Fortran

par **Patrice LIGNELET**
École Nationale Supérieure d'Électronique et de ses Applications (ENSEA)
(division des Administrateurs)
Animateur du groupe Fortran à l'AFNOR

Fortran est le plus ancien (1954) langage de programmation évolué en activité. Il est très utilisé pour les applications requérant des calculs numériques intensifs, domaine où on ne lui connaît pas de concurrent sérieux.

Si la norme Fortran 77 a permis des progrès sensibles dans l'art de (bien) programmer avec Fortran, son absence de réformes profondes le laissait techniquement désarmé pour affronter la dernière décennie du siècle. Conscient de ce danger d'obsolescence, le groupe d'experts chargé de cette nouvelle révision lui a intégré quelques-uns des concepts actuels les plus puissants, comme la modularité (déjà présente en Ada et Pascal Étendu par exemple) et le calcul vectoriel.

Parmi les autres innovations majeures, on peut citer le paramétrage des types numériques, les types cartésiens, les pointeurs, la récursivité, et, le plus visible, une syntaxe libérée du zonage rigide vestige des cartes perforées. En outre, grâce au concept de bloc interface couplé aux modules, elle fiabilise l'exploitation du considérable investissement logiciel représenté par les importantes bibliothèques de sous-programmes disponibles.

Fortran 90 reste entièrement compatible avec la version antérieure (Fortran 77) du langage, ce qui permet d'exploiter l'existant, mais induit la possibilité de deux styles d'écriture des programmes. Toutefois, la nouvelle norme s'insère dans un processus d'évolution à long terme du langage, en dégageant un certain nombre de caractéristiques reconnues comme surannées, et susceptibles de disparaître lors de la prochaine révision de la norme ; le langage n'est donc pas voué à croître indéfiniment.

Ainsi dynamisé par ces apports novateurs, Fortran 90 demeure plus que jamais irremplaçable pour l'ensemble de la programmation numérique, scientifique et technique. Dès lors, on doit également souhaiter qu'il reprenne toute sa place dans la formation initiale de nos futurs ingénieurs et techniciens.

1. Structure générale du langage

1.1 Exemple de programme Fortran 90

Le programme de la figure 1 détermine les principales statistiques sur un échantillon d'une variable qualitative (par exemple, une couleur). Il illustre essentiellement les aspects vectoriels de Fortran 90, notamment les constructeurs de tableaux, la transmission en paramètre par descripteur, des fonctions prédéfinies variées (telles MAXLOC ou COUNT), etc. Il présente aussi le MODULE comme outil d'exploitation fiable d'un service logiciel (ici la procédure Q_STATS).

1.2 Architecture des programmes

Fortran appartient à la famille des langages impératifs ; il est (en principe) compilé. Un programme Fortran se présente matériellement sous la forme d'un ou plusieurs fichiers de texte source, chacun comportant un certain nombre de composants de l'application, ou unités de compilation : modules, procédures externes, blocs de définition de données (BLOCK DATA), ainsi qu'un et un seul programme principal. La seule contrainte d'ordre à respecter est que tout module soit compilé avant son utilisation par une autre unité.

Le programme exécutable est créé par l'édition des liens, comme dans tout langage compilé, à partir du programme principal et des unités appelées, en utilisant s'il y a lieu d'autres composants antérieurement compilés et rangés en bibliothèque.

Procédures externes et BLOKC DATA sont des vestiges de l'ancien style Fortran : le module devrait désormais constituer l'unique outil de décomposition, exportant diverses ressources logicielles, dont des procédures (figure 1). Un programme Fortran 90 typique comportera donc un programme principal, et n ($\geqslant 0$) modules, répartis en un ou plusieurs fichiers-sources (le programme principal étant alors la dernière unité de compilation du dernier fichier-source compilé).

Un programme principal a la structure suivante :

```
[PROGRAM identificateur]
      [déclarations]
      instructions exécutables
[CONTAINS
      procédures internes]
END [PROGRAM [identificateur]]
```

1.3 Structure lexicale

Fortran s'écrit à l'aide d'un jeu de caractères qui est un sous-ensemble de l'alphabet ASCII (*American Standard Code for Information Interchange*) :

— les caractères alphanumériques, qui comprennent les 26 lettres (majuscules et minuscules étant équivalentes, sauf dans un libellé ou chaîne de caractères), les 10 chiffres décimaux et le blanc souligné (_) ;

— 19 caractères spéciaux ou de ponctuation, qui sont, outre l'espace typographique :

= + – * / ( ) , . ' : ! " % & ; < >

```
> f90
f90 compiler Version 1.1 NAGware f.90

 1 ! Ce module exporte une procedure calculant des statistiques
 2 ! elementaires sur un echantillon d'une variable qualitative.
 3 ! Les modalites de la variable sont supposees numerotees de
 4 ! LBOUND (listeff) a UBOUND (listeff).
 5
 6 ! Donnees : le vecteur LISTEFF des effectifs par modalite (classe).
 7 ! Resultats fournis :
 8 !     EFF_MOY  = eff. moyen des classes non vides (0 si toutes vides).
 9 !     MODES    = liste des modes de la distribution.
10 !     NB_MODE = nombre de modalites modales (0 si LISTEFF est vide).
11 ! N.B. : Les classes modales sont les NB_MODE premiers elements
12 ! de MODES.
13 MODULE STATS_QUALITATIVES
14 CONTAINS
15
16   SUBROUTINE Q_STATS (LISTEFF, EFF_MOY, MODES,
17                       NB_MODE)
18         INTEGER, INTENT (IN)  :: LISTEFF (:)
19         INTENT (OUT)          :: EFF_MOY, NB_MODE
20         INTEGER, INTENT (OUT) :: MODES (:)
21
22     IF (SIZE (LISTEFF) > 0) THEN      ! vecteur non vide.
23       MODE   = MINVAL (MAXLOC (LISTEFF) )
24       MODEFF = LISTEFF (MODE)
25       NB_MODE= COUNT (LISTEFF==MODEFF)
26       MODES  = RESHAPE (PACK ((/                                  &
27                          (I, I= MODE, UBOUND (LISTEFF , 1) ) /) ,  &
28                           Mask= LISTEFF (MODE:) == MODEFF) ,       &
29                        SHAPE (MODES) , pad= LBOUND (LISTEFF) -1)
30     ELSE
31       NB_MODE= 0
32     ENDIF
33     NON_VIDES= COUNT (LISTEFF > 0)
34     IF (NON_VIDES > 0) THEN
35       EFF_MOY= SUM (LISTEFF, Mask= LISTEFF >0) / REAL (NON_VIDES)
36     ELSE
37       EFF_MOY= 0 . 0
38     ENDIF
39   END SUBROUTINE Q_STATS
40
41 END MODULE STATS_QUALITATIVES
42
43
44 ! Programme de test du module precedent.
45
46 USE STATS_QUALITATIVES
47 INTEGER, PARAMETER: : LISTE (30) =                                 &
48            (/157, 34, 142, 72, 255, 124, 30, 28, 0, 255, 3, 0, 17, 123,  75, 124, 79, &
49              38,  4, 0, 255, 113, 138, 30, 4, 5, 24, 3, 21, 0/)
50 DIMENSION MODES (5) , IDATE (8)
51
52 CALL DATE_AND_TIME (Values=IDATE)
53 PRINT 10, IDATE (3 : 1 : -1)
54 PRINT*
55 CALL Q_STATS ( (LISTE, EFF_MOY, MODES, NB_MODES)
56 PRINT* , "Effectif moyen des classes non vides =" , EFF_MOY
57 PRINT* , "Il y a", NB_MODES, "classes modales."
58 PRINT* , "Ce sont", MODES ( : NB_MODES)
59 10 FORMAT ("Execution le", 2 (12, '/'), 14)
60  END PROGRAM

> run
 Execution le 13/ 5
 Effectif moyen des classes non vides = 82. 7692337
 Il y a 3 classes modales.
 Ce sont :  5   10   21
> logout

        Les numéros des lignes sont extérieurs au programme.
```

Figure 1 – Exemple de programme vectoriel

Ces caractères servent à construire les éléments lexicaux, qui sont les plus petites entités interprétables des programmes. On distingue les **mots-clés**, prédéfinis (qui ne sont pas réservés en Fortran), en nombre assez élevé (de l'ordre de 75) ; ils définissent les caractéristiques de données (ex : INTEGER, INTENT, etc.), précisent le rôle d'une instruction (ex : IF, CALL), etc.

Les **identificateurs** sont les noms choisis par le programmeur pour désigner ses propres entités (ex. STATS_QUALITATIVES, Q_STATS, LISTEFF, etc.). Un identificateur est formé d'une suite de 1 à 31 caractères alphanumériques dont le premier est une lettre. Certains identificateurs sont aussi prédéfinis : ils désignent des procédures Fortran (ex : SIZE, MAXLOC, DATE_AND_TIME, etc.).

Les éléments lexicaux comprennent encore les opérateurs, et les constantes (numériques, logiques ou chaînes de caractère).

L'**instruction** est la phrase élémentaire d'un programme ; elle regroupe diverses entités lexicales. Toute unité de compilation comporte des instructions de déclaration, et des instructions exécutables. En Fortran 90, il y a une relative indépendance entre l'instruction et la ligne de texte qui lui sert de support dans un fichier source. Plusieurs instructions courtes peuvent apparaître sur une même ligne, séparées par un point-virgule. Une instruction peut s'écrire sur plusieurs lignes, en terminant toutes les lignes (sauf la dernière) par un caractère &. Le cas échéant, la coupure peut intervenir au milieu d'une entité lexicale ; on doit alors écrire un second & juste avant le premier caractère de l'entité sur la ligne suite :

```
PRINT *," Ce message est peut-et&
        &re un peu long..."
```

Les **commentaires** sont de type fin de ligne (comme en Ada), introduits par le caractère point d'exclamation : le commentaire s'étend depuis le ! jusqu'à la fin de la ligne. Ex : figure 1, lignes 1 à 11, 21, etc. Un commentaire peut apparaître derrière un & à la fin d'une ligne initiale d'une instruction répartie sur deux ou plusieurs lignes.

L'ancien style d'écriture du texte source (avec découpage des lignes en zones) reste utilisable, avec des améliorations héritées du nouveau style. On note que, dans ce nouveau style, l'espace est séparateur, ce qui rend l'analyse lexicale des programmes plus efficace.

Le compilateur doit bien sûr être averti du style d'écriture utilisé dans un fichier-source. Cela se fera par une option de compilation, un suffixe spécifique dans le nom du fichier, etc.

2. Types et opérateurs

Tout programme manipule de l'information, qui peut revêtir un certain nombre de formes simples prédéfinies dans un langage, ou être construite par composition de telles formes simples : ce sont les types de données, caractérisés pour l'ensemble des valeurs possibles, et les opérateurs définis sur ces valeurs.

2.1 Types scalaires prédéfinis

Ils sont au nombre de 5 en Fortran : entier, réel, complexe, logique et caractère. Ils sont tous paramétrables par un paramètre de type (KIND) pour en définir diverses variantes ou sous-types :

nom_de_type (KIND = paramètre de sous-type)

Il existe dans chaque cas un sous-type privilégié, correspondant à la valeur par défaut du paramètre KIND pour ce type, et désignant le type du Fortran 77 (pour des raisons de compatibilité ascendante).

