Programmation statique, Le Langage Z

Extrait du livre Méthodes de programmation, Bertrand Meyer, Claude Baudoin, Collection de la Direction des Etudes et Recherches d'Electricité de France, Eyrolles, 1978, ISBN-10 : 221201581X

"Méthodes de Programmation (Programming Methodology), Eyrolles, Paris, 1978; third revised edition, 661 pages, 1984. Translation: Russian (Mir Publishing).

Histoire de ce livre par B. Meyer (https://se.inf.ethz.ch/old/people/meyer/publications/)
" We wrote this book fresh out of school and managed to convince the publisher to include everything (other publishers wanted us to trim it down to 250 pages). It is a compendium of programming methodology, programming techniques, fundamental algorithms and data structures. It emphasizes program correctness, through assertion techniques, and software architecture. The chapter on programming methodology contains the first ever published description of the Z specification language anywhere (as far as I know), in a very early form. The book was extremely successful in France, both as a textbook and for engineers in industry; incredibly, it still seems to be in print.

The Russian translation was also widely circulated and I still meet people from Russia who tell me this is where they learned programming. There never was an English translation: I accepted Prentice Hall's and Tony Hoare's suggestion that I do the translation myself — a huge mistake, as I started rewriting the book instead of translating it, and never finished, although that effort, titled Applied Programming Methodology, fed later work. In particular the object-oriented pseudocode that I used throughout, an extension of the notation in Méthodes de Programmation, led directly to Eiffel. "

Remerciements : Nous remercions Bertrand Meyer qui a autorisé cette reproduction et Marc Guyomard qui a conservé ses ouvrages d'informatique.

Nota : " La description de Z donnée ici est sommaire et partielle; nous avons pris quelques libertés avec les notations d'Abrial pour assurer la compatibilité avec celles de cet ouvrage." Meyer, Baudoin

et ajouter en fin

Claude Delobel et Michel Adiba en 1983 publieront Bases de données et systèmes relationnels, Dunod. Ils y utilisent la notation Z d'origine sous le nom de "modèle relationnel binaire".

Les sources de Z en 1980

J.R. Abrial, Data Semantics in Data Base Management (Klimbie, Koffeman, eds), North-Holland, 1974, pp. 1-59
J.R. Abrial, Manuel du langage Z. IMAG, Grenoble, 1977
J.R. Abrial, Manuel du langage Z et mécanismes de transformation. Notes internes Z/1 à Z/15 non publiées, EDF. notes Z2, Z3, Z7 (mars 1977), Z13, Z14 et Z15 (novembre 1977)
J.R. Abrial, B. Meyer, Notes de cours sur la spécification formelle ; étude de cas (un système de Course notes on formal specification; case study (Système d'annuaire automatisé), École d'été de l'INRIA-EDF-CEA sur la méthodologie de la programmation, Le-Bréau-sans-Nappe, France, Juillet 1978
J-R Abrial (consultant), S.A. Schuman (Intermetrics), B. Meyer (EDF), « Specification Language », Proceedings of Summer Scholl on Program Construction, Belfast, sept. 1979.
J-R Abrial, « The specification language Z : syntax and “semantics” », Oxford University Computing Laboratory, Programming Research Group, avril 1980.

Merci à J.P. Giraudin

de minimiser les interactions entre modules, et préserver l'unité fonctionnelle de chacun d'entre eux.

On notera que l'approche modulaire ne favorise ni n'interdit en aucune façon la conception descendante ou la conception ascendante ; l'une et l'autre peuvent en fait déterminer la stratégie de développement des modules. Tout au plus l'approche modulaire permet-elle de faire entrevoir des horizons plus larges, où le schéma de développement serait moins strictement hiérarchisé, plus "autogestionnaire", que les schémas ascendant et descendant traditionnels ; mais l'élaboration de méthodes permettant de réaliser un tel schéma, tout en préservant la rigueur de développement et en évitant le désordre conceptuel, est un problème ouvert.

VIII.3.5 Programmation statique. Le langage Z

VIII.3.5.1 Définitions : le langage Z_0

Dans tout ce qui précède, nous avons insisté sur l'idée de *spécification fonctionnelle*. Appliquée aux données, nous avons vu au chapitre V que cette notion permettait de définir une structure de données de façon parfaitement externe, à l'aide d'une liste de fonctions et de leurs propriétés abstraites, sans aucune référence aux modes de représentation en mémoire. Une caractérisation semblable a été appliquée aux programmes à partir du chapitre III, et en particulier aux sous-programmes à partir du chapitre IV, lorsque nous avons intercalé, sous forme de commentaires, des *assertions* permettant d'énoncer certaines propriétés sur l'état du programme ; si l'on s'efforce de rendre ces assertions exhaustives, une unité de programme est complètement caractérisée par son assertion initiale et son assertion finale.

Lorsqu'on cherche à développer ces assertions et à les détailler jusqu'au bout, on s'aperçoit vite qu'il s'agit d'un travail difficile ; d'un travail *aussi difficile que la programmation elle-même.*

A partir de cette remarque, une idée décisive, due en particulier à J.R. Abrial, consiste à franchir un pas supplémentaire et à poser que le développement des assertions *est une activité de programmation* : en d'autres termes, qu'un programme peut être entièrement décrit par une suite de relations statiques, qui fournissent simplement une image se situant à un niveau d'abstraction différent de celui de la description habituelle – et plus fondamental.

Le langage Z [Abrial 77] [Abrial 77a] pour la conception et la description des programmes est, ainsi, fondé sur l'idée qu'un programme peut et doit être décrit à plusieurs niveaux. Le premier de ces niveaux, appelé Z_0, permet une description purement statique de la tâche à résoudre ; il est entièrement non-algorithmique, c'est-à-dire que les programmes Z_0 ne contiennent pas d'instructions – d'ordres gouvernant le comportement dynamique d'une machine réelle ou virtuelle – mais seulement une liste de fonctions et de leurs propriétés. Il s'agit donc d'une spécification fonctionnelle, non plus d'une structure de données, mais, plus généralement, de programmes et de problèmes [1].

