MATHÉMATIQUES ET SCIENCES HUMAINES

C. PAIR

La formalisation des langages de programmation

Mathématiques et sciences humaines, tome 34 (1971), p. 71-86

<http://www.numdam.org/item?id=MSH_1971__34__71_0>

Math. Sci. hum. (9e année, n° 34, 1971, p. 71-86)

LA FORMALISATION DES LANGAGES DE PROGRAMMATION

par

C. PAIR [1]

Un langage de programmation est une « langue artificielle », véhicule de la communication homme-machine. On aurait pu penser qu'il serait plus facile de formaliser ces langages que les langues naturelles. Pourtant, et c'est un exemple à méditer pour ceux qui veulent construire une théorie formalisée des langues naturelles, le problème — pour ces langages de programmation — reste très ardu et la formalisation en demeure inachevée. Les linguistes certes n'ont ni à copier, ni même à s'inspirer de la formalisation des langages de programmation, mais certaines techniques mises en œuvre par leurs « homologues » peuvent les intéresser. Et surtout, la prise de conscience de ce qui apparente ou oppose deux domaines contribue à enrichir les deux champs de recherche. Ainsi, les deux termes de « Syntaxe » et de « Sémantique » qu'emploie aussi la logique mathématique n'ont pas la même acception dans l'étude des langages de programmation et en linguistique ; cependant, la problématique de délimiter une frontière (si floue) entre Syntaxe et Sémantique paraît en partie commune aux deux domaines. On pourrait multiplier les exemples où l'on peut espérer éclairer les rapports ou différences et ainsi faire mieux affluer les difficultés profondes et complexes. Dans cet ordre d'idées, nous nous bornons à citer : rapports locuteur-énoncé-situation, problèmes des référents et des anaphores...

A. LENTIN

RÉSUMÉ

Depuis une dizaine d'années se développent des langues d'un nouveau genre, les langages de programmation. L'article indique les problèmes que pose leur formalisation, tant du point de vue syntaxique que du point de vue sémantique, et expose les solutions qui ont été apportées à ces problèmes.

1. INTRODUCTION

Puisque programmer c'est donner des ordres à un ordinateur, la programmation demande une langue qui soit support de ces ordres. La première idée est d'employer une langue directement « compréhensible » par l'ordinateur, c'est-à-dire dont les mots, ou instructions, provoquent directement les actions requises des circuits de l'ordinateur ; dans ce cas, le programmeur doit donc employer la langue de l'ordinateur, ce qu'on appelle le *langage-machine*. C'est ainsi que procédaient les premiers programmeurs. Mais on

1. Université de Nancy 2.

s'est vite rendu compte que le langage-machine, adapté à la technologie de construction des ordinateurs, ne l'est guère à la transmission des algorithmes de calcul. En particulier, il exige une décomposition très poussée des opérations à effectuer. D'autre part, puisqu'il dépend de la construction de la machine, il change avec l'ordinateur. Enfin, il est peu compréhensible à l'homme, même entraîné ; on pourrait penser que ce dernier aspect est secondaire puisque les programmes sont destinés à un ordinateur, mais il n'en est rien car, au cours de son existence, un programme doit être souvent relu, corrigé, modifié.

On a donc défini des langues plus aptes à servir de support à la communication des algorithmes, et qu'on appelle des *langages de programmation.* Ils ne sont pas directement compréhensibles par l'ordinateur et un programme écrit dans un tel langage (programme source) doit être traduit en un programme exprimé en langage-machine (programme objet) ; cette traduction est effectuée par l'ordinateur lui-même au moyen d'un programme écrit une fois pour toutes et qu'on appelle un *compilateur* (fig. 1).

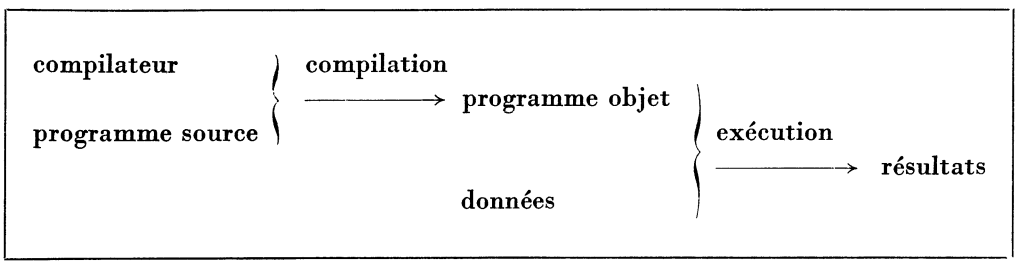

Fig. 1

Les plus évolués des langages de programmation sont indépendants de l'ordinateur. Ce sont des langages totalement artificiels, définis d'après les besoins de la programmation ou d'un certain domaine de la programmation (calcul scientifique, gestion, traitement de textes, simulation...).

Il est donc possible d'en donner une définition complètement précise de manière que la correction d'un programme et sa signification ne soient en rien affaire d'appréciation personnelle. Une telle définition est d'ailleurs nécessaire, car un programmeur doit savoir très exactement ce qu'il peut écrire, un compilateur doit accepter les programmes corrects et rejeter les autres, en indiquant si possible les fautes de syntaxe, et doit traduire un programme-source en un programme-objet qui provoque les actions demandées par le programmeur.

On aperçoit donc la nécessité de donner, pour un langage de programmation, une *syntaxe* et une *sémantique* totalement précises : la syntaxe définit l'ensemble des programmes corrects et la sémantique les actions qu'ils entrainent.

Il s'agit de savoir quels outils utiliser pour décrire syntaxe et sémantique. Pour Fortran [3] premier langage évolué apparu, tout a été décrit dans une langue naturelle, l'anglais. Mais une langue naturelle manque de la précision nécessaire et beaucoup de points restaient dans l'ombre, conduisant à des interprétations variées. D'où l'idée de donner des définitions formelles. Cette idée a été en partie exploitée pour définir la syntaxe d'Algol 60 [33] et de nombreux efforts ont été faits depuis en ce sens, pour formaliser soit la syntaxe plus complètement, soit la sémantique.

