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 ?**	H 2 040-2
1,1	Définition	– 2
1,2	Niveaux	– 3
1,3	Langages généraux. Langages spécialisés	– 3
1,4	Mise en œuvre	– 3
2.	**Ossature d'un langage**	– 4
2,1	Syntaxe et sémantique	– 4
2,2	Données	– 4
	2,21 Droits d'accès : constantes et variables	– 5
	2,22 Notion de type	– 5
	2,23 Types prédéfinis	– 5
	2,24 Données composées	– 5
2,3	Calculs	– 7
	2,31 Opérations. Structures de contrôle	– 7
	2,32 Instructions. Expressions	– 8
2,4	Modularité	– 8
	2,41 Sous-programmes	– 9
	2,42 Classes	– 9
2,5	Syntaxe	– 10
3.	**Conception et choix d'un langage**	– 12
3,1	Enjeu	– 12
3,2	Grands critères	– 12
	3,201 Homogénéité. Régularité	– 12
	3,202 Orthogonalité	– 12
	3,203 Simplicité	– 12
	3,204 Généralité	– 12
	3,205 Extensibilité	– 13
	3,206 Compilabilité	– 13
	3,207 Clarté	– 13
	3,208 Sécurité et fiabilité	– 13
	3,209 Souplesse et commodité d'emploi	– 14
	3,210 Puissance expressive	– 14
	3,211 Complétude de la définition et portabilité	– 14
3,3	Méthodes : définitions formelles	– 14
	3,31 Intérêt d'une étude formelle	– 14
	3,32 Syntaxe	– 14
	3,33 Sémantique statique	– 15
	3,34 Sémantique	– 15
4.	**Un peu d'histoire**	– 17
4,1	Première génération : les pionniers	– 17
	4,11 Fortran	– 17
	4,12 Algol	– 17
	4,13 Lisp	– 17
	4,14 Cobol	– 17
4,2	Deuxième génération : l'ambition	– 18
	4,21 PL/I	– 18
	4,22 Algol 68	– 18
	4,23 Algol W. Pascal	– 18
	4,24 Simula 67	– 19
	4,25 Snobol	– 19
	4,26 APL	– 19
4,3	Troisième génération : l'industrialisation	– 19
5.	**Au-delà des langages de programmation**	– 20
5,1	Langages et progiciels	– 20
5,2	Langages pour non-informaticiens	– 21
5,3	Langages de très haut niveau	– 21
5,4	Langages de spécification	– 21
Index bibliographique		– 22

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;

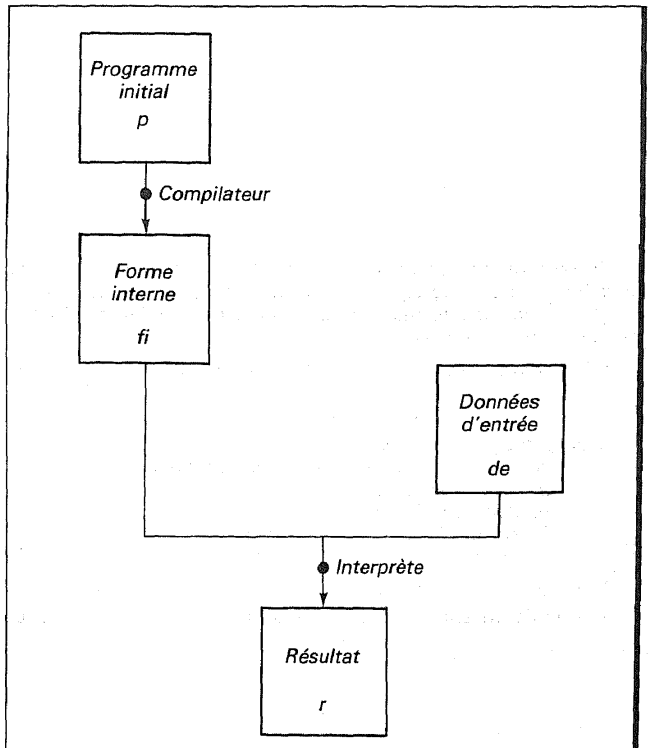

Fig. 1. – Compilation et interprétation.