(1) La description de Z donnée ici est sommaire et partielle ; nous avons pris quelques libertés avec les notations d'Abrial pour assurer la compatibilité avec celles de cet ouvrage.

VIII.3.5.2 Le niveau Z_0

Z_0 est entièrement fondé sur un langage de communication et de description qui a eu l'occasion de faire ses preuves au cours des ans : le langage mathématique, et en particulier le langage de la théorie des ensembles, augmenté d'un petit nombre d'extensions correspondant aux problèmes précis rencontrés en programmation. Les éléments de base manipulables en Z_0 sont ainsi des "objets", des ensembles, et des ensembles ordonnés ou "tuples" ; les opérations de base sont les opérations ensemblistes habituelles (test d'appartenance d'un objet à un ensemble, test d'inclusion d'un ensemble dans un autre, union de deux ensembles, intersection, différence, produit cartésien, etc.) et des opérations sur les tuples comme la concaténation, notée $*$: la concaténation des tuples $t_1 = [a, b, c]$ et $t_2 = [d, e, f, g]$ est $t_1 * t_2 = [a, b, c, d, e, f, g]$; enfin, un programme Z_0 contient des définitions de fonctions et leurs propriétés. Par exemple, A et B étant deux ensembles, l'existence d'une fonction f de A dans B est notée :

$$A \xrightarrow{f} B$$

L'intérêt de cette notation (par rapport à celle que nous avons utilisée : $f : A \to B$) est qu'elle permet de préciser visuellement un certain nombre de propriétés possibles des fonctions, importantes pour le programmeur. Par exemple, il pourrait exister une fonction inverse $f^{-1} = g$; on notera :

$$A \underset{g}{\overset{f}{\longrightarrow}} B$$

f pourrait être une fonction partielle, c'est-à-dire ne pas donner de résultat pour tous les arguments ; il s'agit d'une information importante, qu'on notera en indiquant la borne inférieure 0 du nombre de résultats de f :

$$A \xrightarrow[g]{f(0)} B$$

Plus généralement, f pourrait être éventuellement multivaluée, c'est-à-dire associer à tout élément de A un sous-ensemble f(A) de B (plutôt que 0 ou un seul élément). Les noms des fonctions multivaluées, comme ceux des ensembles, commencent par des majuscules ; si F associe à tout élément de A entre 0 et 10 éléments de B, on notera :

$$A \xrightarrow[G(1,\ -)]{F(0,\ 10)} B$$

(F est ici surjective). Il faut ici préciser deux bornes, l'inférieure et la supérieure. Nous avons supposé en outre que la fonction réciproque G était totale, mais que la borne supérieure du nombre de ses résultats était inconnue, ce que représente le tiret. Là où une fonction partielle comme f ou F n'est pas définie, on conviendra que sa valeur est l'élément spécial **vide**. Les relations portant sur les fonctions sont des propriétés logiques, exprimées à l'aide des opérateurs habituels $\forall$ (pour tout), $\exists$(il existe), **et**, **ou**, **non**, etc.

VIII.3.5.3 Le niveau Z_1 et les transformations.

Le niveau suivant de langage, Z_1, permet une "algorithmisation" du programme Z_0 en ce sens qu'il introduit des instructions. En fait Z_1 est très proche

de la notation algorithmique utilisée dans cet ouvrage, avec la particularité qu'il emploie largement des expressions ensemblistes (**si** x ∈ A **alors** . . .) et des boucles ensemblistes du type (cf. III.3.1.3) :

```
pour x ∈ A répéter
    | .........
```

ou

```
pour x ∈ A tant que c(x) répéter
    | .........
```

L'intérêt tout particulier de cette caractéristique de Z_1 est qu'elle permet d'obtenir de façon systématique des formes "algorithmisées" des programmes statiques exprimés en Z_0. [Abrial 77a] donne ainsi toute une série de règles de transformation de relations Z_0 en instructions Z_1. Par exemple, pour évaluer la condition (expression logique) Z_0 suivante :

$$\forall x \in A \,.\, c(x)$$

on écrira en Z_1 :

```
| e ← vrai ;
| pour x ∈ A tant que e répéter
|     | e ← c(x)
```

Le programme Z_1 ainsi obtenu n'est pas définitif : il faut encore se rapprocher , par des transformations successives, d'un niveau proche de celui des langages usuels. Le niveau suivant de langage, qui n'a pas reçu de nom particulier (on serait tenté de l'appeler Z_2), est débarrassé de toute référence ensembliste : les ensembles, représentés par des fichiers, des tableaux, des listes, etc., sont explicitement parcourus grâce à des ordres **ouvrir** (ouverture du "fichier" et accès au premier élément) et **lire** (accès aux éléments suivants) ; les éléments ont été épuisés si et seulement si l'élément lu est l'élément spécial **vide**. Par exemple :

```
| pour x ∈ A répéter
|         | p(x)
```

s'écrit maintenant :

```
| x ← ouvrir(A) ;
| tant que x ≠ vide répéter
|     | p(x) ;
|     | x ← lire(A)
```

Les transformations suivantes permettront de préciser encore la réalisation du programme : élimination éventuelle de la récursivité, choix de représentation des ensembles par des structures de données plus précises (cf. V.4) en vue de permettre des accès efficaces selon les besoins du problème, etc. Le but est, bien sûr, d'arriver finalement à une version FORTRAN, ALGOL, PL/1, COBOL, etc.

VIII.3.5.4 Un exemple

Pour pouvoir évaluer la méthode d'Abrial, il convient de l'esquisser sur un exemple – hélas trop simple. Il s'agit du traitement d'un fichier documentaire, destiné à en permettre l'"inversion". (1)

(1) Cet exemple a été étudié en collaboration avec M. Demuynck, C. Maillard et P. Moulin.

Soit un fichier B consitué d'"enregistrements". On veut écrire dans un fichier D des "enregistrements" déduits de certains de ceux de B selon des "formats de sortie" contenus dans un fichier C. Un fichier A contient une liste de "clés" permettant de sélectionner les éléments de B qui doivent être traités (figure VIII.4).

