $y \in Y$, $h(y)$ is the pair $(f(y), g(y))$. Then for any binary operation p on X ($p : X \times X \to Z$ for some Z) we may define a vector extension of p, $ext(p) : VEC[X](n) \times VEC[X](n) \to VEC[Z](n)$, whose value for any two vectors v and w in $vec[X](n)$ is

$$ext(p)(v,w) = p \circ (v \& w)$$

where $\circ$ is functional composition; in other words, for any $i \in 1, \ldots, n$,

$$ext(p)(v,w)(i) = p(v(i), w(i)).$$

It is possible to define in the same way (at least if p is associative) a vector reduction of functionality

$$red(p) : VEC[X] \to X$$

where $red(+) = \sum$, etc.

We shall interpret the rules of Section 2 as implying that, in designing programs for vector computers, one should work on objects of data type vector, restricting oneself to extension operations as much as possible. When an extension operation cannot be applied, a reduction will still be preferable to operations which would perform arbitrary shifting of indices

(e.g. $p \circ ((v \circ pred) \& w)$, where *pred* is the predecessor function on integers, which would give $p(v(i-1), w(i))$ for any i); such operations would introduce hopeless backward dependencies.

The situation may be depicted using a hierarchy of abstract machines (Fig. 1). At the matrix level, machine MAT offers the operations of matrix algebra: multiplication, inversion, etc. At the vector level, several machines are available to implement these operations: the extension machine EXT, the reduction machine RED, and others. Choosing one of them will lead to a definite algorithm , the scalar machine SCAL, which corresponds to conventional programming languages. It is clear that the standard matrix multiplication algorithm given above (3.1) stems from the RED machine, while its vectorizable counterpart will come out naturally if one uses the EXT machine.

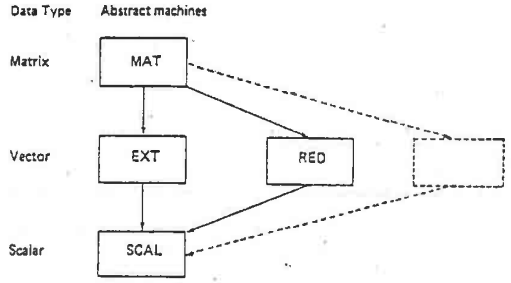


Fig. 1. Hierarchy of types and virtual machines.

Using the above approach, we will derive vector algorithms by working on vector objects from the beginning. This should lead to programs which are both properly structured and efficient on a vector processor. This should be contrasted with the results obtained through more 'ad hoc' methods. For example Higbie [6], in a paper on how to write code which will vectorize on the Cray, warns that 'overly modular or structured programs' will not be vectorizable (because of the rule which we called 'primitive operations only', precluding subprogram calls inside a vectorizable loop). If this were true, the situation might be considered quite sad for the programmer, forced to choose between structure and vectorization. On the other hand, if one agrees that a program is 'structured' at least as much from its proper adequation of control structure to data structure as from its observance of rules regarding control structure only (e.g. many subprograms, etc.), then the answer is clear: rather than in-line expansion of subprogram calls in loop bodies, one should strive to write subprograms working on entire arrays (to use expressions found in Cray publications, "put the loop in the subroutine rather than the subroutine in the loop"). This will, in effect, implement the 'vector' data type abstraction. If the program is indeed vectorizable, i.e. if it does have vectors as its principal objects, there is a good chance that the version thus 'vectorized' will be clearer and better 'structured' independently of any machine consideration.