L'intérêt de la formalisation n'est pas seulement d'apporter la précision nécessaire au programmeur et surtout au constructeur de compilateurs, et aussi parfois la clarté. Une formalisation est également indispensable lorsqu'on veut effectuer des démonstrations concernant des programmes écrits dans le langage étudié, ou concernant un compilateur, notamment la démonstration de la correction

d'un compilateur. Elle est plus encore nécessaire pour automatiser l'écriture des compilateurs, et la formalisation de la syntaxe a été effectivement utilisée à cet effet [6], [22], [8]. Enfin, elle peut constituer un guide pour les concepteurs de langages.

Dans la suite de cet article, nous examinerons successivement la définition de la syntaxe et celle de la sémantique des langages de programmation. Pour fournir un support à cette étude, la figure 2 reproduit un programme simple écrit dans le langage Algol 60.

```
début entier tableau t [1 : 500] ; entier max, i ;
        lire (t) ;
        max : = t [1] ;
        pour i : = 2 pas 1 jusqua 500 faire
                    si t [i] > max alors max : = t [i] ;
        imprimer (max)
fin
```

Ce programme a pour donnée un tableau de 500 nombres entiers ; son résultat est le plus grand entier du tableau ; pour le déterminer, on parcourt le tableau et on note en max le plus grand entier déjà rencontré.

Fig. 2

L'article résume les principales approches déjà utilisées ; il propose aussi succinctement certaines idées de l'auteur sur les points encore mal élucidés.

Comme pour les langues naturelles, la frontière entre syntaxe et sémantique est d'ailleurs floue. En effet, on est en général amené à considérer que, parmi les programmes corrects, certains sont « sans signification », par exemple parce qu'ils ne se terminent pas ; il en est ainsi pour un programme Algol 60 où est exécutée l'instruction

```
e : allera e ;
```

un tel programme peut d'ailleurs posséder une signification pour certaines données, celles qui ne provoqueront pas l'exécution de cette instruction. On peut considérer comme syntaxiques les règles qui permettent de rejeter des programmes avant toute exécution. Mais ce critère, s'il fournit une idée intéressante, ne suffit pas à tracer une frontière acceptable, car par exemple, il conduit à rejeter le programme

```
début entier i ;
        i : = 1 ;
        e : si i > 0 alors allera e
fin
```

et à accepter

```
début entier i
        lire (i) ;
       e : si i > 0 alors allera e
fin.
```

Certains auteurs [33] nomment d'ailleurs « syntaxe» ce qu'ils formalisent et « sémantique» tout le reste... D'autres [13] ne tracent aucune séparation et formalisent simultanément syntaxe et sémantique.

2. FORMALISATION DE LA SYNTAXE

L'alphabet d'un langage de programmation est formé d'un certain nombre de signes tels que + (> ; , des lettres, des chiffres et éventuellement de mots privilégiés comme **début** ou **si**. De plus, selon les cas, l'espace peut avoir ou non une importance, autrement dit faire ou non partie de l'alphabet : en Algol 60, les espaces sont sans importance. A l'aide des lettres et des chiffres, on peut en général former des noms, tels que max dans l'exemple de la figure 2, mais ces noms sont de simples repères (on les appelle d'ailleurs *identificateurs*), de sorte que la manière dont ils sont formés n'importe pas dans la plupart des cas. La définition d'un langage de programmation ne possède donc aucune composante morphologique et on peut passer tout de suite à l'étude de la syntaxe.

2.1. Emploi de C-grammaires

Comme pour les langues naturelles, la première idée est d'employer une grammaire de constituants (ou C-grammaire [19]) pour décrire la syntaxe. Cette idée ne manque pas de justifications : les phrases d'un langage de programmation possèdent une structure fortement parenthétique, non seulement parce qu'elles peuvent en général contenir des expressions algébriques comme :

$$(a + b) \times (c - (a \times b - d))$$

mais aussi parce que les instructions peuvent venir s'emboîter l'une dans l'autre, comme :

max : = t [i]

dans :

si t [i] > max **alors** max : = t [i]

qui est elle-même partie de l'instruction plus complexe :

pour i : = 2 **pas** 1 **jusqua** 500 **faire si** t [i] > max **alors** max : = t [i].

Cette imbrication peut en général être récursive, ce qui permet de construire des instructions arbitrairement longues et compliquées : ceci est une caractéristique des langages de programmation évolués, qui les distingue des langages proches de la machine pour les rapprocher des langues naturelles.

D'autre part, les C-grammaires sont bien les plus intuitives pour spécifier aux programmeurs comment constituer un programme. On écrira par exemple :

< instruction **si** > → **si** < expression booléenne > **alors** < instruction inconditionnelle >
< instruction conditionnelle > → < instruction si >
< instruction conditionnelle > → < instruction si > **sinon** < instructions >
< instruction > → < instruction inconditionnelle >
< instruction > → < instruction conditionnelle >

Les C-grammaires sont même ici tellement naturelles qu'il semble bien que les auteurs d'Algol 60 en ont employé une sans se rendre compte qu'il s'agissait d'un modèle déjà bien connu.

Cependant, nous allons voir que les C-grammaires ne suffisent pas à décrire toutes les règles syntaxiques.

