UNE EVOLUTION DES LANGAGES VERS

UNE MEILLEURE RELATION HOMME-MACHINE

Nadia Thalmann
Docteur ès Sciences
Université du Québec à
Montréal
C.p. 8888
Montréal, Québec

Daniel Thalmann
Docteur ès Sciences
Université de Montréal
C.P. 6128
Montréal, Québec

RESUME

L'évolution des langages de programmation a été longtemps limitée par les contraintes dues aux machines sur lesquelles ces langages devaient être implantés. Aujourd'hui, grâce d'une part au développement technologique, d'autre part à l'apport méthodologique de la programmation structurée, les langages évoluent dans le sens d'une meilleure relation entre l'homme et la machine. Dans cet article, on analyse une évolution parallèle dans trois cas précis:

1. Les structures de contrôle du langage PASCAL face aux sauts et étiquettes du FORTRAN.

2. Les bases de données relationnelles face aux bases de données traditionnelles (CODASYL, IMS)

3. Les langages basés sur des types graphiques face aux logiciels graphiques traditionnels.

ABSTRACT

The evolution of programming languages has been long limited by the constraints imposed by the machines on which those languages were to be used. Today, with technological improvements and a new methodology of structured programming, languages are evoluting towards a better relation between man and machine. This article analyses a parallel evolution in three cases:

1. The structures of control of PASCAL language vs jumps and tags of FORTRAN

2. The relational data bases vs traditional data bases (CODASYL, IMS)

3. Languages based on graphic types vs traditional graphic programs

1. INTRODUCTION

La communication homme-ordinateur a commencé par une adaptation de l'homme à la machine, par le moyen du langage binaire, auquel succéda un langage plus humain, mais toujours proche du matériel: l'assembleur. Considérons en assembleur Nova (Data General, 1974) l'exemple suivant: "Combien de termes, faut-il prendre dans la somme des carrés des entiers pour que celle-ci reste inférieure à une valeur donnée?"

```
A:  ISZ     N          ;  N <- N + 1
    SUB     0,0        ;  effacement du registre AC0
    LDA     1,N        ;  chargement du N dans le registre AC1
    MOV     1,2        ;  copie du registre AC1 dans AC2
    DOCP    2,MDV      ;  multiplication (AC1)×(AC2) + (AC0) -> AC1
    JMP     .+1        ;  temporisation
    LDA     2,SOM      ;  chargement de SOM dans le registre AC2
    ADD     1,2        ;  (AC1) + (AC2) -> AC2
    STA     2,SOM      ;  rangement du contenu de AC2 à l'adresse SOM
    LDA     0,MAX      ;  chargement de MAX dans le registre AC0
    SUBZ#   2,0,SZC    ;  saut d'une instruction si (AC0) < (AC2)
    JMP     A          ;  saut à A (boucle)
    DSZ     N          ;  N <- N - 1
    JSR     IMP        ;  appel à un sous-programme d'impression
    N
```

Si cet assembleur n'est pas ancien, il a un certain intérêt, car on remarque la difficulté d'effectuer une simple multiplication qui est en fait une opération d'entrée-sortie faisant intervenir des registres bien définis et nécessitant l'introduction d'une instruction de temporisation, pour s'assurer que la multiplication a bien eu le temps de se terminer. De telles considérations sortent complètement du cadre de l'algorithme proprement dit.

2. LES STRUCTURES DE CONTROLE DU LANGAGE PASCAL FACE AUX SAUTS ET ETIQUETTES DU FORTRAN.

Le FORTRAN semble, à première vue, être un langage beaucoup plus près de l'homme que de la machine. Pourtant, seule sa forme est effectivement proche d'un langage évolué, la structure logique étant semblable à celle de l'assembleur. Par exemple, l'instruction GOTO est synonyme de l'instruction JUMP rencontrée dans les assembleurs de tous les ordinateurs, ce qui correspond à changer la valeur du compteur d'instructions. Pourtant, dans le FORTRAN, la notion explicite d'adresses et de registres a disparu. Aussi, l'exemple donné en Assembleur Nova devient en FORTRAN.

```
      INTEGER SOMME, SOMMAX
      DATA SOMMAX/.../
      N = 0
1     N = N + 1
      SOMME = SOMME + N*N
      IF(SOMME .LE. SOMMAX)GOTO 1
      K = N - 1
      WRITE(6,100)K
100   FORMAT(I5)
      STOP
      END
```

Dans ce programme, l'algorithme apparaît beaucoup plus clairement, mais l'introduction du test et de l'instruction GOTO sont artificielles. En outre, le résultat cherché N - 1 ne peut pas être imprimé directement et nécessite l'utilisation d'une variable auxiliaire, ce qui alourdit le programme. Actuellement, les techniques d'analyse syntaxique telles que la descente récursive, utilisée dans le compilateur PASCAL (Ammann, 1974) montrent très clairement qu'il est tout aussi aisé de permettre directement l'écriture du résultat d'expressions.

