Cours de Fortran 90 / 95

CLAIRE MICHAUT
Claire.Michaut@obspm.fr
LUTH - Observatoire de Paris
5, place Jules Janssen - 92195 Meudon Cedex

Master Recherche (M2)

Spécialité Astronomie & Astrophysique

MCours.com

Année 2005 - 2006

Table des Matières

I HISTORIQUE

Au début de l'informatique, le fait de programmer se révélait fastidieux et peu fiable. Puis apparut une codification directe des instructions machines, connue sous le nom de code assembleur. Dans les années 50, il devint évident qu'une nouvelle étape dans les techniques de programmation devait être franchie.

C'est pourquoi, l'année 1954 voit naître le projet du premier langage symbolique par John Backus de la société IBM ; Mathematical Formula Translating System dit FORTRAN. Ce langage permet pour la première fois à l'utilisateur de se concentrer sur le problème à résoudre et non sur les instructions de la machine.

1956 - Premier manuel de référence qui définit le Fortran I. Les noms de variables ont jusqu'à six caractères et l'instruction FORMAT est introduite.

1957 - Le Fortran II fait son apparition ainsi que les premiers compilateurs commerciaux. Les sous-programmes et fonctions sont acceptés et deviennent compilables indépendamment les uns des autres.

1958 - À la fin de l'année, le Fortran III est disponible, mais reste interne à la compagnie IBM. Certaines des nouveautés implantées sont le type logique, les paramètres de procédures.

1962 - Le Fortran IV est né et restera le langage informatique des scientifiques pendant seize ans. Il apporte les déclarations de type explicites et l'instruction d'initialisation DATA. Entre temps, les autres constructeurs développent leur compilateur ; il faut bientôt établir une norme.

1966 - Création de la norme ANSI (American National Standards Institutes), le Fortran IV est rebaptisé Fortran 66.

1978 - La norme Fortran 77 (ou Fortran V) modernise le langage, car c'est la fin des cartes perforées. On lui doit le type caractère, de meilleures entrées / sorties, un début de programmation structurée et des fonctions prédéfinies portant un nom générique.

Dans le milieu des années 80, la reconversion à Fortran 77 était bien engagée alors que Fortran 66 perdait considérablement du terrain.

Après 30 ans d'existence, Fortran n'est plus le seul langage de programmation disponible, mais il est resté prédominant dans le domaine du calcul numérique à l'usage des chercheurs, des techniciens et des ingénieurs. Dans le but de maintenir le langage correctement à jour, le Comité technique X3J3 accrédité par l'ANSI a élaboré une nouvelle norme.

1991 - La norme internationale ISO/ANSI Fortran 90 apparaît dotée d'innovations changeant profondément le langage. Le Fortran devient modulaire, il permet un calcul vectoriel et un contrôle de

la précision numérique. De plus il devient plus fiable grâce aux blocs interfaces et s'oriente vers une programmation objet.

1995 - Les caractéristiques dépréciées ou obsolètes forment le Fortran 95 et il admet une structure vectorielle supplémentaire (FORALL) et quelques autres améliorations.

2002 - Fin des travaux pour la norme Fortran 2000, publication prévue pour fin 2004 pour la norme ISO.

Fortran 2005 : Correction de Fortran 2000 et probablement dernière norme Fortran.

Plus tard, va-t-on vers un langage F, FOO, F++ ?

II STRUCTURE D'UN PROGRAMME ET SYNTAXE

II.A UNITÉS DE PROGRAMME ET PROCÉDURES

De façon très générale, la taille des programmes que l'on est amené à développer aujourd'hui devient considérable. Les ordinateurs sont de plus en plus puissants, ils sont alors de plus en plus sollicités dans la vie courante. Et l'exigence de l'utilisateur n'a plus de limite. Quant aux chercheurs et à leurs désirs, rien ne saurait les décourager, pas même des dizaines de milliers de lignes de calculs. Or, dès que l'on envisage un travail un tant soit peu ambitieux, on a de plus en plus souvent besoin de compétences multiples. C'est-à-dire que les programmes sont développés par plusieurs personnes, qui hormis le langage informatique commun ne parlent pas forcément le même jargon technique.

C'est pourquoi, nous avons besoin d'un langage structuré et d'un langage modulaire. La structure permet une bonne lisibilité d'un programme, donc une bonne compréhension et une maîtrise du codage. C'est-à-dire que l'on résout effectivement le problème donné et pas un ersatz qui fonctionne dans 97 % des cas, voire moins ! La modularité permet de répondre aux besoins du Génie Logiciel. Les modules fonctionnent quasiment comme des boîtes noires, ce qui permet d'incorporer dans un programme des parties indépendantes les unes des autres et ne nécessitant pas forcément aux autres développeurs de prendre connaissance de leur contenu. Il suffit de connaître le nom d'appel des procédures dont on a besoin.

La notion d'objet est plus forte encore, et on les manipule alors complètement comme des boîtes noires. L'objet réalise lui-même l'encapsulation de ses données et empêche tout accès depuis l'extérieur.

En récapitulant, Fortran 90 facilite la manipulation des structures (objet de type dérivé), en offrant les possibilités suivantes.

- Il permet de définir des fonctions de manipulation de ces structures :
  - surcharge des opérateurs +, -, * ,
  - de nouveaux opérateurs en les associant aux fonctions correspondantes,
  - autres fonctions ou procédures agissant sur ces structures ;
- Il permet d'encapsuler le type et les fonctions opérant sur les objets de ce type dans un module séparé. Ce qui permettra d'y accéder facilement (USE <module>) en diminuant les sources d'erreurs, dans toutes les unités de programme en ayant besoin ;
- Il permet de rendre privée les composantes de la structure. Si la définition du type se fait dans un module séparé (encapsulation), seules les fonctions ou procédures internes au module ont accès aux composantes privées (PRIVATE) et peuvent les modifier (sécurité).

Sans être un vrai langage orienté objet, Fortran 90 fournit ainsi des extensions objet bien utiles pour la structuration du programme et sa fiabilité.

Un programme est composé d'unités de programme, qui sont :

- un programme principal, contenant une procédure interne ;
- des procédures externes ;
- des modules.

Les procédures externes peuvent être écrites dans des modules, elles deviennent internes au module. Un module peut contenir des procédures, des déclarations, des affectations de variables, des types dérivés et des blocs interfaces.

Les mots clés sont :

```
PROGRAM, END PROGRAM ;
MODULE, END MODULE ;
CONTAINS ;
SUBROUTINE, END SUBROUTINE ;
FUNCTION, END FUNCTION.
```

II.A.1 Le programme principal

Le programme principal est une unité de compilation, mais il a la particularité d'être la seule qui soit exécutable. Donc il est la seule qui soit totalement obligatoire. Le programme principal prend toujours la forme suivante :

```
PROGRAM nom_du_programme
    [instructions de spécification]
    [instructions exécutables]
CONTAINS
    [procédures internes]
END PROGRAM nom_du_programme
```

L'instruction PROGRAM est facultative, mais conseillée. L'instruction END qui génère l'arrêt du programme est obligatoire. Les instructions de spécification sont des instructions de déclaration, des définitions de type, des blocs INTERFACE. L'instruction CONTAINS indique la présence d'une ou plusieurs procédures.

Exemple II.1

```
PROGRAM ETOILE
USE DONNEES
IMPLICIT NONE
REAL :: r1, u1, T1

r1 = 1.e-3 ; u1 = 45000. ; T1 = 55635.
r1 = G * u1 * T1
PRINT*,"r1 vaut ", r1

END PROGRAM ETOILE
```

II.A.2 Les procédures externes

Une procédure externe est appelée depuis une autre unité de programme. Elle a une forme similaire à celle du programme principal, tant du point de vue de sa syntaxe que de son exécution. elle prend place On en distingue deux sortes : les sous-programmes et les fonctions. De plus, on distingue des procédures globales, contenues dans un module et utilisables par l'ensemble des autres unités de programme, des procédures internes, contenues dans une procédure globale et utilisables uniquement par leur hôte.

a) Les sous-programmes

Les sous-programmes, définis par le mot-clé SUBROUTINE, définissent une action au moyen de toutes les instructions du langage, dont l'appel prend lui-même la forme et l'apparence d'une instruction du langage. La syntaxe générale d'un sous-programme est la suivante :

```
SUBROUTINE nom_du_sous-programme [(arguments)]
    [instructions de spécification]
    [instructions exécutables]
[CONTAINS
    [procédures internes]]
END SUBROUTINE [nom_du_sous-programme]
```

Les sous-programmes sont appelées, depuis une autre procédure (ou programme principal), avec l'instruction CALL nom_du_sous-programme [(arguments)].

b) Les fonctions

Les fonctions, définis par le mot-clé FUNCTION, calculent une valeur au moyen de toutes les instructions du langage, dont l'appel prend place au sein d'une expression. La syntaxe générale d'une fonction est la suivante :

```
[TYPE] FUNCTION nom_de_la_fontion (arguments)
    [instructions de spécification]
    [instructions exécutables]
[CONTAINS
    [procédures internes]]
END FUNCTION [nom_de_la_fonction]
```

Les fonctions sont appelées directement à l'aide de leur nom propre et de leurs arguments, depuis une autre procédure (ou programme principal). On peut déclarer leur type avant l'instruction FUNCTION.

Exemple II.2

```
SUBROUTINE LOCATE (rr, TT, m, n, hmn)
IMPLICIT NONE
REAL, INTENT(IN)  :: rr, TT
INTEGER, INTENT(OUT) :: m, n
REAL, OPTIONAL, DIMENSION(4), INTENT(OUT) :: hmn
REAL:: hm, hn, hm1, hn1, Vol

    m = 1 ; n = 1
    DO
        …
        m = m + 1
        n = n + 1
   END DO

IF (PRESENT(hmn)) THEN
   ! Calcul des poids pour l'interpolation des donnees.
      hm = TT - TeV(m) ; hm1 = TeV(m+1) - TT
      hn = rr - rho(n) ; hn1 = rho(n+1) - rr
      Vol = (hm + hm1) * (hn + hn1)
      hmn(1) = hm  * hn  / Vol
      hmn(2) = hm1 * hn  / Vol
      hmn(3) = hm  * hn1 / Vol
      hmn(4) = hm1 * hn1 / Vol
   END IF

END SUBROUTINE LOCATE
```

Exemple II.3

```
REAL FUNCTION ZMOY (rr, TT)
IMPLICIT NONE
REAL, INTENT(IN) :: rr, TT

CALL LOCATE (rr, TT, m, n, hmn)
zmoy = zbar(m,n)   * hmn(4) + zbar(m+1,n)   * hmn(3) &
     + zbar(m,n+1) * hmn(2) + zbar(m+1,n+1) * hmn(1)
zmoy = 1. + zmoy

END FUNCTION ZMOY
```

c) Les procédures récursives

Dans certains cas particuliers, on a besoin qu'une procédure puisse s'appeler elle-même. D'ordinaire, cette structure de programmation est impossible. C'est pourquoi, il faut tout d'abord avertir le compilateur de ce mode de fonctionnement : on utilise l'attribut RECURSIVE devant le nom de la procédure, qu'elle soit un sous-programme ou bien une fonction. Attention toutefois au temps de calcul, qui peut devenir prohibitif, lors de plusieurs appels récursifs successifs !

Syntaxe générale :

```
recursive subroutine nom_proc (list_arg)
recursive function nom_proc (list_arg) result (nom_res)
```

Remarque : dans le cas d'une fonction, on est obligé d'utiliser un nom de résultat différent du nom de la fonction, défini à l'aide du mot clé RESULT.

Les fonctions récursives peuvent être, par exemple, très utiles pour traiter des suites mathématiques, qui par définition, sont construites en référence au rang inférieur. Ainsi pour calculer la suite arithmétique :

$$u_1 = 1 ;\ u_2 = 3 ;\ u_n = 3u_{n-1} - u_{n-2} \quad n \geq 3$$

Exemple II.4

```
RECURSIVE FUNCTION SUITE (n) RESULT (u)
    INTEGER, INTENT (IN) :: n
    INTEGER :: u
    SELECT CASE (n)
        CASE (1)
            u = 1
        CASE (2)
            u = 3
        CASE DEFAULT
            u = 3 * suite (n-1) - suite (n-2)
    END SELECT
END FUNCTION SUITE
```

Remarque : Le mot-clé RESULT peut être utilisé dans toutes les fonctions pour définir le nom du résultat, qu'elles soient récursives ou non. Le type du résultat est toujours celui de la fonction.