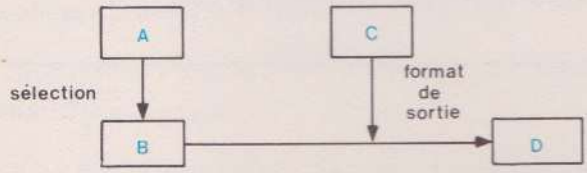


Figure VIII.4

Un élément de B est un "article" composé d'une "clé" et d'un nombre quelconque de couples [nom de champ, valeur] ; un "nom de champ" est un code comme C10, C20, etc ; une "valeur" est un qualificateur, comme "pompe", "turbine", "explosion", etc. Deux éléments distincts ne peuvent avoir la même clé.

Si le fichier A est vide, tous les articles de B seront traités ; sinon, A contient des clés, et seuls les articles de B dont la clé se trouve dans A seront traités.

Dans le traitement d'un article b de B, des enregistrements seront écrits dans D pour chacun des couples [nom de champ, valeur] appartenant à b ; ces enregistrements auront la forme [clé, valeur, numéro de sous-fichier], où la "clé" est celle de l'article b, la "valeur" est celle qu'on trouve dans le couple traduit, et le "numéro de sous-fichier" est déduit du "nom de champ" grâce au fichier C, qui contient une liste de couples [nom de champ, numéro de sous-fichier]

Nous avons volontairement exposé le problème dans son intégralité avant de passer en Z ; pour le traiter, il est utile de séparer les niveaux. Cherchons d'abord à l'exposer en Z_0 au niveau global. (Nous numéroterons les définitions d'ensembles : E_1, E_2, etc. ; les définitions de fonctions : F_1, etc ; les relations : R_1, etc.)

Les ensembles manipulés sont d'abord les fichiers A, B, C, D. B (le fichier de départ) contient des "articles" ; nous postulons donc l'existence d'un ensemble :

Articles

et l'on a :

$B \subseteq$ Articles

Nous avons vu que tout article était formé d'une clé et d'un nombre quelconque de couples [nom de champ, valeur]. Ceci s'écrit comme un produit cartésien :

Articles = Clés × **Tuple** (Nomchamps × Valeurs) (E5)

où Clés est l'ensemble des clés, Nomchamps celui des noms de champs, Valeurs celui des valeurs. X étant un ensemble, la notation **Tuple**(X) désigne en Z_o l'ensemble :

$$\textbf{Tuple}(X) = X \cup X \times X \cup X \times X \times X \cup \ldots$$

c'est-à-dire l'ensemble des suites finies d'éléments de X. Ainsi, [3], [8, 5, 9], [45, 12, 50, 10], etc. appartiennent à **Tuple** (Entiers).

Enfin, un élément de D est un triplet [clé, numéro de sous-fichier, valeur] ; on aura donc ;

$$D \subseteq \text{Clés} \times \text{Numsousfich} \times \text{Valeurs} \qquad \text{(E9)}$$

où Numsousfich est l'ensemble des numéros de sous-fichier possibles.

En récapitulant, les ensembles manipulés sont donc :

Clés	(E1)
Nomchamps	(E2)
Valeurs	(E3)
Numsousfich	(E4)
Articles = Clés x **Tuple**(Nomchamps x Valeurs)	(E5)
$B \subseteq$ Articles	(E6)
$A \subseteq$ Clés	(E7)
$C \subseteq$ Nomchamps x Numsousfich	(E8)
$D \subseteq$ Clés x Valeurs x Numsousfich	(E9)

Au niveau global, le traitement à réaliser peut s'exprimer comme une fonction sans argument, qu'on exprimera comme une fonction dont l'ensemble de départ est l'ensemble spécial à un élément noté **1** :

$$\mathbf{1} \xrightarrow{\text{Trait}\,(0,-)} D \qquad \text{(F1)}$$

Trait est une fonction multivaluée, comme l'indique l'emploi de la majuscule : on crée 0, 1 ou plusieurs enregistrements de D.

Les articles de B sont traités, "traduits", un à un, ce qui suggère d'écrire :

$$\text{Trait} = \bigcup_{b \in B} \text{Traduction}(b) \qquad \text{(R'1)}$$

avec la fonction

$$B \xrightarrow{\text{Traduction}\,(0,-)} D \qquad \text{(F2)}$$

où Traduction est bien multivaluée puisque chaque article de B peut donner lieu à la création de plusieurs articles de D. La relation (R'1) ci-dessus est cependant incomplète car nous avons vu que tous les éléments de B ne sont pas traités si A n'est pas vide ; elle doit être remplacée par :

$$\text{Trait} = \bigcup_{b \in B\,.\,\gamma(b)} \text{Traduction}(b) \qquad \text{(R1)}$$

notation qui signifie que l'"union" n'est opérée que sur les b qui vérifient la condition $\gamma(b)$, avec :

$$\text{Article} \xrightarrow{\gamma(1)} (\textbf{vrai}, \textbf{faux}) \qquad \text{(F3)}$$

et :

$$\forall\, b \in \text{Articles}\,.\,\gamma(b) = (A = \phi \textbf{ ou } b(1) \in A) \qquad \text{(R2)}$$

Articles étant le produit cartésien Clés × **Tuple** (Nomchamps × Valeurs), b(1) désigne le premier composant d'un article b, sa clé.

Nous devons maintenant exprimer la fonction Traduction. Elle portera sur chacun des couples [nom de champ, valeur] formant le second composant, b(2), d'un article b. Ces couples sont traduits séparément, ce que nous exprimerons en introduisant la fonction de traduction d'un couple :

$$\text{Nomchamps} \times \text{Valeurs} \xrightarrow{\text{tradélém (1)}} \text{Valeurs} \times \text{Numsousfich} \qquad (F'3)$$

et en exprimant Traduction par

$$\forall\, b \in B\ .\ \text{Traduction}(b) = \mathop{\mathfrak{U}}_{x \in b(2)} b(1) * \text{tradélem}(x) \qquad (R'3)$$

L'astérisque, rappelons-le, représente la concaténation de tuples.