4. Language Considerations

Before we turn to the derivation of a few vector algorithms, we must pay some attention to language issues. The Cray approach uses a standard language, FORTRAN, and places the task of detecting vectorizable portions of code upon the compiler. The BSP also has a 'vectorizer' for standard FORTRAN code (an introduction to the techniques used for such program transformations may be found in [10]). Other methods have been used or suggested (see [9] or [14] for a survey); for example, the Cyber 205 supercomputer only vectorizes calls to special array processing subroutines. Perrott [14, 15, 16] has argued repeatedly in favor of using a language designed specifically for vector programming; he describes such a language, ACTUS, based on PASCAL. This approach can be justified on several grounds:

- In the Cray and BSP approach to optimization, the programmer has to present his code in a 'favorable' way so that the compiler will be able to detect vectorizable pieces of code; he thus has to know the compiler's idiosyncracies in this respect. This, however, has to be balanced with the considerations on program structuring expressed above.
- The search for vectorizable code amounts to de-compilation (reconstructing higher-level vector constructs, such as they might be expressed in ALGOL 68, PL/I or APL, from lower-level FORTRAN scalar operations), which is a rather silly activity;
- It is quite natural to specify the amount of allowable parallelism in

connection with the data structure definition rather than with the description of the operations performed on it.

On the other hand, the 'vector language' approach seems extremely difficult to implement in the context of a large scientific computing center (the typical target for supercomputers), where it is not realistic to imagine that programmers will turn to a new language for every new kind of application and every new machine – especially at a time when concerns for portability are at last making their way into the scientific programming community.

Given the failures experienced by all previous efforts to impose languages other than FORTRAN to this community, it is doubtful that a proposal applying to vector computers would succeed. In view of the current state of the art, the Cray approach seems sensible as far as program coding is concerned. Languages such as ACTUS may, however, be very useful as intermediary notations for vector program design, and we shall use similar ways of expression in the examples which follow.

5. Examples of Systematic Vector Program Construction

We turn now to the application of the principles expounded in Section 3 to the construction of some practical programs. We shall use a method and set of heuristics for constructing programs from specifications which were exposed in [13]. A similar approach was applied to classical (scalar) numerical algorithms in [2].

The following notation will be used in addition to the ones defined in Section 3:

- $VEC(n)$ stands for $VEC[REAL](n)$, the set of vectors of n real elements;
- $MTR(m,n)$ is the set of (m,n) real matrices;
- $P_l v$, where $v \in VEC(n)$ and $l \leq n$, is the projection of v on $VEC(l)$.

For a matrix $s \in MTR(m,n)$, if $i \leq m$ and $j \leq n$, we will consider line $s[i,*]$ and column $s[*,j]$ as vectors in $VEC(m)$ and $VEC(n)$ respectively.

5.1. *Triangular systems*

We saw in Section 3 a vector algorithm for matrix multiplication. Let us proceed with the inverse operation: solving linear systems. We first examine triangular systems. This will be a simple example of top-down synthesis of a numerical algorithm.

The first step in the design of the program (called *trisolv*) is to express it as a matrix algorithm (which could run on the virtual machine MAT):

in s: $MTR(n,n)$, b: $VEC(n)$; *out* x: $VEC(n)$;
(P) $\{1 \leq i \leq n \Rightarrow P_{i-1} s[*,i] = 0$ *and* $s[i,i] \neq 0\}$
trisolv
(Q) $\{sx = b$, i.e. $\sum_{k=1}^{n} s[*,k] * x[k] = b\}$

We must refine *trisolv* into a predicate transformer (on the vector machine EXT) from the precondition (P) to the postcondition (Q). Let us try twice the heuristic called 'uncoupling' [13], i.e. add an auxiliary vector variable y, and an integer one l, noticing that

$$(Q) \Leftrightarrow b = \sum s[*,k] * x[k] \Leftrightarrow (y + \sum_{k \leq n} s[*,k] * x[k] = b \text{ and } y = 0)$$
$$\Leftrightarrow (y + \sum_{k \leq l} s[*,k] * x[k] = b \text{ and } P_l y = 0) \text{ and } l = n.$$

So $(Q) \Leftrightarrow (I(l)$ *and* $l = n)$ if we set $I(l) =$ the first term of the *and* above. Here, $I(l)$ is a 'weakening' of the exit condition (Q) (which is $I(n)$). We notice that $I(0)$ can be trivially obtained. Thus a refinement of *trisolv*, using $I(l)$ as an invariant and $l = n$ as the goal (exit condition) will be:

```
var l: Integer;
l := 0; y := b{I(l)}
while l < n do
  l := l + 1;
  reestablish I(l);
{l = n and I(l)}
```

This program is correct (by construction): $I(l)$ being a loop invariant, it is true after the completion of the loop, and the exit condition $l = n$ is also true, hence $I(n)$. The statement *reestablish* is now (just as *trisolv* was, one step backwards) a specification for what is to be done.