Dans notre langage habituel, on utilise certaines constructions logiques de pensée. Par exemple, si une condition se réalise, on exécute un certain nombre d'actions, sinon on en fait d'autres.

Cette manière de penser est exprimée respectivement en langage assembleur et en FORTRAN de la manière suivante:

```
      TST    C                IF(NOT.C)GOTO n1
      JMP    E1                    S1
         S1                    GOTO n2
      JMP    E2             n1     S2
E1      S2                   n2
E2
```

A la fin des années 50, on était déjà conscient de l'artificialité de cette formulation et c'est ainsi que dans le langage ALGOL 60, on a introduit la construction if C then S1 else S2. Ce n'est que vers la fin des années 60 que la polémique contre le goto (Dijkstra, 1968, Knuth 1974) a été soulevée et que naissaient les concepts de la programmation structurée (Dijkstra, 1972, Hoare, 1972). Ceux-ci ont été souvent présentés comme le moyen de programmer avec moins de risques d'erreurs, mais ont permis finalement de programmer d'une manière plus proche de l'homme et, par conséquent, ont diminué le nombre d'erreurs. On a pris conscience qu'un programme doit être clair et sûr avant d'être efficace. De plus, c'est le logiciel qui est devenu coûteux alors que le matériel baisse constamment de prix. Ces concepts ont conduit à la définition et à l'implantation de nouveaux langages. Parmi ces langages, le langage PASCAL (Wirth, 1972) offre des constructions très proches de la pensée humaine. Pour montrer ceci, nous allons reprendre l'exemple traité respectivement en assembleur Nova et en Fortran:

```
somme: = 0; n: = 0;
repeat n: = n+1; somme: = somme + n*n
until somme > sommemax;
write(n-1);
```

On remarque que la construction PASCAL repeat...until est très proche du langage naturel.

3. LES BASES DE DONNEES RELATIONNELLES FACE AUX BASES DE DONNEES TRADITIONNELLES

Un autre domaine dans lequel la relation homme-machine est très importante est la création et l'interrogation de bases de données. Un des objectifs majeurs dans la conception des systèmes de bases de données est l'indépendance des données. Les programmeurs ont été trop souvent confrontés à des problèmes relevant du matériel sur lequel ils opéraient ou liés au système d'exploitation gouvernant leur machine. L'indépendance des données, c'est l'indépendance vis-à-vis de l'implantation, elle protège l'humain de descriptions liées à la technologie et améliore ainsi la relation homme-machine.

L'indépendance des données peut être vue sous 3 aspects:

1. L'indépendance des données proprement dite. Le programmeur ne doit pas avoir à mentionner la taille des champs qu'il utilise en termes de mots-machine ou d'octets. Il ne doit pas non plus manipuler explicitement des valeurs de pointeurs.

2. L'indépendance des structures. Le programmeur ne doit pas avoir accès aux liens entre les enregistrements et les fichiers qui constituent la base de données.

3. L'indépendance des programmes. Cet aspect est, en principe, garanti, lorsque les 2 précédents le sont. En effet, c'est seulement dans des systèmes où les liens sont tout-à-fait visibles, que le risque d'accéder par programme à des informations non autorisées est le plus grand. En effet, l'utilisateur a à sa disposition des connaissances sur l'organisation de l'information qui peuvent lui permettre de découvrir de nouveaux moyens d'accès.

Le programmeur ne doit pas être concerné par des objets autres que les éléments qu'il désire manipuler; les types d'accès à l'information, les méthodes de stockage ne doivent pas lui être apparents.

Les systèmes de bases de données traditionnels, apparentés aux modèles de données hiérarchisé et réseau répondent mal aux critères d'indépendance des données.

Considérons IMS (I.B.M., s.d.), le système de bases de données hiérarchisé le plus connu, une description d'enregistrement a la forme:

```
DBD   NAME = personne, ACCESS = HDAM,RMNAME = ...
                        ...
SEGM  NAME = personne, BYTES = 154, PARENT = 0, FREQ = 100
FIELD NAME = nom, BYTES = 30, START = 1, TYPE = 0
FIELD NAME = âge, BYTES = 4, START = 31, TYPE = P
                        ...
```

Il est tout-à-fait regrettable, qu'on doive spécifier le type d'organisation des fichiers, HDAM par exemple pour des fichiers à accès direct. Les informations de longueur (BYTES) d'enregistrements et de champs sont inacceptables, de plus, il est incompréhensible que le programmeur soit obligé de spécifier une adresse-octet (START) pour chaque champ d'un enregistrement.

