Quelques concepts importants des langages de programmation modernes, et leur expression en SIMULA 67

B. MEYER *

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

Cet article a fait l'objet d'une communication aux journées de travail sur le thème "Panorama des Langages d'aujourd'hui" organisées par le groupe GROPLAN (Programmation et Langages) de l'AFCET, à Cargèse (Corse), du 14 au 22 mai 1979. Actes dans le Bulletin GROPLAN, numéros 8 et 9.

E.D.F. – BULLETIN DE LA DIRECTION DES ETUDES ET RECHERCHES
SERIE C – MATHEMATIQUES, INFORMATIQUE n° 1, 1980, pp. 89-150.

I - INTRODUCTION

Il est devenu presque de règle, lorsqu'on discute quelques-unes des principales idées actuelles sur la conception des langages de programmation, et en particulier tout ce qui tourne autour du concept d'"abstraction de données", de citer au passage un langage précurseur, SIMULA 67, et d'inclure en bibliographie le document de base [4]. SIMULA mérite mieux qu'une mention incantatoire : il s'agit d'un langage de programmation actif, pratiqué dans le monde par de nombreux utilisateurs, mis en oeuvre sur un grand nombre de systèmes(1) et dans certains cas très efficacement, convenablement normalisé, et pouvant prétendre à la qualité d'un langage "industriel".

Le but du présent article est d'étudier de façon critique les techniques introduites par SIMULA pour résoudre certaines des questions que se posent aujourd'hui les concepteurs de langages, en particulier dans cinq domaines : modularité et types abstraits ; composition descendante ; conception de modules d'application (et de langages) spécialisés ; généricité ; quasi-parallélisme et coprogrammes.

Les spécialistes de SIMULA pourront trouver que certaines caractéristiques du langage ne sont pas présentées ici dans toute leur richesse : précisément, le but poursuivi ici n'est pas d'explorer toutes les finesses de SIMULA mais, au contraire, de chercher à en retenir les constructions simples qui peuvent aider à mettre en pratique quelques concepts méthodologiques importants. Il y a en fait deux écueils - entre autres - dans la discussion d'un langage de programmation. L'un consiste à prêter une attention exagérée aux finesses ; l'autre, opposé, à négliger l'importance des constructions linguistiques en remarquant qu'on peut tout faire dans n'importe quelle notation théoriquement assez puissante : FORTRAN, BASIC, la machine de Turing, etc.

(1) CDC 6600/7600, CYBER 70, IBM 360/370, UNIVAC 1108/1110, DEC-10, C11 10070/IRIS 80,....

Pour éviter de tomber dans le "fossé de Turing", il convient de garder présent à l'esprit que le choix d'un langage de programmation, ou plutôt d'un système de programmation résulte d'un compromis entre :

- d'une part, la facilité d'expression, c'est-à-dire (l'inverse de) la quantité de contorsions nécessaire pour faire passer dans le langage les structures conceptuelles qui déterminent l'organisation d'un programme ;

- et d'autre part, tous les aspects pratiques qui font la différence entre les systèmes de qualité "industrielle" et les autres : compilation séparée ; possibilité d'appeler des sous-programmes écrits dans d'autres langages ; possibilité de constituer des bibliothèques de modules d'application ; outils d'aide à la mise au point ; entrées et sorties réalistes ; possibilités de manipulation de fichiers, accès direct.

Nous reviendrons en conclusion sur ces critères. Il nous paraît cependant important de garder présent à l'esprit, dans la discussion ci-après, où l'on considère l'application à SIMULA de concepts qui font leur entrée en force dans une pléiade de propositions récentes (GREEN-ADA [9], CLU [14], ALPHARD [21], MEFIA [20], EUCLID [13], MESA [7], CSP [8], etc.), que SIMULA correspond à un ensemble de systèmes solides et utilisables dès aujourd'hui.

II - DONNEES DE BASE

SIMULA 67, faisant suite à un langage de simulation appelé SIMULA tout court [3], a été conçu en 1967 par Dahl et Nygaard au Centre de Calcul Norvégien (NCC) d'Oslo. Il s'agit en fait d'un langage de programmation tout à fait général, dans lequel la simulation n'est qu'une application possible ; plus précisément, la simulation est traitée par un "module" prédéfini, mais de même nature que tous ceux que l'on peut créer à volonté, grâce aux propriétés du langage en ce domaine, que nous aborderons au paragraphe IV.4. L'erreur historique des promoteurs du langage a sans doute été de ne pas changer son nom en conséquence.

Le document de référence officiel est [4], complété par des guides propres aux versions mises en oeuvre sur différentes machine. L'introduction la plus lisible est l'ouvrage "SIMULA BEGIN"[1] .

SIMULA 67 est une extension, presqu'entièrement "compatible vers le haut[(1)]", d'ALGOL 60. Nous donnons ici rapidement, pour le lecteur ne connaissant pas ALGOL 60, les quelques concepts algoliques de base nécessaires à la compréhension de la suite.

Structure des programmes

La notion fondamentale d'ALGOL 60 est celle de bloc. Un bloc a la forme

begin
déclaration 1;
déclaration 2;
.....
déclaration m;
} peuvent être absentes
instruction 1;
instruction 2;
⋮
instruction n
end

(1) différences essentielles : pas de déclarations *own (rémanent)* ; le passage par nom n'est pas le mode par défaut.

où les *"instructions"* peuvent utiliser les variables ou tableaux définis dans les *"déclarations"*, s'il y en a.

Un programme est un bloc de cette forme. Une instruction peut être une instruction de base (affectation par exemple), mais aussi un bloc, c'est-à-dire que les blocs peuvent être imbriqués les uns dans les autres. Une instruction d'un bloc interne peut utiliser des objets déclarés dans un bloc englobant.

Instructions de base et structures de contrôle

L'affectation est notée

variable := expression

(utilisateurs de FORTRAN, notez l'emploi de := plutôt que = utilisé comme opérateur d'égalité, *.EQ.* en FORTRAN).

L'instruction conditionnelle s'écrit

if condition then
instruction 1
else
instruction 2

où l'un des deux *instructions* sera exécutée selon que la *condition* est vraie ou fausse. Notez que grâce au mécanisme des blocs ces instructions peuvent être aussi complexes qu'on le désire. La partie *else instruction 2* peut être absente.

Les boucles s'écrivent

for i:= a step b until c do
instruction

(boucle avec compteur), ou

while condition do
instruction

(exécution de *instruction* tant que *condition* est vraie, donc peut-être jamais).

Procédures

Les sous-programmes s'appellent en ALGOL procédures. Une procédure sans résultat (*subroutine* en FORTRAN) est déclarée comme

type des arguments

procedure p (arg_1,....arg_n) ; *integer* arg_1;.....;

instruction (en général un bloc)

Une procédure à résultat, par exemple réel (*fonction* en FORTRAN), est déclarée comme :

real procedure q (arg_1,...arg_n) ;.....;

instruction, où l'on affecte à q
(en général un bloc)

Dans les deux cas, il s'agit de déclarations pouvant être imbriquées comme les autres : une procédure est donc appelable dans le bloc où elle est déclarée et dans les blocs internes ; elle peut être déclarée locale à une autre procédure.

Pour appeler une procédure, on écrit simplement son nom suivi d'une liste d'arguments réels, s'il y a lieu (une procédure peut ne pas avoir d'arguments) :

$p(a_1, ...a_n)$;
$r := q(b_1,.....b_n)$

Les procédures peuvent être récursives :

procédure s(.....) ; .....
begin
.....
.....;s(.....);..
end

Déclarations et allocation dynamique

Tout objet utilisé dans une instruction doit être déclaré dans le même bloc ou un bloc englobant. Une déclaration de variable a la forme

```
integer x,y,z ;
real t ;
etc.
```

Une déclaration de tableau a la forme

```
boolean array b(0:5)
real array a(1:m, 1:n);
etc.
```

Dans le second cas, m et n doivent avoir été déclarées et initialisées dans un bloc englobant. A chaque nouvelle exécution du bloc dans lequel a est déclaré, il aura une nouvelle taille dépendant de m et n. Les valeurs précédentes seront perdues.

Manipulation de textes, entrées, sorties

SIMULA ajoute à ALGOL des mécanismes puissants, et particulièrement agréables à utiliser, pour :

- la manipulation de chaînes de caractères,
- les entrées et sorties.

Nous renvoyons à la note Atelier logiciel n°16 pour la description de ces propriétés.

III - MODULARITE ET REPRESENTATION DE TYPES ABSTRAITS

III.1 - Une vue systémique de la programmation

Nous commencerons notre revue de quelques-uns des grands concepts actuels par deux problèmes aujourd'hui reconnus comme étroitement liés : celui de la modularité et celui des types de données.

La difficulté majeure de la conception d'un programme d'une certaine taille tient à la méthode choisie pour le diviser en entités homogènes ou "modules". C'est elle qui conditionne en effet l'organisation du projet, la répartition éventuelle du travail, le succès ou l'échec de la phase d'intégration des différents éléments, et la faculté d'adaptation du produit final à des modifications de ses spécifications.

La méthode traditionnelle de division d'un programme en "sous-programmes", fondée sous une décomposition du traitement à effectuer, ne répond pas entièrement à ces exigences. Plusieurs travaux, en particulier ceux de Parnas [17], [18], [19] ont montré qu'on obtenait une décomposition plus solide en se fondant plutôt sur le critère dual du précédent, celui des structures de données, ou types, intervenant dans le système. On peut brièvement justifier ce principe en remarquant qu'un programme est un modèle d'un certain système physique, et qu'une structure satisfaisante pour un tel modèle est celle qui décrit le système en termes d'objets et de relations entre ces objets ; les objets en question sont, soit primitifs (types de base), soit eux-mêmes complexes (sous-systèmes). Dans l'évolution d'un tel modèle, on peut penser que les objets sont un élément plus stable que les relations, et qu'il est donc naturel de fonder sur eux la structure du programme. C'est ce qu'on peut appeler une conception "systémique" de la programmation.

Un "objet" intervenant dans une telle décomposition est un ensemble de données qui doit pouvoir être caractérisé par des propriétés purement externes, et indépendamment des problèmes de représentation. L'exemple le plus fréquemment cité est celui d'une pile d'entiers, qui peut être caractérisée par un ensemble d'opérateurs possédant certaines propriétés formelles (exemple 1). La classe des objets vérifiant ces propriétés est appelée un type abstrait dont l'exemple 1 donne la spécification fonctionnelle [15].

Fonctions

pilevide : *PILE* → *BOOLEEN*
empiler : *ENTIER* x *PILE* → *PILE*
dépiler : *PILE* → *ENTIER* x *PILE*
— *fonction partielle*

Assertions

pour tout p : PILE, e : ENTIER alors

| ∿ *pilevide (empiler (e,p))* *et*
| *dépiler (empiler (e,p))* = *(e,p)* *et*
[∿ *pilevide (p)* => *empiler (dépiler (p))* = *p*]

fin

Exemple 1 : Spécification d'une pile d'entiers

Un grand nombre de langages de programmation récents cherchent à fournir ce mode de décomposition en proposant une structure linguistique permettant de regrouper la description d'une structure de données et de tous les opérateurs permettant de la manipuler, c'est-à-dire la mise en oeuvre d'un type abstrait. Ce mécanisme est fourni sous le nom de "grappe" en CLU, "forme" en ALPHARD, "module" en EUCLID,"package" en ADA, "enveloppe" en PASCAL PLUS, etc.