Next step: develop *reestablish*. One must go from $I(l-1)$, i.e.

$$y + \sum_{k<l} s[*,k] * x[k] = b \text{ and } P_{l-1} y = 0$$

to $I(l)$, i.e.

$$y + \sum_{k<l} s[*,k] * x[k] = b - s[*,l] * x[l] \text{ and } P_l y = 0.$$

Without modifying b, which is part of the input, we must use the assignment $y := y - s[*, l] * x[l]$ after an $x[l]$ such that $P_l(y - s[*, l] * x[l]) = 0$ has been found. But $P_{l-1}s[*, l] = 0$ by hypothesis, and $P_{l-1}y = 0$ also. The equation thus becomes $y[l] - s[*, l] * x[l] = 0$, thence $x[l]$. The final version of the program is:

```
l := 0; c := b; I(0)
while l < n do
| l := l + 1;
| {reestablish I(l) :}
| | x[l] := y[*, l]/s[l, l]
| | y := y - s[*, l] * x[l]
```

Starting from a matrix specification and aiming at the EXT vector target machine, we have just synthesized a program which must be, by construction, vectorizable.

5.2. *Vectorized Choleski*

We shall now introduce a more difficult algorithm, Choleski factorization: given a symmetric positive-definite matrix A, find a lower triangular S such that $SS^t = A$ (in view of the resolution in two easy steps, using e.g. the above program, of the linear system $Ax = b$). What follows is also valid for the LU factorization.

We again apply systematic top-down synthesis. Here are the successive steps. First the specification, expressed in terms of MAT objects:

```
                in a: MTR(n, n); out s: MTR(n, n);
(R)             {symmetric(a) and positive-definite(a)}
                | Choleski
                | {1 ≤ i ≤ n ⇒ P_{i-1}[*, i] = 0}
(S)             | {A = SS^t, i.e. a = Σ_{k≤n} s[*, k] * s[*, k]}
```

As before, we uncouple (S), after introducing the auxiliary variable c of type $MTR(n, n)$:

$$(S) \Leftrightarrow ((c + \textstyle\sum_{k \le l} s[*, k] * s[*, k] = a \text{ and } P_l c = 0) \text{ and } l = n)$$
$$\Leftrightarrow \quad (I(l) \text{ and } l = n).$$

The next refinement is, quite naturally:

```
l := 0; c := a; {I(0)}
while l < n do
| l := l + 1; {c + Σ_{k<l} = a and P_{l-1}c = 0}
| reestablish I(l);
| {c + Σ_{k<l} = a - s[*, l] * s[*, l] and P_l c = 0}.
```

To *reestablish* $I(l)$, one must perform the assignment $c := c - s[*, l] * s[*, l]$ once an $s[*, l]$ such that $P_{l-1}s^l = 0$ and

$$P_l(c - s[*, l] * s[*, l]) = 0$$

has been found. As $P_{l-1}c = 0$, row l is the only one concerned, and must satisfy l-*column*$(c - s[*, l] * s[*, l]) = 0$, that is to say $c[l, *] - s[l, l] * s[*, l] = 0$, which implies ($l$ component)

$$c[l, l] = (s[l, l])^2.$$

Thence the two instructions for *reestablish* $I(l)$:

$$s[l, l] := sqrt(c[l, l])\text{; } s[*, l] := c[l, *]/s[l, l].$$

As c is symmetric (this fact is itself a loop invariant), $P_{l-1}c[*, l] = 0$ implies $P_{l-1}c[l, *] = 0$, therefore $P_{l-1}s[*, l] = 0$.