2.2. Le problème des accords

Si les langues naturelles possèdent des genres, les langages de programmation admettent des *types* d'information. Le programme de la figure 2 manipulait des entiers, ou plus exactement des variables entières, max et i, et aussi un tableau d'entiers, t. On peut aussi traiter des nombres réels, des valeurs logiques... Les opérations et instructions à effectuer portent sur des valeurs de certains types, et il doit exister certains accords de type. Par exemple, on ne peut écrire :

$$2 + \textbf{faux}$$

et dans une instruction d'affectation, telle que :

$$\text{max} : = \text{t [i]}$$

si le premier membre est une variable d'un certain type, le second membre doit être une expression de même type ; par exemple, dans le programme précédent, on ne doit pas écrire :

$$\text{max} : = \textbf{vrai}.$$

Tant qu'il n'existe qu'un nombre fini de types d'information, on peut décrire ces accords à l'aide de C-grammaires, bien que ce ne soit pas une solution économique car elle conduit à multiplier les règles pour y introduire les diverses combinaisons acceptables de types. Mais, en réalité, l'ensemble des types d'un langage de programmation n'est pas en général fini. Par exemple, on peut traiter non seulement des tableaux à une dimension comme dans le cas précédent, mais aussi des tableaux dont le nombre de dimensions est quelconque ; il est possible que dans la pratique on dépasse rarement trois dimensions, mais c'est là une question de performance et non de compétence. De même, on peut définir et utiliser des fonctions à un nombre quelconque d'arguments, qui peuvent eux-mêmes être de types quelconques. En résumé, on peut dire que le problème des accords est plus difficile pour les langages de programmation que pour les langues naturelles.

Pour décrire les accords possibles, on peut conserver une grammaire de constituants à condition d'admettre une infinité de catégories syntaxiques (on symboles non terminaux) et une infinité de règles qui seront obtenues par des schémas de règles. Par exemple, on écrira pour l'instruction d'affectation le schéma :

$$< \text{affectation de type } \tau > \rightarrow < \text{variable de type } \tau > : = < \text{expression de type } \tau > .$$

Il restera à décrire d'autre part l'ensemble infini des types, dont un élément peut remplacer les trois occurrences de τ dans le schéma précédent et les autres schémas de la grammaire. Le plus simple est de définir les types par des chaînes de caractères et d'engendrer leur ensemble par une grammaire, par exemple par une C-grammaire. Le langage est ainsi défini par deux *grammaires superposées*. C'est en gros de cette manière que procède [49] pour décrire le langage Algol 68, bien que dans ce cas les types ne soient pas exactement définis par des chaînes de caractères [39], [36].

Il est intéressant de se demander quelle classe de langages peut être décrite par deux grammaires superposées du genre précédent : une C-grammaire G_0 et une C-grammaire généralisée G_1 admettant une infinité de règles données par un nombre fini de schémas où entrent des symboles à remplacer par un élément d'une catégorie syntaxique engendrée par G_0. La réponse [38] est que cette classe est aussi

vaste que possible ; on peut décrire ainsi tout langage récursivement énumérable, donc en particulier des langages indécidables. On se trouve dans une situation hélas fréquente en théorie des langages : en généralisant une classe de grammaires insuffisante pour décrire une classe de langages donnée, on obtient une classe beaucoup trop puissante. L'inconvénient n'est pas seulement théorique. Rappelons en effet que l'un des buts de la formalisation est de conduire à automatiser l'écriture de compilateurs. Une condition nécessaire pour cela est de pouvoir automatiser l'écriture de programmes de *reconnaissance*, car tout compilateur contient nécessairement un tel programme ; cette automatisation, facile pour les C-grammaires, est évidemment impossible lorsqu'il n'existe pas d'algorithme de reconnaissance pour toutes les grammaires de la classe considérée.

Parmi les autres types de grammaires qui ont été envisagés et permettent de résoudre le problème des accords, citons les grammaires indexées [1] : elles engendrent des langages plus généraux que les C-langages, mais qui ne sortent pas de la classe des langages contextuels. Cependant, elles sont beaucoup moins intuitives à utiliser que les grammaires superposées.

2.3. Le problème des références

Nous avons dit plus haut que dans la définition d'un langage de programmation ne se trouve aucune composante morphologique, car la forme d'un identificateur n'a aucune conséquence. Cependant, chaque identificateur désigne un objet d'un certain type et ce type doit être connu pour appliquer les règles d'accord : par exemple, pour écrire

max : = t [1]

je dois savoir que t est un tableau à une dimension, et qu'il est formé d'éléments dont le type (ici entier) est compatible avec celui de la variable max. Le type des divers identificateurs est en général précisé par une *déclaration*, telle que

entier max

ou

entier tableau t [1 : 500].

Il faut donc écrire des règles qui précisent que, pour toute occurrence d'identificateur, doit exister une déclaration précisant son type et qui indiquent comment retrouver cette déclaration. Il s'agit là d'un problème fort analogue à celui de la correspondance entre pronom et groupe nominal dans le cas des langues naturelles. Il est cependant clair que, pour éviter les ambiguïtés, on devra donner pour les langages de programmation des règles beaucoup plus strictes que pour les langues naturelles. Dans le cas le plus simple, on impose qu'un identificateur utilisé dans un programme doive y faire l'objet d'une déclaration et d'une seule. Plus généralement, on peut partager le programme en *blocs* disjoints ou emboîtés (fig. 3) ; toute occurrence d'identificateur doit alors pouvoir être associée à une déclaration de cet identificateur, qui se trouve dans le plus petit bloc contenant à la fois l'occurrence considérée et une déclaration de l'identificateur ; pour assurer l'unicité de cette déclaration, le plus petit bloc contenant une déclaration d'un identificateur et le plus petit bloc contenant une autre déclaration du même identificateur ne peuvent être identiques.

Brièvement, une déclaration est valide dans le plus petit bloc qui la contient, à l'exclusion des blocs intérieurs où le même identificateur serait redéclaré. Dans certains langages existent des règles plus complexes, comme celles qui gouvernent l'emploi des identificateurs des champs des structures en PL 1, ou les déclarations d'opérateurs en Algol 68 (où, en plus des identificateurs, les opérateurs peuvent être déclarés).