En reprenant l'exemple 1, ainsi, un module de gestion de pile conforme à la spécification précédente devra offrir aux autres modules la possibilité de manipuler une ou plusieurs piles grâce aux opérateurs *dépiler*, *empiler*, *pilevide*.

Lorsqu'on inclut une telle possibilité dans un langage de programmation, plusieurs choix doivent être faits :

- Les types peuvent être "statiques" ou "dynamiques". Dans le premier cas, un module décrit un exemplaire de la structure de données associée ;

dans le second, il ne représente aucun objet concret, mais seulement un modèle de la structure de données, dont les utilisateurs peuvent alors créer à volonté de nouveaux exemplaires (par des opérateurs de création, *créerpile* dans notre exemple, qui doivent être ajoutés à la spécification abstraite).

- Certains langages permettent d'accéder de l'extérieur aux données internes d'un module. On peut au contraire n'autoriser que l'emploi des opérations de la spécification externe, en restreignant les "droits d'accès" à certains sous-programmes et à certaines données désignées comme "exportables". L'intérêt d'une telle restriction est de limiter la propagation des erreurs entre modules, et de faire en sorte qu'une modification de représentation interne dans un module (par exemple, le passage d'une représentation contiguë à une représentation chaînée pour une pile) soit sans effet sur les autres.

- On peut poursuivre plus loin encore l'objectif de protection, en permettant comme GIPSY [2] de définir des "droits d'accès" différents à un même module ; ou l'objectif de séparation entre spécification et représentation, en offrant comme ADA la compilation séparée de ces deux parties d'un module.

III.2 - Les classes en SIMULA

SIMULA fut le premier langage important à offrir une représentation de module correspondant aux définitions précédentes : la classe. Pour replacer cette notion par référence à la discussion ci-dessus, on peut définir une classe comme une représentation de type abstrait, dynamique, sans protection des données internes ni séparation physique entre la partie "spécification" et la partie "représentation".(1)

(1) Note : Les programmes présentés dans cet article n'ont pas été testés. Une version corrigée sera adressée sur demande.

Déclaration d'une classe

Une classe est définie en SIMULA par une déclaration de la forme

class nom_de_classe ; corps_de_classe

ou *class nom_de_classe (liste_de_paramètres_formels) ; corps_de_classe*

où le *corps_de_classe* est un bloc de la forme

begin

déclaration d'attributs { *variables* / *procédures* } ;

actions d'initialisation

end

ou, dans les cas dégénérés, une seule instruction d'initialisation (il n'y a pas alors d'attributs). Les "attributs" sont la représentation concrète des propriétés communes à tous les objets d'un type abstrait associé à la classe. Les "actions d'initialisation" sont destinées à être appliquées à tout exemplaire de la classe à sa création.

A titre d'illustration, on trouvera à l'exemple 2 la déclaration d'une classe réalisant une représentation concrète restreinte du type abstrait "pile d'entiers" du paragraphe précédent. On a choisi une représentation contiguë par un tableau, et ajouté à la spécification une restriction de taille représentée par le paramètre formel n. Les "attributs" sont le tableau p, l'indicateur *sommet*, et les procédures *dépiler, empiler, pilevide*. On a omis les procédures *erreur_pile_pleine* et *erreur_pile_vide*.

Objet dont le type est défini par une classe

Dans un programme incluant une telle déclaration de classe, on pourra déclarer des variables du type correspondant en écrivant :

ref (nom_de_classe) $o_1, o_2, \ldots, o_n$;

L'emploi du mot-clé *ref* est lié à la représentation physique d'une telle variable (pointeur), mais il est souvent utile conceptuellement

```
class pilentier (n) ; integer n ;
    begin
    comment attributs ;
       comment variables ;
            integer array p (1 : n) ;
            integer sommet ;
       comment procédures ;
            procedure empiler (x) ; integer x ;
                  if sommet = n then erreur_pile_pleine
                  else
                     begin sommet := sommet + 1 ;
                           p (sommet) := x
                     end empiler ;
            integer procedure dépiler ;
                  if pilevide then erreur_pile_vide
                  else
                     begin   dépiler := p (sommet) ;
                     sommet := sommet - 1 ;
                     end dépiler ;
            boolean procedure pilevide ;
                   pilevide := (sommet = 0) ;
    comment action d'initialisation (à la création d'une pile
                                     initialement vide) ;
       sommet := 0
    end pilentier
```

EXEMPLE 2 : PILE D'ENTIERS CONTIGUË BORNEE EN SIMULA

de considérer qu'il s'agit d'une simple règle syntaxique et qu'on manipule en fait des objets du nouveau type *nom_de_classe* de la même façon que des objets *integer*, *text* etc. Un tel type,défini par une classe, peut servir à déclarer non seulement des variables mais aussi, bien entendu, des paramètres formels ou des résultats de procédure :

```
procedure p (x,y,z) ; integer x ; ref (c) y,z ;

begin... corps de p... end p ;

ref (c) procedure q(...) ; ... ;
```

Un objet d'un tel type est à tout instant du déroulement d'un programme dans l'un des deux états suivants :

- vide. C'est l'état initial. On considère que sa valeur est alors celle de la constante spéciale (générique) *none*.

- créé. L'objet possède alors chacun des attributs (variables, procédures) définis dans la déclaration de classe.

Un objet o_1 de type *ref (nom_de_classe)* passe dans l'état "créé" par l'exécution d'une instruction

```
o1 :- new nom_de_classe
```

ou

```
o1 :- new nom_de_classe (liste_de_paramètres_réels)
```

selon que la déclaration de classe contenait ou non des paramètres. Par exemple :

```
ref (pilentier) X ;....;

X :- new pilentier (5000) ; ....
```

Comme nous l'avons annoncé, les types "classes" sont dynamiques en SIMULA, c'est-à-dire que l'évaluation d'une expression commençant

par *new* entraîne la création d'un nouvel exemplaire du type, et l'allocation de mémoire correspondante pour les attributs définis dans la déclaration de la classe ; ainsi, dans notre exemple, l'entier *sommet* et le tableau *p*, ici de 5000 éléments.

L'évaluation d'une "expression" *new*.... entraîne aussi l'exécution des actions d'intitialisation sur l'exemplaire nouvellement créé de la classe. Ici l'attribut *sommet* de *X* sera mis à zéro. On notera que les variables locales à une classe sont rémanentes.

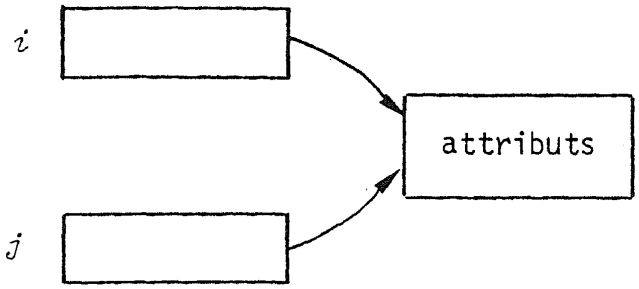
Opérations sur des objets de types définis par des classes

On aura noté le *:-* utilisé pour l'affectation à o_1 et *X* au lieu du *:=* traditionnel d'ALGOL qui est utilisé pour des objets de types classiques (ex. *i := 3*) mais serait illégal ici. De même, on emploiera comme opérateur de test de l'égalité == pour les objets de type *ref (nom_de_classe)* et = pour les autres. Ces différences de traitement sont dues à l'influence déjà mentionnée de la représentation physique des *ref* par des pointeurs : = et *:=* opèrent sur des valeurs, == et *:-* sur des adresses. On notera en particulier que la sémantique de ces opérations n'est pas la même, comme le montre la comparaison des exemples suivants :

```
integer i, j ;                          ref (c) i, j ;
.....;                                  .....;
comment ici prop (i) est vraie ;        comment ici prop (i) est vraie ;
j := i ;                                j :- i ;
mod (j) ;                               mod (j) ;
comment ici prop (i) est vraie ;        comment ici prop (i) peut être
                                                              fausse ;
```

où *mod (j)* est une opération ne pouvant modifier que son argument. Dans le premier cas toute propriété *prop (i)* portant sur *i* et sur des variables autres que *j* est invariante pour les deux instructions indiquées ; dans le

second, elle peut être rendue fausse si *mod (j)* modifie un attribut de *j*.



Cette différence de sémantique oblige à une certaine prudence lorsqu'on s'efforce de considérer en principe une déclaration de classe comme une définition d'un nouveau type(1).

Attributs et actions d'initialisation

Une fois créé par un *new*..., un objet de type *ref*(...) possède tous les attributs apparaissant dans la définition de classe ; au moment de l'évaluation du *new*..., les actions d'initialisation sont exécutées. Ainsi, dans

```
ref (pilentier) pil1  ;

pil1  : - new pilentier (100)
```

l'allocation de mémoire nécessaire au nouvel exemplaire *pil1* (en particulier pour le tableau *p*) est effectuée, et l'attribut *sommet* est mis à zéro.

Par la suite, on peut accéder à tout attribut, variable ou procédure, de l'objet grâce à une notation pointée

```
objet.attribut
```

semblable à celle de PL/1 ou PASCAL. A titre d'exemple, l'extrait de programme suivant créée et utilise une pile d'entiers :

(1) Pour le type *text*, mais pour celui-là seulement, les deux sémantiques coexistent : := est une recopie de caractères, :- une recopie de pointeur.

```
ref (pilentier) X ;
integer somme, i, j
integer array a[1 : 100] ;
.... initialisation de a ... ;
X :- new pilentier (1000) ; i := 1 ;
while i < 50 and a(i) > 10000  do
        X.empiler (a(i)) ;
..... ;
somme := 0 ;
while not X.pilevide do
      somme := somme + X.dépiler ;
.....
```

On notera qu'à l'intérieur d'une déclaration de classe, on désigne les attributs de l'"exemplaire courant" par leur simple nom : ainsi *sommet* dans la déclaration de *pilentier*. Par contre, on désignera un autre exemplaire de la même classe ou d'une autre par la notation pointée. Nous en aurons des exemples dans le paragraphe suivant, consacré à un exemple moins simple.

III.3 - Un exemple : les nombres complexes

Considérons le type abstrait "nombre complexe". Il est défini par la spécification de l'exemple 3 (page suivante) ; on a nommé les opérations par leur symbole mathématique, entouré d'un cercle ($\oplus$, $\ominus$ etc.) pour les distinguer des opérations sur les réels utilisées plus bas ; les fonctions d'accès x, y, ρ, θ sont la partie réelle, la partie imaginaire, le module et l'argument.