Dans le langage de manipulation, la structure hiérarchique est beaucoup trop apparente et une instruction de type GET-NEXT fait trop nettement apparaître l'organisation de l'information.

Le système de base de données réseau le plus connu est certainement CODASYL (Data Base Task Group, 1971). Dans le langage de description, il apparaît un certain nombre de déclarations complètement liées à l'accès aux données; de plus ces déclarations ont des répercussions au niveau du langage de manipulation (Engles, 1971). Par exemple, l'ordre FIND utilise le mode d'adressage spécifié dans le langage de description par:

```
LOCATION MODE IS { DIRECT pointeur
                 { CALC [clé]
                 {     USING attribut-1 [,attribut-2...]
                 {     DUPLICATES ARE [NOT] ALLOWED
                 { VIA lien SET
```

Une telle déclaration fait apparaître au programmeur les notions de pointeurs, de transformations de clés et de liens par des SET. La notion de SET, base du système CODASYL, qui permet de définir une relation binaire entre 2 enregistrements contient des descriptions que le programmeur ne devrait pas avoir à définir, par exemple:

```
MODE IS { CHAIN [LINKED] TO PRIOR
        { POINTER-ARRAY [DYNAMIC]
        { autre-implantation
```

La spécification "autre-implantation" montre bien que le programmeur peut être confronté à une implantation propre à la machine qu'il utilise.

Face à ces systèmes traditionnels, Codd (1970) a proposé un nouveau modèle, le modèle relationnel. Dans un système basé sur un tel modèle, la description des données se limite à la déclaration des relations représentées par des tableaux et les liens entre relations se font par l'intermédiaire de constituants communs. Une telle démarche est très proche de notre manière de penser. Prenons un exemple, avant de fixer un rendez-vous le 15 juin, on regarde dans un calendrier quel jour est le 15 juin; une fois qu'on a découvert que c'est un mercredi, on cherche son emploi du temps le mercredi. On a passé d'une relation CALENDRIER(DATE,JOUR-DE-SEMAINE) à une relation EMPLOI-TEMPS(JOUR-DE-SEMAINE,OCCUPATION), par l'intermédiaire du jour de la semaine mercredi.

La description des relations et les requètes ne font intervenir aucune caractéristique d'implantation. Considérons un exemple exprimé dans un langage

développé par les auteurs:

relations élève (nom, prénom: alpha 15; num-classe:entier) représenté par el;

classe (num-classe;nom-prof:alpha 15) représenté par c;

Si l'on désire imprimer la liste des élèves de Monsieur Martin, la requête s'écrira:

avec c, el sélectionner tous si c.nom-prof = "MARTIN" et c.num-classe= el.num-classe alors imprimer el. nom,el.prénom;

On remarque la simplicité d'une telle requête.

4. LES LANGAGES BASES SUR DES TYPES GRAPHIQUES FACE AUX LOGICIELS GRAPHIQUES TRADITIONNELS

Aujourd'hui, on ne trouve guère de documents, livres et autres moyens de communication qui n'essaient pas d'expliquer une idée, un phénomène, sans le moyen de l'image. Les informaticiens ont aussi voulu développer pour l'ordinateur ce moyen d'expression, en particulier, pour des résultats numériques qu'on peut exprimer sous forme de courbes. Déjà dès 1956, le système ORACLE (Fike, 1959) était développé au laboratoire d'Oak Ridge. Ce système était composé d'un tube cathodique dont une des faces était exposée à un film; ceci pour tracer des courbes et générer des caractères. Seules 4 instructions contrôlaient ce système:

i) Placer la coordonnée Y
ii) Placer la coordonnée X et allumer le point
iii) Placer la coordonnée X et préparer le dessin d'un caractère
iv) Avancer le film

Avec ces 4 instructions, les auteurs d'ORACLE envisagaient déjà de multiples applications de traçage de courbes et des histogrammes. Ceci est resté pendant plusieurs années à ce stade. Avec l'avènement du FORTRAN, on a commencé par piloter des traceurs graphiques par programmation sous forme d'instructions simples commandant le traçage de points et le lever de plume. Cette manière de traiter est très proche du matériel utilisé.

En FORTRAN, des ensembles de sous-programmes ont été développés. Ces logiciels graphiques ne traitent qu'une variable simple, la coordonnée et qu'une variable structurée, le tableau de coordonnées.