II.A.3 Les modules

Un module est une unité de compilation qui a pour fonction de mettre à disposition, aux autres unités de compilation, des ressources logicielles comme des constantes, des variables, des types et des procédures. C'est une abstraction des données et des procédures qu'il encapsule, réalisant ainsi une boîte noire pour l'utilisateur et favorisant la maintenance des applications. Le module n'est pas exécutable, il n'y a que le programme principal qui le soit. La forme générale d'un module est la suivante :

```
MODULE nom_du_module
    [instructions de spécification]
[CONTAINS
    [procédures globales]]
    [procédures internes]]
END MODULE [nom_du_module]
```

Exemple II.5

```
MODULE DONNEES
IMPLICIT NONE
REAL, PARAMETER :: Rgaz = 8.314e7

CONTAINS
    SUBROUTINE INIT (r1, u1, T1)
    IMPLICIT NONE
    REAL, INTENT(IN) :: r1, u1, T1
    REAL :: gam, Mach

    gam = 5./3.
    ! Mach = u1/Csound ; nombre de Mach
      Mach = u1 * sqrt(131/(gam*Rgaz*T1))
      WRITE (6,*) "Le nombre de Mach est de ", Mach

END SUBROUTINE INIT
END MODULE DONNEES
```

a) Accès aux ressources d'un module

L'accès aux ressources d'un module se fait grâce à l'instruction *use nom_module*. Cette ligne d'instruction se place toujours en tête de l'unité utilisatrice, avant le bloc déclaratif et permet l'importation de l'intégralité du module dans cette unité de compilation.

```
PROGRAM (ou SUBROUTINE, FUNCTION)
USE Nom_de_Module1
USE Nom_de_Module2
IMPLICIT NONE
…
END PROGRAM  (ou SUBROUTINE, FUNCTION)
```

On peut toutefois restreindre l'accès à certaines ressources du module uniquement, en important partiellement le module. Pour ce faire, on place l'instruction *only :* suivie d'une liste exhaustive des ressources souhaitées, derrière la commande d'importation du module.

```
USE Nom_de_Module, ONLY: var1, var2, sous-programme1
```

b) Renommer les importations

Pour des raisons de lisibilité d'un programme, ou bien de compatibilité de plusieurs procédures, il est parfois souhaitable de renommer les entités importées. On utilise la double flèche qui est aussi le symbole d'affectation des pointeurs.

```
USE Nom1_de_Module => Nom2, ONLY: var1 => xy, var2
```

II.B FORMAT ET SYNTAXE

II.B.1 Le format

En Fortran 90, on utilisera de préférence le format libre, qui est une nouveauté de cette norme. Les lignes ont une longueur maximale de 132 caractères. On peut coder plusieurs instructions sur une même ligne en les séparant par un point-virgule « ; ». Une instruction peut s'écrire sur plusieurs lignes en utilisant le caractère de continuation esperluette « & ». Une chaîne de caractères écrite sur plusieurs lignes doit être précédée du « & » en plus de celui de continuation de la ligne précédente.

Exemple II.6

```
rr = rho_init ; uu = Cshock ; TT = Temp
PRINT*, "Dans ce calcul le flux final est ",
& Frad, "et l'opacite est ",
& KRoss
```

Exemple II.7

```
PRINT*, "Dans ce calcul le flux final et&
& l'opacite sont ", Frad,
& KRoss
```

II.B.2 Les identificateurs et la syntaxe

Un identificateur est composé d'une suite de caractères, limitée à 31, choisis parmi les lettres (non accentuées), de chiffres (sauf en première position) et du caractère « espace souligné » : « _ ». L'identificateur sert à nommer les objets que le programmateur veut manipuler. Le compilateur ne distingue pas les majuscules des minuscules, mais on peut toutefois s'en servir par soucis de lisibilité.

Exemple II.8 : identificateurs autorisés

```
clight
Nom_de_famille
TypeOfGas
M2
```

Exemple II.9 : identificateurs non autorisés

```
Nom de famille
§TypeOfGas
2M
```

En Fortran 90, il n'existe pas de mots réservés, on peut donc utiliser des mots clés du langage comme identificateur. Dans l'idéal, mieux vaut éviter de le faire, pour limiter les risques de confusion.

Un espace est nécessaire entre identificateurs, mots clés et nombres.

II.B.3 Commentaires

Un commentaire commence par le caractère point d'exclamation « ! » et se termine par la fin de ligne. Donc, il est possible d'insérer des commentaires après des instructions, sur une même ligne, mais pas avant.

Exemple II.10

```
! gam : rapport des capacites de pression & volume
! constants : Cp / Cv.
  gam = 5. / 3.
! Mach = u1/Csound ; nombre de Mach
  Mach = u1 * sqrt(AtW/(gam*Rgaz*T1)) !adimensionne
  PRINT*, "Le nombre de Mach est de ", Mach
  gM2 = gam*Mach*Mach                  ! adimensionne
```

II.B.4 Chaînes de caractères

Les chaînes de caractères sont encadrées par des cotes simples « ' » ou doubles « " ». Elles neutralisent l'effet de tous les autres signes ou mots clés.

Exemple II.11

```
! Le point d'exclamation est utilise comme tel
! dans une chaine de caracteres. dans ce cas
PRINT*, "Le nombre de Mach est grand ! "
"L'éléphant est + grand que la souris."
'3 * plus souvent'
```

II.C LES TYPES ET LES DÉCLARATIONS

Comme toute information stockée en mémoire est codée en binaire, il faut une clé pour connaître la représentation initiale de tel ou tel octet. Ainsi une chaîne de caractères, un nombre ou une valeur logique peuvent avoir dans l'absolu la même représentation binaire, suivant le codage utilisé. C'est pourquoi, nous devons annoncer quel type d'objets nous manipulons, pour différencier leur codage binaire. Le Fortran 90 prévoit cinq types de bases et des combinaisons.

```
logical, character, integer, real, double precision
complex, type
```

Il est possible de définir de nouveaux types de données à partir d'un groupe de données appartenant à des types intrinsèques, à l'aide de la déclaration type. Nous verrons ce point ultérieurement.

De plus on distingue les constantes, les variables scalaires, les vecteurs et tableaux au moyen d'attributs. Ces attributs servent à spécifier ce que le programmateur souhaite faire de ces données déclarées et permettent un contrôle de la bonne utilisation de ces données.

La forme générale de la déclaration d'une variable est la suivante :

```
type [[,liste_attributs] :: ] liste_objets
```

La liste des attributs est la suivante :

```
parameter, dimension, allocatable, pointer, target,
save, intent, optional, public, private, external,
intrinsic
```

Nous expliquerons leur signification au fur et à mesure de leur introduction dans les différents chapitre du cours.

II.C.1 Déclaration de variables scalaires

```
CHARACTER (LEN=12) :: elmt
INTEGER, PRIVATE :: mmax, nmax
INTEGER :: AtN ; REAL :: AtW
REAL :: gM2, CvM2, ww, gam = 5./3.
DOUBLE PRECISION :: x=5.d-4
COMPLEX :: z1=(3,5), z2
```

On voit que l'on peut initialiser les variables lors de leur déclaration. Toute variable initialisée devient permanente, elle reçoit donc implicitement l'attribut « save ».

II.C.2 Déclaration de constantes

Ces objets reçoivent l'attribut « parameter », et doivent obligatoirement être initialisés lors de leur déclaration. Il devient alors interdit de changer leur valeur dans une instruction.

```
CHARACTER (LEN=12) :: elmt='nomdefamille'
REAL, PARAMETER :: clight = 2.99791e10, Rgaz = 8.314e7
COMPLEX, PARAMETER :: z1=(3,5), z2=(6,9)
DOUBLE PRECISION, PARAMETER :: x=5.d-4
```

Toutefois, certains programmateurs auront recours au typage par défaut. On commence par une déclaration du type :

```
IMPLICIT REAL (a-h, o-z)
```

Ainsi tous les identificateurs commençant par une lettre de cet ensemble alphabétique (de a à h ou de o à z) seront automatiquement considérés comme des réels. Sauf, si une déclaration contraire est faite. Cette méthode de déclaration est pratique au commencement, car moins fastidieuse. Mais elle est source d'innombrables erreurs ! Par exemple, elle ne détecte pas les fautes de frappe, pourtant si nombreuses. Le Fortran 90 est justement un langage développé dans le but de prévenir au maximum des erreurs de programmation, donc il est absurde de se passer de cet atout.

C'est pourquoi, il est vivement conseillé d'insérer l'instruction :

```
IMPLICIT NONE
```

avant tout bloc de déclaration. Ce qui oblige le programmateur à déclarer explicitement chaque variable.

La déclaration à l'aide du mot clé TYPE sert à créer des types dérivés définis par le programmateur. Nous verrons ce point ultérieurement, dans le chapitre II.D.

II.C.3 Sous-type : la précision des nombres

a) Avec le mot clé KIND

Les types de base en Fortran 90 sont en fait des noms génériques renfermant chacun un certain nombre de sous-types que l'on peut sélectionner à l'aide du mot clé KIND, lors de la déclaration des objets. Pour chaque type prédéfini correspond un sous-type par défaut, c'est la simple précision en général.

Les différentes valeurs du paramètre KIND sont dépendantes du système utilisé et elles correspondent au nombre d'octets désirés pour coder l'objet déclaré. Pour les chaînes de caractères, KIND sert à indiquer combien d'octets sont nécessaires pour coder chaque caractère ! C'est utile dans des alphabets complexes.

Dans une déclaration, on indique soit directement le nombre d'octets, soit on donne un exemple du sous-type désiré.

Exemple II.12

```
REAL (KIND=8) :: x      réel codé sur 8 octets :
    double précision pour la plupart des calculateurs,
    et simple précision sur Cray C90, par exemple.
INTEGER (KIND=2) :: k entier codé sur 2 octets.
```

En donnant le sous-type directement en exemple, au lieu du nombre d'octets.

Exemple II.13

```
REAL, (KIND = KIND(1.d0)) :: x
réel double précision quelque soit la machine utilisée.
```

b) Avec des suffixes ou préfixes

Une autre manière de préciser le sous-type désiré lors de l'écriture des variables est de leur ajouter un suffixe pour les constantes numériques, ou bien un préfixe pour les chaîne de caractères, qui contient la valeur du sous-type choisi. On insère le caractère « _ » entre la variable et le suffixe ou préfixe.

Exemple II.14

```
123456_2              entier codé sur 2 octets
34.567005678001_8     réel codé sur 8 octets
2_'fleur'             chaîne de caractères, chacun codé
                      sur 2 octets
```

ou bien

Exemple II.15

```
INTEGER, PARAMETER :: deux = 2, huit = 8
123456_deux           entier codé sur 2 octets
34.567005678001_huit  réel codé sur 8 octets
deux_'fleur'          chaîne de caractères, chacun codé
                      sur 2 octets
```

II.C.4 Les fonctions intrinsèques liées à la précision

a) selected_int_kind (m)

Cette fonction retourne un paramètre de sous-type qui permet de représenter les entiers n, en fonction de l'étendue choisie à l'aide de l'entier m, tels que :

$-10^m < n < 10^m$

Elle retourne -1 si aucun paramètre de sous-type ne répond à la demande.

Exemple II.16

```
SELECTED_INT_KIND(30)     retourne -1
SELECTED_INT_KIND(10)     retourne 8
SELECTED_INT_KIND(10)     retourne 4
```

b) selected_real_kind (p, r)

Cette fonction reçoit au moins un des deux arguments p et m qui sont respectivement la précision et l'étendue choisies. Elle renvoie un paramètre de sous-type qui permet de représenter les réels répondant à cette précision, tels que :

$10^{-r} < |x| < 10^r$

La fonction renvoie :

- -1 si la précision demandée n'est pas disponible ;
- -2 si l'étendue n'est pas disponible ;
- -3 si la précision et l'étendue ne sont pas disponibles.