Réciproquement, le paramètre de type d'un objet x s'obtient par la fonction prédéfinie KIND (x). L'ensemble des paramètres KIND disponibles dépend du compilateur.

Le sous-type d'une constante s'écrit en suffixant sa valeur par le paramètre KIND voulu (sous forme d'une constante) ; ex : 347_INT_SHORT avec la déclaration préalable :

Exemple

```
PARAMETER (INT_SHORT = 2)
```

■ Le type **entier** (INTEGER) est un intervalle fini de l'ensemble mathématique des entiers relatifs. On peut choisir de manière portable le sous-type voulu en spécifiant le nombre minimal r de chiffres décimaux requis, en prenant comme valeur du paramètre KIND celle de la fonction prédéfinie :

```
SELECTED_INT_KIND (r)
```

Réciproquement, la fonction prédéfinie RANGE (x) fournit le nombre maximal de chiffres décimaux autorisés par le sous-type (entier) de l'objet x.

■ Le type **réel** (REAL) est une approximation de l'ensemble mathématique des réels, formée de nombres rationnels (y compris 0) de la forme :

$$m \cdot b^e$$

où m (la mantisse) et e (l'exposant) sont des entiers relatifs, et b = 2 ou 16 en pratique selon l'architecture de la virgule flottante disponible.

On peut choisir de manière portable un sous-type réel en spécifiant la précision p souhaitée, et/ou l'intervalle des valeurs représentables désigné par r (pour 10^{-r} à 10^{r} en valeur absolue), hors 0, en prenant comme valeur du paramètre KIND celle de la fonction prédéfinie :

```
SELECTED_REAL_KIND (p, r)
```

L'ancien type DOUBLE PRÉCISION correspond à la nouvelle spécification :

```
REAL (Kind = KIND (0D0))
```

où l'objet 0D0 représente le réel zéro dans ce sous-type ($= 0 \times 10^{0}$)

■ Le type **complexe** (COMPLEX) correspond au produit cartésien de deux types réels identiques : les composantes réelle et imaginaire. Un sous-type se définit donc par le même paramètre de type que pour les réels. Les constantes complexes se dénotent par une paire de nombres entre parenthèses ; exemple : (0 , 1.0).

Les trois types numériques partagent le même ensemble d'opérateurs, dénotés classiquement, par ordre de priorité algébrique croissante :

+ et – (unaires ou binaires)
* et /
* * (exponentiation)

Ces opérateurs polymorphes agissent pour chaque type selon ses règles propres ; c'est ainsi que par exemple 8 / 5 donne l'entier 1 (troncature vers zéro).

■ Le type **logique** ou booléien (LOGICAL) correspond aux deux valeurs de vérité de la logique binaire, dénotées .FALSE. et .TRUE.. Le paramétrage de ce type correspond seulement à des encombrements restreints, et dépend du compilateur. Par défaut, l'encombrement est d'un mot numérique (comme pour les types « par défaut » INTEGER et REAL, le type COMPLEX correspondant à deux mots). Les opérateurs logiques sont les suivants, par ordre de priorité croissante :

.EQV. et .NEQV. (équivalence et ou exclusif respectivement)
.OR.
.AND.
.NOT.

■ Le type **caractère** (CHARACTER) correspond au jeu de caractères disponibles sur la machine cible (celle sur laquelle le programme s'exécute). Le paramètre de type KIND pour les caractères est une exigence asiatique ; il peut aussi donner accès à des jeux de caractères mathématiques, chimiques, musicaux...

En fait, ce type correspond en Fortran aux chaînes de caractères : il admet donc un second paramètre : LEN, qui définit la longueur de la chaîne (vide le cas échéant) ; un caractère est une chaîne de longueur 1 (valeur par défaut du paramètre LEN).

Les constantes chaînes s'encadrent au choix par des guillemets (") ou des apostrophes. Le type caractère admet un opérateur interne : la concaténation, notée //.

On peut accéder à une sous-chaîne d'une chaîne par la notation : chaîne (a : b) ; exemple : "0123456789" (N : N). Si a est omis, il vaut 1 ; si b est omis, c'est la longueur de la chaîne.

Par ailleurs, Fortran dispose d'opérateurs de comparaison, dont le résultat est du type logique par défaut, admettant deux noms distincts :

<	<=	==	/=	>=	>
.LT.	.LE.	.EQ.	.NE.	.GE.	.GT.

Ils sont applicables à deux opérandes d'un type scalaire quelconque, sauf complexe (qui n'admet que == et /=) et logique.

2.2 Types structures (ou « dérivés »)

C'est un type construit correspondant à un produit cartésien d'ensembles (champs) hétérogènes. Il est considérés comme scalaire par opposition aux tableaux, bien qu'il puisse contenir des champs tableaux. Ce type s'appelle article en Pascal, et enregistrement en Cobol. Le nom de type dérivé adopté par la norme est regrettable, car il reprend celui d'un concept du langage Ada qui n'a aucun rapport.

Un type structure se définit par une énumération de champs, chacun ayant un nom particulier et ses propres caractéristiques.

Exemple

```
TYPE client
     INTEGER CODE
     CHARACTER (LEN = 50) nom
     CHARACTER (30)       ville
     INTEGER  code-postal,          &
              solde
END TYPE client
```

Un objet de ce type se déclare ainsi :

```
TYPE (client) quidam
```

En Fortran 90, les structures ne sont pas paramétrables, quant à leur forme (comme en Pascal, en C ou en Ada), mais on peut dimensionner un champ POINTER sur tableau (§ 2.4). On peut obliger le compilateur à implémenter les champs dans l'ordre spécifié en dotant le type de l'attribut SEQUENCE :

```
TYPE client ; SEQUENCE
     INTEGER code, etc.
```

L'accès à un champ d'un objet structure se dénote à l'aide du caractère %.

Exemple

```
quidam % nom
```

Une valeur structure se construit en énumérant les valeurs des champs dans l'ordre.

Exemple

```
quidam = CLIENT (1, "Dupont", "Paris", 75001, 0)
```

2.3 Tableaux

Un tableau est une collection de données homogènes. Cette homogénéité a permis de bâtir des architectures matérielles massivement parallèles, dites vectorielles, pour appliquer simultanément une même opération à tout ou partie d'un tableau. Fortran 90 s'adapte à ce progrès technologique, en traitant le tableau comme un objet de « première classe » (comme en PL/1), escorté d'une panoplie d'outils favorisant le calcul vectoriel.

En Fortran, la structure de tableau est une propriété de l'objet (et non un type) ; elle se définit par les caractéristiques suivantes :

— son **rang**, ou nombre de dimensions ($\leqslant 7$ en Fortran) ; par convention, tout scalaire est de rang 0 ;

— chaque dimension possède un **intervalle de définition**, donné par deux entiers, et une **étendue**, qui est la longueur de cet intervalle (bornes comprises). En Fortran 90, une étendue peut être nulle, le tableau étant alors vide ;

— la **taille** d'un tableau est le produit de ses étendues. Un tableau vide est de taille 0 ;

— son **profil** est le vecteur (tableau de rang 1) de ses étendues. Le profil d'un scalaire est le vecteur vide. Deux tableaux sont dits **compatibles** s'ils ont le même profil ; par convention, un scalaire est compatible avec tout tableau.

Fortran 90 permet de définir des tableaux à profil explicite (constant ou calculable), et à profil différé (tableaux dynamiques).

Dans le cas d'un profil explicite, la structure du tableau est indiquée dans un **attribut de dimension** par la liste des intervalles de définition de ses diverses dimensions :

$$(i_1 : s_1 , i_2 : s_2 , \ldots , i_n : s_n) \quad \text{avec} \quad n \leqslant 7$$

la borne inférieure (i_n :) pouvant être omise lorsqu'elle vaut 1.

Exemple

```
PARAMETER     (N = 5, M = 10)
DIMENSION T3  (N , 0  : M , N + M)
```

Un tableau à profil différé est défini dynamiquement ; son attribut de dimension n'en précise que le rang, par une liste de caractères « : ». Il reçoit en outre l'attribut ALLOCATABLE (à rapprocher de CONTROLLED en PL/1).

Exemple

```
REAL, ALLOCATABLE : : MAT (: , :)        ! rang = 2
```

Dans le cours du calcul, une fois les dimensions déterminées, le tableau est « créé » avec les dimensions voulues par une instruction ALLOCATE (§ 4.1).

L'accès à un élément de tableau s'opère à l'aide d'une liste d'indices (expressions entières quelconques) en nombre égal au rang du tableau.

Exemples

```
MAT (i , j + k)
```

ou

```
T3 (1 , i * j , k/i + 2)
```

On peut aussi accéder à un **sous-tableau**, de deux manières :

— avec une notation par **triplet**, correspondant à une **pseudo-boucle** qui dénote une suite d'indices en progression arithmétique.

Exemple

```
T3 (2 : N , M : 1 : – 1 , 1 : M : 2)
```

La notation a : b : r représente la progression de premier terme a et de raison r extraite de l'intervalle a : b. La raison par défaut est 1 (équivalent à la notion de tranche en Ada). Des notations simplifiées existent, pour désigner tout le début de l'intervalle de définition dans une dimension (: b), toute la fin (: a), ou l'intervalle en entier (:) ;

— avec un **indice vectoriel**.

Exemple

```
INTEGER, DIMENSION (4) : : INDEX = (/ 2, 0, 1, 10/)
DIMENSION VECT (0 : 20)
```

alors VECT (INDEX) désigne le vecteur :

```
(/ Vect (2), Vect (0), Vect (1), Vect (10) /)
```

Un indice vectoriel peut contenir des valeurs en double, avec certaines limitations quant à leur emploi, lorsque l'opération entraîne une sémantique ambiguë.

2.4 Objets pointés

Un objet pointé est créé dynamiquement par le programmeur, et reste sous son contrôle. Un tel objet se définit par son **modèle** (type + rang le cas échéant), et par une variable (POINTER), qui recevra l'**adresse** de l'objet une fois créé.

Exemple

```
INTEGER, POINTER : : P          ! pointeur vers un objet entier
TYPE maillon                    ! élément d'une liste chaînée
     INTEGER valeur
     TYPE (maillon), POINTER : : succ
END TYPE maillon
TYPE (maillon), POINTER : : premier, temp
REAL, POINTER : : PVECT (:) ! pointeur vers un vecteur
                            ! réel
```

Fortran 90 laisse une certaine ambiguïté dénotationnelle : l'identificateur déclaré avec l'attribut POINTER désigne tantôt le pointeur, tantôt l'objet pointé, selon le contexte syntaxique ; ainsi :

```
temp = maillon (10, premier)     ! objet pointé
temp => premier                  ! pointeur
```

De même, le champ Succ du maillon d'adresse Premier se dénote (à la manière d'Ada) : Premier % Succ.

La constante (valeur symbolique) adresse notée NIL en Pascal et NULL en PL/1 ou Ada s'obtient par une instruction sur pointeur (s) en Fortran 90 :

```
NULLIFY (temp)      ! temp : = NIL en Pascal.
```

La création dynamique d'un objet pointé se réalise par le même ordre ALLOCATE que pour les tableaux à profil différé.

Exemple

```
ALLOCATE (temp, pvect (0 : 10))
```

Fortran 90 permet encore de redéfinir dynamiquement une variable statique ou automatique, voire un paramètre, qui doit alors recevoir l'attribut TARGET dans sa déclaration (pour permettre au compilateur d'optimiser la gestion des autres variables).