Pour exprimer la fonction tradélem, associant à tout couple [nom de champ, valeur] un couple [valeur, numéro de sous-fichier], nous introduisons la fonction permettant d'associer un numéro de sous-fichier à un nom de champ, soit numsous :

$$\text{Nomchamps} \xrightarrow{\text{numsous (1)}} \text{Numsousfich} \qquad (F4)$$

Cette fonction, rappelons-le, est représentée par le fichier C ; D'après (E8), $C \subseteq \text{Nomchamps} \times \text{Numsousfich}$, et l'on a précisément :

$$\forall\, c \in C\ .\ \text{numsous}(c(1)) = c(2) \qquad (R4)$$

On notera que l'existence du fichier C et celle de la fonction numsous sont deux propriétés équivalentes ; cette équivalence est exprimée par la relation (R4). Une telle redondance est une des caractéristiques de la programmation au niveau Z_0 : ce n'est qu'au moment de l'"algorithmisation" qu'on choisit entre une représentation par une fonction (= sous-programme) ou par un ensemble (= structure de données). Au niveau Z_0, on se refuse à choisir entre l'espace et le temps car cette question est prématurée, et l'on garde les deux formulations redondantes.

numsous nous permet d'expliciter tradélem :

$$\forall\, b \in B.\ \ \forall x \in b(2).\ \text{tradélem}(x) = [x(2), \text{numsous}(x(1))] \qquad (R'4)$$

d'après la description donnée du traitement d'un couple.

Il reste à exprimer que A contient, s'il n'est pas vide, des clés permettant d'accéder aux éléments de B, et ceci de façon univoque, deux éléments distincts de B ayant des clés différentes. Cela se traduit par l'existence d'une fonction injective :

$$B \xrightarrow[\text{article de clé (0)}]{\text{clé (1)}} \text{Clés} \qquad (F5)$$

La fonction importante, ici, est la fonction inverse article de clé ; en l'écrivant comme une fonction monovaluée, nous avons exprimé qu'on pouvait accéder de façon unique à un élément de B à partir de sa clé. Cette fonction est partielle, comme l'indique le 0 dans sa définition : toutes les clés possibles ne sont pas nécessairement représentées dans B. Notez que, comme précédemment, l'affirmation de l'existence d'une fonction clé totale est redondante avec les relations

ensemblistes précédentes (E5) et (E6) ; la correspondance est exprimée par la relation :

$$\forall\, b \in B \,.\, \text{clé}(b) = b(1) \qquad \text{(R5)}$$

Finalement, nous pouvons récrire (R'3) en nous dispensant de la fonction intermédiaire tradélem grâce à (R'4), ce qui permet d'obtenir, en récapitulant, le programme Z_0 final suivant :

ENSEMBLES

Clés	(E1)
Nomchamps	(E2)
Valeurs	(E3)
Numsousfich	(E4)
Articles = Clés x **Tuple** (Nomchamps x Valeurs)	(E5)
$B \subseteq$ Articles	(E6)
$A \subseteq$ Clés	(E7)
$C \subseteq$ Nomchamps x Numsousfich	(E8)
$D \subseteq$ Clés x Numsousfich x Valeurs	(E9)

FONCTIONS

$\mathbf{1} \xrightarrow{\text{Trait}\,(0,-)} D$	(F1)
$B \xrightarrow{\text{Traduction}\,(0,-)} D$	(F2)
$\text{Article} \xrightarrow{\gamma\,(1)} (\textbf{vrai}, \textbf{faux})$	(F3)
$\text{Nomchamps} \xrightarrow{\text{numsous}\,(1)} \text{Numsousfich}$	(F4)
$B \xrightarrow[\text{article de clé}\,(0)]{\text{clé}\,(1)} \text{Clés}$	(F5)

RELATIONS

$\text{Trait} = \mathop{\mathfrak{U}}_{b \in B \,.\, \gamma(b)} \text{Traduction}\,(b)$	(R1)
$\forall\, b \in \text{Articles} \,.\, \gamma(b) = (A = \phi \textbf{ ou } b(1) \in A)$	(R2)
$\forall b \in B.\ \text{Traduction}(b) = \mathop{\mathfrak{U}}_{x \in b(2)} [b(1), x(2), \text{numsous}(x(1))]$	(R3)
$\forall\, c \in C \,.\, \text{numsous}\,(c(1)) = c(2)$	(R4)
$\forall\, b \in B \,.\, \text{clé}\,(b) = b(1)$	(R5)

On notera que si l'ordre des enregistrements dans D était important, et nécessairement le même que celui des éléments de B, il faudrait remplacer les unions par des concaténations (*).

Pour passer à un programme Z_1, nous traduirons d'abord (R1) par la version (V1) suivante :

(V1)
```
Trait ← ∅ ;
pour b ∈ B répéter
    si γ (b) alors
        Trait ← Trait ∪ Traduction (b)
```

c'est-à-dire, en explicitant la condition $\gamma(b)$:

(V2)
```
Trait ← ∅ ;
pour b ∈ B répéter
    si A = ∅ ou b(1) ∈ A alors
        Trait ← Trait ∪ Traduction (b)
```

Comme le test "$A = \emptyset$?" est indépendant de b, on peut le "sortir de la boucle" pour traiter séparément les cas $A = \emptyset$ et $A \neq \emptyset$. Il s'agit de l'une des "transformations" standard répertoriées dans [Abrial 77a]. Elle nous donne la version (V'3) suivante :

(V'3)
```
Trait ← ∅ ;
si A = ∅ alors
    pour b ∈ B répéter
        Trait ← Trait ∪ Traduction (b)
sinon {A ≠ ∅}
    pour b ∈ B répéter
        si b(1) ∈ A alors
            Trait ← Trait ∪ Traduction (b)
```

Nous pouvons cependant poursuivre ici le développement du programme sur une voie légèrement différente. L'existence de la fonction monovaluée partielle article de clé (propriété (F5)) nous montre en effet que, dans le cas où $A \neq \emptyset$, on peut accéder à un élément b vérifiant $b(1) \in A$, c'est-à-dire clé (b) $\in$ A (relation (R5)), c'est-à-dire encore de clé donnée, à partir de sa clé dans A. On peut donc fonder l'algorithme, dans le cas où $A \neq \emptyset$, sur un parcours séquentiel de A plutôt que de B. C'est ce que nous choisissons de faire ici ; notez que cette décision ne se justifie que si l'on en espère un gain d'efficacité, c'est-à-dire s'il y a beaucoup moins de clés dans A que d'articles dans B.