Exemple II.17

```
SELECTED_REAL_KIND(2,30)        retourne 4
SELECTED_REAL_KIND(20,30)       retourne 16
SELECTED_REAL_KIND(10,10)       retourne 8
SELECTED_REAL_KIND(40,10)       retourne 4
SELECTED_REAL_KIND(20,5000)     retourne -2
```

c) range et precision

En complément, l'étendue d'un nombre est connue à l'aide de la fonction RANGE.

La précision d'un réel (nombre significatif de chiffres derrière la virgule) est connue avec la fonction PRECISION.

Exemple II.18

```
RANGE(1234567890)          retourne 9
RANGE(123456.7890)         retourne 37
PRECISION(1.234567890)     retourne 6
```

II.D AUTRES TYPES

Dans un souci de fiabilité dans l'exécution et l'utilisation d'un programme, ou une partie, on est amené à définir un ensemble de variables. Le fait de les ranger ainsi sous un même nom de type et dans un seul type prévient d'erreurs de manipulation de ces variables et facilite leur déclaration. Les tableaux sont déjà des objets qui regroupent en ensemble de variables, mais c'est un cas particulier qui impose que toutes ces variables soient strictement du même type.

Dans le cas d'objets plus complexes, regroupant des données (champs ou composantes) de différents types, nous allons définir un type dérivé (ou structure) qui étend les types prédéfinis, et qui permet de manipuler ces objets globalement. Chaque composante est identifiée par son nom, elle peut être soit de type intrinsèque soit également de type dérivé.

II.D.1 Les tableaux

Un tableau est un ensemble de variables qui sont toutes du même type, il peut être prédéfini ou de type dérivé. Chacune des variables est repérée par un indice numérique. La forme déclarative générale est la suivante :

```
TYPE, DIMENSION (…) [,liste_attributs] :: nom_tableau
```

Un tableau peut avoir une taille statique, dans ce cas, sa dimension est explicitement écrite lors de la déclaration. Ou bien il a une taille dynamique, c'est-à-dire qu'elle sera définie en fonction de l'exécution du programme, dans ce cas, la dimension est suggérée par le caractère deux-points « : ».

Dans le paragraphe consacré aux tableaux et à leur manipulation, nous reviendrons sur les formes déclaratives des tableaux de taille dynamique.

```
CHARACTER (LEN=8), DIMENSION (15) :: mois
INTEGER, DIMENSION (12,31) :: jour
LOGICAL, DIMENSION (2) :: reponse = (/.true.,.false./)
REAL, DIMENSION (:,:,:) :: position
INTEGER, DIMENSION(4), PRIVATE :: hmn
REAL, DIMENSION (:), ALLOCATABLE :: TeV, rho
```

II.D.2 Les types dérivés

a) Construction d'un type dérivé (ou structure)

Pour construire une nouvelle structure, on utilise la syntaxe suivante :

```
type nom_type
    bloc_declaratif_des_champs

end type nom_type
```

Restriction du bloc déclaratif :

- L'attribut PARAMETER n'est pas autorisé au niveau d'un champ.
- L'attribut ALLOCATABLE est interdit au niveau d'un champ.
- L'attribut POINTER est autorisé au niveau d'un champ, mais pas TARGET.
- L'initialisation d'un champ n'est possible qu'en Fortran 95.

Attention :

- Un objet de type dérivé est toujours considéré comme un scalaire, même si un champ possède l'attribut DIMENSION.
- Les tableaux d'objets de type dérivés sont autorisés et ils peuvent bénéficier de l'attribut ALLOCATABLE.
- Définition multiple de la structure : si le type dérivé n'est pas défini de manière unique, par exemple lorsque la structure est passée en argument de procédure, l'attribut SEQUENCE est obligatoire. Il assure un stockage en mémoire dans le même ordre des composantes en mémoire.

b) Manipulation d'un objet de type dérivé

Pour utiliser des variables du type dérivé précédemment défini, on déclare simplement :

```
type (nom_type) [,liste_attributs] :: nom_var
```

Pour accéder aux champs de l'objet que nous venons de définir, nous utiliserons le caractère pour cent « % ». Pour le reste des opérations, la structure ainsi définie se manipule comme n'importe quelle variable.

c) Exemples

Par exemple, un vétérinaire voudrait créer un type dérivé « animal », dans lequel seraient recensés la race, l'âge, la couleur et l'état de vaccination de ses « patients ».

Exemple II.19

```
TYPE animal
    CHARACTER (LEN=20) :: race
    REAL :: age
    CHARACTER (LEN=15) :: couleur
    LOGICAL, DIMENSION(8) :: vacc
END TYPE animal
```

Il peut ensuite manipuler globalement l'objet animal, ou bien accéder à chacun de ses champs, grâce à l'opérateur « % ».

Pour déclarer de nouveaux animaux, il utilise la forme suivante :

Exemple II.20

```
TYPE (animal) :: ponpon, rox, roucky
```

Pour rentrer les informations concernant ces nouveaux patients :

Exemple II.21

```
ponpon%race = 'chat_de_gouttiere'
ponpon%age = 12
ponpon%couleur = 'noir'
ponpon%vacc = (/.true.,.false., .true., .true.,&
               .false. ,.false., .true., .true./)
```

II.D.3 Vers un langage orienté objet

Fortran 90 n'est pas un langage orienté objet, à proprement parlé, mais il s'en rapproche, car il possède des extensions objet qui accroissent la fiabilité des programmes et procurent un confort de programmation. En fait, il lui manque quelques notions, comme la hiérarchie de types dérivés avec héritage. La norme Fortran 2000 les prévoit.

Fortran 90 facilite cependant la manipulation des structures (objets de type dérivé), car il permet :

- la définition d'autres fonctions de manipulation des structures, comme :
  - la surcharge des opérateurs +, -, * ,
  - de nouveaux opérateurs en les associant aux fonctions correspondantes,
  - des autres fonctions ou procédures agissant sur ces structures.
- d'encapsuler le type et les fonctions opérant sur les objets de ce type, dans un module séparé. Ce qui permettra d'y accéder facilement, avec l'instruction USE *nom_module*
- de rendre privées les composantes interne d'une structure, avec l'instruction PRIVATE.

Ces différents point seront abordés ultérieurement.

II.E LES FONCTIONS INTRINSÈQUES

Fortran 90 possède une librairie très complète de fonctions sur les scalaires. On trouvera bien sûr toutes les fonctions mathématiques habituelles sur les entiers, les réels, et les complexes. Nous ne présenterons pas ces fonctions dans ce cours.

De plus Fortran 90 possède des fonctions :

- de manipulation de bits ;
- de manipulation de chaînes de caractères ;
- de conversion de type ;
- d'interrogation sur les variables ;
- des fonctions intrinsèques liées à l'utilisation de tableaux (voir le Chapitre IV.C).

Ces fonctions sont nommées et expliquées tout au long du cours, suivant le sujet développé.

III STRUCTURES DE CONTRÔLE

À l'origine, les instructions d'un programme sont séquentielles : elles sont exécutées au fur et à mesure de leur ordre d'apparition. Mais rapidement, un programme plus élaboré a besoin d'une souplesse de contrôle : dans tel cas faire ceci, par contre dans tels autre cas faire cela, le refaire n fois, etc. On a donc besoin d'instructions pour faire des choix et des boucles.

Le programme sera alors écrit comme une suite de blocs. Un bloc peut-être contenu dans un autre, mais seulement au niveau inférieur. Il n'y a pas d'imbrication possible, sur le même niveau de priorité. Ces suites de blocs forment le langage structuré et augmentent la lisibilité du programme.

Un bloc est composé d'une instruction initiale contenant le mot clé de la structure, de mots clés intermédiaires et est fermé par un END mot clé de la structure.

Les mots clés définissant les blocs sont :

```
IF… THEN, ELSE IF… THEN, END IF
DO, CYCLE, EXIT, END DO
DO WHILE, END DO
SELECT CASE, CASE, CASE DEFAULT, END SELECT
WHERE, ELSEWHERE, END WHERE
FORALL, END FORALL
```

III.A LE CHOIX AVEC IF

La construction typique d'un bloc est la suivante :

```
IF (expression_logique) THEN
    bloc1
ELSE
    bloc2
END IF
```

Avec ses variantes :

```
IF (expression_logique1) THEN
    bloc1
ELSE IF (expression_logique2) THEN
    bloc2
…
END IF
```

Ou plus directement :

```
IF (expression_logique) THEN
    bloc1
END IF
```

Et encore plus simple, en une seule phrase :

```
IF (expression_logique)    bloc1
```

Le bloc IF peut être étiqueté, comme les autres structures présentées dans la suite. Dans ce cas, l'étiquette doit être répétée au niveau du END, à sa suite. L'étiquette est un identifiant de la structure, c'est-à-dire un nom ; lorsqu'elle est optionnelle, elle augmente la lisibilité du programme ou bien elle est rendue obligatoire par la logique du programme. La syntaxe est la suivante :

```
nom : IF (expression_logique) THEN
    bloc1
END IF nom
```

Exemple III.1

```
stab = f(1,2) * f(2,2)
IF (stab >= 1.01) THEN
    ecart = stab - 1.
    h = 0.5 * h
ELSE IF (stab <= 0.99) THEN
    ecart = 1. - stab
    h = 0.5 * h
ELSE
    count = count + 1
    z = z + h
    f(3,1) = f(1,2)
END IF
```

III.B LE CHOIX AVEC SELECT CASE

C'est une construction complètement nouvelle en Fortran, elle permet de faire un choix multiple, sans expression logique, car le test porte sur une valeur unique de la variable.

La syntaxe générale est la suivante :

```
[nom :] SELECT CASE (var)
    CASE (val1)
        bloc1
    CASE (val2)
        bloc2
    …
    CASE DEFAULT
        bloc default
END SELECT [nom]
```

Où val1, val2.. sont des valeurs que peut prendre la variable selectionnée *var*.

Le cas par défaut englobe tous ceux qui n'ont pas été explicitement mentionnés, il est optionnel.

Exemple III.2

```
SELECT CASE (K)
    CASE (1) ! On integre en montant.
        PRINT*,"Integration directe : "
        h =  Long*1.2 / REAL(nstep)
        f(1,1) = 1. ; f(2,1) = 1.5 ; f(3,1) = 0.
        z = 0.
        nom_fichier1 = name

    CASE (2) ! On integre en descendant.
        PRINT*,"Integration inverse : "
        h = -Long*1.2 / REAL(nstep)
        f(1,1) = r2/r1 ; f(2,1) = u2/u1 ;
                         f(3,1) = T2/T1
        z = Long
        nom_fichier2 = name

END SELECT
```

Exemple III.3

```
element : SELECT CASE (AtN)
! AtN : n° atomique - AtW : Poids atomique
    CASE (1:9)
        elmt = 'trop_leger' ; AtW = 0.
    CASE (12)
        elmt = 'magnesium' ; AtW = 24.305
    CASE (14)
        elmt = 'silicium' ; AtW = 34.855
    CASE (18)
        elmt = 'argon' ; AtW = 39.948
    CASE (47)
        elmt = 'argent' ; AtW = 107.868
    CASE (54)
        elmt = 'xenon' ; AtW = 131.29
    CASE DEFAULT
        elmt = 'non_utilise' ; AtW = 0.
    END SELECT element
```

III.C LE CHOIX AVEC WHERE

Cette instruction ne s'applique qu'aux tableaux. L'instruction WHERE permet d'effectuer un choix, à l'aide d'un filtre, puis d'effectuer des instructions uniquement aux éléments filtrés. Ce bloc ressemble beaucoup à un bloc IF, mais il permet en plus une écriture compacte pour les tableaux.