The final version will thus be:

```
l := 0; c := a;
while l < n do
| l := l + 1;
| pivot := sqrt(c[l, l]);
| s[*, l] := c[l, *]/pivot;
| c := c - s[*, l] * s[*, l]
```

A FORTRAN translation appears on Fig. 2 and 3. It exhibits some of the nice properties of programs resulting from top-down design (high-level built-in documentation, etc.) and the safety guaranteed by the systematic synthesis method.

```
      SUBROUTINE
     *                    C H O L V E C
     *                   (N, A, S, NDP)
C
C                          PURPOSE:
C
C     FACTORIZATION OF A SYMMETRIC MATRIX, VECTORIZABLE VERSION.
C
C     INPUT
C
            INTEGER    N
C                                 Order of the matrix A
            REAL       A   ( 1 )
C                                 Array of the entries of A. Aij is at
C                                 the position ((J - 1)(2N - J) + 2I)/2
C                                 ('column-symmetric storage mode')
C     OUTPUT
C
            REAL       S   ( 1 )
C                                 Array of the entries of A. Aij is at
C                                 the position ((J - 1)(2N - J) + 2I)/2
C                                 On exit, if NDP = N, A = S tr(S)RT
            INTEGER    NDP
C                                 Number of columns actually taken into
C                                 account during the factorization.
C                                 If NDP < N, a non-positive radix ap-
C                                 peared in the treatment of column
C                                 NDP + 1
C-----------------------------------------------------------------------
C
C     LOCAL VARIABLES:
C
            INTEGER L, NNP1S2, ADRLL, ADRJL, ADRJJ, I, J, LP1
            REAL PIVOT, MUL, RADIC
C
C     ARITHMETIC FUNCTION:
C
            INTEGER ADDRESS.
            ADRESS(I, J) = ((J - 1)*(2*N - J) + 2*I)/2
C
```

Fig. 2. Head of the vectorizable Choleski program (FORTRAN).

6. Conclusion

The field of numerical and scientific programming, although the oldest and one of the best established among the application domains of computers, has shown strong resistance to the practical implementation of software research and advances in programming methodology. With the popularization of new 'number-crunching' machines, there is again a strong temptation to go back to low-level, machine-dependent, programming techniques, and to dismiss any attempts at better software engineering as incompatible with the efficient use of these very fast computers. We hope to have shown that such an attitude has no justification, and that systematic methods can be applied for the rational and efficient use of this new technology.


```
C-----------------------------------------------------------------------
C
            NDP = 0
C     i <--- 0 ;
              L = 0
C     C <--- A ;
            NNP1S2 = (N*(N + 1))/2
            DO 1 I = 1, NNP1S2
1             S(I) = A(I)
C             -- The array S contains both C and A
C     while i < n do
2     IF (L .GE. N) GOTO 7
C           i <--- i + 1 ;
                L = L + 1
                ADRLL = ADRESS(L, L)
C           pivot <--- sqrt(C11) ;
                RADIC = S(ADRLL)
                IF (RADIC .LE. 0.) GOTO 7
C                 -- Exception if A is not positive definite
                PIVOT = SQRT(RADIC)
                NDP = L
C           S1 <--- C1/pivot ;
                DO 3 I = L, N
3                 S(ADRLL + I - L) = S(ADRLL + I - L)/PIVOT
C           C <--- C - S1 * S1 ;
                LP1 = L + 1
                IF (LP1 .EQ. N) GOTO 6
                DO 5 J = LP1, N
                     ADRJJ = ADRESS(J, J)
                     ADRJL = ADRESS(J, L)
                     MUL = S(ADRJL)
                     DO 4 I = J, N
C                         S(ADRJJ+I-J) = S(ADRJJ+I-J) - MUL*S(ADRJL+I-J)
C                         -- This loop is the only vectorizable one
4                         CONTINUE
5                    CONTINUE
6             CONTINUE
              GOTO 2
7     RETURN
      END
```

Fig. 3. Body of the Choleski program.