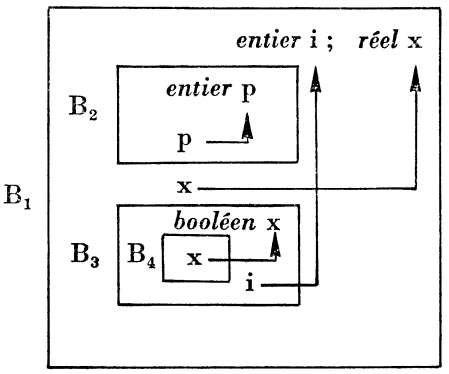

Fig. 3

Les règles concernant les références sont des règles de syntaxe *globale*, par opposition aux règles de constituants qui sont locales. On voit par ce qui précède qu'il est facile d'écrire en langue naturelle ces règles de syntaxe globale, ce qui suffit d'ailleurs pour apprendre au programmeur à utiliser le langage, mais qu'il est beaucoup moins facile de les formaliser. Les tentatives faites jusqu'alors conduisent à des ormalisations fort complexes et non utilisées dans la pratique. En particulier, il est théoriquement possible d'utiliser des grammaires superposées, puisqu'on peut ainsi engendrer tout langage récursivement énumérable ; mais l'expérience montre qu'on parvient alors à une formalisation peu naturelle et d'une extrême complication.

2.4. Conclusion

La solution qui consiste à ne formaliser que la partie de syntaxe pouvant l'être à l'aide de C-grammaires a été souvent adoptée, avec éventuellement des variantes purement notationnelles pour indiquer dans un second membre de règle qu'une composante est facultative, que l'ordre des composantes est arbitraire, etc., [11], [20]. Cette solution est utile car elle est simple et permet d'automatiser l'analyse syntaxique.

Deux difficultés l'empêchent cependant d'être pleinement satisfaisante. Le problème des accords peut être traité fort naturellement en employant des grammaires superposées. Pour éviter d'engendrer des langages indécidables comme nous l'avons signalé plus haut, on peut suggérer de restreindre ces grammaires en exigeant la condition suivante, qui assure la décidabilité : *l'ensemble des règles obtenues à partir d'un schéma est tel qu'il n'existe qu'un nombre fini de règles de second membre donné.*

Le problème des références ne se poserait pas si la morphologie des identificateurs déterminait leur type ; or les déclarations ont pour but de permettre d'associer un type à chaque identificateur. La solution à ce problème apparaît donc comme étant de nature transformationnelle. Et on peut finalement suggérer la solution suivante qui conduirait à une formalisation complète de la syntaxe des langages de programmation. La syntaxe serait définie en deux parties :

a) Deux grammaires superposées, au sens du paragraphe 2.2., vérifiant la condition de décidabilité indiquée à l'alinéa précédent, pour engendrer un langage où tout identificateur est accompagné de son type (et plus généralement toute indication est accompagnée des renseignements nécessaires à la vérification des règles d'accord) ;

b) Une transformation, fonction récursive de ramification [37] portant sur les marqueurs de phrases engendrés par les grammaires superposées, qui supprime les renseignements superflus en vérifiant qu'il existe des déclarations adéquates.

La composante transformationnelle pourrait d'ailleurs être utilisée aussi à d'autres fins. Comme dans le cas des langues naturelles, elle peut permettre à la grammaire générative *a*) de n'engendrer qu'un langage-noyau restreint, étendu ensuite par transformations. Cette approche, timidement employée par [42], a pour principal mérite de simplifier la définition de la sémantique, si on demande aux transformations de conserver la signification d'un programme. On peut d'autre part songer aussi à vérifier les règles d'accord transformationnellement, ce qui permettrait d'utiliser en *a*) une unique C-grammaire comme grammaire générative. Cette idée [27], intéressante car elle formalise les méthodes de reconnaissance utilisées dans la pratique, a l'inconvénient de conduire à une formalisation peu naturelle et assez complexe.

3. FORMALISATION DE LA SÉMANTIQUE

Les travaux sont ici moins avancés qu'en syntaxe, de sorte qu'on devra se borner à un panorama superficiel. Dans la pratique, la sémantique est décrite en langue naturelle, avec un effort plus ou moins grand pour employer la langue naturelle de manière non-ambiguë, en définissant soigneusement les termes employés, jusqu'à parvenir à un langage presque formel : [42] offre un remarquable exemple d'un tel effort (*cf.* [39]). Les études faites sur la formalisation de la sémantique n'ont été qu'exceptionnellement appliquées à de véritables langages de programmation. Il faut cependant citer ici les travaux du laboratoire IBM de Vienne, portant notamment sur PL 1 [28].

Définir la sémantique d'un langage de programmation, c'est indiquer comment associer à tout programme (syntaxiquement correct) une « valeur sémantique ». Le problème est donc double : définir un ensemble de valeurs sémantiques, définir une application de l'ensemble des programmes dans celui des valeurs sémantiques.

Plutôt que de partir des programmes eux-mêmes, il est plus pratique et plus normal de partir des marqueurs de phrase déterminés par la syntaxe ; on peut même transformer ces ramifications de manière à ne conserver que leurs parties signifiantes. Le passage du texte à des ramifications est parfois appelé passage de la syntaxe concrète à la syntaxe abstraite [29], [28]. Il n'est cependant pas utilisé par tous les auteurs, notamment par ceux qui se refusent à étudier séparément syntaxe et sémantique.

3.1. Équivalence de programmes

Une première idée consiste à refuser de s'appesantir sur la nature des valeurs sémantiques et à considérer que la seule chose importante est de savoir dans quel cas deux programmes sont équivalents. Formellement, on définit une relation d'équivalence dans le langage et l'ensemble des valeurs sémantiques est l'ensemble-quotient.

Pratiquement, on imposera certaines équivalences : par exemple, en Algol 60, si V désigne une variable réelle, E une expression arithmétique simple, T un terme, on peut remplacer l'instruction d'affectation :

$$V := E + T$$

par le bloc :

début réel x, y ; x : = E ; y : = T ; V : = x + y **fin**

où x, y sont les identificateurs n'ayant pas d'occurrence dans V, E, T ; ou encore, on peut remplacer la suite des deux instructions d'affectation :

$$V := V_1 ; V := V_2$$

par :

$$V := V_2$$

si V, V_1, V_2 désignent des variables, V et V_2 désignant des variables différentes.

L'équivalence sémantique sera alors la plus petite relation d'équivalence satisfaisant aux conditions imposées.