Le logiciel de base se réduit généralement à 3 sous-programmes:

SUBROUTINE LINE(X,Y,N)	relie les N points de coordonnées X(I), Y(I)
SUBROUTINE POINT(X,Y,N)	trace les N points de coordonnées X(I), Y(I)

SUBROUTINE TEXT(XT,YT,T) écrit le texte T au point de coordonnées (XT,YT)

Comme on le voit, la seule évolution est l'utilisation de tableaux de coordonnées. On reste très près des premiers outils graphiques tels que ORACLE. Pourtant des logiciels graphiques beaucoup plus élaborés ont été développés et permettent le tracé de courbes planes, de courbes de niveaux (Morse, 1969) et de représentations dans l'espace (Thalmann, 1977).

Dans la vie courante, on parle de la forme des objets, mais on les voit rarement comme un ensemble de points. L'évolution des langages graphiques (Shapiro, 1975), s'est donc faite en concevant directement l'objet.

Il nous a semblé intéressant de pouvoir définir les formes géométriques naturelles dans un langage structuré tel que PASCAL. Dans ce langage, on a introduit un nouveau type structuré, la figure. Par exemple, dans cette extension graphique de PASCAL, qui comprend le type de base vector et les opérations associées, un type cercle peut être défini comme:

```
type  cercle = figure(centre:vector; r:real);
      var  pas, theta:real; x1,x2:vector;
      begin theta: = 0.0; x1: = <<r, 0.0>> + centre; pas: =0.1;
      while theta < 2.0 * pi do
            begin theta: = theta + pas;
            x2: = << r * cos(theta), r * sin(theta) >> + centre;
            connect(x1,x2); x1: = x2
            end
      end;
```

Des variables de type cercle peuvent alors être déclarées; une instruction create permet la création dynamique de la figure en mémoire. Par exemple, un cercle centré au point <3,4> et de rayon 5 sera créé ainsi:

```
var c: cercle;
begin create c(<<3.0,4.0>>,5.0); ...
```

Un type figure peut être utilisé pour construire d'autres types graphiques. Par exemple, un type concentrique qui sera une série de n cercles de même centre. Ces types peuvent être utilisés comme d'autres types PASCAL. On peut avoir des tableaux de cercles et des procédures dont les paramètres sont des cercles. Cette manière structurée de programmer des applications graphiques correspond beaucoup mieux à la représentation des objets dans notre esprit. Par exemple, le cercle ici existe en tant qu'entité.

5. CONCLUSION

Nous avons analysé l'évolution des langages de programmation vers un rapprochement de l'expression humaine. Bien que les progrès soient manifestes,

on peut souhaiter une communication orale systématique entre l'homme et la machine. Même si le langage à disposition était restreint, l'être humain pourrait s'exprimer plus aisément que dans une forme écrite. Nous pensons que l'évolution des langages va se faire dans ce sens lors des prochaines décennies.

REFERENCES BIBLIOGRAPHIQUES

1. Ammann, U. "The Method of Structured Programming Applied to the Development of a Compiler", Intern. Comp. Symp. 73, ACM, Amsterdam, North-Holland, 1974, pp. 93-99.

2. Codd, E.F. "A Relational Model of Data for Large Shared Data Banks", Comm. ACM, 13(1970)6, pp. 377-387.

3. Data General Corporation. "How to Use the Nova Computers", Southboro, DGC, 1974.

4. Data Base Task Group "Report of the CODASYL Data Base Task Group", ACM, N.Y., 1971.

5. Dijkstra, E.W. "Goto Statement considered harmful", Comm. ACM, 11(1968)3, pp. 147-148.

6. Dijkstra, E.W. "Notes on Structured Programming", N.Y., Academic Press Inc, 1972.

7. Engles, R.W. "An Analysis of the April 1971 Data Base Task Group", Proc. 1971 ACM-SIGFIDET Workshop, ACM, 1971, pp. 69-91.

8. Fike, C.T. "Oracle Curve Plotter", Comm. ACM, 2(1959)10, pp. 38-39.

9. Hoare, C.A.R. "Notes on Data Structuring", Academic Press Inc, 1972.

10. IBM "IMS/360 Applications Description Manual", White Plains, N.Y., GH-20-0765.

11. Knuth, D.E. "Structured Programming with Goto Statements", Comp. Surveys, 6(1974)4, pp. 261-301.

12. Morse, S.P. "Concepts of Use in Contour Map Processing", Comm. ACM 12(1969)3, pp. 147-152.

13. Shapiro, L.G. "ESP^3: A High-level Graphics Language", Comp. Graphics, 9(1975)1, pp. 70-77.

14. Thalmann, N. "A New Computer Program for Generating Three-Dimensional Plots of Electronic Densities and Related Contour Levels", Chimia 31(1977)9, pp. 361-362.

15. Wirth, N. "The Programming Language PASCAL", Acta Informatica, 1(1972)4, pp. 241-259.