Si l'on cherche à réaliser un "module" représentant ce type, deux représentations s'offrent naturellement : la représentation cartésienne, où l'on conserve les parties réelles et imaginaires; la représentation polaire, où l'on conserve le module et l'argument. La première est appropriée aux opérations x, y, $\oplus$, $\ominus$, $\bar{}$, *cartésien* et rend les autres malaisées ;

fonctions

création

cartésien : *REEL* x *REEL* → *COMPLEXE*
(création d'un complexe à partir de ses parties réelle et imaginaire)

polaire : *REEL* x *REEL* → *COMPLEXE*
(à partir de son module et de son argument)

accès

x, y, ρ, θ : *COMPLEXE* → *REEL* x *REEL*

modification

$\oplus$, $\ominus$, $\otimes$, $\oslash$: *COMPLEXE* x *COMPLEXE* → *COMPLEXE*

$\overline{\ }$: *COMPLEXE* → *COMPLEXE* (conjugué)

$\equiv$: *COMPLEXE* *COMPLEXE* *BOOLEEN*

assertions

pour tout a, b, r, t : *REEL*, c, c' : *COMPLEXE* *alors*

cartésien $(x(c), y(c)) = c$ *et* *polaire* $(\rho(c), \theta(c)) = c$ *et*

x (*cartésien* $(a,b)) = a$ *et* y (*cartésien* $(a,b)) = b$ *et*

ρ (*polaire* $(r,t)) = r$ *et* θ(*polaire* $(r,t)) = t$ *et*

$\rho(c) = \sqrt{x(c)^2 + y(c)^2}$ *et* $\theta(c) =$ *si* $x(c) = 0$ *alors* $\pi/2$ *sinon* *arc tg* $(y(c)/x(c))$ *et*

$x(c) = \rho(c) \cos(\theta(c))$ *et* $y(c) = \rho(c) \sin(\theta(c))$ *et*

$x(c \oplus c') = x(c) + x(c')$ *et* $y(c \oplus c') = y(c) + y(c')$ *et*

$x(c \ominus c') = x(c) - x(c')$ *et* $y(c \ominus c') = y(c) - y(c')$ *et*

$\rho(c \otimes c') = \rho(c) \times \rho(c')$ *et* $\theta(c \otimes c') = \theta(c) + \theta(c')$ *et*

$\rho(c \oslash c') = \rho(c) / \rho(c')$ *et* $\theta(c \oslash c') = \theta(c) - \theta(c')$ *et*

$x(\overline{c}) = x(c)$ *et* $y(\overline{c}) = - y(c)$ *et*

$\rho(\overline{c}) = \rho(c)$ *et* $\theta(\overline{c}) = - \theta(c)$ *et*

$c \equiv c' \Leftrightarrow (x(c) = x(c')$ *et* $y(c) = y(c'))$

fin

EXEMPLE 3 : LE TYPE ABSTRAIT *COMPLEXE* - SPECIFICATION FONCTIONNELLE

la seconde est appropriée aux opérations ρ, θ, x, (/), ¯, *polaire*.

Face à ce dilemme, le mieux est de ne pas choisir a priori, mais de passer d'une représentation à l'autre au gré des opérations demandées. Nous utiliserons deux variables locales à la classe, soit *cart_rep* et *pol_rep*, indiquant respectivement si les représentations cartésienne et polaire sont disponibles, et des variables *x_rep*, *y_rep*, *ro_rep*, *theta_rep* susceptibles de représenter partie réelle, partie imaginaire, module et argument. La classe doit être écrite de telle façon que la création d'un nouvel objet établisse et que chaque chaque appel de procédure maintienne l'invariant de représentation suivant :

(cart_rep ou pol_rep) (l'une des des deux au moins est bonne)

et *(cart_rep ⇒ x_rep et y_rep significatifs)*

et *(pol_rep ⇒ ro_rep et theta_rep significatifs)*.

La condition sine qua non de validité de ce module est qu'on l'emploie seulement à travers les procédures *cartésien*, *polaire*, *x*, *y*, *ro*, *theta*, *conjugué*, et jamais en modifiant les variables de représentation *x_rep*, *y_rep*, *ro_rep*, *theta_rep*, *cart_rep et pol_rep* ni en utilisant d'autres procédures internes (*cart_calcul* et *pol_calcul* ci-après).

Si cette condition est respectée, le module décrit à l'exemple 4, donné aux pages suivantes, répondra à la question.

```
class complexe ;
    comment représentation ;
        boolean cart_rep, pol_rep ;
        real x_rep, y_rep, ro_rep, theta_rep ;
        procedure cart_calcul ;
            comment procédure à usage exclusivement interne calcu-
                    lant s'il y a lieu la représentation carté-
                    sienne, l'autre étant supposée connue ;
            if not cart_rep then
               begin
               x_rep := ro_rep * cos(theta_rep) ;
               y_rep := ro_rep * sin(theta_rep) ;
               cart_rep := true
               end cart_rep ;
        procedure pol_calcul ;
            comment procédure à usage exclusivement interne calcu-
                    lant s'il y a lieu la représentation polaire,
                    l'autre étant supposée connue ;
            if not pol_rep then
               begin
               ro_rep := sqrt (x_rep ** 2 + y_rep ** 2) ;
               theta_rep := if x_rep = 0 then pi/2
                            else arctg (y_rep/x_rep) ;
               pol_rep := true
               end pol_calcul ;
    comment procédures (exportées) de création ;
        procedure cartésien (a,b) ; real a,b ;
            begin
            x_rep := a ; y_rep := b ;
            cart_rep := true ; pol_rep := false
            end cartésien ;
        procedure polaire (r,t) ; real r,t ;
            begin
            ro_rep := r ; theta_rep := t ;
            pol_rep := true ; cart_rep := false
            end polaire ;
```

EXEMPLE 4 (DEBUT) : MODULE "COMPLEXE" EN SIMULA

```
comment procédures (exportées) d'accès ;
        real procedure x ;
                begin
                cart_calcul ; x :=  x_rep
                end x ;
        real procedure y ;
                begin
                cart_calcul ; y  := y_rep
                end y ;
        real procedure ro ;
                begin
                pol_calcul ; ro :=  ro_rep
                end ro ;
        real procedure theta ;
                begin
                pol_calcul ; theta  := theta_rep
                end theta :


      boolean procedure égal_complexe (c) ; ref (complexe) c ;
              égal_complexe := if  cart_rep  then
                                    (x_rep = c.x  and  y_rep = c.y)
                               else (ro = c.ro  and  theta = c.theta) ;
```

EXEMPLE 4 (SUITE) : MODULE "COMPLEXE" EN SIMULA

```
comment procédures (exportées) de modification ;

    ref (complexe) procedure plus (c) ; ref (complexe) c ;
            begin ref (complexe) p ;
            p :- new complexe ; p.polaire (x+c.x, y+c.y) ;
            plus :- p
            end plus ;

    ref (complexe) procedure moins (c) ; ref (complexe) c ;
            begin ref (complexe) m ;
            m :- new complexe ; m.cartésien (x-c.x, y-c.y) ;
            moins :- m
            end moins ;

    ref (complexe) procedure mult (c) ; ref (complexe) c ;
            begin ref (complexe) m ;
            m :- new complexe ; m.polaire (ro * c.ro, theta+c.theta) ;
            mult :- m
            end mult ;

    ref (complexe) procedure div (c) ; ref (complexe) c ;
            begin ref (complexe) d ;
            d :- new complexe ; d.polaire (ro/c.ro, theta-c.theta) ;
            div :- d
            end div ;

    ref (complexe) procedure conj ;
            begin ref (complexe) cj ;
            cj :- new complexe ;
            if cart_rep then cj.cartésien (x_rep, y_rep)
            else cj.polaire (ro_rep, theta_rep) ;
            conj :- cj
            end conj ;

comment actions d'initialisation ;

    x_rep := y_rep := ro_rep := theta_rep := 0 ;
    cart_rep := pol_rep := true

end classe complexe
```

EXEMPLE 4 (FIN) : MODULE "COMPLEXE" EN SIMULA

Un certain nombre de remarques s'imposent sur cet exemple.

Remarque 4.1 On aurait pu éviter la nécessité pour l'utilisateur d'une double initialisation par *c :- new complexe*, puis *c.cartésien (...)* ou *c.polaire (...)* en donnant trois paramètres à la classe (représentation, réel 1, réel 2).

Remarque 4.2 On a pris le parti de calculer systématiquement la représentation cartésienne (resp. polaire) dès que x ou y (resp. ρ ou θ) est demandé.

Remarque 4.3 Une des contraintes des types représentés par des classes en SIMULA apparaît bien sur cet exemple : dans une déclaration de classe, on "parle" toujours d'un certain exemplaire courant de la classe. C'est pourquoi les procédures *plus, moins, mult, div* n'ont qu'un seul paramètre (et *conjugué* aucun) : l'autre est implicite ; c'est à ses attributs que se réfère une écriture comme x dans $x + c.x$ (procédure *plus*), $c.x$ désignant le même attribut propre à l'argument c de la procédure. Cette dissymétrie artificielle est assez gênante. On voit bien, en comparant par exemple avec ADA où le "package" *complexe* comprendrait une procédure *plus* à deux paramètres, qu'il y a eu dans la conception de la "classe" en SIMULA fusion de deux notions bien distinctes : une structure syntaxique pour regrouper ("encapsuler") toutes les propriétés d'un type abstrait ; une structure syntaxique décrivant la composition (attributs) et la vie (initialisation) d'un exemplaire de ce type.

Remarque 4.4 On peut éviter l'allocation de mémoire entraînée par l'exécution de chacune des procédures de modification *plus, moins, mult, div, conjugué* en les remplaçant par des procédures d'affectation du résultat de l'opération correspondante, par exemple :

```
procedure affplus (c,c') ; ref (complexe) c,c' ;
        comment affecte à c, supposé créé, la somme
                de c' et de l'"exemplaire courant" ;
        if  c == none then erreur
        else c.cartésien (x + c'.x, y + c'.y)
```

et de façon correspondante pour les autres.

Remarque 4.5 Nous avons vu que la règle d'emploi de ce module était de passer par les procédures exportées, et elles seules. Il n'existe en SIMULA aucun moyen de faire respecter cette règle. On notera cependant que l'adjonction au langage de clauses restreignant la visibilité ne poserait aucune difficulté ; voir une proposition en ce sens dans [16].

Remarque 4.6 Dans les procédures calculant de nouveaux objets de type *complexe* (procédures *plus, moins, mult, div, conj*), l'emploi de variables locales (respectivement *p, m, m, d, cj*) est nécessaire pour éviter des récursivités non désirées. En effet, si l'on écrit dans le corps de la procédure *plus*

```
plus :- new complexe ;
plus.cartésien (x+c.x, y+c.y)
```

le *plus* de la première instruction, apparaissant à gauche du signe d'affectation, désigne bien le résultat de la procédure *plus*, mais celui de la seconde instruction représente un appel récursif (ici syntaxiquement illégal) de la même procédure. La règle (qui est une source bien connue de difficultés dans l'enseignement de la programmation avec les langages comme ALGOL 60) est qu'une apparition du nom d'une fonction dans le corps de cette fonction désigne un appel récursif sauf si elle se trouve à gauche d'un symbole d'affectation.