La syntaxe générale est la suivante :

```
WHERE (expression_logique)
    bloc1
ELSEWHERE
    bloc2
END WHERE
```

Remarque : Les blocs d'instructions *bloc1* et *bloc2* ne peuvent agir que sur des affectations concernant des tableaux. Dans le membre de droite des instructions, les tableaux sont obligatoirement conformant avec celui de l'expression logique.

```
WHERE (tab1 > 0.)
    tab2 = SQRT (tab1)
ELSEWHERE
    tab2 = 0.
END WHERE
```

En Fortran 95, la syntaxe d'un bloc WHERE s'étend grâce à l'ajout de filtre de choix logique ELSEWHERE(filtre), de façon identique au bloc IF, ELSE IF, ELSE, END IF ; de plus il peut être étiqueté comme les structures précédentes. Le bloc WHERE s'écrit alors sous la forme suivante :

```
[nom :] WHERE (expression_logique1)
    bloc1
ELSEWHERE (expression_logique2)
    bloc2
    …
ELSEWHERE
    bloc3
END WHERE [nom]
```

III.D LA BOUCLE DO

Très souvent en programmation, nous avons besoin de reproduire un grand nombre de fois les mêmes instructions. La boucle DO sert alors à itérer.

La syntaxe générale est la suivante :

```
[nom :] DO [bloc de contrôle]
        bloc
        END DO [nom]
```

III.D.1 Clause de contrôle itérative

Si d'avance, on sait combien de fois, on a besoin de faire la même série d'opérations, on utilise un compteur entier.

```
[nom :] DO compt = int1, int2 [, int3]
        bloc
        END DO [nom]
```

Où *int1* est la valeur entière de départ du compteur ;

- *int2* est la valeur entière d'arrivée ;
- *int3* est l'incrément entier, par défaut il vaut 1 et est donc optionnel ;
- et *compt* est une variable entière qui prendra les valeurs de *int1* à *int2*, en ajoutant *int3* à chaque fois. Bien entendu, la dernière itération ne tombe pas forcément sur la valeur de *int2*.

Le nombre d'itérations est le plus grand nombre entre *(int2 + int3 – int1) / int3* et *0*.

Exemple III.4

```
DO num = 1, 300
    T(num) = dx * num + y
END DO
```

Attention : num = 4 en fin de boucle !

Exemple III.5

```
alenvers : DO i = 201, 2, -1
    ux = (i - 1) * 5.e-3
    Tx = FUNTEMP (u1, ux, T1, Tx, GT)     ! appel de
fonction
    RT = FUNREST (u1, ux, T1, Tx, GT)     ! appel de
fonction
    PRINT*, ux, Tx, RT
END DO alenvers
```

III.D.2 Clause de contrôle WHILE

Dans ce cas, on ne connaît pas d'avance le nombre d'itérations, mais on connaît plutôt un domaine dans lequel on veut que la série de calculs soit exécuté. Ce domaine est défini par une expression logique et l'itération continue tant que cette expression est VRAIE.

Structure générale de la boucle DO WHILE :

```
[nom :] DO WHILE (expression_logique)
        bloc
    END DO [nom]
```

Exemple III.6

```
Integre_choc : DO WHILE (n < nstep)
    rr = f(1,1) * r1 ; TT = f(3,1) * T1
    CALL RUNGEKUTTA (z, h, f, 3, n, Gas)
    n = z / h
END DO Integre_choc
```

Cette clause de contrôle est assez souvent utilisée pour lire les données d'un fichier dont on ne sait pas le nombre d'enregistrement. On utilise pour cela le marqueur de fin de fichier « eof », associé au paramètre « iostat ». Nous verrons plus loin, avec les accès de fichiers et les entrées-sorties, comment on procède. Mais le boucle prend la forme suivante :

Exemple III.7

```
DO WHILE (eof == 0)
    READ (unit=65, iostat=eof), x, y, z
END DO
```

III.D.3 Boucle infinie et altération

On peut aussi construire des boucles, qui sont à priori infinies, mais dans laquelle on insère une clause de sortie. Bien entendu, aucune boucle infinie n'est concevable, il faut l'interrompre par un moyen ou un autre.

Pour cela on utilise l'instruction EXIT :

```
[nom :] DO [bloc de contrôle]
        bloc
        IF (expression_logique) EXIT [nom]
        END DO [nom]
```

Pour altérer le déroulement d'une boucle, il est possible de forcer l'exécution du programme à passer à la boucle suivante, avec l'instruction CYCLE.

```
[nom :] DO [bloc de contrôle]
        bloc1
        IF (expression_logique) CYCLE [nom]
        bloc2
        END DO [nom]
```

III.E LE SCHÉMA FORALL

D'un point de vue algorithmique, le schéma FORALL remplace certaines boucles DO. Mais pour le compilateur, il facilite la parallélisation de cette partie de code. La syntaxe du schéma FORALL est la suivante :

```
[nom :] FORALL (bloc de contrôle [, bloc de contrôle]
        [, filtre])
        bloc
        END FORALL [nom]
```

Le bloc de contrôle est une commande sur les indices d'un ou plusieurs tableaux. Si plusieurs plages d'indices sont définies à l'aide de plusieurs blocs de contrôle, le domaine doit être rectangulaire. C'est-à-dire que chaque commande sur les indices ne doit pas dépendre des autres indices. La construction des tableaux, ainsi que leur manipulation est détaillée dans le chapitre suivant.

Exemple III.8

```
FORALL (i=3:10, j=2:5) mat(i,j) = 12
```

Exemple III.9

```
FORALL (i=1:n, j=1:m, mat1>0)
mat2(i,j) = log(mat1(i,j))
END FORALL
```

IV LES TABLEAUX

IV.A GESTION MÉMOIRE DES TABLEAUX

Un tableau est un ensemble d'éléments ayant même type et même valeur de paramètre de sous-type. Il est défini par son type, son rang, ses étendues, son profil, sa taille. Un tableau peut contenir jusqu'à 7 dimensions.

- Le rang (*rank*) d'un tableau est son nombre de dimensions.
- L'étendue du tableau dans une dimension est le nombre d'éléments dans cette dimension.
- Le profil (*shape*) d'un tableau est un vecteur de rang 1 et d'étendue le rang du tableau et dont chaque élément contient l'étendue dans la dimension correspondante.
- La taille (*size*) du tableau est le produit des étendues sur toutes les dimensions.

Deux tableaux de même profil sont dits conformant.

Dans le chapitre II.D, nous avons évoqué la déclaration des tableaux. Distinguons les tableaux qui ont une allocation statique de ceux qui ont une allocation dynamique.

IV.A.1 L'allocation statique

Il suffit de préciser le type du tableau, puis l'attribut dimension lors de la déclaration en indiquant uniquement l'étendue de chaque dimension.

```
type, DIMENSION (b_inf1:b_sup1, b_inf2:b_sup2 …) :: nom_tab
```

Où *b_inf1, b_sup1* sont des entiers qui délimitent l'étendue de la première dimension du tableau *nom_tab*. Il sont respectivement l'indice minimum et l'indice maximum. Il en va de même pour la seconde dimension avec les entiers *b_inf2, b_sup2*.

IV.A.2 L'allocation dynamique

On parle d'allocation dynamique lorsque l'on a pas défini au préalable l'étendue du tableau. Il se peut, en effet, que cette donnée ne soit pas connue lors de l'écriture du programme, par contre elle le deviendra pendant l'exécution du programme. Ce phénomène est fréquemment rencontré lorsqu'on lit des données dans un fichier généré par un autre programme. En Fortran 90, nous avons alors la possibilité de gérer de façon dynamique la mémoire allouée à un tableau. Il existe pour cela deux manières distinctes : soit le passage en argument d'un tableau, et de sa dimension, dans une procédure, soit l'allocation directe de mémoire.

La différence principale entre ces deux méthodes est que :

- dans la première, l'emplacement mémoire des tableaux est alloué dès l'entrée dans la procédure et est libéré à sa sortie ;
- dans la seconde c'est le développeur qui décide du moment de l'allocation et du moment de la libération de la mémoire.

a) En argument de procédure : les tableaux automatiques

Ce cas se présente lorsque au moment du développement d'une procédure, on ne connaît pas encore la dimension du tableau, mais le programme appelant cette routine sera capable d'en fournir la valeur. Ou bien lorsque le même sous-programme est appelé à être exécuté, avec des tableaux de tailles très différentes.

Avec passage en argument de la taille, la syntaxe générale se présente sous la forme suivante :

```
SUBROUTINE nom (arg1, n1, arg2, n2 …)
IMPLICIT NONE
INTEGER :: n1, n2
type, DIMENSION (n1) :: arg1
type, DIMENSION (n2) :: arg2
```

Sans passage de la taille en argument, la syntaxe générale se présente sous la forme suivante :

```
SUBROUTINE nom (arg1, arg2, …)
IMPLICIT NONE
type, DIMENSION (:) :: arg1
type, DIMENSION (:) :: arg2
```

Pour connaître l'étendue des tableaux, on a recourt à l'instruction SIZE. Mais, dans ce cas, l'utilisation d'une interface explicite est obligatoire, sinon le compilateur ne peut pas détecter des erreurs potentielles dans la manipulation de ces tableaux. Nous reviendrons sur ce point lorsque nous verrons la notion d'INTERFACE.

b) Avec l'instruction ALLOCATE : les tableaux dynamiques

Pour générer dynamiquement des tableaux et libérer la mémoire au moment opportun, Fortran 90 possède l'instruction ALLOCATE et DEALLOCATE. Au préalable, le tableau considéré devra être déclaré avec l'attribut ALLOCATABLE.

La forme générale a l'allure suivante :

```
type, DIMENSION(:[,…]), ALLOCATABLE :: nom_tab
INTEGER :: ok
…
ALLOCATE (nom_tab(dim1[,…]) [, stat = ok])
… [bloc_instructions]
DEALLOCATE (nom_tab [, stat = ok])
```

L'utilisation du paramètre *stat* est optionnel, c'est un contrôle du bon déroulement de l'allocation ou de la libération de la mémoire. Lorsque l'allocation (ou la libération) s'est correctement effectuée, l'instruction ALLOCATE (ou DEALLOCATE) renvoie pour la variable entière *ok* la valeur 0, sinon *ok* prend une valeur supérieure à 0.

Exemple IV.1

```
SUBROUTINE SOLVER
IMPLICIT NONE
INTEGER :: npas
REAL, ALLOCATABLE, DIMENSION(:) :: Prad, Frad

ALLOCATE (Prad(npas), Frad(npas), stat = ok)
IF (ok > 0) THEN
    PRINT*,"Il y a un probleme d'allocation dans
SOLVER"
    STOP
END IF

READ (18,*) Prad, Frad
…
DEALLOCATE (Prad, Frad)
END SUBROUTINE SOLVER
```

Il existe en plus une fonction qui permet de s'assurer que la mémoire du tableau que l'on manipule est allouée ou non :

```
allocated (array)
```

Elle renvoie la valeur logique « .TRUE. » si le tableau allouable *array* est alloué, sinon la valeur « .FALSE. ».

IV.B LA MANIPULATION DES TABLEAUX

a) Généralités

Fortran 90 permet la manipulation globale des tableaux, ou de section de tableaux, en indiquant seulement leur nom. Cependant les tableaux doivent être conformants pour être utilisés dans les mêmes instructions. Tous les opérateurs utilisés avec des variables scalaires sont alors utilisables avec des tableaux.

L'initialisation d'un tableau au moment de sa déclaration est permise à l'aide d'un constructeur de tableaux. C'est un vecteur de scalaires borné par les signes « (/ » et « /) ».

Remarque importe : la valeur d'une expression faisant intervenir un tableau est entièrement évaluée avant son affectation.

b) Globale

Attention toutefois à la simplicité de l'écriture, car bien qu'elle allège les lignes de programme, il ne faut pas oublier que derrière un simple identifiant peut se cacher un tableau monstrueux ! D'un autre côté, cette écriture compactée fait ressortir l'essentiel de la programmation et évite bien des erreurs sur les bornes et dimensions de tableaux.

Exemple IV.2

```
REAL, DIMENSION(4) :: tab
…
tab = 100.
```

Chacun des 4 éléments du tableau *tab* prend la valeur 100.

Exemple IV.3

```
REAL, DIMENSION(4) :: tab1, tab2, tab3
…
tab1 = 100.
tab2 = 2 + tab1
```

Chacun des 4 éléments du tableau *tab2* prend la valeur 102.