Nous obtenons :

(V3)
```
Trait ← ∅ ;
si A = ∅ alors      {comme précédemment : B traité séquentiellement}
    pour b ∈ B répéter
        Trait ← Trait ∪ Traduction (b)
sinon               {A ≠ ∅ : accès "direct" aux éléments de B
                     à partir de leur clé}
    pour a ∈ A répéter
        b ← article de clé (a) ;
        si b ≠ vide {c'est-à-dire si la fonction "article de
                     clé" est définie sur a} alors
            Trait ← Trait ∪ Traduction (b)
```

Nous utilisons maintenant (R3) pour développer Traduction (b) ; la notation "X :∋ x", où X est un ensemble et x un objet, désigne en Z l'instruction d'inclusion ajoutant l'élément x à l'ensemble X (1).

```
|Trait ← ∅ ;
|si A = ∅ alors
|    |pour b ∈ B répéter
|    |        |pour x ∈ b(2) répéter
|    |        |        |Trait :∋ [b(1), x(2), numsous (x(1))]
(V4) |sinon
|    |pour a ∈ A répéter
|    |        |b ← article de clé (a) ;
|    |        |si b ≠ vide alors
|    |        |        |pour x ∈ b(2) répéter
|    |        |        |    |Trait :∋ [b(1), x(2), numsous
|    |        |        |    |         (x(1))]
```

L'étape suivante consiste à éliminer les boucles **pour** ensemblistes en les exprimant par des parcours séquentiels explicites. D'après les définitions données plus haut des opérations **ouvrir**, **lire**, et de la valeur spéciale **vide**, on obtiendra la version (V5) :

```
|Trait ← ∅ ;
|a ← ouvrir (A) ;
|si a = vide alors
|        |b ← ouvrir (B) ;
|        |tant que b ≠ vide répéter
|        |    |x ← ouvrir (b(2)) ;
|        |    |tant que x ≠ vide répéter
|        |    |    |Trait :∋ [b(1), x(2), numsous (x(1))] ;
|        |    |    |x ← lire (b(2))
|        |    |b ← lire (B)
(V5)     |sinon
|        |répéter
|        |    |b ← article de clé (a) ;
|        |    |si b ≠ vide alors
|        |    |    |x ← ouvrir (b(2)) ;
|        |    |    |tant que x ≠ vide répéter
|        |    |    |    |Trait :∋ [b(1), x(2), numsous (x(1))] ;
|        |    |    |    |x ← lire (b(2))
|        |    |a ← lire (A) ;
|        |jusqu'à a = vide
```

(1) Notez la correspondance formelle entre l'instruction d'affectation habituelle a : = b, rendant vraie la condition "a = b", et l'affectation ensembliste X : ∋ x, rendant vraie la condition "X ∋ x", ou "X contient x".

Remarquons que peu de décisions de mise en œuvre ont été prises, hormis la notation b(1) plutôt que clé (b). En particulier, **ouvrir** et **lire** n'impliquent pas nécessairement des entrées-sorties physiques, mais peuvent être l'accès à un tableau (i ← 1 pour **ouvrir** ; i ← i + 1 pour **lire**) ou à une structure chaînée ℓ (x ← premier (ℓ) ; x ← suivant (x)).

De nombreuses transformations resteraient à opérer sur ce programme : il faut choisir un mode d'accès à A et à B, remplacer l'affectation ensembliste à Trait par une instruction d'écriture, etc. La méthode devrait cependant être claire.

A titre de complément, nous fournissons ci-après (une fois n'est pas coutume) un programme COBOL donnant une version exécutable du programme Z ci-dessus, et obtenu par transformation systématique à partir de celui-ci. Seules la *PROCEDURE DIVISION* et une partie de l'*IDENTIFICATION DIVISION* sont fournies.

Pour comprendre ce programme, il pourra être utile de savoir que les représentations physiques adoptées dans cette réalisation sont les suivantes :

- A est un "fichier séquentiel" externe, qui pourrait être sur cartes, sur disque ou sur bande magnétique. On y accède par l'ordre de lecture *READ*, qui lit les enregistrements dans l'ordre jusqu'à la fin du fichier, signalée par la condition spéciale *AT END*. Les enregistrements lus sont placés dans une chaîne de caractères, ici *ENR-A* : l'association entre le fichier et sa "zone de lecture" se fait en COBOL à la compilation, grâce à la *FILE SECTION* de la *DATA DIVISION*.
- D est un fichier séquentiel analogue à A, que l'on constitue par des ordres *WRITE* successifs, qui écrivent chacun l'enregistrement présent dans *ENR-D* ; même remarque que ci-dessus en ce qui concerne l'association entre le fichier et sa "zone d'écriture".
- C est représenté par le tableau *TABC*, rangé par numéro de sous-fichier : la valeur de *TABC(I)* est le nom de champ associé au sous-fichier numéro *I*. Le chargement en mémoire du tableau n'a pas été représenté.
- B, enfin, est une "base de données" gérée par le programme IMS d'IBM, et résidant sur disque. Sa structure hiérarchique est la suivante : chaque enregistrement b de B, tel qu'il a été défini plus haut, est représenté par une "racine" contenant la clé b(1), à laquelle sont rattachés des "segments dépendants" en nombre quelconque, représentant chacun un couple [nom de champ, valeur] de b(2) ; leur "type de segment" est le nom du champ, et leur contenu est la valeur associée.