IV - COMPOSITION DESCENDANTE ET PREFIXATION DE CLASSES

IV.1 - Principe

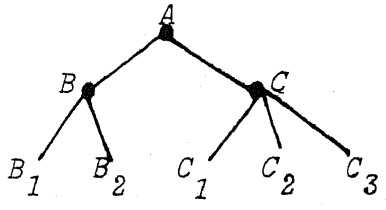
La composition descendante par affinages successifs [22] permet de développer un programme par niveaux d'abstraction en utilisant à chaque étape des éléments de niveau inférieur, non encore détaillés. Il est naturel de chercher à appliquer cette méthode non seulement aux programmes, mais aussi aux données, en particulier si la décomposition en modules est guidée par celles-ci.

Outre les modes d'abstraction de programmes (procédures) et de données (classes) partagés aujourd'hui avec d'autres langages, SIMULA offre un support linguistique original à la composition descendante dirigée par les données : la notion de préfixation de classe. Le principe est le suivant : étant donné une classe *A*, définissant un certain nombre d'attributs, variables et procédures, communs à tous les objets de type *ref (A)*, il est possible de définir de nouvelles classes *B*, *C*,... comme sous-classes de *A*, en préfixant leur déclaration par le nom de *A* :

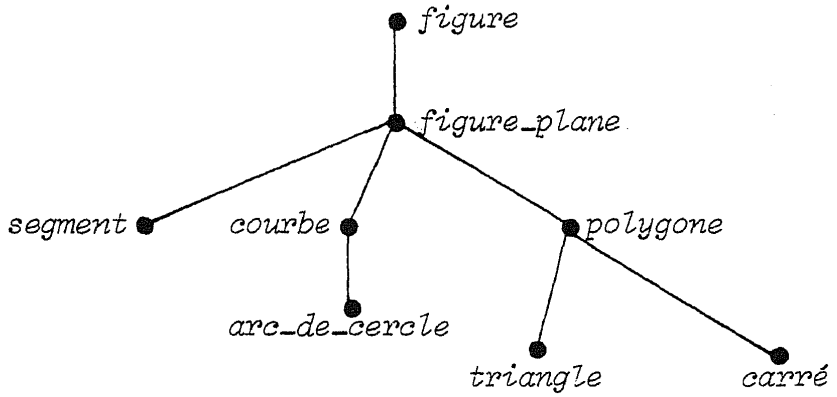
A class B ; begin ... attributs des objets B ... end ;

A class C ; begin ... attributs des objets C ... end ;

Les objets de types *B*, *C*, ... possèderont alors, outre les attributs intervenant dans la déclaration de *B*, *C*, ..., tous les attributs définis dans la déclaration de *A*. *B*, *C*, ... peuvent à leur tour préfixer de nouvelles déclarations de classes ; on obtient ainsi une structure hiérarchisée, arborescente, des déclarations de classes.

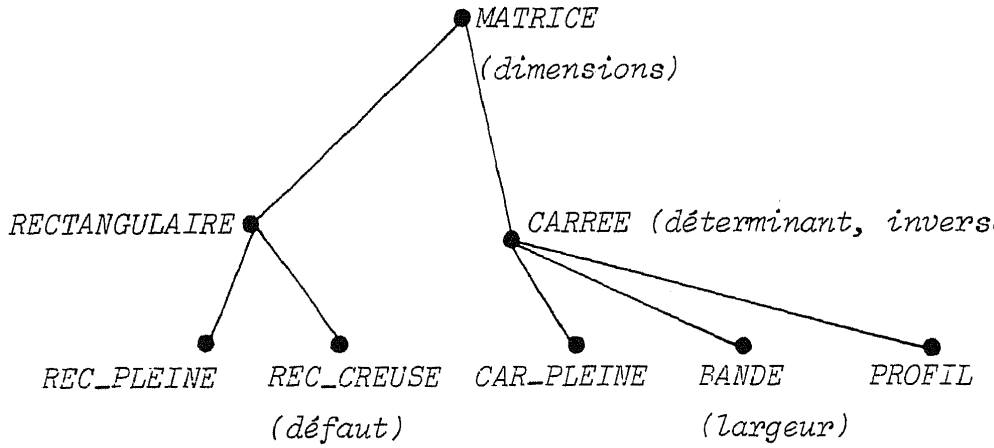


Le mécanisme de la préfixation permet de développer des structures de données progressivement, en spécialisant à chaque étape les objets décrits. Un exemple souvent choisi est celui des structures de données d'un système de manipulation d'images, pour lequel la hiérarchie des classes pourrait reproduire le schéma suivant :



Remarque : On notera que la préfixation d'une classe par une autre exprime la relation "est un exemple de", et non "utilise". Cette dernière relation est simplement représentée par des déclarations d'objets d'une autre classe. Ici, par exemple, les attributs de la classe *polygone* pourront comprendre des objets de type *ref (segment)*.

Un autre exemple est celui d'un logiciel numérique, offrant des opérations variables selon les caractéristiques des matrices manipulées : carrées ou non, pleines ou creuses, etc. Une hiérarchie possible pour décrire la structure du type *MATRICE*, et celle des classes SIMULA correspondantes, est la suivante :



On a indiqué au niveau de chaque noeud quelques-uns des attributs caractéristiques de ce noeud (et partagés par ses descendants).

Un autre exemple de la même démarche est, dans [6], la spécification en SIMULA d'un système relationnel de gestion de bases de données, où les choix de représentation des relations sont cachés dans des sous-classes.

Remarque : On notera le caractère fondamental de la structure en arbre des préfixations de classes. Cette limitation explique pourquoi nous n'avons pas utilisé la préfixation pour définir la classe *COMPLEXE* en III.3 : une telle classe pourrait bien admettre des sous-classes *COMPLEXE_CARTESIEN* et *COMPLEXE_POLAIRE*, mais celles-ci seraient disjointes et l'on ne pourrait pas tirer parti du fait que les deux représentations sont disponibles pour certains nombres complexes. De la même façon, le mécanisme de la préfixation, transposé aux mathématiques, permettrait de définir la hiérarchie monoïde-groupe-espace vectoriel, et la théorie des espaces topologiques, mais non de les réunir pour introduire la notion d'espace vectoriel topologique. Une structure telle que celle qui est induite en ADA par la relation *use* permet au contraire de définir des graphes acycliques, et non pas seulement des arbres.

IV.2 - Les objets virtuels

En recensant les attributs d'une classe possédant des sous-classes, comme *figure* ou *MATRICE*, on trouve fréquemment, comme conséquence du principe même de la conception descendante, des objets qui sont véritablement caractéristiques de cette classe-"souche", mais dont la réalisation précise dépend des sous-classes. Ainsi, on peut imaginer que toute *figure* possède parmi ses attributs des procédures *translation* et *rotation*, qui seront détaillées différemment selon qu'il s'agit d'un arc de cercle, d'un carré, etc. De même, une *MATRICE* sera sujette à des procédures *valeur (i, j)* et *changer_valeur (r, i, j)*, dont la mise en oeuvre dépend de la représentation physique.

De telles procédures sont dites virtuelles ; elles sont regroupées en SIMULA dans un paragraphe spécial en tête de la déclaration de la classe-souche, où l'on indique leur nom et, s'il y a lieu, le type de leur résultat, mais sans mentionner les arguments [1] :

(1) Cette dernière clause, assez désagréable en pratique, semble liée à l'interdiction de faire référence dans une classe au nom de toute sous-classe de cette classe, pour ne pas rendre impossible la compilation en un seul passage.

```
class MATRICE ;
    virtual : real procedure valeur ;
              procedure changer_valeur ;
    begin integer m, n ; comment les dimensions ;
    boolean procedure indices_corrects (i, j) ; integer i, j ;
          comment cette procédure n'est pas virtuelle ;
          indices_corrects := (i >= 1) and (i <= m) and
                              (j >= 1) and (j <= n)
    end MATRICE
```

Bien entendu, chaque procédure virtuelle devra être définie effectivement pour toute sous-classe susceptible de l'utiliser ; par exemple :

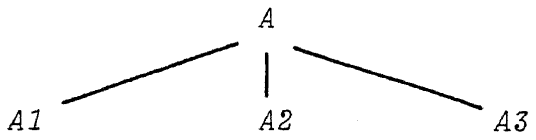
```
MATRICE class CARREE (ordre) ; integer ordre ;
    virtual : real procedure déterminant ;
              ref (CARREE) procedure inverse ;
    begin
    m := n := ordre
    end CARREE ;

CARREE class CAR_PLEINE ;
    begin
    real array représentation (1:ordre, 1:ordre) ;
    real procedure valeur (i, j) ; integer i, j;
              valeur := représentation (i, j) ;
    procedure changer_valeur (r, i, j) ; real r ; integer i, j ;
              représentation (i, j) := r ;
    real procedure déterminant ; ..... ;
    ref (CAR_PLEINE) procedure inverse ; ..... ;
    ..............
    end CAR_PLEINE
```

On notera que seuls les attributs "procédures" peuvent être virtuels, non les attributs "données".

IV.3 - Discrimination entre sous-classes

Soit la hiérarchie suivante :

```
class A ; ..... ;
A class A1 ; ..... ;
A class A2 ; ..... ;
A class A3 ; ..... ;
```



Un objet *X* de type *ref (A)* est susceptible de recevoir des valeurs de type *ref (A1)*, *ref (A2)*, *ref (A3)* :

X :- new A1 , etc.

Pour savoir ce qu'il en est à un moment donné de l'exécution, on peut utiliser l'instruction *inspect* :

```
inspect X
        when A1 do   action_1
        when A2 do   action_2
        when A3 do   action_3
```

On notera que cette instruction est à la fois une discrimination entre sous-types (*case* de PASCAL) et un mécanisme d'importation de noms externes (*with* en PASCAL, cf. *use* en ADA) car *action_1*, *action_2*, *action_3* peuvent utiliser directement les attributs de *X* relatifs à *A1*, *A2*, *A3* respectivement, sans passer par la notation pointée habituelle.

Il faut mentionner une autre méthode de discrimination utilisant l'opérateur *is* :

```
if X is A1 then .....
else if .....
```

La méthode utilisant *inspect* est de toute évidence préférable : outre qu'elle est plus rigoureuse, elle permet un contrôle statique de la validité des références. Notons cependant qu'un contrôle est toujours effectué (en partie à l'exécution dans le pire des cas) : le problème de la "référence folle", présent dans de nombreux langages, n'existe pas en SIMULA.