Exemple

```
TARGET : : CIBLE (10, 10)
```

Un pointeur est associé dynamiquement à une cible par une affectation d'adresse (=> ; § 4.2.1).

3. Instructions de déclaration

3.1 Typage des objets

Il existe deux sortes d'objets (ou données) : les constantes, et les variables. Les variables se désignent toujours par un identificateur. Les constantes peuvent aussi le cas échéant être nommées (§ 3.2, attribut PARAMETER). Les objets nommés (constantes et variables) doivent être typés en Fortran. Ce typage s'opère soit implicitement,

d'après la lettre initiale du nom de l'objet ; soit explicitement, au moyen d'une déclaration de type de l'objet. Par défaut, tout objet nommé non déclaré est du type entier si son nom commence par une lettre de I à N, et réel sinon.

Le typage implicite peut être modifié au moyen d'une instruction de déclaration IMPLICIT associant un ensemble de lettres à une spécification de type. Par défaut, on a ainsi :

IMPLICIT INTEGER (I-N), REAL (A-H, O-Z)

Au contraire, si l'on veut proscrire tout typage par défaut (pour des logiciels hautement critiques par exemple), on peut utiliser la déclaration :

IMPLICIT NONE

Une déclaration IMPLICIT se place en premier lieu dans une unité de programme (après d'éventuelles déclarations d'importation USE : § 6).

3.2 Les deux syntaxes orthogonales de déclaration

En Fortran 77, un objet pouvait avoir au plus quatre propriétés. Aussi existait-il une instruction de déclaration spécifique pour chacune :

— une déclaration de type explicite le cas échéant ;
— PARAMETER pour les constantes nommées ;
— DIMENSION pour les tableaux ;
— SAVE pour les variables rémanentes (statiques) ;
— DATA pour initialiser les variables (incompatible avec PARAMETER).

Fortran 90 distingue une dizaine de propriétés (dont certaines incompatibles). Plutôt que d'émietter la déclaration d'une entité entre autant d'instructions de déclaration, on peut désormais, à côté du style précédent (une déclaration par propriété), regrouper les propriétés d'une même entité dans une déclaration de type explicite. Ainsi, les deux systèmes de déclarations suivants sont équivalents :

1)
```
LOGICAL BOOL
DIMENSION BOOL (5)
SAVE BOOL
```

2)
```
LOGICAL, DIMENSION (5), SAVE : : BOOL
```

En Fortran 90, une déclaration complète se présente donc ainsi :

```
spécification de type [, liste d'attributs]                    &
                    : : identificateur [= valeur] {, id [=v]}
```

où la spécification de type (scalaire) prend l'une des formes indiquées au paragraphe 2.1.

Nota : les entités entre crochets sont d'un usage facultatif, les entités entre accolades peuvent figurer zéro ou plusieurs fois.

3.3 Liste des attributs de déclaration

L'attribut est (ou commence par) un mot-clé décrivant une propriété de l'objet. En sus du type, on peut ainsi utiliser :

PARAMETER	pour une constante nommée ;
DIMENSION (....)	pour un tableau ;
SAVE	pour une variable rémanente ;
ALLOCATABLE	pour un tableau à profil différé ;
POINTER	pour une variable accessible par pointeur ;
TARGET	pour une variable redéfinissable par pointeur ;
INTENT (....) / OPTIONAL	pour un paramètre de procédure ;
PUBLIC / PRIVATE	pour une entité déclarée dans un module ;
EXTERNAL / INTRINSIC	pour une procédure transmise en paramètre

Remarque : l'attribut DIMENSION peut être remplacé par un attribut de dimension (§ 2.3) placé à la suite du nom de l'objet.

Exemple

```
LOGICAL, SAVE : : BOOL (5)
```

4. Charpente algorithmique des programmes

Toute unité procédurale (programme principal ou procédure) comprend deux parties : une section déclarant les entités locales à l'unité, et une séquence d'instructions exécutables exprimant la suite des calculs à mener pour résoudre un problème donné. Ces instructions comprennent les instructions simples, et des instructions composées traduisant la structure algorithmique de l'unité.

Ces diverses instructions font intervenir des expressions, au sens mathématique du terme, que l'on présente en premier lieu.

4.1 Expressions

Une expression exprime un calcul, et possède une valeur, résultat de ce calcul lorsque l'expression est évaluée. Elle prend la forme d'une constante, d'une variable (la valeur étant celle de la variable au moment de l'évaluation), d'un sous-objet (d'une constante ou d'une variable tableau, structure ou chaîne de caractères), d'un constructeur de tableau ou de structure (§ 2.2 et § 2.3), d'un appel de fonction (prédéfinie ou non), d'une opération portant sur des facteurs qui sont des expressions, ou d'une expression entre parenthèses.

Dans le cas d'une expression combinant plusieurs opérateurs, l'ordre d'évaluation repose sur des règles de priorité entre opérateurs, comme en algèbre :

$$a + b / c - d \text{ est ainsi évalué comme } a + \frac{b}{c} - d$$

Nous avons donné les priorités relatives des opérateurs pour chaque type de données (§ 2.1). Ces catégories d'opérateurs sont eux-mêmes hiérarchisés ainsi, selon l'ordre croissant des priorités :

1) opérateurs logiques
2) concaténation
3) opérateurs de comparaison
4) opérateurs arithmétiques.

Lorsqu'une opération arithmétique ou de comparaison porte sur 2 facteurs de types numériques différents, il y a conversion de l'un des facteurs en une valeur aussi semblable que possible du type de l'autre facteur, selon l'ordre hiérarchique croissant des types :

1) entier 2) réel 3) complexe

Exemple : (0 , 1.0) + 2 donne le complexe (2.0 , 3.0).

Lorsqu'une opération porte sur deux facteurs de même type mais de paramètres KIND différents, le facteur du sous-type le moins exigeant est converti en une valeur de l'autre sous-type.

Exemple : 4.1 + 3.2 d 0 donne le réel 7.3 d 0 en double précision.

Enfin, si les deux facteurs sont scalaires, le résultat de l'opération est scalaire. Si l'un est un tableau et l'autre un scalaire, le résultat est un tableau de même profil que le facteur tableau.

Exemple

(/ (I , I = 1, 3) /) + 2 donne le vecteur (/ 3, 4, 5 /)

Si les deux facteurs sont des tableaux, ils doivent être compatibles (§ 2.3), le résultat étant un tableau du même profil.

Exemple

INTEGER, PARAMETER : : INDEX (0 : 3) = (/ 1, 0, 2, 4 /)

alors :

INDEX (0 : 2) + INDEX (1 : 3) donne le vecteur (/1, 2, 6 /)

Il existe deux catégories d'expressions utilisables dans les déclarations, et soumises à certaines restrictions spécifiques :

— les expressions constantes d'initialisation, utilisables pour initialiser les variables dans une déclaration de type, ou définir les constantes nommées.

Exemple

PARAMETER (MAXI = 50, IRESTE = MOD (MAXI, 6))

— les expressions de spécification (scalaires de type entier), définissant les bornes des tableaux ou les longueurs des chaînes automatiques.

Exemple

SUBROUTINE SSP (N, C)
CHARACTER (LEN = *) C ; DIMENSION IV (N + LEN (C))

4.2 Instructions simples

4.2.1 Affectation

L'affectation donne une nouvelle valeur à tout ou partie d'une variable (scalaire ou tableau). Il y a deux sortes d'affectations en Fortran 90.

■ L'**affectation de valeur classique**, présente dans tous les langages impératifs :

(sous-)variable = expression

où la sous-variable est, le cas échéant, une sous-chaîne, un champ de structure, ou un sous-tableau. L'expression est d'abord évaluée, puis sa valeur donnée à la (sous-)variable, qui devient ainsi définie le cas échéant. Expression et variable doivent être du même type (pas obligatoirement du même sous-type sauf pour les caractères), ou tous deux numériques ; dans ce dernier cas, la valeur de l'expression est convertie avant affectation si les types diffèrent.

Si la (sous-)variable est scalaire, l'expression doit l'être aussi ; si c'est un tableau, l'expression doit être compatible (§ 2.3).

■ L'**affectation d'adresse**, qui s'écrit :

(sous-)variable pointeur => cible

où la cible peut être une valeur d'adresse (pointeur, ou fonction à résultat POINTER), ou un (sous-)objet ayant l'attribut TARGET.

Exemples (§ 2.4) :

PREMIER => TEMP
PVECT => CIBLE (: , 1)

4.2.2 Appel de sous-programme

Les sous-programmes, prédéfinis ou non, peuvent être considérés du point de vue de l'utilisateur comme des super-instructions spécialisées. En Fortran, on les appelle à l'aide d'une instruction CALL (comme en PL/1) :

CALL < nom du sous-programme > [(liste des arguments d'appel)]

Exemple : figure 1, lignes 52 et 55.

Les traitements prédéfinis qui, dans d'autres langages comme Pascal ou Ada, se réalisent par l'appel de sous-programmes, se traduisent en Fortran par des instructions spécialisées (entrées-sorties notamment, § 7).

4.2.3 Instruction de branchement GOTO

L'enrichissement algorithmique de Fortran 90 devrait limiter considérablement son emploi. Elle s'écrit :

GOTO étiquette

où l'étiquette est un entier positif préfixant l'instruction visée.

4.2.4 Instructions de retour

L'instruction (facultative) RETURN dans une procédure permet de quitter cette procédure pour revenir au point d'appel.

L'instruction STOP arrête l'exécution du programme et redonne la main au système d'exploitation. Le mot-clé STOP est suivi facultativement d'une chaîne de caractères, alors affichée par le système.

4.2.5 Instructions pour l'allocation dynamique

Fortran 90 introduit trois instructions pour l'allocation dynamique :

NULLIFY (liste de (sous-)variables pointeurs) ! (§ 2.4)
ALLOCATE (liste de (sous-)objets [, STAT = variable scalaire entière])
DEALLOCATE (liste de (sous-)objets [, STAT = variable scalaire entière])

Ces deux dernières instructions permettent de gérer l'existence de tableaux ALLOCATABLE, comme d'objets pointés (dans ce dernier cas, elles correspondent à des procédures prédéfinies en Pascal : new (pseudo-fonction en Ada), et **dispose** respectivement).

Exemples

ALLOCATE (MAT (0 : N **2, N-1 : N+M))
DEALLOCATE (P, MAT, PVECT)

Dans les deux cas, la variable scalaire optionnelle reçoit la valeur 0 si tout s'est bien passé, et une valeur positive sinon.

En dehors de celles-ci, il existe d'autres instructions simples spécialisées pour les entrées-sorties (§ 7), ou liées aux structures algorithmiques (§ 4.3.2).

4.3 Schémas algorithmiques

La programmation structurée repose sur trois schémas de base : l'enchaînement ou séquence, le choix et l'itération ou boucle. Tardivement « civilisés », les schémas algorithmiques du Fortran 90 disposent d'une syntaxe parenthésée qui, à l'instar d'Ada, les dispensent de recourir occasionnellement à une forme syntaxique spécifique pour leur associer une séquence (comme DO ; .... END ; en PL/1, BEGIN .... END en Pascal, ou {....} en C).