Les accès à la base de données se font par l'intermédiaire du programme *CBLTDLI*, auquel on fournit :

- un argument qui spécifie le type d'opération d'entrée-sortie désirée : *GU* signifie "lire la première racine de la base de données" ; *GN* veut dire "lire le segment ou la racine suivante" ; *GNP* signifie ici "lire en séquence les segments qui dépendent hiérarchiquement de la dernière racine lue" ;
- un argument, le *PCB*, qui fournit au système d'exploitation certains renseignements indispensables à la bonne gestion de la base de données, et dont deux sous-ensembles utiles au programme utilisateur sont le *CODE-RETOUR* (*'GE'* signifie "racine ou segment non trouvé", *'GB'* veut dire "fin de fichier atteinte") et le *TYPE-SEGMENT* qui, dans ce cas précis, fournit le nom de champ auquel est associé une valeur lue ;
- un argument spécifiant la zone à utiliser pour placer la racine ou le segment lu ;
- un argument facultatif qui permet de sélectionner certains enregistrements, et qui est formé d'une chaîne de caractères, écrite dans un langage de commande spécial appelé DL/1, et décodée à l'exécution par un interprète contenu

dans le système de gestion de bases de données IMS : ici, *SELECT-RACINE* impose de ne lire que des racines, et *SELECT-RACINE-CLE* requiert l'accès direct à une racine de clé donnée (lue dans le fichier A). C'est ce qui a été exprimé en Z par l'affirmation de l'existence d'une fonction monovaluée partielle "article de clé", permettant d'accéder à un élément de B à partir de sa clé contenue dans A.

D'un point de vue méthodologique, on notera que dans ce cas précis la structure concrète des fichiers était connue a priori : le programme en Z_0 représente donc un effort d'abstraction de ses propriétés caractéristiques. La forme en Z_0 est plus facilement utilisable, nous semble-t-il, que la description détaillée de la base dans le système de gestion de bases de données utilisé (IMS), et plus indépendante de choix techniques non fondamentaux, qui peuvent être remis en cause ultérieurement. Mais la programmation en Z permet de revenir au système concret, comme nous le faisons ici, tout en gardant une description de plus haut niveau d'abstraction.

COBOL

```
PROCEDURE DIVISION.
        .
        .
        .
        OPEN OUTPUT D
        OPEN INPUT A
        MOVE 'B1' TO NOM-SELECT
        READ A AT END GO TO A-VIDE.
ETIQ1.
        MOVE ENR-A TO CLE-SELECT
        CALL 'CBLTDLI' USING GU PCB ZONE-B1 SELECT-RACINE-CLE
        IF CODE-RETOUR NOT = 'GE' CALL 'TRAD' USING ZONE-B1 PCB.
        READ A AT END CLOSE A GOBACK.
        GO TO ETIQ1.
A-VIDE.
        CALL 'CBLTDLI' USING GU PCB ZONE-B1 SELECT-RACINE
        IF CODE-RETOUR = 'GE' CLOSE A, D GOBACK.
ETIQ2.
        CALL 'TRAD' USING ZONE-B1 PCB
        CALL 'CBLTDLI' USING GN PCB ZONE-B1 SELECT-RACINE
        IF CODE-RETOUR = 'GB' CLOSE A, D GOBACK.
        GO TO ETIQ2.
```

COBOL

```
IDENTIFICATION DIVISION.
PROGRAM-ID.
     TRAD.
      .
      .
      .
      .
PROCEDURE DIVISION USING ZONE-B1 PCB.
ETIQ1.
     CALL 'CBLTDLI' USING GNP PCB ZONE-X2
     IF CODE-RETOUR = 'GE' OR 'GB' GOBACK.
     PERFORM RECH-NUMSOUS THRU FINR IF I = 0 GO TO ETIQ1.
     MOVE ZONE-B1 TO ENR-D (1)
     MOVE ZONE-X2 TO ENR-D (2)
     MOVE NUMSOUS (I) TO ENR-D (3)
     WRITE ENR-D
     GOTO ETIQ1.
RECH-NUMSOUS.
     MOVE 0 TO I.
BOUCLE.
     ADD 1 TO I
     IF I > IMAX MOVE 0 TO I GO TO FINR.
     IF TYPE-SEGMENT NOT = TABC (I) GO TO BOUCLE.
FINR.
     EXIT.
```

VIII.3.5.5 Discussion

L'utilisation de Z a le mérite principal de permettre de poser clairement quelques-unes des vraies questions qui sont la clé de la réalisation d'un programme. Or la détection des vraies questions est souvent en programmation – comme plus généralement dans toutes les disciplines scientifiques – l'étape la plus délicate de la recherche des solutions.

Dans la réalisation d'un projet important de programmation, une grande partie de la difficulté tient souvent à l'imprécision de la tâche à résoudre : on se trouve confronté à un "cahier des charges" lourd, multiforme, incomplet, contradictoire, désordonné. Chacune des tâches élémentaires demandées est souvent simple ; presque toujours, en tous cas, les difficultés conceptuelles véritables sont limitées à un petit nombre de sous-problèmes. La véritable difficulté tient à la multiplicité des demandes, à la définition de leurs relations mutuelles, à cette espèce de grouillement de spécifications devant lequel programmeurs et chefs de projets se sentent désespérés.

> Face à une telle situation, typique de l'informatique de gestion, mais qui se rencontre aussi dans tous les autres domaines d'application,

on a souvent le sentiment qu'il suffirait de poser complètement le problème pour que la suite du développement coule de source. Or c'est précisément de poser complètement et rigoureusement le problème que s'occupe Z_0. Mais, à la différence d'un cahier des charges rédigé en français (et par des non-spécialistes de l'informatique), un programme Z_0 est directement exploitable pour donner un programme au sens habituel, algorithmique, du terme, à l'aide des transformations mentionnées ci-dessus, de façon en principe mécanisable – bien que dans l'état actuel de la technique l'intuition et l'expérience du programmeur continuent bien sûr à jouer un rôle.

L'expérience de la programmation en Z montre que l'on est parfois tenté, dans la résolution d'un problème, d'abandonner assez vite le niveau Z_0 pour passer à un programme "algorithmique" en Z_1, alors même qu'on n'a pas entièrement spécifié le problème en Z_0 : l'habitude des langages de programmation usuels donne en effet à croire que les sous-problèmes que l'on a négligés d'exprimer en détail sont triviaux, et qu'on pourra les résoudre en cours de route en écrivant le programme Z_1. Or bien souvent, on se heurte en Z_1 à des difficultés ou à des contradictions qui paraissent inextricables ; si l'on cherche à en déceler les raisons, on s'aperçoit que la démarche "algorithmisante" tend à *surspécifier* constamment le problème, en obligeant à prendre des décisions arbitraires qui peuvent se révéler trop contraignantes, et lier le programmeur au point qu'il aboutira à une impasse lors d'une étape ultérieure.