Il est immédiat d'associer à cette définition un système formel dont les théorèmes sont les $p \sim q$, où p et q sont des programmes équivalents : les axiomes sont les équivalences imposées ; les règles d'inférence traduisent la symétrie et la transitivité de la relation d'équivalence. Par conséquent, on peut espérer, à partir de cette approche, parvenir à *démontrer* des équivalences de programmes. En particulier, on peut espérer justifier la correction d'un compilateur traduisant un langage L dans un langage L_1, en définissant ainsi la sémantique d'un langage qui contient à la fois L et L_1 ; L_1 peut être considéré comme un *langage-noyau* pour L.

Il ne s'agit pas d'une définition explicite de la sémantique, mais d'une définition axiomatique, où on impose certaines propriétés en espérant pouvoir en déduire toutes les propriétés intuitivement souhaitées. Là est la difficulté, car pour que le système formel soit complet, c'est-à-dire traduise à lui seul toute la sémantique du langage, on doit imposer un très grand nombre de conditions, rendant ainsi la définition peu maniable. Les exemples donnés sur des cas très simples [21] sont instructifs à cet égard.

Heureusement, pour les applications, on peut se borner à définir une équivalence plus faible, qui permet de démontrer l'équivalence de certains programmes, mais ne prétend pas définir complètement la sémantique. Une idée proche a été suivie pour Algol 60 par [34] qui associe à chaque programme un représentant canonique dans un langage-noyau, définissant par là une équivalence. C'est également une équivalence plus faible qui est utile au programmeur ou au constructeur de compilateurs, pour lui indiquer par exemple la signification d'une instruction complexe, comme une instruction d'itération, en termes d'instructions plus simples [33], [42] :

pour i : = a **tantque** b **faire** S

est équivalent à :

```
i : = a ;
e : si ¬ b alors allera termine ;
    S ;
    allera e ;
termine :
```

La définition de la sémantique par équivalence, malgré l'intérêt théorique qu'elle présente, ne semble donc pas suffisante. On peut d'ailleurs aussi lui reprocher, puisqu'elle refuse de considérer la nature des valeurs sémantiques, de ne pas tenir compte du fait qu'on traite un langage de programmation et non pas n'importe quelle langue.

3.2. Structures d'information

Un programme fait passer de données à des résultats. La valeur sémantique d'un programme est donc une fonction (calculable) transformant les données en des résultats.

Le premier problème est de préciser la nature des données et des résultats. Plusieurs auteurs, après [30] et [41], se contentent de considérer des « vecteurs » ayant un certain nombre de composantes.

Cette vue idéalise assez bien l'état de la mémoire d'un ordinateur, qui est formée d'un certain nombre de mots admettant chacun un contenu. Mais en réalité, les langages de programmation traitent des informations abstraites, de structure pouvant être assez complexes (listes, arborescences, tableaux...), dépendant du domaine auquel le langage s'applique, qui doivent bien sûr être représentées dans la mémoire d'un ordinateur, mais qu'il est dommage de confondre avec leur représentation. L'étude de la sémantique des langages de programmation demande une théorie des structures d'information [14], [32], recouvrant aussi bien les structures logiques ou abstraites que les structures physiques les représentant dans une mémoire d'ordinateurs.

On peut proposer de définir une information comme un couple (E, A) formé d'un ensemble E (pouvant contenir des nombres, des caractères..., mais aussi des «repères» permettant l'accès à d'autres éléments) et d'un ensemble A de fonctions [1] à zéro, une ou plusieurs variables de E, à valeurs dans E (fonctions d'accès dans l'information). La figure 4 schématise une information où les flèches représentent

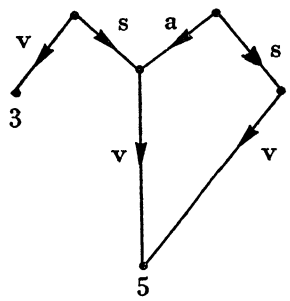

Fig. 4

les fonctions de A, qui sont ici des fonctions à une variable : A = { a, s, v }. Les informations sont sujettes à variation et on peut définir une structure d'information comme un couple formé d'un ensemble d'informations et d'un ensemble de transformations élémentaires sur elles ; en composant ces transformations élémentaires, on obtient les transformations permises dans la structure considérée.

Le cas où l'on se limite à des informations arborescentes [28] permet d'unifier la représentation des données et des résultats avec celle du programme.

3.3. Valeur sémantique d'un programme

Après avoir précisé la nature des données et des résultats, la définition de la sémantique d'un langage de programmation peut donc consister à indiquer une fonction « sémantique » qui associe à tout programme une fonction faisant passer des données aux résultats :

$$\text{programme} \xrightarrow[\text{sémantique}]{} (\text{données} \longrightarrow \text{résultats}).$$

La fonction sémantique doit être effectivement calculable si l'on veut qu'elle soit utilisable. Une variante consiste à indiquer une fonction calculable à deux variables :

$$\text{programme, données} \xrightarrow[\text{sémantique}]{} \text{résultats}.$$

1. Dans tout l'article, « fonction » veut dire « fonction partielle », c'est-à-dire non nécessairement définie dans tout son ensemble de départ.