De manière générale, Fortran 90 unifie la syntaxe des schémas algorithmiques sur le modèle suivant :

```
[identificateur:] mot-clé [commande]
                          corps du schéma
                  END mot-clé [identificateur]
```

L'identificateur optionnel sert de repère syntaxique, permettant au compilateur de vérifier l'appariement, et améliorant la lisibilité dans le cas de schémas emboîtés.

■ **Remarque** : l'espace entre END et le mot-clé est facultatif.

4.3.1 Schémas décisionnels

Un tel schéma exécute une séquence et une seule (au plus) parmi un ensemble de séquences placées en exclusion mutuelle par le schéma.

L'**alternative**, ou schéma de choix logique, s'écrit :

```
IF (expression scalaire logique) THEN
   séquence d'instructions exécutables
{ELSE IF (expression logique) THEN
         séquence}
[ELSE
         séquence]
ENDIF
```

Les expressions sont évaluées (dans l'ordre), et dès que l'une s'évalue à vrai, la séquence associée est seule exécutée ; si les expressions ont toutes la valeur faux, la dernière séquence (derrière ELSE) est exécutée si elle est présente.

Exemple : figure 1, lignes 22 à 32.

Ce schéma admet une forme simplifiée (appelée IF logique) :

```
IF (expression logique) instruction simple exécutable
```

Fortran 90 introduit une forme particulière de l'alternative pour conditionner des affectations de tableaux :

```
WHERE (expression de tableau logique)
         séquence d'affectations de tableaux
[ELSEWHERE
         séquence d'affectations de tableaux]
END WHERE
```

avec sa forme simplifiée :

```
WHERE (expression de tableau) affectation de tableau
```

Tous les tableaux figurant dans un même schéma doivent être compatibles. L'expression ou filtre est d'abord évaluée. Dans la séquence qui suit WHERE, l'affectation et l'évaluation ont lieu sur les seuls éléments de tableau correspondant à une valeur « vrai » du filtre, et faux pour la séquence qui suit ELSEWHERE s'il y a lieu.

Exemple

```
WHERE (MAT > 0.0) MAT = LOG (MAT) / MAT
```

Le schéma de choix proprement dit (choix discret) s'écrit :

```
SELECT CASE (expression scalaire d'un type discret)
   CASE (sélecteur1)
         séquence1
   {CASE (sélecteur)
         séquence}
   [CASE DEFAULT
         séquence]
END SELECT
```

L'expression (critère du choix) et les sélecteurs sont d'un même type discret (entier, chaîne de caractères, voire logique). Chaque sélecteur est une liste de valeurs simples et/ou d'intervalles fermés ou ouverts de valeurs, chaque valeur étant une expression constante d'initialisation (§ 3.1). L'intersection de deux ensembles de valeurs correspondant à deux sélecteurs doit être vide. Est exécutée l'unique (au plus) séquence dont le sélecteur contient la valeur du critère ; si aucun ne la contient, la séquence DEFAULT est exécutée si elle est présente.

Exemple : (schéma de choix numérique ternaire, ancien IF arithmétique) :

```
SELECT CASE (critère)
      CASE (:– 1)    ; PRINT * , "critère < 0"
      CASE (0)       ; PRINT * , "critère nul"
      CASE (+ 1 :)   ; PRINT * , "critère > 0"
END SELECT
```

4.3.2 Boucles DO

Les boucles permettent de répéter à volonté l'exécution d'une séquence. En Fortran 90, leur écriture subit un profond rajeunissement syntaxique. Elles enrichissent en outre de schémas algorithmiques absents du langage antérieurement.

Il y a deux sortes de boucles DO en Fortran 90 :

■ **Boucle de parcours** :

```
DO variable indice = a, b [, r]
   séquence d'instructions exécutables
END DO
```

donne successivement à l'indice les valeurs de la progression arithmétique de premier terme a et de raison r extraite de l'intervalle [a, b] (§ 2.3, notation par triplet), et exécute la séquence pour chaque valeur (**0 fois si la progression est vide**).

Exemple

```
HARMO     = 0.0
DO I      = 100, 1, – 1
   HARMO = HARMO + 1.0 / I
END DO
```

a, b et r peuvent être des expressions scalaires quelconques de type entier. L'indice doit aussi être scalaire et de type entier ; après sortie normale de la boucle, il est égal à la première valeur de la progression extérieure à l'intervalle (soit 0 dans l'exemple ci-dessus), comme en PL/1 ou en C.

■ **Boucles de recherche** :

```
DO WHILE (expression scalaire logique)
   séquence d'instructions exécutables
END DO
```

qui exécute la séquence tant que l'expression vaut vrai.

Exemple : figure 2, lignes 78-85.

Le test d'arrêt d'une boucle de recherche WHILE est effectué au début de chaque itération. Fortran 90 permet de le placer en un point quelconque du corps de la boucle grâce au schéma (dit boucle n + 1/2 de Dijkstra) :

```
DO
   séquence1
   IF (expression logique test d'arrêt) EXIT
   séquence2
END DO
```

La boucle WHILE correspond à une séquence1 vide, la boucle repeat....until du Pascal (ou do....while de C) à une séquence2 vide.

```
> f90
f90 compiler Version 1.1 NAGware f.90
 1 ! Module exportant un type abstrait Liste lineaire d'entiers.
 2
 3  MODULE LISTES_D_ENTIERS
 4
 5    TYPE, PRIVATE : :  MAILLON              ! type secret.
 6          TYPE (MAILLON) , POINTER : :  SUIVANT
 7          INTEGER VALEUR
 8    END TYPE
 9
10    TYPE LISTE ; PRIVATE    ! type visible a structure secrete.
11          TYPE (MAILLON) , POINTER : :  DEBUT , FIN
12    END TYPE
13
14  CONTAINS
15
16    ! Initialisation d'une liste a l'etat vide :
17
18   SUBROUTINE  L_CREER  (LLE)
19
20          TYPE (LISTE) ; INTENT (OUT) : :  LLE
21
22          NULLIFY  (LLE%Debut) ; NULLIFY  (LLE%Fin)
23    END SUBROUTINE
24
25
26    ! Test de l'etat vide d'une liste :
27
28    LOGICAL FUNCTION  L_VIDE  (LLE)
29
30          TYPE (LISTE) , INTENT (IN) : :  LLE
31
32          L_VIDE= .NOT.  ASSOCIATED  (LLE%Debut)
33    END FUNCTION
34
35
36    ! Ajout d'un entier en fin de liste :
37
38    SUBROUTINE  L_ALLONGER  (LLE , INT)
39
40          TYPE (LISTE) , INTENT (INOUT) : :  LLE
41          INTENT (IN) : :  INT
42
43          TYPE (maillon) , POINTER : : NEU
44
45          ALLOCATE  (NEU)
46          NEU%valeur =INT ; NULLIFY  (NEU%Suivant)
47          IF  (L_VIDE  (LLE) )  THEN
48              LLE%Debut => NEU
49          ELSE
50              LLE%Fin%Suivant => NEU
51          ENDIF
52          LLE%Fin => NEU
53    END SUBROUTINE  L_Allonger
54
55
56    ! Iterateur de recherche :
57    ! Recherche du premier element d'une liste (LLE)
58    ! satisfaisant un predicat  COND.
59    ! TROUVE = "Cet element existe, et est renvoye
60    ! s'il y a lieu dans  LAUREAT"
61
62    SUBROUTINE  L_RECHERCHE  (LLE, COND, TROUVE,          &
63                                        LAUREAT)
64          TYPE (LISTE) , INTENT (IN) : :  LLE
65          LOGICAL, INTENT (OUT))    : :  TROUVE
66          INTEGER, INTENT (OUT), OPTIONAL : :  LAUREAT
67          INTERFACE
68              LOGICAL FUNCTION COND  (INT)
69                  INTENT (IN) : :  INT
70              END
71          END INTERFACE
72
73          TYPE (maillon) , POINTER : :  P
74
75    !.....corps du ssp.....
76
77          P=>  LLE%Debut
78          DO WHILE  (ASSOCIATED  (P) )
79              IF  (COND (P%Valeur) ) THEN
80                  TROUVE= .TRUE.
81                  IF  (PRESENT (LAUREAT) ) = P%valeur
82                  RETURN
83              ENDIFF
84              P = > P%Suivant
85          END DO
86          TROUVE= . FALSE.
87    END SUBROUTINE  L_Recherche
88
89  END MODULE  LISTES_D_ENTIERS

        Les numéros des lignes sont extérieurs au programme.
```

Figure 2 – Exemple de module exportant un type abstrait

Plus généralement, toute boucle peut contenir une instruction EXIT, ou CYCLE, provoquant les effets suivants :

EXIT (instruction de même nom en Ada, et **break** en C) permet de quitter la boucle ;

CYCLE (**continue** en C) permet de passer à l'itération suivante.

Ces deux instructions opèrent sur la boucle la plus interne les contenant ; en cas de boucles emboîtées elles doivent mentionner l'identifcateur-repère préfixant la boucle concernée si ce n'est pas la plus interne.

5. Procédures

5.1 Définitions

Les procédures (ou sous-programmes selon les langages, en Ada par exemple) sont définies dans l'article *Langages de programmation. Introduction* [H 2 000]. Elles concourent à découper une application en unités réduites, plus facilement compréhensibles, et accroissent de ce fait sa maintenabilité.

Une procédure est comparable à un programme principal, tant au plan de sa logique d'exécution, que de sa structure syntaxique. La différence majeure est que le programme est lancé par une commande du système d'exploitation, tandis qu'une procédure s'exécute par appel depuis une instruction Fortran.

Il y a deux catégories syntaxiques de procédures en Fortran :

— les **sous-programmes** (SUBROUTINE), qui définissent une action, et qui sont appelés par une instruction CALL (§ 4.2.2) ; ils correspondent aux procédures de Pascal ou Ada, et aux fonctions de type void en C ;
— les **fonctions** (FUNCTION), qui calculent une valeur scalaire ou tableau (procédure .... returns .... en PL/1), et dont l'appel se place dans une expression.

Une procédure admet un nombre fixe (éventuellement nul) de paramètres (ou arguments formels) permettant un échange d'information avec l'unité appelante.

La figure 1 montre un exemple de sous-programme (Q_STATS, lignes 16-39), la figure 2 d'autres exemples de sous-programmes et un de fonction (L_VIDE, lignes 28-33).

En Fortran 90, une procédure peut en contenir d'autres (§ 5.2). Les procédures peuvent être optionnellement récursives : leur entête doit alors être préfixé du mot-clé RECURSIVE (comme en PL/1).

Fortran offre en outre un grand nombre de procédures prédéfinies (plus d'une centaine) (§ 8).

5.2 Classification des procédures par localisation

Il existe trois catégories de procédures selon leur localisation :

— les procédures **externes** (les seules existant en Fortran 77), formant chacune une unité de compilation ;
— les procédures **de module** (§ 6) ;
— les procédures **internes** à une autre procédure (leur hôte), et qui ne peuvent pas elles-même contenir de procédure interne (contrairement aux langages à structure de bloc comme PL/1, Pascal ou Ada), ni être transmises en argument.

Les procédures externes sont les composants actuels des bibliothèques Fortran. Mais, avec Fortran 90, elles devraient être remplacées par des procédures regroupées en modules, garantissant une réelle fiabilité aux appels.