IV.4 - Constitution de modules d'application

Un aspect intéressant du mécanisme de préfixation est qu'il permet de constituer de façon commode des modules d'application ("packages"). On peut en effet préfixer par un nom de classe non seulement de nouvelles classes, mais aussi des programmes ou plus généralement des blocs, qui héritent alors des attributs de la classe préfixante. Il suffit donc d'"encapsuler" convenablement dans celle-ci les données et les opérations, éventuellement fort complexes,relatives à un domaine d'application spécialisé, et d'en fournir le mode d'emploi à l'utilisateur qui, dans le cas le plus simple, n'aura à connaître de SIMULA que les déclarations d'objets, l'affectation et l'appel de procédure. Des applications à l'enseignement de l'informatique se présentent naturellement : on peut créer un "univers" permettant à un projet d'étudiant d'aborder d'emblée des problèmes fins d'un système. SIMULA peut de ce point de vue être considéré comme un générateur de langages spécialisés - langages à qui, selon une interview récente /-23-/, l'avenir appartiendrait.

V - GENERICITE

V.1 - Principe et exemple simple

Une étape supplémentaire dans l'effort d'abstraction caractéristique de l'évolution actuelle de la programmation, et qui a donné naissance aux concepts introduits dans les deux paragraphes précédents (abstraction par rapport aux modalités de représentation des données : types abstraits, abstraction par rapport à la suite du développement du système : conception descendante), consiste à suivre le chemin tracé par les mathématiciens, qui cherchent à appliquer les mêmes résultats à des objets concrètement différents, mais caractérisés par des structures identiques. Deux catégories importantes d'objets susceptibles d'être ainsi traités sont :

- les opérations : il est loisible d'utiliser le même symbole + pour l'addition d'entiers, de réels, de matrices, ou le ou booléen, les propriétés externes étant les mêmes (associativité, élément neutre, etc.) ;

- les ensembles : les mêmes raisonnements, fondés sur la théorie des groupes, peuvent être appliqués à des ensembles de nombres, de transformations etc. munis de cette structure.

En programmation, un objet tel que l'opérateur + ou le "type" GROUPE, caractérisé par des propriétés fixes de structure mais susceptible de plusieurs interprétations concrètes, est dit générique. Certains langages de programmation récents permettent de définir de tels objets. Les deux grandes catégories d'objets génériques correspondent aux deux cas définis ci-dessus ; selon qu'on considère l'ordre des traitements ou celui des données, on distingue, selon la terminologie de [11] :

- La généricité "incrémentale", permettant d'interpréter différemment des opérateurs selon le type des données auxquels ils sont appliqués. C'est ce qui se passe dans les langages de programmation courants avec des opérateurs comme "+" et "*" qui représentent des calculs entiers ou flottants selon le type de leurs arguments ; de tels opérateurs sont dits "surchargés". ALGOL 68 permet de définir de nouvelles surcharges d'opérateurs.

- La généricité "structurale", intéressante surtout si on l'étend à la notion de "type abstrait générique", permettant de définir des types de données paramétrés par d'autres types. En remarquant ainsi dans l'exemple 1 (pile d'entiers) que le type des éléments empilés, *ENTIER*, ne joue aucun rôle dans la compréhension de la structure de pile, on est naturellement amené à définir le type générique *PILE (t)*, dont les opérations associées *(empiler, dépiler, pilevide)* peuvent être étudiées indépendamment du type *t* des objets empilés ; des "cas" particuliers du type *PILE (t)* sont par exemple *PILE (ENTIER)*, *PILE (TEXTE)*, voire *PILE (PILE (ENTIER))* etc.

On notera que la seconde espèce de généricité inclut la première : un type de données étant défini, comme nous l'avons vu au paragraphe III, par les opérations associées, ces opérations devront être génériques si le type l'est. *Empiler*, *dépiler*, etc., seront ainsi génériques dans notre exemple. La généricité incrémentale étant, prise isolément, d'un intérêt limité, nous étudierons ici la généricité de la seconde espèce.

V.2 - Types génériques en SIMULA

Une façon simple de représenter en SIMULA un type générique *typgen*, paramétré par un type *t*, est de définir d'abord une classe correspondant au paramètre *t* ;

class t ; ..... ;

et, dans la définition de la classe associée à *typgen*, de représenter tout objet du type *t* par un objet SIMULA de type *ref (t)*.

Les types correspondant à des arguments réels d'un cas particulier de *typgen* seront alors définis par des sous-classes de *t* :

t class t_1 ; ..... ;
t class t_2 ; ..... ; etc.

Cette méthode est appliquée à la définition d'une pile générique avec cette fois une représentation chaînée (exemple 5), et donc plus de restriction de taille.

```
class pile ; comment générique en t ;
      begin
      comment partie représentation ;
            ref (t) tête ;
            ref (pile) corps ;
      comment opérateurs de la spécification ;
            ref (t) procedure dépiler ;
                  begin
                  dépiler :- tête ;
                  tête :- corps.tête ; corps :- corps.corps
                  end dépiler ;
            procedure empiler (x) ; ref (t) x ;
                  begin ref (pile) c ;
                  c :- new pile ;
                  c.tête :- tête ; c.corps :- corps ;
                  tête :- x ; corps :- c
                  end empiler ;
            comment une pile vide a été créée par un new mais a un
                  corps vide (none) ;
            boolean procedure pilevide ;
                  pilevide := (corps == null)
end classe pile
```

EXEMPLE 5 : PILE GENERIQUE CHAINEE

On peut faire sur cet exemple les commentaires suivants :

Remarque 5.1 On ne peut pas avec cette méthode définir des piles d'entiers, de textes, de réels, de tableaux etc. ; il faut passer par des types intermédiaires préfixés par *t* :

```
t class tentier ;
        begin integer x end tentier ;

t class tréel ;
        begin real x end tréel ;

t class ttabréel ;
        begin real array x[1 : 100] end ttabréel ;
```

Remarque 5.2 Cette méthode ne permet pas de garantir l'intégrité d'une pile : rien n'empêche l'utilisateur de constituer une pile contenant à la fois des *tentier*, des *tréel* etc.

Remarque 5.3 Il n'est pas très satisfaisant d'avoir à déclarer une nouvelle classe, *t*, sans lien syntaxique avec la classe *pile*, pour rendre cette dernière générique. Le lecteur aura peut-être fait le rapprochement avec la remarque de la fin du paragraphe IV.2 : seuls les attributs "procédures" peuvent être virtuels.

V.3 - L'arbre binaire

L'exemple précédent ne permet pas de saisir tous les problèmes de la généricité, car il n'existe dans l'absolu aucune contrainte particulière sur le type des objets "empilables"(1). Il est au contraire assez fréquent qu'un type générique *typgen (t)* impose certaines conditions sur son paramètre formel, le type *t*.

(1) Cette affirmation n'est pas tout à fait exacte : le lecteur aura peut-être remarqué dans l'exemple 5 qu'on utilisait dans *empiler* l'affectation de pointeurs :- sur le type *t*, ce qui peut entraîner le partage de structures de données et des effets de bord non désirés. En fait, il aurait fallu même ici inclure une opération d'affectation dans la spécification de *t*.

Si l'on cherche par exemple à définir le type *arbin (t)*, pour arbre binaire de recherche contenant des objets de type *t*, il faut pouvoir disposer sur *t* d'une relation d'ordre (permettant les comparaisons), d'une procédure permettant l'affectation (pour les insertions) et du test d'égalité.

Le mécanisme des procédures virtuelles permet d'exprimer élégamment ces contraintes :

```
class scalaire ; comment un objet "insérable" dans un arbre binaire ;
      begin
      virtual : boolean procedure inférieur ;
                procedure affectation ; boolean procedure égal ;
      comment on peut à ce niveau compléter par les procédures suivantes : ;
      boolean procedure infouégal (s) ; ref (scalaire) s ;
              infouégal := inférieur (s) or égal (s) ;
      boolean procedure différent (s) ; ref (scalaire) s ;
              différent := not égal (s) ;
      boolean procedure supérieur (s) ; ref (scalaire) s ;
              supérieur := not infouégal (s) ;
      boolean procedure supouégal (s) ; ref (scalaire) s ;
              supouégal := not inférieur (s)
      end scalaire.
```

Exemple 6 : scalaire (à mettre dans un arbre binaire de recherche)

Des exemples de types "scalaires" sont alors :

```
scalaire class scalent ;

        begin scalaire entier ;

        integer i ;

        boolean procedure inférieur (s) ; ref (scalent) s ;
                inférieur := (i < s.i) ;

        procedure affectation (s) ; ref (scalent) s ;
                i := s.i ;

        boolean procedure égal (s) ; ref (scalent) s ;
                égal := (i = s.i)

        end scalent ;

scalaire class scal_chaîne_car ; ..... ;

scalaire class entrée_table_symbole ; ..... ;
```

On notera que le mécanisme permettant de contraindre le type *t* (ici *scalaire*) est purement syntaxique : des objets d'une classe préfixée par *scalaire* et à laquelle manquerait une des procédures *inférieur*, *affectation* ou *égal* pourraient être insérés dans un arbre binaire (erreur à l'exécution) ; mais aucun moyen n'existe pour spécifier dans la déclaration de *scalaire* que la réalisation de ces procédures dans des sous-classes devra répondre à une sémantique compatible avec la notion habituelle de relation d'ordre, d'affectation et d'égalité.

Nous donnons ci-dessous une réalisation du type générique *arbin*, utilisant la classe *scalaire* précédente (exemple 7).

```
class arbin ; comment arbre binaire de recherche ;
      ref (scalaire) racine ; ref (arbin) gauche, droite ;

      procedure insérer (s) ; ref (scalaire) s ;
            if racine == null then racine.affectation (s)
            else
              begin ref (arbin) père, fils ;
              fils :- this arbin ;
              while fils =/= null do
                        begin
                        père :- fils
                        fils :- if s.inférieur (père.racine) then
                                père.gauche
                                else père.droite
                        end ;
              fils :- new arbin ; fils.affectation (s) ;
              if s.inférieur (père.racine) then père.gauche :- fils
              else père.droite :- fils
              end insérer ;
      boolean procedure recherche (s) ; ref (scalaire) s ;
            if racine == null then recherche := false
            else
              begin ref (arbin) noeud ; boolean b ;
              noeud :- this arbin ; b := true ;
              while b do
                        begin

                        if noeud == null then b := false
                        else if s.égal (noeud.racine) then b := false
                        else if s.inférieur (noeud.racine) then
                                              noeud :- noeud.gauche
                        else noeud :- noeud.droite
                        end boucle ;
              recherche := (noeud =/= null)
              end cas non vide ;

..... suite de la classe .....
```

EXEMPLE 7 : ARBRE BINAIRE GENERIQUE

Cet exemple appelle quelques remarques :

Remarque 7.1 *=/=* représente l'inégalité pour les objets de type *ref(...)*.

Remarque 7.2 On utilise dans les procédures *recherche* et *insérer* la construction *this c* qui, dans une déclaration d'une classe *c*, désigne l'objet de type *ref (c)* qui est précisément l'exemplaire courant de *c*, celui qu'on "est en train de décrire" (cf. la remarque 4.3).