Pour définir des fonctions calculables, on peut se contenter de définitions directes, utilisant notamment la récurrence, de manière à définir par exemple des fonctions récursives sur les marqueurs de phrase, [23] donne une méthode qui revient à définir une fonction implicitement par un système d'équations dont les inconnues sont liées aux nœuds du marqueur de phrase. Mais ces définitions directes sont souvent considérées comme insuffisamment formelles et on est amené à les préciser en utilisant l'un ou l'autre des outils suivants :

a) Les machines abstraites, du genre de la machine de Turing, mais en réalité on emploie des machines plus élaborées que les machines de Turing [15], [28] ; les descriptions d'algorithmes par une suite de règles comme les algorithmes de Markov [31], ou par un organigramme [17], [35], [4] peuvent être considérées comme des variantes de ce type.

b) Le λ-calcul de Church [10] (et plus généralement la logique combinatoire [12], [40]).

c) Des systèmes formels, qui, contrairement aux outils précédents, conduisent à des définitions non explicites ; ils donnent deux possibilités essentielles pour définir une fonction f à k variables :
— définir f axiomatiquement en lui associant un système formel dont les théorèmes sont les

$$a_1, \ldots, a_k \rightarrow f(a_1, \ldots, a_k) ;$$

— utiliser un système formel connu et associer à f une forme $F(x_1, \ldots, x_k, y)$ telle que $F(a_1, \ldots, a_k, b)$ soit un théorème si, et seulement si, $b = f(a_1, \ldots, a_k)$ [16] ; par exemple, la fonction valeur absolue peut être définie par l'expression booléenne :

$$(x \geqslant 0 \wedge y = x) \vee (x < 0 \wedge y = -x).$$

La représentation de la valeur sémantique d'un programme par une λ-expression ou grâce à un système formel est moins riche que sa représentation par une machine abstraite, car elle ne donne qu'une vue globale de la transformation des données en les résultats, alors qu'une machine abstraite marque les étapes de la transformation. En particulier, une machine abstraite peut fort bien représenter un programme ne se terminant pas, ou ne se terminant que pour certaines données, deux tels programmes ne suivant pas les mêmes étapes étant représentés par des machines différentes. Or, il peut être intéressant d'étudier des programmes ne se terminant pas, par exemple dans le cas des langages de simulation.

En revanche, la représentation à l'aide d'un système formel, si elle présente l'inconvénient de n'être pas explicite, semble celle qui permet le plus facilement de faire des démonstrations concernant le langage, bien qu'on puisse espérer, en construisant une théorie de certains types de machines abstraites, obtenir des résultats assez puissants pour effectuer des démonstrations commodes [15].

En combinant les deux sortes de fonctions sémantiques et les divers outils de définition, on obtient un grand nombre de manières de décrire la sémantique, dont beaucoup ont été proposées ou utilisées.

[25] fait usage du λ-calcul et emploie une fonction du premier type transformant un programme en une λ-expression (voir aussi [5]).

Représenter une fonction du second type par une machine abstraite, c'est définir un *interpréteur ;* [28] procède de cette manière, ainsi que [7] et [13] qui emploient des outils proches des algorithmes de Markov.

Représenter une fonction du premier type par une machine abstraite qui transforme un programme en une autre machine abstraite transformant les données en les résultats, c'est employer une approche analogue à la description d'un compilateur. On a parfois proposé de définir la sémantique d'un langage en en donnant effectivement un compilateur [10] ; mais il ne saurait être question de traduire le langage donné dans le langage d'une machine particulière, faute de manquer d'universalité ; des tentatives sans lendemain ont été faites pour définir un langage machine universel, UNCOL ; plus récemment, elles ont été reprises et ont donné naissance à L M U [35]. Ce genre de définition de la sémantique, peut-être utile pour le constructeur de compilateurs, risque d'être rebutant, car très orienté vers la machine. Ici, comme au paragraphe 3.2. concernant la représentation des données et des résultats, on touche d'ailleurs un point important qui divise les informaticiens (voir [2]) : l'informatique est-elle d'abord la science des ordinateurs, auquel cas il faut tout ramener à la machine, ou est-elle celle des algorithmes de traitement de l'information, qui existent indépendamment des ordinateurs ?

3.4. Tentative de synthèse

Nous avons vu que la définition de la sémantique par équivalence présente des aspects intéressants, mais devient fort compliquée lorsqu'elle prétend tout traiter à elle seule. Plus précisément, elle se prête bien à ramener un langage à un langage noyau en brisant les constructions complexes, mais beaucoup moins à étudier les constructions élémentaires. Les définitions utilisant le λ-calcul et surtout une machine abstraite se prêtent mal aux démonstrations et sont aussi assez compliquées, mais pour une raison opposée, car elles peinent à traiter les constructions complexes. Le traitement des sauts (**aller à**) est également source de complication, notamment avec le λ-calcul, car il s'agit d'une notion dynamique, essentielle en programmation, mais qui se laisse mal approcher par les outils mathématiques, plus propres à décrire des situations statiques.

On peut donc songer à combiner les deux approches. Il est un peu ennuyeux d'employer simultanément des outils différents ; mais l'emploi d'un système formel, déjà rencontré avec la définition par équivalence, semble permettre de garder un unique outil, en traitant par des axiomes et règles d'inférence de types divers aussi bien la transformation des constructions complexes en constructions élémentaires que l'étude des constructions élémentaires elles-mêmes. Elle permet aussi d'éviter une définition explicite qui est difficile à obtenir et à maîtriser dans le cas des langages complexes, comme le sont les véritables langages de programmation.

Nous proposons donc un système formel, dont les formules sont des triplets : (pg, do, res) où :

— pg représente une ramification appartenant à l'ensemble des marqueurs de phrase du langage étudié (ou d'un langage plus vaste, ce qui est souvent très commode),

— do et res représentent des informations, au sens défini précédemment, d'une certaine structure qui dépend essentiellement du domaine d'application du langage.

Les théorèmes (pg, do, res) indiquent que le programme de marqueur de phrase pg, appliqué à la donnée do, fournit (ou peut fournir) le résultat res.

Les axiomes (pg, do, t (do)), où t est une transformation permise par la structure considérée, ne portent que sur des pg très simples : ils formalisent donc la sémantique des constructions élémentaires.

Les règles d'inférence seront de plusieurs types, notamment :

(1) $$(\Phi\ (pg),\ do,\ res) \vdash (pg,\ do,\ res)$$

où Φ est une fonction récursive de ramification : ce sont des règles d'équivalence, qui formalisent la sémantique des constructions complexes et permettent de justifier leur suppression ; on peut, grâce à elles, se ramener à un langage-noyau, ne contenant par exemple aucun saut, et aussi interpréter les appels de procédure.