Syntaxiquement les procédures internes sont regroupées à la fin de leur hôte, dans une section spécifique introduite par le mot-clé CONTAINTS (comme pour un module : figure 2).

5.3 Paramètres et modes de transmission

Fortran admet des paramètres de tout type, y compris procéduraux. L'instruction d'entête de la procédure cite le nom seul des paramètres : leurs caractéristiques et leur mode de transmission se déclarent dans la section déclarative de la procédure.

Fortran 90 reconnaît quatre modes de transmission des données, dont trois, analogues à ceux du langage Ada, se déclarent explicitement par l'attribut INTENT (cf. les diverses procédures des figures 1 et 2) :

— paramètres d'entrée, qui sont des constantes locales à la procédure (comme en Ada), et reçoivent l'attribut INTENT (IN) (c'est le seul mode existant en C) ;
— paramètres résultats, définis par la procédure (qui peut les utiliser ensuite, contrairement à Ada), qui reçoivent l'attribut INTENT (OUT) ;
— les paramètres mixtes (à la fois d'entrée et résultats), qui reçoivent l'attribut INTENT (INOUT).

Le mode de transmission par défaut est le quatrième mode, le seul disponible en Fortran 77 (c'est aussi celui de PL/1) ; retenu par nécessité de compatibilité, il n'offre aucune garantie. C'est l'emploi qui est fait de paramètre par la procédure qui dicte son mode « logique » : s'il est modifié, l'argument d'appel (ou argument réel) doit être une variable (comme dans les nouveaux modes OUT et INOUT), sinon ce peut être une expression quelconque.

Il n'y a pas de mode de transmission explicite pour les paramètres procéduraux. On peut déclarer, comme en Pascal, l'interface formelle d'une procédure paramètre au moyen d'un bloc d'interface (§ 5.5 et figure 2, lignes 67-71).

L'argument associé devra être déclaré EXTERNAL dans la procédure d'appel (ou INTRINSIC si cet argument est une procédure prédéfinie).

Les paramètres d'entrée ne peuvent avoir, comme en Ada, de valeur par défaut explicite, mais Fortran 90 offre un mécanisme plus général : les paramètres optionnels, recevant l'attribut OPTIONAL. La présence ou non d'un argument associé lors d'un appel est détectée par la fonction logique prédéfinie PRESENT.

Exemple

```
SUBROUTINE P (N, X, RES)
    INTEGER, INTENT (IN), OPTIONAL : : N
    REAL, INTENT (OUT), OPTIONAL : : RES
    IF (PRESENT (N)) THEN ; NN = N
                          ELSE ; NN = 0 ! valeur par défaut.
    ENDIF
    IF (PRESENT (RES)) RES = ....
```

Lors de l'appel d'une procédure, l'association des arguments d'appel avec les paramètres utilise les deux procédés énoncés dans l'article *Langages de programmation. Introduction* [H 2 000], comme en Ada :

— positionnel, commun à la plupart des langages ;
— associatif, utilisant les noms des paramètres comme mots-clés locaux ; ce dernier procédé, outre sa grande lisibilité, est souvent indispensable lorsque des paramètres optionnels sont omis.

Exemple

```
CALL P (X = 3.0 * A + B, RES = SORTIE) ! N est omis
```

Il doit y avoir identité de sous-type entre arguments et paramètres appariés. Si le paramètre est un tableau, l'argument doit être un tableau de même rang ; mais le profil du paramètre n'est pas spécifié : il hérite de celui de l'argument (figure 1, lignes 18 et 20) ; il en est de même pour la longueur d'un paramètre de type chaîne de caractères. Un tableau paramètre ne peut pas recevoir l'attribut ALLOCATABLE.

De façon plus générale qu'en Pascal (niveau 1 de la norme ISO), on peut imposer des contraintes aux bornes d'un paramètre tableau, ou à la longueur d'une chaîne, à l'aide d'expressions de spécification faisant intervenir d'autres paramètres.

Exemple

```
SUBROUTINE SSP (U, X, Y)
    DIMENSION U (: , :) ! profil hérité.
    REAL, INTENT (IN), DIMENSION (SIZE (U,1)) : : X, Y
```

5.4 Caractéristiques spécifiques aux fonctions

Le type d'une fonction est soit implicite, soit explicite. S'il est explicite, la spécification de son type peut préfixer l'instruction FUNCTION d'entête (comme en C ; figure **2**, ligne 28), ou figurer dans la section déclarative. Ce type doit être connu au point d'appel, au besoin par une déclaration de type du nom de la fonction.

Une fonction renvoie sa valeur par une variable-résultat qui est normalement le nom de la fonction (figure **2**, ligne 32). Mais on peut aussi suffixer l'instruction FUNCTION par une clause RESULT (variable) : c'est alors cette variable qui doit être utilisée. Ce dernier procédé est obligatoire si la fonction est RECURSIVE, le nom de la fonction étant alors réservé aux appels récursifs.

Il faut noter qu'en Fortran, la variable-résultat est une vraie variable : elle doit être définie, mais peut ensuite être évaluée (contrairement à Pascal).

5.5 Blocs d'interface et applications

Les bibliothèques de procédures externes thésaurisées jusqu'à présent offrent le grand inconvénient de ne pas permettre au compilateur de contrôler la validité des appels (saufs à utiliser un outil logiciel spécialisé). Fortran 90 formalise un procédé pour décrire l'interface formelle d'une procédure : le bloc d'interface, que l'utilisateur peut inclure dans son programme d'appel (ou mieux, dans un module : § 6).

Exemple : figure **2**, lignes 67-71 (pour un paramètre procédural, mais la forme est identique pour toute procédure).

On y définit complètement l'interface formelle d'une ou plusieurs procédures, telle qu'elle figure dans le texte original de la procédure (à de possibles variations syntaxiques près, telles que les noms des paramètres, l'ordre des déclarations, etc.). Ce(s) bloc(s) d'interface se place(nt) dans la section déclarative de l'unité utilisatrice.

Au-delà de leur intérêt en terme de fiabilité, les blocs d'interfaces ont de nombreuses applications.

■ Rendant visible l'interface formelle d'une procédure externe, un bloc d'interface permet d'utiliser l'**appel associatif à mot-clé** (§ 5.2).

■ Définition d'**opérateurs associés à des fonctions unaires ou binaires**.

Exemple

```
INTERFACE OPERATOR (+)
    TYPE (ENSEMBLE) FUNCTION UNION (E1, E2)
        TYPE (ENSEMBLE), INTENT (IN) : : E1, E2
    END
END INTERFACE
```

qui, pour deux ensembles A et B, permet d'appeler la fonction à volonté comme UNION (A, B) ou A + B.

Cette faculté n'est pas, comme en Ada, réservée aux seuls noms d'opérateurs prédéfinis : l'utilisateur peut créer ses propres noms (identificateurs) écrits entre deux points (comme les opérateurs logiques), par exemple : .IN.

■ **Affectation généralisée**

Un sous-programme ayant deux paramètres, dont le premier a le mode de transmission OUT ou INOUT et le second le mode IN, peut être « appelé » par une affectation de valeur au premier paramètre :

```
INTERFACE ASSIGNMENT (=)
    SUBROUTINE CHAR_TO_NUM (N, CH)
            INTENT (OUT) : : N
            CHARACTER (LEN = *), INTENT (IN) : : CH
    END SUBROUTINE
END INTERFACE
```

permet à toute affectation d'une chaîne de caractères à une variable scalaire entière d'être interprétée comme un appel de CHAR_TO_NUM.

■ **Généricité**

Il faut entendre ce terme à la manière de PL/1 : définition d'un nom unique pour la même opération conceptuelle réalisée techniquement par diverses procédures selon le contexte ; en Ada, l'équivalent est assuré par la **surcharge**, ou homonymie naturelle propre à ce langage (la généricité y définit un modèle paramétré à la compilation). Ce concept n'est pas nouveau en Fortran : depuis la norme 77, la plupart des fonctions prédéfinies sont génériques.

Ainsi, pour définir une opération générique d'échange de la valeur de deux variables :

```
INTERFACE ECHANGER ! la procédure générique
    SUBROUTINE I_SWAP (I, J)
        INTENT (INOUT) : : I, J
    END
    SUBROUTINE R_SWAP (U, V)
        INTENT (INOUT) : : U, V
    END
    SUBROUTINE MAT_SWAP (M1, M2)
        REAL, INTENT (INOUT), DIMENSION (: , :) : : M1, M2
    END
END INTERFACE
```

5.6 Macrofonctions

À la manière de C (#define), Fortran permet de définir des macrofonctions, d'intérêt purement local aux unités procédurales qui les définissent, et destinées à écrire une seule fois une expression de structure donnée.

Exemple

```
IMPLICIT LOGICAL (B)
BON_DIM (I, J, K) = K > 0. AND. K < I + J
BON_TRIANGLE (I, J, K) =                                    &
Bon_dim (i, j, k) .and. Bon_dim (j, k, i).and. Bon_dim (k, i, j)
```

6. Modularité

6.1 Présentation

Le module Fortran correspond à la classe introduite par Simula 67, au package (paquetage) de Ada, et à l'unité de même nom en Pascal Étendu et Modula-2. Le concept est présenté dans l'article *Langages de programmation. Introduction* [H 2 000] de ce traité. Son adoption permet à Fortran 90 d'opérer un saut qualitatif fondamental, et rend

obsolètes de fait un certain nombre de caractéristiques antérieures parmi les plus discutées (tel le COMMON).

Le module est une unité de programme non exécutable exportant un ensemble de ressources logicielles de manière fiable. Compilable séparément, il constituera la base des futures bibliothèques de composants Fortran 90, remplaçant à terme (en les intégrant) les procédures externes jusqu'alors seules disponibles.

On distingue généralement quatre emplois du module, par ordre de complexité croissante :

— mise à disposition d'un ensemble de données et de types (élargissant le rôle traditionnel des COMMON et BLOCK DATA associés) ;

— fonction bibliothèque de procédures indépendantes (mais ayant un certain rapport sémantique) ; dans un premier temps, on écrira des modules de cette nature pour exporter des blocs d'interface (§ 5.5) fiabilisant l'utilisation des bibliothèques Fortran existantes. Le module STATS_QUALITATIVES de la figure **1** appartient à cette catégorie ; il exporte une procédure (Q_STATS), rendant visible sont interface formelle ;

— module objet abstrait gérant les transitions d'état d'un objet, à partir d'un état initial ; caché à l'utilisateur, cet objet évolue et est consulté au moyen de procédures exportées par le module ;

— classe proprement dite (au sens des langages à objets) : cette catégorie de module exporte un type abstrait de données dont la structure reste secrète, avec son lot de primitives (procédures) publiques définissant les opérations sur le type. Tel est le module LISTE_D_ENTIERS de la figure **2**, exportant un type abstrait appelé LISTE.

En dehors du premier cas, tout module exporte des procédures dont l'implémentation est cachée. Toutefois, en Fortran 90, le module forme toujours une seule unité de compilation : il n'y a pas séparation entre une partie spécification visible, et une partie implémentation cachée, comme en Pascal Étendu (interface et implémentation) ou en Ada (interface et package body). Cela peut obliger à recompiler une unité utilisatrice alors qu'à spécification (et sémantique) constante, seule la réalisation technique évolue.

Accessoirement, un module Fortran n'a pas de partie optionnelle exécutable (séquence to begin do en Pascal Étendu, séquence finale begin en Ada).