Remarque 7.3 La programmation pour le moins maladroite de la boucle dans *recherche* vient de ce que la norme de SIMULA ne garantit pas (comme le fait celle d'ALGOL W par exemple) que le *and* est un "et conditionnel", c'est-à-dire qu'on peut écrire *a and b* si *a* est faux et *b* non défini.

VI - PROGRAMMATION QUASI-PARALLELE : LES COPROGRAMMES

VI.1 - Généralités

En décrivant les deux composantes d'une classe, introduites au paragraphe II :

- attributs (variables et procédures)
- actions (d'initialisation)

nous avons jusqu'ici mis l'accent sur la première.

L'éclairage inverse permet de considérer une classe non pas seulement comme une "capsule" de données assurant la représentation d'un type abstrait, mais comme un modèle de processus exécutant répétitivement des actions, qui ne sont plus simplement d'"initialisation".

Cette possiblité ouvre la voie à une méthode de programmation intéressante, qui considère le système physique modélisé comme un ensemble de sous-systèmes coopérants dont l'activité se poursuit en parallèle, et qui doivent de temps à autre se synchroniser et échanger de l'information.

Pour pouvoir modéliser convenablement une telle situation, il faut disposer d'opérations représentant la synchronisation et l'échange. Bien entendu, dans un contexte d'exécution classique, le modèle lui-même est strictement séquentiel, et non pas parallèle.

La "synchronisation" est assez bien représentée en SIMULA par la primitive *resume*, dont l'effet peut être défini (au moins pour les utilisations simples) de la façon suivante : si *X* et *Y* sont des variables de type *ref (...)* désignant des processus, telles que les processus associées aient été créés (par *X :- new..., Y :-new...*), alors l'exécution par le processus associé à *X* de *resume Y* a pour effet de suspendre l'exécution de ce processus et de reprendre celle du processus désigné par *Y*, au dernier endroit où elle avait précédemment cessé. L'exécution du processus désigné par *X* pourra se poursuivre ultérieurement, à l'instruction devant suivre le *resume Y*, si un processus quelconque, ou le programme principal, exécute une instruction *resume X*.

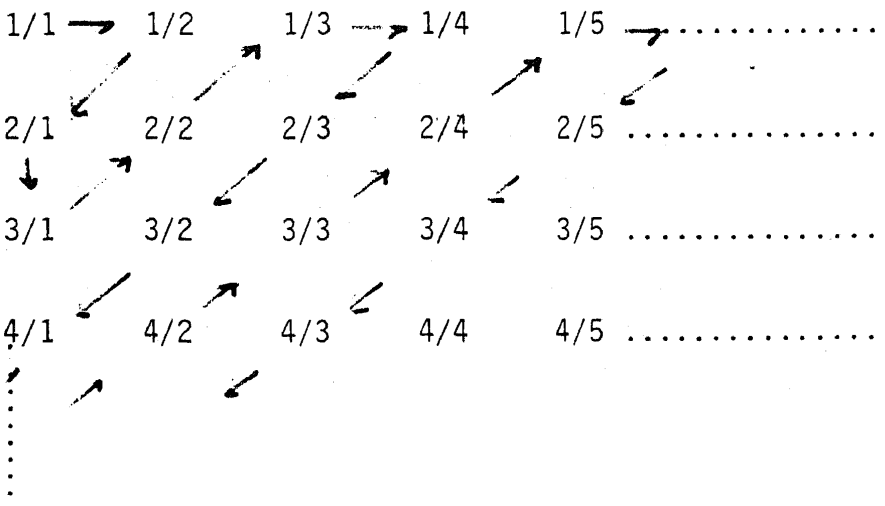
L'aspect "communication" est moins bien traité ; en fait SIMULA n'offre pas d'instruction particulière permettant l'échange de messages entre deux processus représentés par des exemplaires de classes ; on notera que l'instruction *resume X* ne permet pas de transmettre de paramètres à *X*. La communication d'informations entre unités de programme quasi-parallèles se fait généralement par le moyen de variables globales, ce qui n'est pas une méthode de programmation très sûre, ni très élégante.

Au-delà des détails de mise en oeuvre qui en obscurcissent souvent la présentation, la programmation quasi-parallèle à la SIMULA permet, par le simple jeu de la notion de classe et de l'instruction *resume*, de représenter sans les trahir des situations réelles où intervient un parallélisme véritable ; c'est sur cette base que sont en particulier définies les primitives de simulation du langage. La programmation quasi-parallèle est cependant également intéressante lorsque la réalité modélisée n'est pas fondamentalement parallèle, mais simplement constituée d'un certain nombre de sous-systèmes qui se déterminent les uns par rapport aux autres sur un pied d'égalité, plutôt que selon une dépendance hiérarchique correspondant à la décomposition classique en sous-programmes. Le programme SIMULA sera alors formé de classes de type "processus" se comportant comme des coprogrammes.

Nous étudierons l'utilisation des coprogrammes à l'aide de deux exemples, l'un simple, l'autre plus compliqué. Le but de ces exemples n'est pas de prôner systématiquement cette approche, mais d'illustrer un style peu classique de programmation.

VI.2 - Coprogrammes : Un exemple simple

Nous appliquerons d'abord cette démarche à un exemple simple. Il s'agit d'un problème tiré de Jackson [11]: imprimer les rationnels selon un ordre diagonal inspiré de la méthode de Cantor :



On impose de n'imprimer que les 1000 premiers rationnels de cette énumération (ou une autre condition d'arrêt *c(i,j)* dépendant du rationnel *(i/j)*).

En ignorant d'abord cette condition d'arrêt, on peut écrire le programme d'impression comme une boucle infinie

```
integer i, j ;
i := 1 ; j := 1 ;
while true do
    begin
    while  i >= 1  do
        begin
        imprimer_fraction (i, j) ;
        i := i-1 ; j := j+1
        end descente ;
    i := 1 ;
    while  j >= 1  do
        begin
        imprimer_fraction (j, i) ;
        j := j-1 ; i := i+1
        end montée ;
    j := 1
    end
```

La difficulté, si l'on ajoute une condition d'arrêt, est qu'elle peut intervenir après le calcul d'un rationnel quelconque, c'est-à-dire dans l'une des boucles internes. Une solution classique amènerait à mélanger le calcul des fractions successives avec le contrôle de la condition d'arrêt (et il faut imaginer, au-delà de cet exemple précis, un programme comprenant beaucoup de tels calculs dans des boucles profondément imbriquées).

Dans une solution quasi-parallèle, il suffit d'avoir deux coprogrammes : l'un, *cont*, contrôlera l'arrêt ; l'autre, *calc*, calculera et imprimera les rationnels successifs.

Le programme prend alors la forme suivante :

```
begin
class calcul ;  begin ..... cf. ci-après ..... end calcul  ;
class contrôle ; begin ..... cf. ci-après ..... end contrôle ;

ref (calcul) calc ; ref (contrôle) cont ;
calc :- new calcul ; cont :- new contrôle ;
resume calc
end programme principal
```

A l'occasion de *calcul* et *contrôle*, nous introduisons ci-dessous la primitive *detach* qui joue un rôle important, à côté de *resume*, dans l'expression des coprogrammes ; elle permet de rendre le contrôle à l'élément de programme qui a créé l'exemplaire de classe dans lequel elle intervient. Elle sera indispensable pour qu'un objet de type *ref (calcul)* ou *ref (contrôle)* (ici *calc* ou *cont*), après sa création par un *new* ..., rende temporairement le contrôle au programme principal ci-dessus.

Le programme de contrôle est simplement :

```
class contrôle ;
    begin  integer n ;
    detach ;
    for n := 1 step 1 until 1000 do
        resume calc ;
    detach
    end contrôle ;
```

Le programme de calcul devient :

```
class calcul ;
    begin  integer i, j ;
    detach ;
    ..... le programme initial, où l'on a fait précéder
          chaque appel à "imprimer_fraction" de
          "resume cont ;" .....
    end calcul ;
```

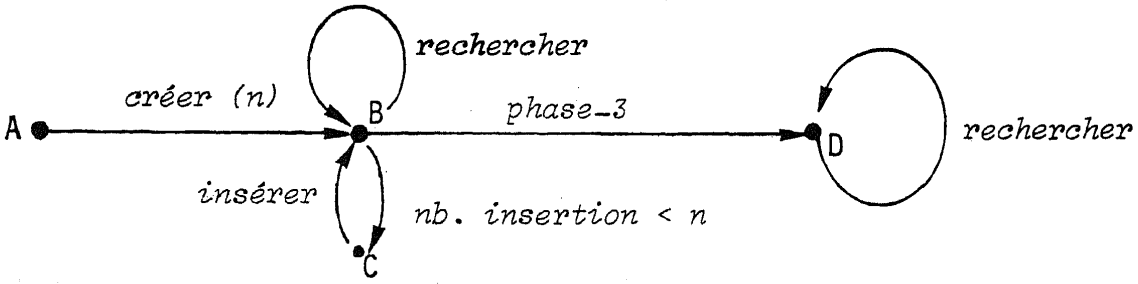
En séparant ainsi la partie "calcul" et la partie "contrôle", nous avons obtenu une décomposition en deux coprogrammes faiblement couplés. C'est l'un des avantages de ce style de programmation que de produire des unités de programme pouvant dans une large mesure être lues et comprises indépendamment les unes des autres. On notera en particulier que *n* est une variable locale à *contrôle*, *i* et *j* à *calcul*. Cependant, une condition d'arrêt plus complexe aurait obligé à introduire des variables globales, qui seront également nécessaires dans l'exemple suivant où les différents coprogrammes se transmettent plus d'informations.

VI.3 - Types abstraits avec scénario

Nous introduirons, pour illustrer la démarche quasi-parallèle, un exemple de ce qu'on peut appeler un "type abstrait avec scénario". Il s'agit d'un module défini, comme à la section II, par son aptitude à répondre d'une façon bien spécifiée à certaines opérations, mais avec la contrainte supplémentaire que seules certaines successions de ces opérations doivent pouvoir être acceptées. En d'autres termes, on impose un "scénario" réglant les vies possibles du module.

L'exemple choisi est celui d'un module de gestion de table, dont la spécification inclut comme opérations de base *créer (n)* (résultat : une table pouvant abriter jusqu'à *n* éléments), *rechercher (c, tab)* (résultat : un élément de clé *c* dans *tab*, s'il y en a), *insérer (x, tab)*, (la nouvelle table résultant de l'ajout de *x* à *tab*). Les propriétés formelles du type abstrait correspondant (de la forme *rechercher (clé(x), insérer (x, tab))* = *x* etc.) sont faciles à écrire.

Nous ajoutons ici les restrictions suivantes. La vie d'une table se déroule en trois phases, symbolisées par le diagramme de transition ci-dessous. Au cours de la première phase, l'utilisateur doit exécuter l'opération *créer (n)*. Au cours de la seconde, il peut effectuer librement des recherches et des insertions, sans toutefois exécuter plus de n insertions. Au cours de la troisième, il doit effectuer une fois une opération sans argument notée *phase_3* (un signal de changement de phase), et ne peut plus ensuite demander que des recherches. (On peut interpréter cet exemple comme celui d'un compilateur en plusieurs "passages" qui, après un certain temps, recherche mais n'insère plus dans la table des symboles).