References

[1] J.R. Abrial, S.A. Schuman and B. Meyer, Specification language, in: Proceedings Summer School on Program Construction, Belfast (September 1979).

[2] A. Bossavit and B. Meyer, On the constructive approach to programming: the case for partial Choleski factorization (a tool for static condensation), in: Vichnevetsky and Stepleman (Eds.), Advances in Computer Methods for Partial Differential Equations III (IMACS, 1979).

[3] Cray-1 Computer System, FORTRAN (CFT) Reference Manual, Cray Document No. 2240009, Version E (1981).

[4] M. Dungworth, The Cray 1 computer system, in: Infotech State of the Art Report on Supercomputers, Volume 2: Invited papers (Maidenhead, 1979) pp. 51–76.

[5] P.M. Flanders, FORTRAN extensioms for a highly parallel processor, in: Infotech State of the Art Report on Supercomputers, Volume 2: Invited Papers (Maidenhead, 1979) pp. 117–134.

[6] L. Higbie, Vectorization and conversion of FORTRAN programs for the Cray-1 (CFT) compiler, Cray Document No. 2240207 (June 1979).

[7] Infotech State of the Art Report on Supercomputers, Volume 1: Total Systems Issues; Volume 2: Invited papers (Maidenhead, 1979).

[8] E.W. Kozdrowicki and D.J. Theis, Second-generation of vector supercomputers, Computer (IEEE), Special Section on Supersystems for the 80's 13 (11) (1980) 71–83.

[9] D.J. Kuck, Languages and compilers for parallel and pipeline machines, in: CREST Conference on Design of Numerical Algorithms for Parallel Processing, Bergamo, Italy (June 1981).

[10] D.J. Kuck, Automatic program restructuring for high-speed computation, in: W. Händler (Ed.), CONPAR 81, Nürnberg, June 1981, Lecture Notes in Computer Science 111 (Springer, Berlin, 1981) pp. 66–84.

[11] B. Meyer and C. Baudoin, Méthodes de programmation (Eyrolles, Paris, 1978).

[12] B. Meyer, Un calculateur vectoriel: Le Cray-1 et sa programmation, EDF Report HI/3452-01, Atelier logiciel No. 24 (May 1980).

[13] B. Meyer, A basis for the constructive approach to programming, in: S.H. Lavington (Ed.), Information Processing 80 (North-Holland, Amsterdam, 1980).

[14] R.H. Perrott, Parallel languages, in: Infotech State of the Art Report on Supercomputers, Volume 1: Total Systems Issues (Maidenhead, 1979) pp. 117–149.

[15] R.H. Perrott, A standard for supercomputer languages, in: Infotech State of the Art Report on Supercomputers, Volume 2: Invited Papers (Maidenhead, 1979) pp. 291–308.

[16] R.H. Perrott, A language for array and vector processors, TOPLAS (Transactions on Programming Languages and Systems, ACM) 1 (2) (1979) 177–195.

Résumé

Les recherches présentées dans cette thèse sur travaux couvrent une période de dix ans (1975-1985). Elles portent sur divers aspects du génie logiciel et plus particulièrement de la méthodologie de la programmation ; on y trouvera des études sur la structure macroscopique des systèmes logiciels, la spécification formelle, la modularité, les structures de données, l'algorithmique, les langages de programmation, la conception par objets, les méthodes de construction systématique de programmes, les outils logiciels interactifs, l'édition structurelle et la programmation des super-ordinateurs.

Vint-et-un articles sur ces différents sujets sont regroupés dans la troisième partie de la thèse. La première partie est un document de synthèse sur les principaux problèmes du génie logiciel ; la seconde, une introduction thématique aux articles qui suivent.

MOTS-CLES

Génie logiciel, méthodologie de la programmation, systèmes interactifs, outils logiciels, super-ordinateurs, programmation vectorielle, modularité, spécification formelle, conception et programmation par objets, construction systématique de programmes, langages de programmation, structure des systèmes logiciels, structures de données, algorithmes.