6.2 Entités visibles et privées

Par défaut, toute entité définie dans un module est exportée, donc visible par l'utilisateur du module. Cela correspond à l'attribut par défaut PUBLIC (équivalent d'export en Pascal Étendu) ; il faut alors donner l'attribut PRIVATE à toute entité (type, objet ou procédure) cachée. on peut inverser cette situation en introduisant une instruction de déclaration :

```
PRIVATE
```

qui oblige ensuite à déclarer toute entité exportée avec l'attribut explicite PUBLIC.

Dans le cas des types cartésiens, il existe trois possibilités, comme en Ada :

— le type est entièrement visible, nom et structure (il a, implicitement ou explicitement, l'attribut PUBLIC) ;

— le type est visible, mais sa structure est cachée (type t is private ; en Ada) ; exemple : figure **2**, le type LISTE (lignes 10-12) ;

— le type est entièrement caché (en Ada, défini dans la section private de l'interface, ou dans le corps du module) ; le type reçoit, implicitement ou explicitement, l'attribut PRIVATE ; exemple : figure **2**, le type MAILLON (lignes 5-8).

Remarque : l'équivalent des types Ada limited private n'existe pas en Fortran 90 : l'affectation reste toujours utilisable pour un type privé ; mais le module peut la redéfinir (§ 5.5).

6.3 Importation des ressources d'un module

Un module doit être compilé avant toute unité utilisatrice, ce qui crée un fichier décrivant l'interface visible du module, géré automatiquement par le compilateur.

Fortran 90 confond en une seule déclaration (USE) à la fois l'importation d'un module (clause with en Ada), et la déclaration de visibilité (use en Ada).

L'utilisation la plus simple (correspondant en Ada à with m ; use m ;) est :

```
USE nom_de_module
```

qui importe le module, et donne accès à toutes ses ressources visibles sous leur nom d'origine, sauf en cas de conflit de nom ; on peut néanmoins accéder à de telles entités en les renommant, à la manière de Pascal Étendu.

Exemple

```
USE Liste_d_Entiers, INT_LISTE   => LISTE, L_EMPTY        &
                                 => L_VIDE
```

baptise localement INT_LISTE et L_EMPTY les entités associées du module.

Ce même procédé est à employer si l'on recourt à deux modules exportant deux entités distinctes homonymes que l'on souhaite utiliser. Le mélange with / use est compensé en Fortran par la possibilité de limiter l'accessibilité aux entités visibles d'un module, toujours comme en Pascal Étendu, à l'aide d'une clause ONLY.

Exemple

```
USE Liste_d_Entiers, ONLY : ! équivalent d'un with Ada
.....
CONTAINS
    SUBROUTINE P
    USE Liste_d_Entiers ! équivalent d'un use Ada.
```

La première déclaration USE permet d'accéder au module sans en rien importer ; on peut l'inclure en tête de l'unité importatrice pour des raisons de lisibilité. Si la procédure P est l'utilisatrice effective du module, la seconde déclaration USE lui donne toute la visibilité requise.

Plus généralement, en particulier pour des modules de type bibliothèque, on peut restreindre la liste des entités importées, éventuellement en les renommant.

Exemple

```
MODULE M
    PRIVATE
    PUBLIC:: A, B, C, D
.....
PROGRAM principal
    USE M, ONLY : B, VLOC => A
```

n'importe que B (sous son nom), et A renommé localement VLOC.

La relation d'importation est transitive : un module M2 important une entité d'un module M1 peut la réexporter librement. On peut limiter cette réexportation en rendant PRIVATE dans M2 les noms locaux des entités importées.

7. Entrées-sorties

7.1 Fichiers

Paradoxalement, Fortran, langage scientifique par excellence, dispose d'une assez belle panoplie d'outils pour manipuler les fichiers, encore enrichie par la récente norme.

On distingue deux types de fichiers selon la forme de leur contenu :

— les fichiers de texte, dont les enregistrements (ou composants) contiennent exclusivement des suites de caractères (fichiers de type text en Pascal ; offerts par le paquetage prédéfini text_io en Ada ; créés sous le contrôle d'un format d'édition en Fortran, et en mode stream en PL/1) ;

— les fichiers binaires, dont les enregistrements sont des suites de valeurs quelconques (ou fichiers sans format).

Deux méthodes d'accès aux enregistrements sont disponibles en Fortran :

— séquentielle (offerte en Ada par le paquetage prédéfini séquential_io pour les fichiers binaires ; c'est la seule possible en Pascal ISO) ;

— directe par le rang (paquetage direct_io en Ada pour les fichiers binaires ; type file [....]of.... en Pascal Étendu).

Sur certains systèmes, les fichiers à enregistrements de longueur fixe peuvent admettre les deux méthodes.

En Fortran, tout fichier peut être traité en écriture seulement, en lecture seulement, ou en modification ; c'est un paramètre spécifique (ACTION =) de l'instruction OPEN (§ 7.2) qui détermine l'utilisation possible. En accès séquentiel, on peut aussi allonger un fichier existant (grâce au paramètre POSITION = "append" de l'ordre OPEN).

Dans un programme Fortran, un fichier est désigné par un numéro d'unité logique, qui est un entier positif ou nul, choisi à l'ouverture du fichier. Si le fichier est rémanent, il est connu du système sous un nom externe définissable et accessible en Fortran.

Il existe deux unités prédéfinies, dénotées * ; l'une en lecture (sysin en PL/1, input en Pascal, standard_input en Ada), l'autre en écriture (sysprint en PL/1, output en Pascal, standard_output en Ada). En mode d'exécution dialogué, toutes deux correspondent au terminal de l'utilisateur ; elles désignent toujours un fichier de texte séquentiel.

7.2 Ordres généraux

On peut interroger le système sur un fichier connu par son nom, ou par son numéro d'unité logique (ul), grâce à l'instruction :

INQUIRE (f, liste des paramètres résultats)

où f est FILE = nom ou UNIT = ul.

Les paramètres permettent d'obtenir une information sur l'existence de f (EXIST =), sur les permissions accordées par le système (SEQUENTIAL, DIRECT, (UN)FORMATTED), sur les caractéristiques de l'exploitation en cours (OPENED, RECL, ACCESS, FORM, READ, WRITE, READWRITE et ACTION), sur le nom externe (NAMED, NAME) ou le numéro d'unité logique (NUMBER), plus quelques paramètres spécifiques (BLANK, DELIM, PAD pour les fichiers de texte ; NEXTREC pour un fichier direct ; POSITION pour un fichier séquentiel).

Remarque : certains de ces paramètres correspondent à des fonctions prédéfinies en Ada (OPENED = correspond à is-open, ACTION = correspond à mode, NAME = correspond à fonction de même nom) et en Pascal Étendu (NEXTREC correspond à position).

Fortran 90 introduit une nouvelle forme de cette instruction :

INQUIRE (IOLENGTH = variable entière) liste de variables

où la « liste » décrit la structure des enregistrements d'un fichier binaire ; la « variable » reçoit alors la longueur interne des enregistrements associés (valeur utilisable ensuite pour le paramètre RECL de l'instruction OPEN).

Les unités prédéfinies (*) sont accessibles sans formalité dès le début du programme. Les autres fichiers doivent être ouverts avant leur utilisation et associés à un numéro d'unité logique par une instruction :

OPEN ([UNIT =] ul, liste de paramètres d'entrée)

Les paramètres définissent diverses caractéristiques générales du fichier (FILE, STATUS, FORM, ACCESS, RECL, POSITION, ACTION), et précisent certaines options spécifiques pour les fichiers de texte (BLANK, PAD, DELIM).

Remarque : en accès séquentiel :

OPEN (ul, POSITION = "rewind", ACTION = "write" [,....])

correspond à la procédure prédéfinie rewrite en Pascal ; la procédure reset correspond à ACTION = "read".

Une fois le traitement du fichier terminé, il est recommandé de le fermer par l'instruction :

CLOSE ([UNIT =] ul [, STATUS = action à prendre])

qui correspond aux procédures close ou delete en Ada (selon la valeur de STATUS).

Remarque : toutes les instructions d'entrée-sortie Fortran peuvent en outre comporter les paramètres suivants :

ERR = étiquette de transfert en cas d'erreur ;
IOSTAT = variable scalaire entière (= 0, ≠ 0 en cas d'erreur), dont la logique correspond à celle de STAT pour les instructions ALLOCATE / DEALLOCATE (§ 4.2.5).

7.3 Instructions de lecture-écriture

Les instructions ci-après portent sur un fichier désigné par un numéro d'unité logique. L'**écriture** d'un nouvel enregistrement se réalise par l'instruction :

WRITE (liste de commande) [Liste de données]

et la lecture d'un enregistrement par :

READ (liste de commande) [liste de variables]

En Fortran, les « listes » d'entrée-sortie (données ou variables) ne correspondent pas nécessairement à un objet structure unique comme dans la plupart des langages, même si leur introduction en Fortran 90 doit en généraliser l'emploi dans ce contexte.

Les listes de commande sont de la forme :

[UNIT =] ul [, options]

où les options possibles sont en particulier :

END = étiquette pour gérer la fin du fichier en lecture séquentielle ;
REC = expression pour les fichiers directs (paramètres from et to en Ada).

D'autres options sont spécifiques aux fichiers de texte (§ 7.4).

En accès séquentiel, on dispose encore de l'instruction :

REWIND ([UNIT =] ul)

pour replacer un fichier à son début (effet équivalent au paramètre POSITION = "rewind" de l'ordre OPEN).

Il existe en outre deux instructions techniquement obsolètes : ENDFILE (ul) et BACKSPACE (ul).

7.4 Fichiers de texte

Ils sont définis au paragraphe 7.1. Il existe trois façons de diriger la correspondance entre les données du programme transférées, et les caractères qui les représentent dans les enregistrements :

— par un format d'édition explicite (équivalent au mode edit des ordres get/put en PL/1) ;
— par liste, ou en format libre (dénoté * ; mode list en PL/1) ;
— par noms de données (NAMELIST ; mode data du PL/1).

De plus, contrairement aux fichiers binaires, l'exécution d'un seul ordre READ ou WRITE peut traiter plusieurs enregistrements (appelés **lignes** dans le cas des fichiers de texte) successifs.

Pour les unités prédéfinies, il existe des formes abrégées des ordres READ / WRITE avec format explicite ou libre :

READ format [, Liste de variables]

PRINT format [, Liste de données]! figure 1, lignes 53-58.

7.4.1 Édition avec format explicite

Les instructions READ / WRITE utilisent dans leur liste de commande l'option :

[FMT =] format

et, optionnellement :

ADVANCE = et, en lecture seulement : SIZE = et EOR =

Remarque : l'option ADVANCE = "no" permet d'obtenir un effet équivalent aux procédures prédéfinies read et write de Pascal (sans elle, les ordres Fortran READ / WRITE équivalent à readln et writeln respectivement).

L'indication de format est l'étiquette d'une instruction FORMAT (figure 1, ligne 59), ou une expression de type chaîne de caractères dont la valeur est de la forme :

"(flist)", où flist est une liste d'éléments de la forme :

[r] ed où ed est un modèle d'édition de donnée (répétable)
ou /
ned où ned est un modèle non répétable
(tous autres modèles d'édition) ;

[r] (flist)
r : facteur de répétition (constante entière > 0).