Un exemple typique de ce genre de surspécification est l'**ordre de parcours** d'un ensemble : bien souvent, on veut effectuer une action $a(x)$ pour tous les éléments x d'un ensemble A, l'ordre dans lequel ces éléments sont traités étant sans importance. Dans l'écriture d'un programme au sens classique, on est obligé de prescrire un ordre de parcours de l'ensemble, même si celui-ci est implicite ; par exemple, si les éléments de l'ensemble sont triés selon une certaine clé, et si on les consulte séquentiellement, c'est cet ordre qui sera adopté. Pour une étape ultérieure, on peut s'apercevoir que l'ordre de parcours devient significatif, mais qu'un ordre différent est désiré : on se trouve alors lié par une décision de conception prise trop tôt. En Z_0, au contraire, aucun ordre de parcours n'est spécifié lorsqu'on représente la transformation f correspondant à a et ses propriétés :

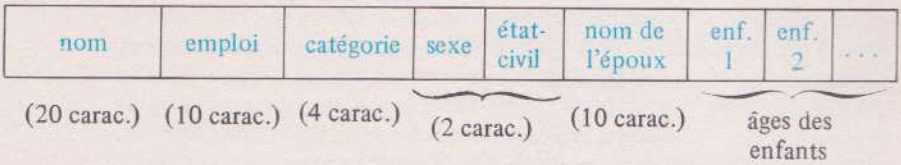
$$\begin{cases} A \xrightarrow{f(\ldots)} B \\ \forall x \in A \,.\, f(x) = \ldots\ldots \end{cases}$$

Notons qu'ici le premier niveau de Z_1 permet encore de ne pas spécifier l'ordre de parcours s'il n'est pas nécessaire :

```
pour x ∈ A répéter
    a(x)
```

Cependant, dès l'étape Z_1, chaque fois qu'on écrit l'enchaînement de deux instructions, on prescrit un ordre d'exécution qui peut ne pas être significatif.

Un problème similaire de surspécification se pose à propos des structures de données et du compromis espace-temps. Que de projets de programmation se retrouvent entravés par un choix de représentation trop hâtif ! Soit un exemple aussi simple qu'un fichier du personnel, destiné à la paye. Supposons qu'on ait choisi une fois pour toutes qu'il sera composé d'enregistrements de la forme :



Que ce choix soit initialement bon ou mauvais importe peu. Mais, ce format étant fixé et connu, tous les programmes accèderont à l'emploi comme aux caractères 21 à 30 de chaque enregistrement, à l'état-civil comme au 36ème caractère, etc. (nous supposons que les auteurs de ces programmes n'ont pas lu le chapitre V).

Supposons maintenant que, les programmes étant corrects, on s'aperçoive au moment du chargement du fichier que celui-ci demande une taille de mémoire (interne ou périphérique) inacceptable. On cherchera à réduire la place nécessaire, et ceci peut très certainement se faire à peu de frais. Le nombre d'emplois possibles est certainement limité à quelques milliers au plus, et 14 bits représentant un pointeur vers une "table des emplois" suffisent sans doute. Les noms se répètent : une méthode de type de l'adressage associatif peut se justifier, voire des methodes spéciales utilisant la redondance naturelle des langages usuels ("*trie*" : voir [Knuth 73], 6.3) et permettant d'économiser de la place dans des proportions considérables ; la simple adoption d'un format variable peut déjà donner de bons résultats si la plupart des noms ont moins de 10 lettres. Le sexe peut se représenter sur un seul bit. Le nom de l'époux est en général le même que celui de l'employé : avec un format variable, on utilisera un bit indiquant si tel est le cas, et, seulement si la réponse est négative, une zone décrivant le nom de l'époux – ou encore un pointeur si l'époux figure également sur le fichier.

De telles modifications sont l'application de techniques de routine pour le programmeur professionnel ; elles peuvent réduire l'encombrement d'un fichier dans des proportions considérables, voir permettre dans certains cas de conserver en mémoire centrale tout un fichier qui paraissait demander un disque entier. Notez que la démarche inverse pourrait être de mise : si le temps de calcul est une denrée précieuse, mais l'espace une ressource abondante, on sera conduit à répéter en de multiples exemplaires des informations redondantes pour diminuer à l'extrême les calculs effectués.

Le point important est que ce sont là des décisons de représentation, qui dépendent de l'environnement, des tests, etc. Prendre ces décisions au moment de la conception est prématuré, et peut amener des réécritures multiples et des modifications en chaîne lorsqu'on les remet en cause – ce qui ne saurait manquer d'arriver.

Par contraste, soit le "programme" Z_0 :

$$\text{Employés} \xrightarrow[\text{Employé de nom}(1,-)]{\text{nom}(1)} \text{Noms}$$

{Employés est l'ensemble des employés, Noms celui des noms ; ils sont reliés par la fonction nom : tout employé a un nom et un seul ; un nom peut être porté par un ou plusieurs employés}

$$\text{Employés} \xrightarrow{\text{emploi (1)}} \text{Emplois}$$

$$\text{Employés} \xrightarrow{\text{catégorie (1)}} \text{Catégories}$$

$$\text{Employés} \xrightarrow{\text{sexe (1)}} (\text{masculin, féminin})$$

$$\text{Employés} \xrightarrow{\text{etat-civil (1)}} (\text{veuf, divorcé, marié, célibataire})$$

$$\text{Employés} \xrightarrow{\text{Enfants (0, –)}} \text{Ages-enfants}$$

Parmi les relations, on aura par exemple (nous supposons pour les besoins de la démonstration que l'entreprise est particulièrement sourcilleuse quant à la moralité de son personnel) :

$$\forall\, x \in \text{Employés}.\ \text{état-civil}(x) = \text{célibataire} \Rightarrow \text{Enfants}(x) = \phi$$