Exemple IV.4

```
INTEGER :: i
REAL, DIMENSION(4) :: tab1, tab2=(/1,2,3,4)/)
…
tab1 = (/10.,15.,20.,25./)
tab2 = tab2 * tab1
```

Chacun des 4 éléments du tableau *tab2* prend respectivement la valeur 10 ; 30 ; 60 ; 100.

Exemple IV.5

```
INTEGER :: i
REAL, DIMENSION(4) :: tab1, tab2=(/ (i, i=1,4)/)
…
tab1 = 100. ; tab2 = tab2 * tab1
```

Chacun des 4 éléments du tableau *tab2* prend respectivement la valeur 100 ; 200 ; 300 ; 400.

On peut aussi repéré par le caractère « : » l'étendue du tableau. La notation est alors plus lourde, mais permet au développeur de ne pas oublier qu'il manipule un tableau à n dimension à ce moment. Pour reprendre l'exemple précédent, on peut écrire sous la forme suivante.

Exemple IV.6

```
INTEGER :: i
REAL, DIMENSION(4) :: tab1, tab2=(/ (i, i=1,4)/)
…
tab1(:) = 100. ; tab2(:) = tab2(:) * tab1(:)
```

Dans le cas de tableau à plusieurs dimension, on mentionnera chacune des dimensions par « : ».

```
REAL, DIMENSION(4,16) :: tab1, tab2
…
tab2(:) = 20. / tab1(:,:)
```

c) Par section

On peut aussi faire référence à une partie d'un tableau appelée section de tableau. Le sous-tableau ainsi défini est aussi un tableau.

Les indices sont donnés soit par une suite arithmétique lorsqu'on a une section régulière du tableau initial, soit par un vecteur d'indices, lorsqu'on veut extraire une section non régulière.

La section régulière prend une forme typique : *b_inf : b_sup : inc*

Où *b_inf* et *b_sup* sont respectivement les indices inférieure et supérieure de la section choisie, et *inc* est un entier qui représente l'incrément. Par défaut *inc* vaut 1. C'est le même principe que dans la boucle DO avec clause itérative.

Exemple IV.7

```
INTEGER :: i
REAL, DIMENSION(20) :: tab1 = (/ (i, i=1,20)/)
REAL, DIMENSION(5)  ::
…
tab2(:) = tab1(1:20:4)
```

De plus, on peut combiner les différentes manipulations possibles, en voici un certain nombre d'exemples.

Exemple IV.8

```
! on sélectionne les éléments de la ligne 2
!                et des colonnes de 4 à 8
    tab1(2,4:8)

! on sélectionne tous les éléments de la ligne 2
    tab1(2,:)

! on sélectionne tous les éléments des lignes 1, 2 et 3
    tab1(:3,:)

! on sélectionne les éléments des lignes 1, 3 et 5
!                    et des colonnes de 4 à n
    tab1(1:6:2,4:n)
```

La section quelconque prend la forme d'un vecteur d'indices : (/i1, i2, i3…/), a l'aide du constructeur de vecteurs.

Exemple IV.9

```
! on sélectionne les éléments des lignes 1, 5 et 19
!                    et des colonnes de 4, 5 et 10
    tab1((/1,5,19/), (/4,5,10/))
```

On peut aussi affecter directement une section avec l'identifiant d'un vecteur.

Exemple IV.10

```
INTEGER, DIMENSION(4) :: vect_ind = (/3,6,7,9/)
    tab1(vect_ind, (/4,5,10/))
```

Lorsqu'un sous-tableau est constitué d'éléments répétés, il ne peut pas recevoir une nouvelle affectation, donc il ne peut être placé à gauche d'une expression.

Exemple IV.11

```
INTEGER, DIMENSION(4)  :: vect_ind = (/3,6,3,9/)
INTEGER, DIMENSION(10) :: tab1
INTEGER, DIMENSION(4)  :: tab2
! on peut ecrire :
    tab2 = tab1(vect_ind)
! mais il est interdit d'écrire :
tab1(vect_ind) = …
```

IV.C LES FONCTIONS INTRINSÈQUES SUR LES TABLEAUX

Fortran 90 propose toute une gamme de fonctions intrinsèques qui offre l'avantage de simplifier l'utilisation des tableaux. Nous distinguerons les fonctions passives qui renvoient seulement une information sur le tableau des fonctions actives qui modifient le tableau.

Dans la suite, on utilisera comme argument les mots suivants :

- *array* qui désigne le tableau sur lequel on applique la fonction ;
- *vect* qui désigne un vecteur ;
- *mask* qui désigne un tableau d'expressions logiques conformant au tableau *array* ;
- *dim* qui restreint la fonction à s'appliquer à la dimension mentionnée.

IV.C.1 Les fonctions d'interrogation

```
maxloc (array [, mask] [, dim])
minloc (array [, mask] [, dim])
```

Ces fonctions retournent respectivement un tableau de rang 1 correspondant à la position de l'élément maximum et de l'élément minimum du tableau *array*. Fortran 95 admet en plus la précision de la dimension *dim*, ce que ne fait pas Fortran 90.

```
lbound (array [, dim])
ubound (array [, dim])
```

Ces fonctions retournent respectivement un tableau d'entiers du type par défaut et de rang 1 qui contient la liste des bornes inférieures du tableau *array* (ou de la dimension *dim* mentionnée) et la liste des bornes supérieures du tableau *array* (ou de la dimension *dim* mentionnée).

```
size (array [, dim])
```

Cette fonction retourne la taille totale du tableau *array* ou l'étendue le long de la dimension *dim*.

```
shape(array)
```

Cette fonction retourne un tableau d'entiers de rang 1 qui contient le profil du tableau *array*.

Exemple IV.12

```
INTEGER, DIMENSION(-1:1,3) :: tab1
```

$$tab1 = \begin{pmatrix} -15 & 2 & 35 \\ 4 & 50 & -36 \\ -7 & 28 & -9 \end{pmatrix}$$

```
maxloc (tab1)    donne (/2, 2/)
minloc (tab1)    donne (/2, 3/)
lbound (tab1)    donne (/-1, 1/)
ubound (tab1)    donne (/1, 3/)
size (tab1)      donne 9
shape(tab1)      donne (/3, 3/)
```

`allocated (array)`

Cette fonction fournit la valeur « .TRUE. » si le tableau *array*, déclaré avec *allocatable*, est alloué et la valeur « .FALSE. » sinon.

IV.C.2 Les fonctions de réduction

Les fonctions de réduction renvoient soit un scalaire soit un tableau d'un rang inférieur à celui passé en argument.

a) Les fonctions all et any

`all (mask [, dim])`

Cette fonction renvoie la valeur logique « .TRUE. », si tous les éléments, pris un par un, répondent aux critères du masque, sinon la valeur « .FALSE. ». Si *dim* est précisé, la fonction renvoie un tableau qui contient les valeurs logiques « .TRUE. » ou « .FALSE. » suivant que les éléments de cette dimension répondent aux critères du masque.

Exemple IV.13

```
tab1(1,:) = (/1, 3, 5, 7/) ; tab2(1,:) = (/1, 2, 3, 4/)
tab1(2,:) = (/2, 4, 6, 8/) ; tab2(2,:) = (/5, 6, 7, 8/)
! reduction globale
ALL (tab1==tab2)        donne .false.
ALL (tab1/=tab2)        donne .false.
! reduction par colonne
ALL (tab1/=tab2, dim = 1)
    donne /(.false., .true., .true., .false./)
! reduction par ligne
ALL (tab1/=tab2, dim = 2)
    donne /(.false., .false./)
```

Si on a déjà des tableaux de variables logiques, on les utiliser directement dans l'instruction ALL.

Exemple IV.14

```
tab1(1,:) = (/.true., .true., .true., .true./)
ALL (tab1)  donne .true.
```

On peut aussi tester les valeurs d'un tableau.

Exemple IV.15

```
REAL, DIMENSION (5,10) :: tab1, tab2
…
IF (ALL(tab1) >= 0) THEN
    tab2 = SQRT (tab1)
END IF
```

De façon symétrique à la fonction *all*, on peut utiliser la fonction suivante :

```
any (mask [, dim])
```

Cette fonction renvoie la valeur logique « .TRUE. », si au moins un des éléments, pris un par un, répondent aux critères du masque, sinon la valeur « .FALSE. ». Si *dim* est précisé, la fonction renvoie un tableau qui contient les valeurs logiques « .TRUE. » ou « .FALSE. » suivant que les éléments de cette dimension répondent aux critères du masque.

Exemple IV.16

```
tab1(1,:) = (/1, 3, 5, 7/) ; tab2(1,:) = (/1, 2, 3, 7/)
tab1(2,:) = (/2, 4, 6, 8/) ; tab2(2,:) = (/5, 6, 4, 8/)
! reduction globale
ANY (tab1==tab2)        donne .true.
ALL (tab1/=tab2)        donne .true.
! reduction par colonne
ALL (tab1/=tab2, dim = 1)
    donne /(.true., .true., .true., .false./)
! reduction par ligne
ALL (tab1/=tab2, dim = 2)
    donne /(.true., .true./)
```

Si on a déjà des tableaux de variables logiques, on les utiliser directement dans l'instruction ALL.

Exemple IV.17

```
tab1(1,:) = (/.true., .true., .false., .true./)
ANY (tab1)  donne .true.
```

On peut aussi tester les valeurs d'un tableau.

Exemple IV.18

```
REAL, DIMENSION (5,10) :: tab1, tab2
…
IF (ANY(tab1) <= 0) THEN
    PRINT*, 'On ne peut pas calculer la racine carree.'
    EXIT
END IF
```

b) La fonction count

`count(mask [, dim])`

Cette fonction compte le nombre d'occurrences qui prennent la valeur logique « .TRUE. », suivant les critères du masque. Si *dim* est précisé, la fonction renvoie un tableau d'entiers contenant le nombre d'occurrences VRAI.

Exemple IV.19

```
tab1(1,:) = (/1, 3, 5, 7/) ; tab2(1,:) = (/1, 2, 3, 7/)
tab1(2,:) = (/2, 4, 6, 8/) ; tab2(2,:) = (/5, 6, 4, 8/)
! compte global
COUNT (tab1==tab2)      donne 3
COUNT (tab1/=tab2)    donne 5
! compte par colonne
COUNT (tab1/=tab2, dim = 1)   donne /(1, 2, 2, 0/)
! compte par ligne
COUNT (tab1/=tab2, dim = 2)     donne /(2, 3/)
```

Si on a déjà des tableaux de variables logiques, on les utiliser directement dans l'instruction ALL.

Exemple IV.20

```
tab1(1,:) = (/.true., .true., .false., .true./)
COUNT (tab1)     donne 3
```

c) Les fonctions maxval et minval

`maxval (array [, dim] [, mask])`
`minval (array [, dim] [, mask])`

Ces fonctions renvoient respectivement la valeur maximum et la valeur minimum du tableau *array*, ou le tableau des maxima et des minima suivant la dimension du tableau imposée et selon les critères du masque s'il est précisé.

Note : Si le tableau est de taille nulle, le résultat est respectivement le plus grand nombre positif et négatif disponible sur l'ordinateur.

Exemple IV.21

```
tab1(1,:) = (/1, 3, 5, 7/) ; tab2(1,:) = (/1, 2, 3, 7/)
tab1(2,:) = (/2, 4, 6, 8/) ; tab2(2,:) = (/5, 6, 4, 8/)
! reduction globale
MAXVAL (tab1)              donne 8
! reduction partielle, par colonne
MINVAL (tab1,DIM=1,MASK=tab1>1) donne (/2, 3, 5, 7/)
```

d) Les fonctions sum et product

```
sum (array [, dim] [, mask])
product (array [, dim] [, mask])
```

Ces fonctions calculent respectivement la somme et le produit des éléments d'un tableau de nombres ; ou suivant la dimension indiquée et selon les critères du masque, s'il est précisé.

Note : Si le tableau est de taille nulle, le résultat est respectivement 0 et 1.