On désire que le module de gestion de la table soit programmé de façon à n'accepter que les suites de transitions correspondant au diagramme ci-dessus, où les transitions non figurées sont des cas d'erreur.

VI.4 - Rappel sur CSP

Il sera particulièrement commode d'illustrer d'abord la programmation quasi-parallèle de cet exemple dans la notation CSP de Hoare [8] [12], qui nous fournira un modèle pour la programmation en SIMULA.

L'intérêt de CSP ("Communicating Sequential Processes") pour traiter ce problème est double. Tout d'abord, il s'agit d'un formalisme conçu pour l'expression de programmes parallèles, mais prévu pour que le programme décrivant le fonctionnement d'un système avec parallélisme puisse être réinterprété, dans un contexte séquentiel, comme le programme de simulation quasi-parallèle du même système.

Une autre caractéristique de CSP qui rend cette notation intéressante comme modèle simple de systèmes parallèles ou quasi-parallèles est que ses opérations primitives, l'entrée et la sortie, réalisent à la fois la synchronisation et la communication. L'écriture est la suivante : un processus P exécute une sortie à l'intention d'un processus Q, notée :

$$Q!sig\ (x_1, \ldots x_n)$$

où *sig* est un nom de signal, et $x_1, \ldots x_n$ des expressions définies dans *P*. A cette sortie doit correspondre dans *Q* une entrée notée :

$$P?sig\ (y_1, \ldots y_n)$$

où *sig* est le même nom de signal, et $y_1, \ldots y_n$ sont des variables de *Q*. A l'exécution de l'une de ces instructions dans l'un des deux processus, ce processus s'arrête jusqu'à ce que l'autre ait effectué l'instruction symétrique, compatible par le nom du signal et le nombre des arguments. Lorsque la "rencontre" se produit, il y a affectation simultanée des valeurs de $x_1, \ldots x_n$ à $y_1, \ldots y_n$, et les deux processus poursuivent leur chemin. Bien entendu, elle peut ne jamais se produire, auquel cas l'un des processus, ou les deux, resteront bloqués indéfiniment. Une synchronisation simple sans échange d'information correspond à un signal sans paramètres *sig( )*.

Les structures de contrôle de CSP sont reprises des structures de Dijkstra [5], avec une notation plus synthétique. L'écriture :

$$[C_1 \rightarrow A_1 \,[]\, C_2 \rightarrow A_2 \,[]\, \ldots\ldots \,[]\, C_n \rightarrow A_n]$$

désigne une instruction conditionnelle exécutant l'une des instructions A_i telle que la condition C_i correspondante est vraie, échouant si aucune ne l'est ; l'écriture :

$$*[C_1 \rightarrow A_1 \,[]\, C_2 \rightarrow A_2 \,[]\, \ldots\ldots \,[]\, C_n \rightarrow A_n]$$

est une boucle exécutant répétitivement l'instruction conditionnelle précédente, à cette différence près que le cas où toutes les C_i sont fausses devient ici la sortie normale de boucle.

CSP ajoute à ce formalisme des "commandes gardées" une extension aux "conditions", leur permettant d'inclure des instructions d'entrée *P?sig* $(y_1, \ldots y_n)$. L'évaluation d'une telle "condition" est suspendue jusqu'à ce que *P* envoie un signal compatible ; si cela se produit, l'instruction d'entrée s'effectue et produit la valeur *vrai*. Si *P* est terminé, la valeur *faux* est produite (on peut ainsi programmer un test de fin de fichier).

Les conditions peuvent être construites à partir de l'opérateur ; qui représente le "et conditionnel" : C_1 ; C_2 ; ... C_n signifie *si* C_1 *alors si* C_2 *alors* ... *si* C_{n-1} *alors* C_n ; une telle condition complexe peut inclure une instruction d'entrée unique, qui en est alors obligatoirement le dernier élément C_n ; la raison évidente de cette restriction est que l'évaluation d'une instruction d'entrée est irréversible. Notons que la présence d'instructions d'entrée dans plusieurs branches d'une instruction conditionnelle ou d'une boucle correspond à l'attente du premier parmi plusieurs événements possibles.

VI.5 - Un modèle en CSP

Ce rappel des bases de CSP permet d'écrire le programme, fort simple, traitant le "type abstrait avec scénario" décrit précédemment.

```
n : integer ;
    [utilisateur?créer (n) →
              i : integer ; phase_2 : boolean ;
              i := 0 ; phase_2 := true ;
             *[phase_2 ; utilisateur?rechercher (x) →
                                  ...recherche (linéaire) de x... ;
                                  utilisateur!résultat (...)
              ▯phase_2 ; i < n ; utilisateur?insérer (x) →
                                  ...insertion de x...
              ▯phase_2 ; utilisateur?phase_3( ) → phase_2 := false
             ] ; réorganiser la table ;
             *[utilisateur?rechercher (x) →
                                  ...recherche (associative) de x... ;
                                  utilisateur!résultat (...)
```

EXEMPLE 7 : TYPE AVEC SCENARIO EN CSP

Une caractéristique intéressante de ce modèle est qu'on spécifie seulement le comportement légal du module, décrit par le schéma précédent ; toute tentative illégale (plus de n insertions, recherche ou insertion non précédées par une création, plusieurs créations) entraîne un blocage , etc.

VI.6 - Traitement de l'exemple en SIMULA

On peut utiliser le programme CSP précédent comme modèle pour l'expression de la même solution en SIMULA. Le résultat ne sera pas tout à fait aussi élégant, mais il est possible, on va le voir, de conserver l'essentiel de la structure.

La différence principale tient à l'absence de primitives de communication. L'information entre le module *U* (utilisateur) et le module *G* (gestionnaire de la table) passera par des variables globales qui serviront de boîtes aux lettres. Plus précisément :

. une boîte *taille* (variable entière) servira à déposer n pour la création ;

. une boîte *dépôt* (variable de type "*clé*") servira pour déposer une clé à rechercher ;

. une boîte *retrait* (variable de type "*info*") servira à abriter l'objet résultat d'une recherche ;

. une boîte *entrée* ("*info*") servira à déposer un objet à insérer ;

. une boîte *demande* (variable entière) contiendra le type de demande (*créer*, *insérer*, *rechercher*, ou *phase-3*).

On notera sur ce dernier point que le réflexe immédiat est d'écrire à la PASCAL :

type demande = (créer, insérer, rechercher, phase-3)

et qu'il faut malheureusement revenir ici à des codes numériques choisis par le programmeur. L'absence des types par énumération est de toute évidence un handicap sérieux dans les langages pré-pascaliens.

On suppose donc que le programme principal a la forme de l'exemple 8-1.

```
begin
integer demande ;
integer créer, insérer, rechercher, phase_3 ;
integer taille ;
class info ; ..... ; ref (info) retrait, entrée ;
class clé  ; ..... ; ref (clé) dépôt ;
class utilisateur ; .... cf. exemple 9 .... ;
class gestionnaire ; .... cf. exemple 10 .... ;
ref (utilisateur) U ; ref (gestionnaire) G ;
créer :=  0 ; insérer :=  1 ; rechercher  := 2 ; phase_3  := 3 ;
U  :- new utilisateur ; G :-  new gestionnaire ; resume U
end  programme principal
```

EXEMPLE 8-1 - PROGRAMME PRINCIPAL

L'"utilisateur" passe alors par des procédures données à l'exemple 8-2. Le module de gestion de table utilise les procédures de l'exemple 8-3, et sa partie algorithmique est donnée en 8-4.

On note dans cet exemple une certaine lourdeur dans le mécanisme de communication entre coprogrammes. Mais, comme le programme CSP, le coeur du processus de gestion de table (exemple 8-4) est remarquable en ce qu'il décrit le déroulement du scénario et rien d'autre. Toute suite de demandes incorrecte entraînera une erreur (bouclage, ou tentative de reprise d'un objet terminé) ; bien entendu, on aurait pu traiter explicitement ces erreurs (ce qui revient à ajouter au diagramme de transition des arcs supplémentaires). Mais on a bien en 8-4 une description du module *gestionnaire* en tant que processus homogène et autonome, ce qui était le but recherché.

```
class utilisateur ;
        begin
        procedure demander_création (n) ; integer n ;
                begin
                demande := créer ; taille := n ; resume G
                end demander_création ;
        procedure demander_insertion (x) ; ref (info) x ;
                begin
                demande := insérer ; entrée :- x ; resume G
                end demander_insertion ;
        ref (info) procedure demander_recherche (c) ; ref (clé) c ;
                begin
                demande := rechercher ; dépôt :- c ; resume G ;
                demander_recherche: :- retrait
                end demander_recherche ;
        procedure demander_phase_3 ;
                begin
                demande := phase_3 ; resume G
                end demander_phase_3 ;
        detach ;  ----------------------------------------
        ----------------------------------------------------
        end utilisateur ;
```

EXEMPLE 8-2 : UTILISATEUR

```
class gestionnaire ;
    begin
    integer tailletab, nombrélem ;
    procedure création ;
            begin
            tailletab := taille ; ... créer la table... ;
            resume U
            end création ;
    procedure insertion ;
            begin ref (info) x ;
            x :- entrée ; ... insérer x dans la table ... ;
            nombrélem := nombrélem + 1 ; resume U
            end insertion ;
    procedure recherche_linéaire ;
            begin ref (clé) c ; ref (info) resul ;
            c :- dépôt ; ... rechercher, dans la table gérée
                             linéairement, un élément résul de clé c... ;
            retrait :- résul ; resume U
            end recherche_linéaire ;
    procedure recherche_associative ;
            begin ref (clé) c ; ref (info) résul ;
            c :- dépôt ; ... rechercher, dans la table gérée
                             associativement, un élément résul de clé
                             c ... ;
            retrait :- résul ; resume U
            end recherche_associative ;
    procedure changer ;
            begin
            ... réorganiser la table linéaire en table associative...
            end changer ;
```

EXEMPLE 8-3 : GESTIONNAIRE (PROCEDURES NON EXPORTEES)

```
detach ;
if demande = créer then

    begin

    nombrélem := 0 ;

    while (demande ≠ phase_3) do

        if demande = recherche then

            recherche_linéaire

        else if demande = insertion and nombrélem < tailletab then
                insertion ;

    changer ;

    while demande = recherche do

        recherche_associative

    end vie de la table

end classe gestionnaire
```

EXEMPLE 8-4 : GESTIONNAIRE (INSTRUCTIONS DE LA CLASSE)

VII - CONCLUSION

Nous nous permettrons, pour conclure, d'abandonner le ton de l'analyse académique pour ouvrir une réflexion plus subjective sur le problème de l'évaluation et du choix d'un langage de programmation.

En examinant la manière dont SIMULA résiste à un examen selon les critères étudiés, qui correspondent à quelques-unes des préoccupations principales dans la réflexion actuelle sur les langages de programmation, on peut suggérer le bilan suivant :

- SIMULA propose des constructions répondant au moins en partie à tous les concepts examinés, à l'exception de celui de la protection, dont l'application reste essentiellement à la charge du programmeur - tout au moins tant que la proposition de Palme /16/ ne fait pas partie de la norme.