(2) $$(\Phi\ (pg, do), do, res) \vdash (pg, do, res) :$$

ce sont des règles d'équivalence conditionnelle, puisque le programme transformé Φ (pg, do) dépend des données ; par exemple, avec :

$$pg = \text{si B alors I},$$
$$\Phi\ (pg, do) = \text{si } < B, do > \text{ alors I},$$

où < B, do > désigne la « valeur» de la variable booléenne B dans la donnée de do ; ces règles portent sur des pg simples ; l'emploi combiné des règles des types (1) et (2) permet de se ramener à des programmes sans aucun saut ni appel de procédure, où toutes les constructions sont élémentaires.

(3) des règles de composition

$$\left.\begin{array}{l}(pg, do, res)\\(pg\ 1, res, res\ 1)\end{array}\right\} \vdash (\Phi\ (pg, pg\ 1), do, res\ 1)$$

permettent de formaliser l'exécution séquentielle d'un programme ; l'exécution collatérale ou parallèle [42] demanderait des règles plus complexes.

4. CONCLUSION

Les difficultés rencontrées et la diversité des points de vue viennent en grande partie du fait que les buts poursuivis en définissant les langages de programmation avec précision présentent une grande diversité. Certains se rapportent à l'utilisateur, et encore peut-on distinguer entre l'utilisateur qui se contente d'une vue globale de l'action obtenue et le programmeur qui entre dans le détail des étapes du calcul ; d'autres touchent ceux qui écriront un compilateur pour le langage, ou encore ceux qui seront chargés de modifier le langage, de le comparer à d'autres langages, de le standardiser. A ces buts purement orientés vers l'homme, qui nécessitent plus une définition claire et brève qu'une définition formelle, s'opposent des buts visant à obtenir des démonstrations, voire des démonstrations automatiques, sur un programme, sur le langage, sur ses compilateurs, et même à produire automatiquement des compilateurs. L'intérêt de ces derniers buts n'est pas infirmé par le fait que de nombreuses questions sont indécidables, même parmi les plus simples, par exemple [19] savoir si une C-grammaire est non ambiguë (ce qui est essentiel pour la grammaire d'un langage de programmation) ou si un programme se terminera pour certaines données, ou, à plus forte raison, si deux programmes sont sémantiquement équivalents.

Les divers buts ne sont pas faciles à satisfaire simultanément, bien qu'on y soit parvenu pour la partie de la syntaxe qui s'exprime par une C-grammaire (aussi appelée notation de Backus par certains spécialistes des langages de programmation). Ce métalangage donne en effet une définition totalement formelle, permettant donc démonstrations et automatisations, mais en même temps il possède une « sémantique» immédiatement accessible à l'homme qui n'a pas toujours à avoir présente à l'esprit sa valeur formelle pour l'utiliser ; ceci est le signe d'une bonne théorie, de même que, par exemple, la théorie mathématique des nombres rationnels est utile car on peut utiliser ces nombres sans songer à la complexité de leur définition ; de plus, ce métalangage permet de fragmenter la définition et n'oblige pas à songer à tout à la fois. Les diverses formalisations proposées pour la sémantique ne sont pas

parvenues à ce stade, mais le but à atteindre est clair. Il est d'autant moins désespéré qu'il faut se souvenir du précédent de la « notation de Backus» qui, à son apparition, a été considérée par certains comme difficile et abstraite.

Il est cependant vraisemblable que la diversité des buts demande la coexistence de plusieurs modes de définition, pour la sémantique surtout, mais il faudrait alors diposer d'une théorie unique permettant de démontrer leur équivalence. On voit apparaître ici une fois de plus l'importance des démonstrations, qui conduit à privilégier les définitions faisant usage d'un système formel.

Une définition formelle d'un langage peut avoir pour pivot une syntaxe abstraite, définissant un ensemble de ramifications, par exemple par une C-grammaire généralisée (grammaires superposées) ; de cet ensemble, on passe au langage par des transformations, et c'est aussi sur cet ensemble qu'est définie la sémantique. On peut d'ailleurs songer à avoir la même syntaxe abstraite pour les langages de toute une famille [26].

Ce schéma général diffère peu de ceux qui peuvent être utilisés pour décrire les langues naturelles. Bien sûr, il est plus facile de donner une définition pour les langages artificiels que sont les langages de programmation, que pour les langues naturelles. Mais nous avons vu que ce n'est pas immédiat pour autant. Du côté syntaxique, les difficultés sont moins nombreuses et mieux délimitées, mais elles sont de même nature que certaines de celles qu'on rencontre pour les langues naturelles, et sont même parfois plus sérieuses. Une différence importante, mais qui est de peu de conséquence sur la difficulté de la définition, est que les grammaires définissant les langages de programmation ne doivent pas être ambiguës. Du côté sémantique, les différences sont plus importantes et les langages de programmation sont dans une situation bien meilleure que les langues naturelles, car la nature des valeurs sémantiques est plus simple à cerner. Il s'agit de langages impératifs et non (ou peu) déclaratifs. Une difficulté est cependant créée par le fait qu'ils décrivent une évolution dans le temps. Par exemple, le calcul d'une fonction peut avoir des effets annexes, faisant évoluer les valeurs de certaines variables.

Cette difficulté conduit certains chercheurs à considérer que les mathématiques (en y incluant la logique mathématique) sont insuffisantes pour représenter les phénomènes informatiques et à partir en quête de nouvelles logiques incluant l'évolution temporelle. Pour ma part, cette assertion ne me semble pas fondée, ne serait-ce que parce que ce n'est tout de même pas la première fois que les mathématiques traitent un phénomène faisant intervenir le temps. Plutôt que de prendre comme notions premières des notions peu claires et mal élucidées, il me paraît plus sage d'essayer d'abord de ne pas négliger et de bien utiliser les outils fournis par les mathématiques.

BIBLIOGRAPHIE

[1] Aho, A. V., *Indexed grammars : An extension of context-free grammars*, Doct. thesis, Princeton University, 1967.

[2] Arsac, J. *La science informatique*, Paris, Dunod, 1970.