Exemple IV.22

$$tab1 = \begin{pmatrix} 1 & 3 & 5 & 7 \\ 2 & 4 & 6 & 8 \end{pmatrix} \quad et \quad tab2 = \begin{pmatrix} 1 & 2 & 3 & 7 \\ 5 & 6 & 4 & 8 \end{pmatrix}$$

```
SUM(tab1)                              donne 36
PRODUCT(tab1, MASK=tab1<6)             donne 120
SUM(tab2, DIM=2)                       donne (/13, 23/)
PRODUCT(tab2, DIM=2, MASK=tab2>4)      donne (/7, 240/)
```

IV.C.3 Les fonctions de construction et de transformation

a) La fonction merge

```
merge (array1, array2, mask)
```

Cette fonction fabrique un tableau, à partir des deux tableaux conformants et de même type *array1* et *array2*, en fonction des critères imposés par le masque. Si la valeur du masque est vrai, on utilise l'élément de *array1*, si la valeur du masque est faux on utilise l'élément de *array2*.

Exemple IV.23

$$tab1 = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \quad et \quad tab2 = \begin{pmatrix} 10 & 20 & 30 \\ 40 & 50 & 60 \\ 70 & 80 & 90 \end{pmatrix}$$

```
MERGE (tab1, tab2, tab1>4)
```

$$\text{donne} \quad tab1 = \begin{pmatrix} 10 & 20 & 30 \\ 40 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

b) La fonction reshape

`reshape (array, shape [, pad] [, order]))`

Cette fonction construit un tableau, à partir des éléments du tableau array, et dont le profil est donné par le tableau d'entiers positifs de rang 1 *shape*. L'attribut optionnel *pad* est un tableau du même type de *array*, dont les éléments seront utilisés si la taille de *shape* est supérieure à celle de *array*.

Exemple IV.24

```
RESHAPE ( (/(i, i=1,12)/), (/3, 4/))
```

donne $\begin{pmatrix} 1 & 4 & 7 & 10 \\ 2 & 5 & 8 & 11 \\ 3 & 6 & 9 & 12 \end{pmatrix}$

c) Les fonctions pack et unpack

`pack (array, mask [, vector])`

Cette fonction construit un tableau de rang 1, en compressant les valeurs du tableau *array*, suivant les critères du masque. Si le vecteur optionnel *vector* est présent, le résultat aura la taille de *vector*.

`unpack (vector, mask [, field])`

Cette fonction est assez symétrique de la précédente. Elle construit un tableau de rang supérieur à 1, en décompressant les valeurs du vecteur *vector*, suivant les critères du masque. Si le tableau optionnel *field* est présent, les valeurs de *field*, qui sont du même type que celle de *vector*, seront utilisées lorsque le masque contient une valeur fausse. Le résultat aura le profil du masque et le type du vecteur.

Exemple IV.25

On a *tab*1= $\begin{pmatrix} 1 & 4 & 7 & 10 \\ 2 & 5 & 8 & 11 \\ 3 & 6 & 9 & 12 \end{pmatrix}$

```
vect1 = (/ (i, i=-12,-1) /)
PACK (tab1, MASK=tab1<10, VECTOR=vect1)
donne   (/1, 2, 3, 4, 5, 6, 7, 8, 9, -3, -2, -1 /)
```

d) La fonction spread

`spread (array, dim, n)`

Cette fonction crée un nouveau tableau en dupliquant *n* fois les valeurs du tableau *array*, le long de la dimension mentionnée. Le résultat est un tableau d'un rang supérieur à *array*.

e) Les fonctions cshift et eoshift

`cshift (array, shift [, dim])`

Cette fonction opère un décalage circulaire sur les éléments du tableau *array*, le long de la dimension indiquée, d'autant d'indices que *shift* indique.

Attention : *shift* est un entier positif, pour un décalage vers les indices décroissants (on effectue i = i-shift), ou un entier négatifs, pour un décalage vers les indices croissants. Par défaut *dim* est à 1.

```
eoshift (array, shift [, boundary] [, dim])
```

Cette fonction agit comme la précédente, mais donne en plus la possibilité de remplacer les éléments perdus à la frontière par des éléments de remplissage donnés par le tableau *boundary*.

Note : Si *boundary* n'est pas présent, les valeurs de remplissage par défaut dépendent du type du tableau *array* :

- pour un INTEGER, la valeur par défaut est 0 ;
- pour un REAL, " " 0.0 ;
- pour un COMPLEX, " " (0.0,0.0) ;
- pour un LOGICAL, " " .FALSE. ;
- pour un CHARACTER, " " un chaîne composée de blancs.

Exemple IV.26

On a *tab*1= $\begin{pmatrix} 1 & 4 & 7 & 10 \\ 2 & 5 & 8 & 11 \\ 3 & 6 & 9 & 12 \end{pmatrix}$

```
CSHIFT (tab1, SHIFT=2, DIM=1)
```

donne $\begin{pmatrix} 3 & 6 & 9 & 12 \\ 1 & 4 & 7 & 10 \\ 2 & 5 & 8 & 11 \end{pmatrix}$

```
EOSHIFT (tab1, SHIFT=-1, DIM=2)
```

donne $\begin{pmatrix} 0 & 1 & 4 & 7 \\ 0 & 2 & 5 & 8 \\ 0 & 3 & 6 & 9 \end{pmatrix}$

IV.C.4 Les fonctions propres aux matrices et vecteurs

a) Multiplication de vecteurs et de matrices

Pour effectuer le produit scalaire de deux vecteurs :

```
dot_product (vect1, vect2)
```

et la multiplication de deux matrices :

```
matmul (mata, matb)
```

b) **matrice transposée**

```
transpose (matrix)
```

Cette fonction renvoie la matrice transposée du tableau *matrix*.

V LES POINTEURS

V.A INTRODUCTION

V.A.1 Définition :

Le pointeur est un objet qui peut désigner des objets différents au cours de l'exécution d'un programme, il reçoit l'attribut POINTER. Il agit comme un alias et est du même type que les objets qu'il pointe. L'objet désigné par le pointeur est appelé la cible du pointeur, il doit recevoir l'attribut TARGET. Pour assigner un objet à un pointeur, on utilise le symbole de la double flèche matérialisée pas les caractères : « => ». Un pointeur a aussi le droit d'être valoriser vers un autre pointeur.

La notion de pointeur est surtout utile dans le contexte de l'allocation dynamique de mémoire.

V.A.2 Déclaration d'un pointeur et d'une cible

```
REAL, TARGET :: targ
REAL, POINTER :: ptr
```

Le type, le paramètre de type et le rang du pointeur et de la cible doivent être identiques.

Et pour affecter *targ* comme cible de *ptr* (ou pour valoriser le pointeur) : ptr => targ.

Remarque : pour déclarer un tableau de pointeurs, seul son rang (le nombre de dimensions) doit être déclaré et ses bornes prendront les valeurs de l'objet pointé.

Exemple V.1

```
REAL, POINTER :: ptr1
INTEGER, DIMENSION(:,:), POINTER :: ptr2
! attention ptr3 n'est pas un tableau de pointeurs.
COMPLEX, POINTER :: ptr3(:)
```

V.B ÉTAT D'UN POINTEUR

Un pointeur se trouve forcément dans un des trois états suivants :

- indéfini : comme lors de sa déclaration,
- nul : il ne pointe sur rien, c'est-à-dire qu'il n'est pas l'alias d'une variable,
- ou associé : il pointe sur une variable.

Note : en Fortran 95, on peut forcer un pointeur à l'état nul, lors de sa déclaration.

```
REAL, POINTER :: ptr => NULL()
```

a) La fonction nullify

L'instruction NULLIFY permet de forcer un pointeur à l'état nul.

```
syntaxe : NULLIFY (ptr1)
! norme 95 : remplacee par la valorisation
ptr1 => NULL()
```

b) La fonction associated

L'état d'un pointeur peut être connu grâce à la fonction intrinsèque ASSOCIATED, dont la syntaxe générale est la suivante :

```
associated (pointer [, pointer] [, target])
```

`associated (pointer)`

Cette fonction renvoie « .TRUE. », si *pointer* est associé à une cible, s'il est dans l'état nul, la fonction renvoie « FALSE. ».

`associated (pointer1, pointer2)`

Cette fonction renvoie « .TRUE. », si *pointer1* et *pointer2* sont associés à la même cible, sinon elle renvoie « FALSE. ». Remarque : si *pointer1* et *pointer2* sont tous les deux dans l'état nul, elle renvoie « FALSE. ».

`associated (pointer, target)`

Cette fonction renvoie « .TRUE. », si *pointer* est associé à la cible *target*, sinon elle renvoie « FALSE. ».

Attention : le pointeur ne doit pas être dans l'état indéterminé pour utiliser cette fonction !

c) Opération sur les pointeurs

On a vu que l'opérateur « => » sert à valoriser le pointeur. Le pointeur prend alors la valeur de la cible.

L'opérateur « = » sert à affecter une valeur, mais attention lorsque les opérandes sont des pointeurs, l'affectation porte sur la cible et non sur le pointeur.

V.C L'ALLOCATION DYNAMIQUE

L'instruction ALLOCATE, que nous avons déjà vu pour l'allocation dynamique de mémoire des tableaux, sert aussi à associer un pointeur et à lui allouer de la mémoire. L'instruction DEALLOCATE sert à libérer la mémoire, comme pour un tableau.

Notes :

- Dans ce cas, l'espace alloué n'a pas de nom et la seule façon d'y accéder est d'utiliser le pointeur. Le pointeur est l'alias direct du chemin qui mène à cette mémoire.
- Après l'instruction DEALLOCATE, le pointeur est dans l'état nul.
- On ne peut pas exécuter l'instruction DEALLOCATE sur un pointeur dont l'état est indéterminé.
- On peut aussi faire de l'allocation dynamique de mémoire pour des variables, par l'intermédiaire de pointeurs.

Exemple V.2 : tri de chaîne de caractères

```
MODULE CHAINE

END MODULE CHAINE
PROGRAM TRI_CHAINE
IMPLICIT NONE
CHARACTER (LEN=80), DIMENSION(:), TARGET  :: chaine
CHARACTER (LEN=80), DIMENSION(:), POINTER :: ptr_ch
CHARACTER (LEN=80), POINTER :: ptr_int
INTEGER :: num, i
LOGICAL :: fini

PRINT*, "Quel est le nombre de chaines ? "
READ(*,*) num

ALLOCATE(chaine(num), ptr_ch(num))
lis : DO i = 1, num
    PRINT*, "Donnez votre message : "
    READ(*,*) chaine(i)
    ptr_ch(i) => chaine(i)
END DO
```

```
tri_tous : DO
    fini = .TRUE.
    tri_chacun : DO i = 1, num-1
        IF (chaine(i) > chaine(i+1)) THEN
            fini = .FALSE.
            ptr_int     => ptr_ch(i)
            ptr_ch(i)   => ptr_ch(i+1)
            ptr_ch(i+1) => ptr_int
        END IF
    END DO tri_chacun
    IF (fini) EXIT
END DO tri_tous
END PROGRAM TRI_CHAINE
```

Dans l'exemple précédent, on aurait pu faire le même tri à l'aide d'un tableau ALLOCATABLE, à la place du tableau de pointeurs. Néanmoins, il paraît que le temps d'exécution d'un programme est amélioré dans le cas de tableaux de grandes tailles et de longues chaînes de caractères ! Car la gestion de la mémoire diffère dans ces deux cas.

V.C.1 En argument de procédure

Il est possible de passer en argument de procédures un pointeur. Nous distinguons le cas où le pointeur est déjà dans la définition de la procédure, du cas où l'argument muet n'est pas un pointeur.

a) L'argument muet a l'attribut pointer

Dans ce cas, le pointeur figure obligatoirement dans l'appel de la procédure et n'est pas forcément associé au moment de l'appel. L'information réelle qui passe dans la procédure est le descripteur du pointeur : c'est-à-dire son adresse, ses dimensions…

Remarques :

- L'interface doit être explicite, pour que le compilateur sache que la procédure possède en argument muet un pointeur.
- L'attribut *intent* ne peut pas être utilisé pour qualifier le pointeur qui est en argument muet.

b) L'argument muet n'a pas l'attribut pointer

Dans ce cas, le pointeur est obligatoirement associé avant l'appel de la procédure. C'est alors l'information concernant la cible qui passe dans la procédure, c'est-à-dire son adresse. On se retrouve alors dans un cas habituel de passage de variables dans un appel de procédure.

c) Cible en argument de procédure

Si un argument en appel de procédure a l'attribut TARGET, tout pointeur l'ayant pour cible au moment de l'appel, devient indéfini au retour de la procédure. De façon générale, la norme ne garantit pas la conservation de l'association du pointeur entre sa cible passée en argument d'appel de procédure et la cible correspondante en argument muet.