– 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 [I.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.

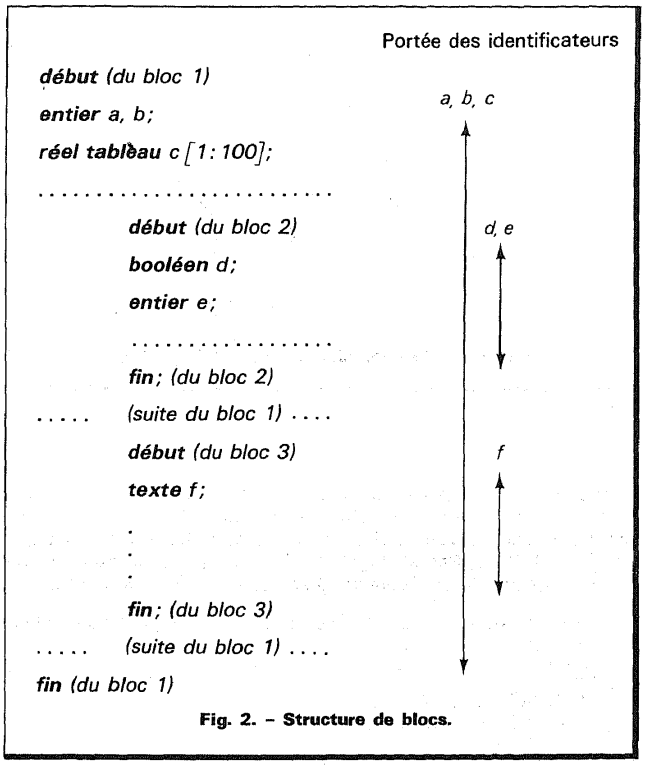

Fig. 2. – Structure de blocs.

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: *3,141592*) 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 [I.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 [I.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

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 [I.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 1000 pour la seconde:

[Fortran] *INTEGER T (10,1000)*

Ce tableau contient 10000 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 a 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*.

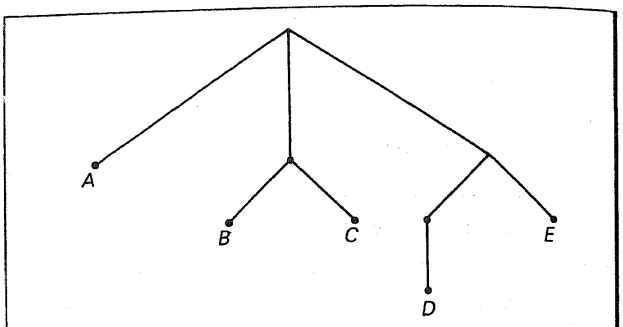

Fig. 3. – Arbre associé à la liste *(A(BC) ((D)E))*.

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**).

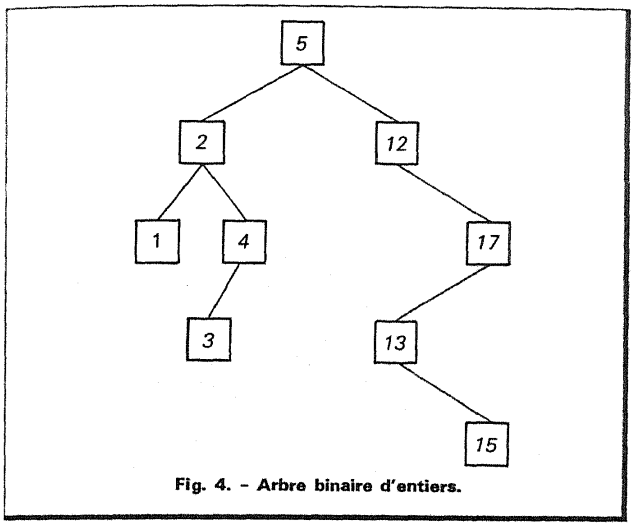

Fig. 4. – Arbre binaire d'entiers.