Le principe est de faire correspondre, à chaque donnée éditée, un modèle d'édition de donnée adapté à son type, et à décrire la mise en place des champs édités à l'aide de modèles auxiliaires.

Liste des modèles d'édition de données

● **Remarque :** w désigne le nombre de caractères du champ édité sur la ligne ;

d : nombre de chiffres de la partie fractionnaire ;
e : nombre de chiffres de l'exposant ;
m : nombre minimal de chiffres à afficher ;
s : nombre de chiffres significatifs ;

• pour un entier : Iw, Bw, Ow, Zw, et leurs variantes Iw.m, etc en sortie (pour les bases de numération respectivement 10, 2, 8 et 16),
• pour un réel : Fw.d, Ew.d [E.e], Dw.d, ENw.d [E.e], ESw.d [E.e] ; en lecture, tous sont équivalents ; en sortie, ils permettent d'obtenir diverses variantes de présentation,
• pour une chaîne de caractères : Aw (avec en sortie troncature ou complétion par des espaces de la valeur éditée),
• pour une valeur logique : Lw (T / F représentent les valeurs logiques),
• pour tous types : Gw.s [E.e] où seul w est utilisé en dehors du type réel.

Une valeur complexe est transmise à l'aide de deux modèles réels ; une structure, à l'aide d'autant de modèles que de champs scalaires.

Modèles précisant l'édition numérique

BN, BZ	interprétation des espaces non initiaux en lecture ;
S, SS, SP	traitement du signe + en sortie ;
kP	facteur d'échelle (k constante entière).

Règles générales pour l'édition numérique

— En lecture, les espaces initiaux ne sont pas significatifs. L'interprétation des autres espaces est dirigée par le contrôle d'espace en vigueur (cf. BLANK =, BN, BZ).

— À la lecture d'un réel, un point décimal dans le champ lu supplante l'indication du modèle.

— En sortie, la représentation est cadrée à droite. Si la valeur transmise est trop grande pour être représentable selon le modèle, le processeur produit un champ de w astérisques.

Modèles de positionnement des champs

La transmission du prochain caractère vers ou depuis la ligne courante s'opérera à partir du caractère :

Tc	de rang c sur la ligne ;
TLc	situé c positions avant la position courante ;
TRc ou cX	situé c positions après la position courante.

Autres modèles

"libellé"	en sortie ; transmis tel quel (figure 1, ligne 59) ;
/	provoque la fin du traitement de la ligne courante ;
:	arrête l'exploration du format après épuisement des données.

7.4.2 Édition en format libre

Elle libère l'utilisateur de toute obligation de cadrage des champs en lecture, et des lourdes élaborations de format en sortie. L'indication de format est alors remplacée par * .

Chaque ordre d'entrée-sortie traite un ensemble de caractères placés sur une ou plusieurs lignes, et formant une suite de valeurs : constante ou valeur vide (le cas échéant répétée par un facteur de

répétition r*), séparées par une virgule ou un espace et, en entrée, terminées le cas échéant par /

Exemple de ligne en entrée (**remarque :** 2* répète la valeur vide) :

'Bonjour' 50 * 1.5 (2.5E3, 3.14) 2* ,1 3

7.4.3 Édition dirigée par noms

Poursuivant des buts voisins du format libre, ce mode d'édition vise à accroître la lisibilité des données sur le support externe. Malheureusement, il rompt l'indépendance données-programme. Le principe consiste à associer à chaque type d'article logique du fichier un nom de liste générique, équivalent à un nom de structure, dont les champs sont définis par une déclaration :

NAMELIST / nom d'article / liste des noms des champs

Dans le fichier, les données relatives à un article logique commencent par & nom de l'article, et se terminent par /. Les valeurs des divers champs sont indiquées sous forme d'affectations séparées par des virgules.

Les instructions d'entrée-sortie prennent la forme :

READ ([UNIT =] ul, [NML =] nom d'article)
WRITE ([UNIT =] ul, [NML =] nom d'article)

avec toujours les options facultatives ERR / IOSTAT, et END = en lecture.

Exemple

```
DIMENSION A (9) ; CHARACTER * 20 TEXTE ; COMPLEX Z
NAMELIST / ARTICLE / TEXTE , Z , A
READ (* , NML = ARTICLE)
```

peut lire les lignes suivantes :

```
&ARTICLE TEXTE = "EXEMPLE", Z = (0, 2),
  A (1) = 47, A (4:8) = 5 * 0.6 /
```

7.4.4 Fichiers internes

Fortran permet de considérer une chaîne de caractères ((sous-)objet variable) comme un pseudo-fichier interne d'une ligne, et un (sous-)tableau de chaînes comme un fichier formé d'autant de lignes que d'éléments dans le tableau. Un fichier interne est toujours traité avec format explicite.

Exemple

```
      CHARACTER (12) : : TEXTE
      WRITE (TEXTE , 10) SOMME
10    FORMAT (F12.2, TL3 , ' ,')
```

8. Liste des procédures prédéfinies

Ces procédures font partie intégrante de la définition du langage, et sont donc disponibles sur tous les compilateurs respectant intégralement la norme. Du point de vue de leur mode de calcul, on distingue quatre sortes de procédures prédéfinies en Fortran 90.

■ Les **procédures distributives**, opérant sur des arguments *a priori* scalaires, mais applicables à des tableaux ; les fonctions renvoient alors un tableau de même profil, résultant de la distribution de la fonction aux éléments du ou des arguments.

Exemple : ABS ((/ – 3 , 2 , 7 /)) donne le vecteur (/ 3, 2 , 7 /)

■ Les **fonctions attributs**, ou d'interrogation, qui fournissent une propriété de leur argument principal indépendamment de sa valeur (qui peut même être indéterminée) ; elles correspondent en Ada au concept d'attribut.

■ Les **fonctions de transformation** : toutes les autres fonctions ; elles portent généralement sur des tableaux.

■ Les **sous-programmes non distributifs**.

L'interface d'une procédure prédéfinie est toujours visible, ce qui permet les deux formes d'appel (§ 5.3), et autorise la présence de paramètres optionnels ; lorsqu'ils sont mentionnés, ils apparaissent ici en lettres minuscules.

● **Remarque :** lorsqu'une procédure prédéfinie est transmise en argument (ce qui est possible en Fortran, contrairement à Pascal), la procédure d'appel doit la déclarer INTRINSIC.

Exemple

```
INTRINSIC COS
CALL CALCUL (COS, ....)
```

8.1 Fonctions distributives

8.1.1 Fonctions de conversion

Ces fonctions ont toutes un paramètre optionnel KIND, précisant le sous-type du résultat. Ce sont : INT, REAL (plus l'archaïque DBLE), CMPLX (X, y), LOGICAL.

8.1.2 Fonctions arithmétiques

Elles admettent des arguments d'un type numérique quelconque, le résultat étant du même type. Ce sont :

valeur absolue (ABS), arrondi (NINT et ANINT), troncature (AINT), reste et modulo (MOD et MODULO), extremums (MIN et MAX), transfert de signe (SIGN (A , B)), différence positive (DIM (X, Y)), conjugué (CONJG (Z)).

Les fonctions suivantes opèrent de plus une conversion de type : multiplication en double précision (DPROD), partie entière (FLOOR et CEILING), partie imaginaire (AIMAG (Z)).

8.1.3 Fonctions mathématiques

À un rangement réel ou complexe ; ce sont : racine carrée (SQRT), exponentielle (EXP), logarithmes (LOG, LOG10), fonctions circulaires (COS, SIN, TAN) et leurs inverses (ACOS, ASIN, ATAN et sa variante ATAN2 (Y, X)), fonctions hyperboliques (COSH, SINH, TANH).

8.1.4 Fonctions relatives aux chaînes de caractères

Comparaisons ASCII (LLT, LLE, LGE, LGT), conversion code ↔ caractère (CHAR et ICHAR, et leurs équivalents ASCII : ACHAR et IACHAR), localisation d'une sous-chaîne (INDEX), justification (ADJUSTL et ADJUSTR), longueur hors espaces finaux (LEN_TRIM), contrôle par rapport à un alphabet (SCAN et VERIFY).

8.1.5 Fonctions de traitement des réels

Accès aux composantes du modèle (FRACTION, EXPONENT et sa réciproque SET_EXPONENT), et quelques fonctions très spécialisées (NEAREST, SCALE, SPACING, RRSPACING).

■ **Remarque :** SPACING (X) donne la valeur absolue du dernier bit de la représentation de X, et permet par exemple de mettre en œuvre la méthode de perturbation et permutation de Vignes.

8.1.6 Fonctions de traitement du bit (à arguments entiers)

Opérations logique bit à bit (NOT, IAND, IOR, IEOR), décalages (absolu ISHFT, circulaire ISHFTC), accès au bit (BTEST et ses réciproques IBSET et IBCLR, IBITS).

8.1.7 Fonction de définition conditionnelle

MERGE (TSOURCE, FSOURCE, MASK)

Pour des tableaux, les fusionne sous le contrôle de MASK (de type logique) (cf. l'opérateur ? en C).

8.2 Fonctions-attributs

8.2.1 Fonctions générales

KIND (X) : paramètre de type de l'objet X.
ASSOCIATED (POINTER, target) : test d'association d'un pointeur (équivaut à la comparaison de pointeurs dans les autres langages).
PRESENT (A) : test de présence d'un argument optionnel.
LEN (STRING) : longueur physique de la chaîne argument.

8.2.2 Caractéristiques des types numériques

■ **Externes**

EPSILON, PRECISION, HUGE (+∞), TINY (plus petit réel > 0), RANGE (réciproque de r dans une fonction SELECTED_t_KIND).

■ **Internes**

BIT_SIZE pour un entier, MINEXPONENT et MAXEXPONENT pour un réel et, pour les deux types : DIGITS (nombre de chiffres significatifs ; attribut de même nom en Ada), et RADIX.

8.2.3 Fonctions relatives aux tableaux

ALLOCATED (ARRAY)	test d'existence d'un tableau ALLOCATABLE (§ 2.3).
LBOUND (ARRAY, dim)	et UBOND (ARRAY, dim) bornes du tableau.
SHAPE (SOURCE)	donne le profil de son argument.
SIZE (ARRAY, dim)	donne une étendue (la taille si dim est absent) du tableau.

8.3 Fonctions de transformation

8.3.1 Fonctions sur tableaux

Un certain nombre de ces fonctions ont un paramètre optionnel :

DIM (entier scalaire)	spécifiant une dimension (1 ⩽ dim ⩽ rang du tableau argument).
MASK (logique)	filtre d'application de la fonction.

■ **Application d'une opération aux éléments d'un tableau**

MINVAL et MAXVAL	éléments extrêmes ;

avec dans le même ordre d'idée (figure 1, ligne 23) :

MINLOC et MAXLOC	position des éléments extrêmes.
SUM et PRODUCT	somme et produit des éléments.
ANY et ALL	somme et produit logique des éléments (idem en PL/1).
COUNT	nombre d'éléments « vrai » dans un tableau logique.

■ **Fonctions de calcul matriciel**

DOT_PRODUCT	produit scalaire.
MATMUL	produit matriciel.
TRANSPOSE	transposée d'une matrice.