[3] Backus, J. W., *et al.*, "The Fortran automatic coding system", *Proc. Western Joint Comp. Conf. 11*, 1957, pp. 188-198.

[4] Blickle, A., *Algorithmically definable functions*, Polish Academy of Sciences, Inst. of Math., Varsovie, 1970.

[5] Böhm, C. et Gross, W., "Introduction to the CUCH", *Automata theory*, Caianiello (ed.), New York, Academic Press, 1966, pp. 35-65.

[6] BROOKER, R. A. et MORRIS, D., "A general translation program for phrase structure languages", *J. ACM* (9), 1962, pp. 1-10.

[7] CARACCIOLO DI FORINO, A., "Generalized Markov algorithms and automata", *Automata theory*, Caianiello (ed.), New York, Academic Press, 1966, pp. 107-114.

[8] CHEATHAM, T. E. et SATTLEY, K., "Syntax directed compiling", *Proc. Eastern Joint Comp. Conf.* (25), 1964, pp. 31-57.

[9] CHOMSKY, N., *Aspects of the theory of syntax*, Cambridge, Mass., MIT Press, 1965.

[10] CHURCH, A., "The calculi of lambda conversion", *Ann. of Math. Studies* (6), Princeton, NJ, 1941.

[11] *Cobol-1961 extended : External specifications for a common business oriented language*, Dept of Defense, Washington, DC, 1962.

[12] CURRY, H. B. et FEYS, R., *Combinatory logic*, Amsterdam, North-Holland, 1968.

[13] DE BAKKER, J.W., *Formal definition of programming languages*, Mathematisch Centrum, Amsterdam, 1967.

[14] D'IMPERIO, M., "Data structures and their representation in storage", *Ann. Rev. in Autom. Progr.* (5), part 1, New York, Pergamon Press, 1968.

[15] ELGOT, C. C., "Machine species and their computation languages", *Formal language description languages*, Steel (ed.), Amsterdam, North-Holland, 1966, pp. 160-178.

[16] FLORENTIN, J. J., "Proving theorems about programs", *Conference faite à l'IRIA*, 1970.

[17] FLOYD, R. W., "Assigning meanings to programs", *Proc. of symp. in appl. math.*, 19, 1967, pp. 19-32.

[18] GARWICK, J. K., "The definition of programming languages by their compilers", *Formal language description languages*, Steel (ed.), Amsterdam, North-Holland, 1966.

[19] GROSS, M. et LENTIN, A., *Notions sur les grammaires formelles*, Paris, Gauthier-Villars, 1967.

[20] IBM Systems Reference Library, File nº 5 366-29, Form C 28-6571. PL/1 Language Specifications.

[21] IGARASHI, S., "An axiomatic approach to equivalence problems of algorithms with applications", Thèse, Report of the Computer Center University of Tokyo, 1, 1968, pp. 1-101.

[22] IRONS, E. T., "The structure and use of the syntax directed compiler", *Ann. Rev. in Autom. Progr.* (3), 1963.

[23] KNUTH, D. "Semantics of context-free languages", *Math. Syst. Theory* (2), 1968, pp. 127-145.

[24] LANDIN, P. J., "A correspondence between Algol 60 and Church's lambda-notation", *Comm. ACM* (8), 1965, pp. 89-101 et 158-165.

[25] — "A formal description of Algol 60", *Formal language description languages*, Steel (ed.), Amsterdam, North-Holland, 1966, pp. 266-294.

[26] — "The next 700 programming languages", *Comm. ACM* (9), 1966, pp. 157-164.

[27] LECLAIRE, J. M., *Définition de la syntaxe des langages de programmation*, Thèse de troisième cycle, Faculté des Sciences de Nancy, 1970.

[28] LUCAS, R. et WALK, K., "On the formal description of PL/1", *Ann. Rev. in Autom. Progr.* 6, part 3, New York, Pergamon Press, 1968.

[29] Mc CARTHY, J., Towards a mathematical science of computation", *Information Processing* (62), Popplewell (ed.), Amsterdam, North-Holland, 1963, pp. 21-28.

[30] — "A formal description of a subset of Algol", *Formal languages description languages*, Steel (ed.), Amsterdam, North-Holland, 1966, pp. 1-12.

[31] MARKOV, A. A., *Teoriya algorifmov*, Trudy Mat. Inst. Steklov 42, Moscou, 1954.

[32] MEALY, G. H., "Another look at data", *Proc. Fall Joint Comp. Conf.* (31), 1967, pp. 525-534.

[33] NAUR, P. (ed.), "Revised report on the algorithmic language Algol 60", *Comm. ACM.* (6), 1963, pp. 1-17.

[34] NIVAT, M. et NOLIN, L., *Sur un procédé de définition de la syntaxe d'Algol*, Institut Blaise-Pascal, Paris, 1963.

[35] NOLIN, L., *Formalisation des notions de machine et de programme*, Paris, Gauthier-Villars, 1969.

[36] PAIR, C., "Concerning the syntax of Algol 68", *Algol Bulletin*, 31, 1970.

[37] QUERE, A., *Étude des ramifications et des bilangages*, Thèse de troisième cycle, Faculté des Sciences de Nancy, 1969.

[38] SINTZOFF, M., "Existence of a van Wijngaarden syntax for every recursively enumerable set", *Ann. Soc. Scientif.*, Bruxelles, 81.2, 1967, pp. 115-118.

[39] — "Introduction à la description d'Algol 68", *Rev. franç. d'informatique et de rech. op.*, série bleue (3), 1969, p. 3-16.

[40] Steel, T. B., "A formalization of semantics for programming languages description", *Formal language description languages*, Steel (ed.), Amsterdam, North-Holland, 1966, pp. 25-36.

[41] STRATCHEY, C., "Towards a formal semantics", *Formal languages description languages*, Steel (ed.), Amsterdam, North-Holland, 1966, pp. 198-220.

[42] VAN WIJNGAARDEN, A.; MAILLOUX, B. J. ; PECK, J. E. L., et KOSTER, C. H. A., *Algol 68*, MR 101, Mathematisch Centrum, Amsterdam.