- Les concepteurs de SIMULA se sont heurtés en 1967 à des difficultés que devaient connaître à leur suite les créateurs de toute la série de langages plus récents cherchant à répondre aux critères cités. Sans qu'une comparaison sérieuse ait été ébauchée dans cet article (on pourra cependant se reporter aux autres communications du même volume), il semble bien que les solutions retenues en SIMULA 67, si elles ne sont pas meilleures, ne sont pas non plus tellement pires que ce qui a été choisi dans ces propositions nouvelles.

La présente étude de SIMULA a été suscitée par une réflexion sur un problème tout à fait concret : quel langage, ou, plutôt, quel système de programmation, peut-on envisager d'introduire dans un environnement industriel de programmation, pour aider à améliorer la manière dont sont écrits les programmes (un des sous-produits d'un tel effort, s'il réussit, étant de permettre aux utilisateurs de faire *moins* de programmation et plus de physique, d'automatique, de comptabilité, ou d'autre chose) ?

Considérons plus précisément le cas de l'informatique scientifique, et de projets d'une certaine taille exigeant des ressources (temps, mémoire centrale, fichiers) importantes. Nous écartons donc la micro- et la mini-informatique. Essayons d'analyser les difficultés qu'éprouvent dans un tel cadre les utilisateurs de calcul avec leur outil principal, en général FORTRAN. Il semble qu'elles se ramènent pour une grande part (sans que les intéressés en aient en général conscience) aux problèmes cités ci-dessous par ordre d'importance décroissante, le premier se détachant nettement comme crucial :

1. La modularité, la conception descendante : comment développer et conserver un grand programme sous forme de "morceaux" cohérents, l'ensemble restant maîtrisable ? Notons en particulier que l'impossibilité dans les langages classiques de construire un module autour d'une structure de données (une classe) entraîne des pratiques conduisant par réaction à une visilité totale ("communs poubelles") qui sont une cause d'anarchie particulièrement répandue.

2. L'allocation dynamique de tableaux. Ce problème peut sembler peu sérieux en théorie, mais joue un rôle important en programmation scientifique. Chaque programmeur a ses techniques propres pour pallier l'absence d'allocation dynamique, de la gestion d'un commun spécial à l'utilisation d'un "préprocesseur" ad hoc. Il s'agit là aussi d'un facteur important d'incohérence dans les programmes (et d'inefficacité).

3. La récursion. On rencontre régulièrement des programmes qui effectuent (souvent à l'insu du programmeur) des opérations fondamentalement récursives, comme des parcours d'arbres. L'absence de récursivité dans le langage entraîne un mélange douteux entre les opérations de l'algorithme et le contrôle de la récursion (gestion de pile), qui sape la structure du programme.

4. (Pour les utilisateurs de FORTRAN).La manipulation de caractères.

5. Les structures de contrôle.

Bien entendu, on ne peut proposer de remplacement aux solutions actuelles qui détruirait leurs qualités les plus appréciables :

A - L'éfficacité. On peut dans la plupart des cas accepter une certaine dégrade performances comme prix d'une programmation plus rapide et plus fiable, mais pas au-delà d'une certaine limite, de 20% à 50%.

B - La compilation séparée, la possibilité de réutiliser des programmes écrits dans d'autres langages et de constituer des bibliothèques.

C - Des entrées et sorties riches, l'accès direct.

D - La qualité "industrielle" de la documentation, la facilité d'emploi du compilateur, l'accès à des opérations du système d'exploitation, etc.

E - La normalisation du langage et la transportabilité des programmes (exigeant en général un certain compromis avec C et D).

Notons que toutes ces qualités s'appliquent non à des langages seuls, mais à des systèmes de programmation - qui sont ce que l'utilisateur "voit"".

Si l'on écarte les langages récents encore à l'état d'expérimentation, pour ne considérer que les langages de large diffusion, on peut faire les constatations suivants :

- FORTRAN ne répond pas aux critères 1,2,3,4 ni 5 ; la nouvelle norme n'améliorera sensiblement les choses que pour 4 (caractères) et un peu 5 (structures de contrôle).

- COBOL satisfait 4 et à peu près 5, mais non 1, 2 ni 3.

- PL/1 ne répond pas au critère 1 (modularité). Il est encore mal normalisé (E).

- PASCAL n'est pas satisfaisant pour 1 ni pour 2 (tableaux alloués à l'exécution). La compilation séparée (B) n'est pas définie de façon normalisée, bien que certains systèmes la permettent.

- SIMULA offre une solution satisfaisante à 1, excellente à 2,3,4 (la gestion du type *text*, à la façon d'un type abstrait, est remarquablement élégante), et bonne à 5, malgré l'absence regrettable d'une instruction *case*...*of*... . Les qualités A à D dépendent évidemment à la fois de E (normalisation) et des divers systèmes. On peut noter que le langage est théoriquement bien normalisé (document [7]), mais que l'épaisseur des "guides du programmeur" propres à chaque système montre qu'il reste des problèmes. Pour A (efficacité), les systèmes "officiels" diffusé par le NCC sont satisfaisants. Le point B (compilation séparée) est fixé par la norme de façon également satisfaisante, mais comme une facilité optionnelle que n'offrent donc pas tous les systèmes, à l'origine en tout cas. Les entrées et sorties et l'accès direct (C) sont bien définis, grâce à la classe système *BASICIO*, et SIMULA est agréable à utiliser pour ce type d'opérations ; noter cependant que pour des raisons d'ailleurs évidentes la norme est assez discrète sur les détails de l'accès direct. Enfin, le critère D (qualité industrielle) est convenablement rempli par les systèmes NCC.

Choisir un langage de programmation et, encore plus, un langage destiné à en supplanter un autre, est une lourde décision, et les critères à considérer ne se limitent pas à ceux que nous venons de voir. Nous nous garderons donc de toute affirmation péremptoire. Nous espérons cependant avoir permis au lecteur de noter que dans le paysage changeant des modes en programmation, et en attendant le langage du futur, UTOPIA 84, ou 94, ou 2004, SIMULA est plus qu'un monument antique et respectable.

REMERCIEMENTS

Je remercie les organisateurs de GROPLAN et de la réunion de Cargèse pour avoir suscité une réflexion comparative sur les langages de programmation actuels, l'auditoire pour ses critiques lors de mon exposé à cette réunion, J. André pour plusieurs remarques importantes, ainsi qu'A. Bossavit, M. Demuynck, B. Logez et N. Penot pour leurs commentaires sur la première version de cet article.

BIBLIOGRAPHIE

[1] Birtwistle, Graham M., Ole Jordan Dahl, Bjorn Myhrhaug, et Kristen Nygaard :
SIMULA BEGIN ; Studentliteratur, Lund (Suède) et Auerbach Publishers, Philadelphie (Pennsylvannie), 1973

[2] Carrez, C. :
La Protection dans les systèmes et son expression dans les langages de programmation ; Réunion AFCET-GROPLAN, Cargèse, 14-22 mai 1979.

[3] Dahl, Ole Jordan :
SIMULA, An Algol-Based Simulation Language
Communications of the ACM, 9, 9, pages 671-681, juillet 1966.

[4] Dahl, Ole Jordan, Bjorn Myhrhaug et Kristen Nygaard :
Common Base Language ; Norwegian Computing Center, Publication S22, Octobre 1970

[5] Dijkstra, Edsger W. :
A Discipline of Programming
Prentice-Hall, Englewood Cliffs (N.J.), 1976

[6] Dufourd, J. F. :
Types abstraits, modèles relationnels, et langage SIMULA
Colloque "Les bases de Données, modèles, mise en oeuvre, évaluation" ; Chapitre français de l'ACM, Université Paris VI, 14-15 juin 1979.

[7] Geschke, Charles M., James H. Morris Jr., et Edwin H. Satterthwaite
Early Experience with Mesa ; Communications of the ACM, 20, 8, pages 540 - 553, août 1977

[8] Hoare C.A.R. :
Communicating Sequential Processes ; Communications of the ACM, 21, 8, pages 162-180, Août 1978

[9] Honeywell, Inc. et Cii Honeywell Bull : *Reference Manual for the GREEN Programming Language* ; 15 Mars 1979.

/10/ Jackson, Michael A. :
Principles of Program Design ; Academic Press, Londres, 1975.

/11/ Jacquet, Paul :
La Généricité comme outil d'abstraction dans les langages de programmation ; Théories et Techniques de l'informatique, Congrès AFCET-TTI, Gif-sur-Yvette, 13-15 Novembre 1978

/12/ Kaubisch W.H. et C.A.R. Hoare :
Discrete Event Simulation Based on Communicating Sequential Processes ; Department of Computer Science, The Queen's University, Belfast (Irlande du Nord), 1978

/13/ Lampson B.W., J.J. Horning, Ralph L. London et G.J. Popek :
Report on the Programming Language Euclid ; SIGPLAN Notices 12, 2, pages 1-79 (n° spécial), Février 1977

/14/ Liskov, Barbara H., Alan Snyder, Russel Atkinson et Craig Schaffert :
Abstractions Mechanisms in CLU ; Communications of the ACM, 20,.8, Août 1977, pages 564-576.

/15/ Meyer, Bertrand, et Claude Baudoin :
Méthodes de Programmation ; Eyrolles, Paris, Collection de la Direction des Etudes et Recherches EDF, 1978.

/16/ Palme, Jacob :
New Feature for Module Protection in Simula ; SIGPLAN Notices, 11, 5, 1976.

/17/ Parnas, David L. :
A Technique for Software Module Specification with Examples ; Communications of the ACM, 15, 5, pages 330-336, Mai 1972.

/18/ Parnas, David L. :
On the Criteria to be used in Decomposing Systems into Modules ; Communications of the ACM, 15, 12, pages 1053-1058, Décembre 1972

/19/ Parnas, David L. :
Designing software for Ease of Extension and Contraction ; IEEE Transactions on Software Engineering, SE-5, 2, pages 128-138, Mars 1979

[20] Scholl, Pierre-Claude, Pierre-Yves Cunin et Michael Griffiths : *Aspects fondamentaux du Langage MEFIA* ; Journées d'Etude sur la Fiabilité des Programmes dans les Applications Industrielles, Chapitre français de l'ACM, EDF (Clamart), 26-27 avril 1978

[21] Shaw, Mary L., William A. Wulf et Ralph L. London : *Abstraction and Verification in Alphard : Defining and Specifying Iteration and Generators* ; Communications of the ACM, 20, 8, pages 553-564, Août 1977.

[22] Wirth, Niklaus : *Program Development by Stepwise Refinement* ; Communications of the ACM, 14, 4, pages 221-227, Juillet 1971.

[23] (Wirth, Niklaus) : *Libres Questions à Niklaus Wirth* ; Interview dans 01-Informatique, 130, pages 106-112, mai 1979.