VI CONTRÔLE DE VISIBILITÉ

VI.A LES RESSOURCES PRIVÉES ET PUBLIQUES

Il est souvent souhaitable que l'intégralité des ressources d'un module ne soit pas accessible par les autres unités de programme. En effet, lorsqu'on définit un module, on souhaite faire partager un certain nombre de variables, de définitions de type, de procédures, via l'instruction *use nom_module* ; mais pour des besoins spécifiques à ce module, on est en droit de définir des ressources propres à ce module.

- Les ressources non visibles depuis l'extérieur sont dites privées (private).
- Tandis que les ressources visibles depuis l'extérieur sont dites publiques (public).

Par défaut, toutes les ressources d'un module sont publiques.

On a souvent intérêt à rendre privées certaines ressources d'un module, cela évite les risques de conflits avec les ressources d'un autre module.

Une autre façon de restreindre l'utilisation des ressources d'un module est d'insérer l'instruction *only nom_var*, lors de l'accès à un module, dans l'instruction *use nom_module* (voir plus haut chapitre II.A.3a)).

a) Les instructions PUBLIC et PRIVATE

Le mode par défaut est le mode public. Les instructions PUBLIC et PRIVATE, sans argument, permettent de changer le mode. Un telle instruction ne peut apparaître qu'une seule fois dans un module et affecte l'ensemble du module.

```
MODULE PARAM
IMPLICIT NONE
INTEGER, DIMENSION(:) :: par
PRIVATE     ! tout le module devient privé.
REAL :: x, y
INTEGER :: i, j
END MODULE PARAM
```

b) Les attributs PUBLIC et PRIVATE

Si l'on veut rendre public ou privé n'importe quel objet d'un module (ou d'une procédure), on lui affecte soit l'attribut PUBLIC ou soit l'attribut PRIVATE, directement lors de sa déclaration.

Syntaxe générale :

```
type, private :: var
type, public  :: var
```

Attention toutefois à ne pas confondre le fonctionnement des attributs PRIVATE et PUBLIC, avec celui des instructions PRIVATE et PUBLIC, malgré leur similitude.

```
MODULE INIT
IMPLICIT NONE
INTEGER, DIMENSION(:) :: par
PRIVATE                 ! rend l'environnement privé.
REAL :: x, y
INTEGER :: i, j
REAL, DIMENSION(:), PUBLIC :: vect
        ! rend public le tableau "vect", donc exportable.
PUBLIC :: lect

CONTAINS
    SUBROUTINE lect (x,y)
    …                   ! cette procédure est publique.
    END SUBROUTINE lect
    FUNCTION dict (vect)
    …                   ! cette procédure est privée.
    END FUNCTION dict
END MODULE INIT
```

VI.A.2 les types dérivés semi-privés

On parle de type dérivé semi-privé lorsqu'un type dérivé, défini au chapitre II.D.2, est public, mais dont toutes les composantes sont privées. Son intérêt est de permettre au développeur de modifier la structure du type dérivé, sans affecter les autres unités de programme qui l'utilisent.

Les attributs PUBLIC et PRIVATE s'appliquent aux types dérivés, comme pour les autres types de variables.

Alors un type dérivé est soit :

- transparent : il est public et ses composantes sont aussi publiques ;
- semi-privé : il est public et ses composantes sont toutes privées ;
- ou privé.

Exemple VI.1

```
MODULE BASE
    TYPE STAR
        PRIVATE
        INTEGER :: num
        REAL, DIMENSION(:), POINTER :: lambda
    END TYPE STAR
END MODULE BASE
```

Dans cet exemple, de définition d'un type semi-privé, seules des procédures internes au *module base* peuvent accéder aux champs de la variable de type *star*.

Exemple VI.2

```
MODULE BASE
    TYPE STAR
        PRIVATE
        INTEGER :: num
        REAL, DIMENSION(:), POINTER :: lambda
    END TYPE STAR
CONTAINS
    FUNCTION VALOR(b1, b2)
    IMPLICIT NONE
    TYPE (STAR) :: b1, b2
    …
    b1%num = b2%num - 1
    valor = (b1%lambda + b2%lambda) / 2.
    …
    END FUNCTION VALOR
    …
END MODULE BASE
```

VI.A.3 Restriction de l'instruction USE

De la même manière que l'on peut rendre privé certaines ressources d'un module, on peut limiter l'accès aux ressources d'un module, lors de l'instruction USE, dans l'unité utilisatrice.

Syntaxe générale :

```
USE nom_module, ONLY : list_var
```

Exemple VI.3

```
MODULE PARAM
IMPLICIT NONE
INTEGER, DIMENSION(:) :: par
REAL :: x, y
INTEGER :: i, j
END MODULE PARAM

PROGRAM MACHINE
USE PARAM, ONLY : x, y
! seules ces 2 variables sont accessibles
…
END PROGRAM MACHINE
```

Il est aussi possible de renommer des ressources d'un module, si les noms proposés par le module ne conviennent pas à l'unité utilisatrice : par exemple lorsqu'un de ces noms est déjà affecté ! Pour renommer, on utilise l'opérateur d'affectation « => », identique à celui des pointeurs.

Syntaxe générale :

```
USE nom_module, ONLY : old_var=>new_var, old_vec=>new_vec
```

Exemple VI.4

```
MODULE PARAM
IMPLICIT NONE
INTEGER, DIMENSION(:) :: par
REAL :: x, y
INTEGER :: i, j
END MODULE PARAM

PROGRAM MACHINE
USE PARAM, ONLY : x=>varx, y=>vary
…
END PROGRAM MACHINE
```

VII CONTRÔLE DE COHÉRENCE

VII.A LES ARGUMENTS DE PROCÉDURE

VII.A.1 Vocation des arguments

Lors de la déclaration des arguments de procédure, il est possible d'affiner le contrôle sur la manipulation de ces arguments, au cours de la procédure, en leur spécifiant une vocation :

- être un paramètre d'entrée seulement, donc ne pas recevoir de nouvelle affectation ;
- être un paramètre de sortie seulement, donc recevoir certainement une affectation ;
- ou bien, être à la fois un paramètre d'entrée et de sortie de la procédure.

Pour spécifier cette vocation à être un paramètre d'entrée ou de sortie de procédure, on utilise l'attribut INTENT, qui peut être *in, out* ou *inout*.

Syntaxe générale :

```
procedure nom_proc (arg1, arg2, …, argn)
     type, INTENT(in)    :: list_arg1
(et/ou)
     type, INTENT(out)   :: list_arg2
(et/ou)
     type, INTENT(inout) :: list_arg3
```

Où *list_arg1*, *list_arg2* et *list_arg3* sont des listes de variables prises parmi les arguments de la procédure.

Exemple VII.1

```
SUBROUTINE INIT (r1, u1, T1)
IMPLICIT NONE
REAL, INTENT(IN) :: r1, u1, T1
REAL :: Mach, Miso, Mrad, Mcon
```

```
! Calcul des constantes du systeme d'equations.
! gam = rapport des gaz parfaits gam = Cp / Cv.
  gam = 5. / 3.
! Mach = Cshock / Csound = u1/Cs ; nombre de Mach
! adimensionne
  Mach = u1 * sqrt(AtW/(gam*Rgaz*Tdeg*T1))
  WRITE (6,*) "Le nombre de Mach est de ", Mach
  gM2  = gam*Mach*Mach        ! adimensionne
  CvM2 = 1.5 / (Mach*Mach)     ! adimensionne


! On a pose dans le calcul : w = (a T1**4) / (3 rho1 u1**2)
  ww = 7.568e-15 * (T1*Tdeg)**4 / (3.*r1*u1*u1)        !
[erg/cm3]
! On a aussi pose : l = c/u1 /(Krossland*L), mais on en a
pas encore besoin.
  Miso = sqrt((3.*gam-1.)/(gam*(3.-gam)))
  PRINT*,"Le seuil isotherme est ", Miso
  Mrad = 7**(7./6.) * (6*gam)**(-0.5) * (ww * gM2)**(-
1./6.)
  PRINT*,"Le seuil radiatif est ", Mrad
  Mcon = 2.38 * Mrad
  PRINT*,"Le seuil continu est ", Mcon


END SUBROUTINE INIT
```

Dans l'exemple précédent, les arguments *r1*, *u1* et *T1* sont déclarés avec l'attribut INTENT(IN), la procédure *init* ne peut donc pas modifier leur valeur. Il est interdit d'écrire : *r1* = …, *u1* = …, ou *T1* = ….

Exemple VII.2

```
SUBROUTINE LECTURE (rr, uu, TT)
IMPLICIT NONE
REAL, INTENT(OUT) :: rr, uu, TT

! Lecture des conditions initiales.
OPEN (unit=12, file="condinit.dat", status="old",&
              form="formatted")
    READ(12,*)
    READ(12,'(11x,e8.3)') rr
        WRITE(6,*) 'Density in g/cm-3: ', rr
    READ(12,'(11x,e8.3)') uu
        WRITE(6,*) 'Shock speed in cm/s: ', uu
    READ(12,'(11x,f7.3)') TT
        WRITE(6,*) 'T in eV: ', TT
CLOSE (12)

END SUBROUTINE LECTURE
```

Dans l'exemple précédent, les arguments *rr, uu* et *TT* sont déclarés avec l'attribut *intent(out)*, la procédure *lecture* doit donc leur fournir une valeur.

Remarque : cependant, l'expérience montre que le comportement du compilateur dépend aussi de son constructeur : certain compilateur ne signale aucune erreur, ni même message d'attention, lors d'une mauvaise utilisation des variables à vocation INTENT(OUT*)*.

VII.A.2 Présence optionnelle des arguments

L'attribut OPTIONAL permet de déclarer certains arguments comme optionnels et leur présence éventuelle est testée à l'aide de la fonction intrinsèque d'interrogation PRESENT. Cette précaution évite l'utilisation systématique de l'argument, or que l'on sait pertinemment qu'il ne sera pas forcément toujours présent.

Syntaxe générale :

```
type, optional [, attributs] :: var
```

Exemple VII.3

```
SUBROUTINE LECTURE (rr, uu, TT, AtN)
IMPLICIT NONE
REAL, INTENT(OUT) :: rr, uu, TT
REAL, OPTIONAL, INTENT(OUT) :: AtN

! Lecture des conditions initiales.
OPEN (unit=12, file="condinit.dat", status="old",&
              form="formatted")
    READ(12,*)
    READ(12,'(11x,e8.3)') rr
        WRITE(6,*) 'Density in g/cm-3: ', rr
    READ(12,'(11x,e8.3)') uu
        WRITE(6,*) 'Shock speed in cm/s: ', uu
    READ(12,'(11x,f7.3)') TT
        WRITE(6,*) 'T in eV: ', TT

    IF (PRESENT(AtN)) THEN
        READ(12,'(11x,i2)')   AtN
            WRITE(6,*)   'Atomic number: ', AtN
    END IF
CLOSE (12)

END SUBROUTINE LECTURE
```

VII.A.3 Passage d'arguments par mots clés

Dans un appel de procédure, il devient possible de repérer les arguments, non plus par leur position, mais aussi par le nom de l'argument correspondant dans l'interface de procédure. Cet appel par mot clé est très pratique, surtout lorsque la liste des arguments contient des arguments optionnels. Il est même recommandé d'utiliser les mot clés pour les arguments optionnels. Toutefois les arguments, qui ne sont encore repérés uniquement par leur position, doivent obligatoirement figurer en tête de liste, et seul le dernier pourra être omis (s'il est optionnel).