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] $X \leftarrow \lceil T$
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 stuctures 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-

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 :

$$C_n^m = \textbf{\textit{si}}\ m = 0\ \textbf{\textit{alors}}\ 1$$
$$\textbf{\textit{sinon si}}\ n = 0\ \textbf{\textit{alors}}\ 0$$
$$\textbf{\textit{sinon}}\ C_{n-1}^m + C_{n-1}^{m-1}$$

– 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 $A \star B$ dans l'instruction d'affectation suivante :

[Fortran] $C = A \star B$

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 (p 1, p 2)
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)
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

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.

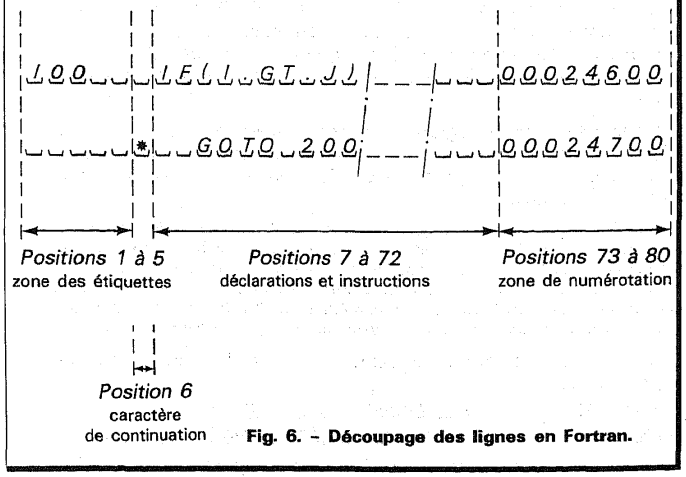

Fig. 6. – Découpage des lignes en Fortran.

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 K);
        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. Écrire 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 $b = (b_1, b_2, \ldots, b_n)$ 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 *8★★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.

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 [I.b. 28]; sur ses applications à la programmation, [I.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 [I.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

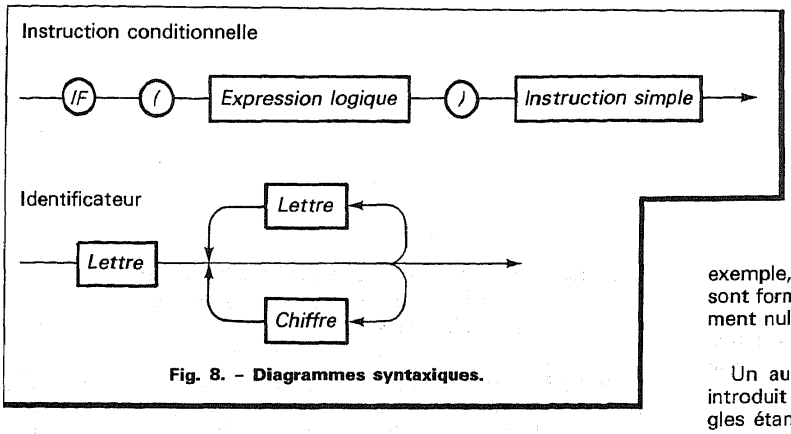

Fig. 8. – Diagrammes syntaxiques.

années cinquante (travaux qui avaient pour objet initial les langues humaines, non les langages artificiels ; cf par exemple [I.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.

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* [I.b. 28] et à une étude synthétique [I.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 $\uparrow v : : =$ *chiffre* $\uparrow c \downarrow c$ | *entier* $\uparrow ve$ *chiffre* $\uparrow vc \downarrow 10 \star ve + vc$
chiffre $\uparrow v : : =$ *'0'*$\downarrow 0$ | *'1'*$\downarrow 1$ | *'2'*$\downarrow 2$ | *'3'*$\downarrow 3$ | *'4'*$\downarrow 4$ | *'5'*$\downarrow 5$ | *'6'*$\downarrow 6$ *'7'*$\downarrow 7$ | *'8'*$\downarrow 8$ | *'9'*$\downarrow 9$

Dans ces équations, les noms précédés de $\uparrow$ 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 $\downarrow$ 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).

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 [l.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 [l.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 [l.b. 11 et 27] et Pascal [l.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.