Si à chaque emploi correspond une catégorie et une seule, on notera qu'il existe une fonction :

$$\text{Emplois} \xrightarrow[\text{Emplois associés (1, –)}]{\text{grade associé (1)}} \text{Catégories}$$

En termes de représentation physique, l'existence de cette fonction signifie qu'il n'est pas strictement nécessaire que chaque enregistrement contienne un champ "catégorie", puisque la catégorie peut être "calculée" à partir de l'emploi (en passant par une table). Mais le programme Z_0 ne prescrit rien : il fournit les éléments permettant de choisir la meilleure représentation physique. Parmi ces renseignements, ceux qui indiquent quels sont, pour une fonction f, les nombres minimal et maximal d'éléments associés à tout élément (par exemple (1, 1) pour la fonction emploi ci-dessus), et ceux qui indiquent les caractéristiques de la fonction inverse, sont parmi les plus importants pour guider les choix ultérieurs en Z_1, ce qui explique qu'une notation spéciale ait été développée pour eux (ils pourraient être exprimés par des relations, comme dans les spécifications fonctionnelles du chapitre V).

La leçon de ces remarques est qu'il faut résister à la tentation, mentionnée ci-dessus, de passer trop vite en Z_1, et *toujours spécifier complètement le problème en* Z_0 avant d'aborder les niveaux suivants.

La méthode de programmation qui a été rapidement présentée, utilisant une approche initiale "statique" des programmes et des transformations répétées, nous paraît apporter des solutions intéressantes à plusieurs des problèmes importants de la programmation. De fait, le langage Z nous paraît refléter une évolution importante. Ceci étant, on peut cependant émettre un certain nombre de réserves :

- l'emploi d'un formalisme fondé sur les notations mathématiques relativement avancées de la théorie des ensembles pose un certain nombre de problèmes psychologiques et pédagogiques – moins peut-être, d'ailleurs, à cause de la difficulté intrinsèque de ces notations, finalement plus simples que FORTRAN, COBOL, ou (certainement !) PL/1, que du statut spécial ("totem et tabou") dont jouissent les mathématiques dans notre société ;
- l'importante séparation entre aspects statiques et dynamiques ne résoud pas le problème de la décomposition d'une tâche complexe. De fait,

un programme en Z_0 pose les mêmes problèmes de division de la complexité, s'il est assez gros, qu'un programme habituel. Les méthodes usuelles s'appliquent : on peut parler de programmation descendante et de programmation ascendante en Z_0, et une stratégie de résolution des problèmes faisant appel à ces notions s'impose dès que les projets dépassent une certaine taille. Schématiquement, dans le processus de programmation en Z_0, qui consiste à définir des ensembles, des fonctions, et des relations, l'approche descendante conduit à dégager les fonctions correspondant au traitement à effectuer avant d'avoir décrit en détail les ensembles auxquels elles s'appliquent ; l'approche ascendante, à définir au contraire complètement les ensembles avant de s'intéresser aux fonctions qui leur sont appliquées ;

– on peut se demander si la somme de travail supplémentaire que demande la construction du programme en Z_0, est justifiée. Ce travail est effectivement assez lourd. La seule réponse à cette objection est la constatation que le programme statique est de toute façon construit implicitement en programmation usuelle, et qu'il est important d'en posséder une version explicite, afin de pouvoir distinguer les choix fondamentaux et les choix de représentation, avec les avantages annexes que sont les possibilités d'optimisation et la présence d'une documentation rigoureuse et exploitable (le programme Z_0) correspondant vraiment aux programmes réalisés.

VIII.3.6 Le programme et ses transformations.

Une conséquence naturelle de toutes les remarques émises dans les paragraphes précédents, sur laquelle il est utile d'insister, est l'idée qu'**on ne peut composer un programme en une seule étape** : la programmation est un processus trop difficile pour qu'on puisse espérer obtenir immédiatement, comme d'un coup de baguette magique, un programme satisfaisant selon tous les critères légitimement applicables. Le programme doit plutôt être considéré comme un objet de travail, éternellement perfectible, que des transformations successives amélioreront pour le rendre plus conforme aux diverses qualités désirées, à partir d'une version initiale qui n'avait peut-être que celle d'être correcte, au détriment de la facilité de mise en œuvre, de l'efficacité, etc. Nous avons étudié ci-dessus de telles transformations : le passage d'un niveau d'abstraction à un niveau plus proche de la machine, le passage d'un programme statique à un programme dynamique, etc. D'autres transformations importantes sont celles qui permettent d'obtenir des programmes de mieux en mieux adaptés à un environnement précis d'exécution (langage, machine), à partir d'une forme conceptuellement exacte, mais peut-être inefficace. C'est en particulier le cas des transformations diminuant le degré de récursivité d'un programme ou l'exprimant de façon non récursive pour une réalisation en FORTRAN par exemple, remplaçant une structure de table par une réalisation plus spécifique (adressage associatif, arbre binaire AVL, B-arbre...) qui tient compte des modalités d'accès, simulant l'allocation dynamique, etc.

Cette méthode d'amélioration itérative des programmes est probablement la seule qui permette d'obtenir des produits totalement satisfaisants. Elle soulève

un certain nombre de problèmes : il faut en effet garantir que les versions successives restent correctes, ou encore que la validité par rapport à une certaine spécification est un "invariant" des transformations successives. Une question qu'on peut légitimement poser, par exemple, est celle-ci : que deviennent les spécifications intermédiaires, les "assertions" associées aux divers points d'un programme, au cours des transformations successives ?

On trouvera dans [Knuth 76] l'exposé d'un tel développement complet de programme (résolvant le problème des "mariages stables"). On peut envisager qu'à l'avenir des systèmes conversationnels aideront le programmeur à réaliser de telles transformations ; un tel système, permettant en particulier d'éliminer les récursions inutiles, est décrit dans [Darlington 76]. On trouvera des fondements théoriques à ces méthodes dans [Burstall 77] et [Arsac 77].