■ **Autres fonctions**

RESHAPE	fonction de restructuration (figure 1, lignes 26-29).
PACK	construit un vecteur d'éléments (figure 1, lignes 26-27).
UNPACK	fonction inverse de PACK.
SPREAD	construit un tableau par duplication d'un autre.
EOSHIFT, CSHIFT	décalages (avec expulsion, ou circulaire).

8.3.2 Fonctions générales

SELECTED_INT_KIND (R) et SELECTED_REAL_KIND (p, r) voir § 2.1.

REPEAT (STRING, NCOPIES)	duplication de la chaîne STRING
TRIM (STRING)	chaîne STRING moins ses espaces finaux.
TRANSFER	transfert de type (cf. unckecked_conversion en Ada.)

8.4 Sous-programmes prédéfinis

RANDOM_SEED	initialise ou interroge le générateur pseudo-aléatoire.
RANDOM_NUMBER	fournit un (tableau de) réel(s) pseudo-aléatoire(s).
DATE_AND_TIME	fournit la date et l'heure (figure 1, ligne 52).
SYSTEM_CLOCK	consultation de l'horloge interne.
MVBITS	transfert de bits entre entiers (procédure distributive).

9. Conclusion

Fortran vient de s'offrir une véritable révolution culturelle, au bon sens du terme, Des constructions saines et rigoureuses, accroissant le pouvoir d'expression du langage et améliorant la fiabilité des programmes, sont venues balayer un certain nombre de conceptions archaïques. Fortran sort ainsi renforcé dans son domaine de prédilection : le calcul intensif, tout en rejoignant – enfin ! – les préoccupations du génie logiciel. Toutefois, cette modernisation poussée a dû conserver les caractéristiques obsolescentes de Fortran 77, pour d'évidents motifs de réutilisation du code existant : cela entraîne la co-existence actuelle de deux styles d'écriture antinomiques, alourdissant considérablement les compilateurs – jusqu'au niveau de l'analyse lexicale ! De même, si le professionnel de la programmation ne peut qu'accueillir avec faveur le rajeunissements de son langage de prédilection, le scientifique ou le technicien, utilisateur occasionnel, pourra rester perplexe devant sa complexification apparente : sensibilisation et formation doivent être les réponses à leur apporter.

Nous l'avons dit dès l'introduction, Fortran 90 n'est qu'une étape dans un processus de progrès désormais irréversible. D'ores et déjà l'avenir se prépare – l'horizon 2000 est pour demain – et des groupes d'experts sont actuellement au travail pour enrichir encore Fortran : le domaine du temps réel (Fortran « Haute Performance »), celui de la programmation par objets sont des voies intensivement explorées, à côté d'autres améliorations importantes comme la gestion des exceptions.

Reprenant la conclusion de la version précédente de cet article, il est plus que jamais permis d'ajouter : « les développements que nous venons de mentionner font penser que Fortran connaîtra encore de beaux jours au XXI^{e} siècle. »

Fortran

par **Patrice LIGNELET**
École Nationale Supérieure d'Électronique et de ses Applications (ENSEA)
(division des Administrateurs)
Animateur du groupe Fortran à l'AFNOR

Caractéristiques obsolescentes

La norme indique comme telles des caractéristiques de Fortran 77 s'exprimant de meilleure façon en Fortran 90, et vouées à une probable disparition dès la prochaine norme. Tout compilateur doit en signaler l'emploi dans un programme. Elles comprennent des caractéristiques déjà obsolescentes de fait en Fortran 77 (le descripteur Hollerith nH, l'instruction PAUSE, les étiquettes de FORMAT définies par ASSIGN), de douteux apports de Fortran 77 (indices ou paramètres de boucles réels !, branchement vers ENDIF depuis l'extérieur d'un IF), d'anciennes caractéristiques rendues obsolètes par de nouvelles constructions : IF arithmétique ternaire (utiliser un CASE pour les entiers, ou un schéma IF pour les réels), boucles DO emboîtées à fin commune, ou sur une instruction exécutable (utiliser un END DO propre à chaque boucle), ASSIGN et GOTO « assigné » (utiliser une procédure interne), et même certaines caractéristiques officiellement normalisées par Fortran 77 (paramètres de type étiquette (*) et retour ventilé, à remplacer par un code de retour testé par un CASE).

Beaucoup d'autres constructions, bien que non déclarées obsolescentes compte tenu de leur taux d'emploi élevé, le sont devenues de fait :

— le format fixe de texte source avec zonage rigide (et espaces non significatifs) ;
— les zones communes (COMMON) et (BLOCK DATA) associés, remplacées chacune par un module ;
— l'association de mémoire (EQUIVALENCE), remplacée par les pointeurs et les cibles (TARGET) ;
— les paramètres tableaux à borne implicite : utiliser un profil hérité ;
— les points d'entrée secondaire (ENTRY) : utiliser un MODULE avec autant de procédures distinctes ;
— le GOTO de ventilation, remplacé par le CASE ;
— les noms spécifiques des fonctions prédéfinies génériques (déjà osbolètes en Fortran 77) ;
— et la nouvelle directive INCLUDE, déjà rendue inutile par les modules, qui opèrent l'inclusion au niveau sémantique.

Normes auxiliaires supplémentaires

Comme pour Ada, le module permet désormais d'étendre le langage sans alourdir sa grammaire. Divers services logiciels spécialisés pourront ainsi être offerts au programmeur Fortran, à l'aide de module les faisant chacun l'objet d'une norme auxiliaire supplémentaire rattachée à la norme principale du langage.

Un type chaîne de caractères de longueur variable (char (n) varying en PL/1 est en voie d'adoption. Le module s'appelle ISO_VARYING_STRING ; il exporte le type abstrait VARYING_STRING, et l'unifie avec le type CHARACTER (à la manière de Pascal Étendu). C'est ainsi que toutes les opérations ou fonctions applicables au type CHARACTER le deviennent au nouveau type, voire en mélangeant les deux types le cas échéant. Des procédures spécifiques sont ajoutées :

— VAR_STR (CHAR) pour convertir explicitement l'argument CHARacter en VARYING_STRING ;
— GET, PUT et PUT_LINE pour les entrées-sorties du nouveau type ;
— enfin, INSERT, REPLACE, REMOVE, EXTRACT et SPLIT pour manipuler le contenu d'une chaîne variable.

D'autres projets sont en crous d'étude pour réaliser des passerelles entre Fortran et des normes relatives à d'autres domaines (le traitement graphique, le langage d'accès aux bases de données SQL, le système d'exploitation POSIX, etc.)

Disponibilité des compilateurs

L'hétérogénéité et la complexité croissantes du langage n'incitent guère les grands constructeurs à accélérer la commercialisation de compilateurs ; n'ont-ils pas souvent introduit dans leur implémentation de Fortran 77 un certain nombre d'extensions recoupant quelques unes des caractéristiques nouvelles, mais avec une syntaxe divergente ? Devant cette carence provisoire, quelques fournisseurs spécialisés, plus dynamiques, ont su prendre le relai. On peut ainsi faire appel à :

— NAG (Numerical Algorithms Group, Oxford), qui offre un compilateur Fortran 90 (f90) sur :

DEC station 3100, HP9000/700, IBM Risc system/6000, NeXT, Silicon Graphics 4D, SUN 3 et 4 Unix 4.2 bsd, Apollo Domain : F90 version 2 (la version 1 est disponible sur HP9000/400 et VAX/VMS).

Ces compilateurs, écrits en C, opèrent en plusieurs passes, dont les premiers traduisent le code Fortran 90 en C.

Ils sont commercialisés, hors maintenance, à des prix hors taxes de 5 940 F à 12 000 F (mars 1993). À noter l'existence d'une version de base pour processeur INTEL 386 SX ou +, proposée à 990 FH.T.

— Salford Software offre un compilateur de même nature (réalisé en collaboration avec NAG) pour PC systèmes DOS ou UNIX (FTN 90). (Salford Software Marketing Ltd., Maxwell Building, The Crescent, Salford M5 4WT, Grande-Bretagne).

— Pacific-Sierra Research diffuse le produit VAST-90, qui traduit le code Fortran 90 en Fortran 77 de l'ancienne norme (qui peut ensuite être pris en charge par un compilateur Fortran 77, avec une bibliothèque complémentaire pour certaines procédures prédéfinies). Ce même produit peut réécrire en Fortran 90 du code Fortran 77 existant. (Pacific-Sierra Research Corp., Computer Products Group, 12340 Santa Monica Belvedere, Los Angeles, CA 90025).

— La société américaine PARASOFT diffuse un autre traducteur de Fortran 90 en Fortran 77 (Parasoft Corp., 2500 E, Foothill Belweder Pasadona, CA 91107).

Autres compilateurs : Microway (pour système DOS, OS/2 et Unix) ; EPC pour Sparc Solaris 1.x et 2.x, IBM RS/6000, Intel 3/486 (SVR 384, Solaris 2.x), Motorola 88000/100/100 (SVR 384).

Un compilateur maison pour IBP 9000 vient d'être installé chez certains clients privilégiés. On annonce encore un compilateur chez Hewlett-Packard, chez SUN (partiellement construit à partir du compilateur NAG), chez Digital Equipement (DEC), chez CRAY (CF90, version Sparc), Microsoft, Lahey, Portland, ACRI (pour machine Alpha)...

Bibliographie

ABERTI (C.). – *Fortran 90. Initiation à partir du Fortran 77*. Série Informatique, SI éditions, Menton (1992).

BRAINERD (W.S.), GOLDBERG (C.H.) et ADAMS (J.C.). – *Programmer's Guide to Fortran 90*. McGraw-Hill, New-York (1991).

BRAINERD (W.S.) et alii. – *The Fortran 90 Hand-book*. Unicomp (1992).

COUNIHAN. – *Fortran 90*. Pitman (1991).

DELANNOY (C.). – *Programmer en Fortran 90*. Eyrolles (1993).

DUBESSET (M.) et VIGNES (J.). – *Les spécifications du Fortran 90*. Technip (1993).

KERRIGAN (J.). – *Migrating to Fortran 90*. O'Reilly (1993).

LIGNELET (P.). – *Fortran 90. Approche par la Pratique*. Série Informatique, SI éditions, Menton (1993).

LIGNELET (P.). – *Les fichiers en Fortran*. Masson (1988).

METCALF (M.) et REID (J.). – *Fortran 90 Explained*. 1990, Oxford Universty Press, Oxford. Ouvrage traduit en français par PICHON (B.) et CAILLAT (M.) sous le titre *Fortran 90 : les concepts fondamentaux*. AFNOR Éditions (1993).

METCALF (M.). – *A first Encounter with Fortran 90* (using the NAG f90 compiler). Fortran Journal, vol. *4*, n° 1, p. 2-8, janv.-fév. 1992.

Cet article est extrait de la revue bimestrielle FORTRAN JOURNAL, The Journal of the Fortran User Community (P.O. Box 201, Fullerton, CA 92634. En France, voir la société OPTIMIZE : cf. Annexe 3).

MORGAN et SCHONFELDER. – *Programming in Fortran 90*. Blackwell (1993).

Normalisation

Norme internationale

ISO/IEC 1539	1991	Programming language Fortran.
ISO/IEC 1539-2		[en voie d'adoption] Varying Length Character Strings in Fortran.

France (AFNOR)

ISO/CEI 1539		Langage de programmation FORTRAN. (indice de classement Z 65-110) [à paraître]

États-Unis (ANSI)

X 3.198	1992	Programming Language Fortran Extended.