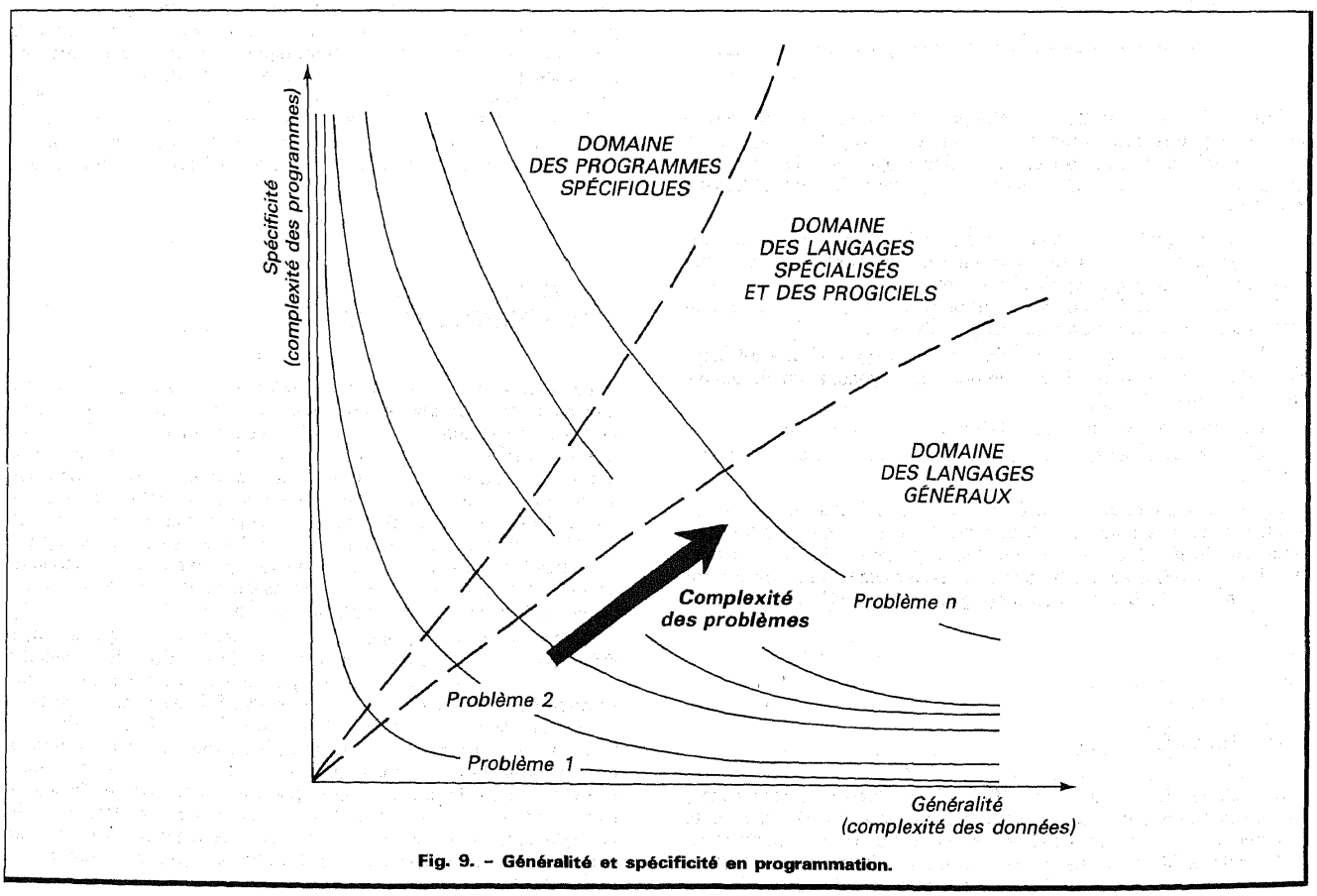

Fig. 9. – Généralité et spécificité en programmation.

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 [l.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 [l.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 [l.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 [l.b. 34] sont non formels et destinés plutôt à la communication avec les commanditaires du système; d'autres comme Special ou Z [l.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.

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 [l.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°s 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 Studentliteratur. 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 l.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 l.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 l.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. (+).