Exemple VII.4

```
REAL SUBROUTINE EVALDIM (a, b, n, min, max, prec)
REAL, INTENT (IN), DIMENSION(n) :: a, b
REAL, INTENT (OUT), OPTIONAL :: min, max, prec
…
END SUBROUTINE EVALDIM
```

```
…
```

```
! Appels corrects
CALL EVALDIM (x, y)
CALL EVALDIM (x, y, min=eps)
CALL EVALDIM (min=eps, max=hug, a=x, b=y)
CALL EVALDIM (min=eps, prec=mic, a=x, b=y)
CALL EVALDIM (x, y, eps, hug)
CALL EVALDIM (x, y, max=hug, prec=mic)
```

VII.B INTERFACE DE PROCÉDURE

Une interface de procédure est constituée des informations nécessaires permettant la communication entre plusieurs unités de programmes. Ces informations sont :

- le type de la procédure : *function* ou *subroutine* ;
- les arguments de la procédure (arguments formels), avec leur type et leurs attributs,
- les propriétés du résultat dans le cas d'une fonction.

Par défaut, on est dans le cas d'une interface implicite : les arguments affectifs de l'appel de la procédure sont définis dans l'unité de programme appelant, et les arguments muets de la procédure sont définis dans la procédure.

- Dans le cas, d'une procédure interne, la compilation de l'unité appelante et de la procédure se fait ensemble : il y a donc un contrôle de cohérence. C'est donc une interface explicite, mais qui limite la visibilité de la procédure, depuis l'extérieur.
- Dans le cas de procédures externes, la compilation de l'unité appelante et de la procédure se fait séparément et les arguments sont passés par adresse. Alors, le contrôle de cohérence entre arguments effectifs et arguments muet n'existe pas à la compilation. De nombreuses erreurs, lors de l'exécution du programme, sont possibles et souvent difficiles à détectées.

C'est pourquoi, on a recours à une interface explicite, à l'aide d'un bloc *interface*. Ainsi, le compilateur connaît toutes les informations nécessaires à l'interface entre l'unité appelante et la procédure.

Syntaxe générale d'un un bloc *interface* :

```
interface
    procedure nom_procedure (list_arg)
        bloc déclaratif de list_arg
    end procedure nom_procedure
end interface
```

où le bloc déclaratif est une duplication de la partie déclarative de la procédure.

VII.B.1 Bloc interface dans l'unité appelante

On écrit directement le bloc interface au niveau du bloc déclaratif de l'unité de programme qui appelle la procédure externe concernée.

Exemple VII.5

```
PROGRAM CHOC
IMPLICIT NONE
REAL :: rr1, uu1, TT1, rr2, uu2, TT2
REAL, EXTERNAL :: FuncGas

INTERFACE
    SUBROUTINE STRUCT (r1, u1, T1, r2, u2, T2, FuncGas, TG)
        REAL, INTENT(IN) :: r1, u1, T1, r2, u2, T2
        REAL, EXTERNAL :: FuncGas
        CHARACTER(LEN=2), INTENT(IN) :: TG
    END SUBROUTINE STRUCT
END INTERFACE
 …
    CALL STRUCT (rr1, uu1, TT1, rr2, uu2, TT2, FuncGas,
'GP')
…
 END PROGRAM CHOC
```

```
SUBROUTINE STRUCT (r1, u1, T1, r2, u2, T2, FuncGas, TG)
IMPLICIT NONE

REAL, INTENT(IN) :: r1, u1, T1, r2, u2, T2
REAL, EXTERNAL :: FuncGas
CHARACTER(LEN=2), INTENT(IN) :: TG

INTEGER, PARAMETER :: nvar = 3, nstep = 1e4
REAL, DIMENSION(nvar, 2) :: f
INTEGER :: n, i, j, k, jm, km, count
REAL :: rr, uu, TT, z, z0, h
```

```
  …
  Integre_choc : DO n = 1, nstep
      rr = f(1,1) * r1 ; TT = f(3,1) * T1
      IF (rr>rho(nmax) .or. TT>TeV(mmax) ) THEN
           EXIT Integre_choc
      END IF
      CALL RUNGEKUTTA (z, h, f, 3, n, FuncGas)
  END DO Integre_choc
  …
END SUBROUTINE STRUCT
```

L'inconvénient, dans ce cas, est qu'il est nécessaire de dupliquer le bloc *interface* dans toutes les unités de programme appelant la procédure concernée. Or l'intérêt d'une procédure externe, avec bloc interface, est justement d'être utilisable, avec cohérence, par toute autre unité de programme. On privilégiera alors l'écriture du bloc *interface* au sein d'un module.

VII.B.2 Bloc interface dans un module

Pour améliorer la fiabilité générale, on préfère insérer le même bloc *interface* dans toutes les unités de programme faisant appel à la procédure en question. On utilise le module, qui via l'instruction *use* insérera le bloc *interface* dans l'unité appelante.

A ce niveau, on a le choix entre deux voies :

- écrire un module dans le seul but de contenir le bloc *interface* ;
- écrire un module qui contient la procédure externe : c'est une excellente solution !

a) Module avec bloc interface

On reprend l'exemple précédent, mais la déclaration du bloc *interface* se fait grâce à un module, nommé *interhydro*.

Exemple VII.6

```
MODULE INTERHYDRO
    INTERFACE
        SUBROUTINE STRUCT (r1, u1, T1, r2, u2, T2,&
                           FuncGas, TG)
            REAL, INTENT(IN) :: r1, u1, T1, r2, u2, T2
            REAL, EXTERNAL :: FuncGas
            CHARACTER(LEN=2), INTENT(IN) :: TG
        END SUBROUTINE STRUCT
    END INTERFACE
END MODULE INTERHYDRO
```

```
PROGRAM CHOC
USE INTERHYDRO   ! Insertion de l'interface
IMPLICIT NONE
REAL :: rr1, uu1, TT1, rr2, uu2, TT2
REAL, EXTERNAL :: FuncGas
…
    CALL STRUCT (rr1, uu1, TT1, rr2, uu2, TT2, FuncGas,
'GP')
…
END PROGRAM CHOC
```

```
SUBROUTINE STRUCT (r1, u1, T1, r2, u2, T2, FuncGas, TG)
IMPLICIT NONE

REAL, INTENT(IN) :: r1, u1, T1, r2, u2, T2
REAL, EXTERNAL :: FuncGas
CHARACTER(LEN=2), INTENT(IN) :: TG

INTEGER, PARAMETER :: nvar = 3, nstep = 1e4
REAL, DIMENSION(nvar, 2) :: f
INTEGER :: n, i, j, k, jm, km, count
REAL :: rr, uu, TT, z, z0, h
…
Integre_choc : DO n = 1, nstep
    rr = f(1,1) * r1 ; TT = f(3,1) * T1
    IF (rr>rho(nmax) .or. TT>TeV(mmax) ) THEN
         EXIT Integre_choc
    END IF
    CALL RUNGEKUTTA (z, h, f, 3, n, FuncGas)
END DO Integre_choc
…
END SUBROUTINE STRUCT
```

b) Module contenant la procédure

On reprend à nouveau l'exemple précédent, mais cette fois-ci le module sert à encapsuler la procédure, qui devient une procédure interne au module grâce au mot clé *contains*. L'interface de procédure est connue vie l'instruction *use nom_module*.

Exemple VII.7

```
MODULE INTERHYDRO
CONTAINS

    SUBROUTINE STRUCT (r1, u1, T1, r2, u2, T2, FuncGas, TG)
    IMPLICIT NONE

    REAL, INTENT(IN) :: r1, u1, T1, r2, u2, T2
    REAL, EXTERNAL :: FuncGas
    CHARACTER(LEN=2), INTENT(IN) :: TG

    INTEGER, PARAMETER :: nvar = 3, nstep = 1e4
    REAL, DIMENSION(nvar, 2) :: f
    INTEGER :: n, i, j, k, jm, km, count
    REAL :: rr, uu, TT, z, z0, h
 …
    Integre_choc : DO n = 1, nstep
        rr = f(1,1) * r1 ; TT = f(3,1) * T1
        IF (rr>rho(nmax) .or. TT>TeV(mmax) ) THEN
           EXIT Integre_choc
        END IF
        CALL RUNGEKUTTA (z, h, f, 3, n, FuncGas)
    END DO Integre_choc
    …
    END SUBROUTINE STRUCT
END MODULE INTERHYDRO
```

```
PROGRAM CHOC
USE INTERHYDRO   ! accès directe a la procédure
IMPLICIT NONE
REAL :: rr1, uu1, TT1, rr2, uu2, TT2
REAL, EXTERNAL :: FuncGas
…
    CALL STRUCT (rr1, uu1, TT1, rr2, uu2, TT2, FuncGas,
'GP')
…
END PROGRAM CHOC
```

VII.B.3 Récapitulatif sur l'interface explicite

a) L'interface explicite

Une interface de procédure est dite explicite dans les cinq cas suivants :

- appel de procédures intrinsèques : elles ont toujours des interfaces explicites :
- appel de procédure interne ;
- présence d'un bloc *interface* dans la procédure appelante,
- accès au module, via l'instruction *use*, contenant le bloc interface de la procédure externe ;
- accès au module, via l'instruction *use*, contenant la procédure externe ;

b) L'interface explicite obligatoire

Le long du cours, on a déjà mentionné des cas où l'interface doit être explicite, sinon le compilateur ne peut pas savoir si le programme est cohérent ou non. On rencontre dix situations de ce type :

- utilisation d'une fonction à valeur tableau,
- utilisation d'une fonction à valeur pointeur,
- utilisation d'une fonction à valeur chaîne de caractères dont la longueur est déterminée dynamiquement,
- utilisation d'une procédure générique,
- passage en argument d'un tableau à profil implicite,
- argument formel avec l'attribut *pointer* ou *target*,
- passage d'arguments à mots clé,
- argument optionnel,
- surcharge ou définition d'un opérateur,
- surcharge de l'opérateur d'affectation.

VII.C INTERFACE GÉNÉRIQUE

Le bloc *interface* peut servir aussi à regrouper une famille de procédure, sous un même nom. La procédure sera appelée dans une unité de programme par son nom de famille, ou nom générique. La procédure qui sera réellement utilisée lors de l'appel est choisie pas le compilateur en fonction des instructions du bloc *interface*. Le bloc *interface* doit alors posséder un nom, qui sera le nom générique des procédures concernées.

Syntaxe générale :

```
interface nom_generique
    procedure nom1 (list_arg1)
        bloc declaratif de nom1
    end procedure nom1
    procedure nom2 (list_arg2)
        bloc declaratif de nom2
    end procedure nom2
…
    procedure nomn (list_argn)
        bloc declaratif de nomn
    end procedure nomn
end interface
```

Note : en Fortran 95, on peut aussi nommer l'instruction end interface :

```
end interface nom_generique
```

Exemple VII.8

```
INTERFACE moyenne
    FUNCTION moy_int (tab_int)
        INTEGER, DIMENSION(:) :: tab_int
    END FUNCTION moy_int

    FUNCTION moy_real (tab_real)
        REAL, DIMENSION(:) :: tab_real
    END FUNCTION moy_real

    FUNCTION moy_comp (tab_comp)
    COMPLEX, DIMENSION(:) :: tab_comp
    END FUNCTION moy_comp
END INTERFACE moyenne
```

Dans cet exemple, la fonction moyenne accepte en argument un tableau de rang1 soit d'entiers, soit de réels, soit de complexes. La moyenne sera effectivement calculée en faisant appel à la fonction correspondant au type de l'argument donné.

VII.D SURCHARGE ET CRÉATION D'OPÉRATEURS

La notion de surcharge des opérateurs prédéfinis par le langage est propre aux langages orientés objets. On entend par surcharge d'un opérateur : un élargissement de son champ d'application. Il faut,

bien entendu, définir les nouvelles opérations entre les objets. La surcharge d'un opérateur impose de respecter sa nature et les règles de priorité précédemment définie dans le langage.

Habituellement, un opérateur retourne une valeur, construite en fonction des expressions données. On emploie alors des procédures de type FUNCTION pour surcharger les opérateurs. Toutefois, une exception est faite pour l'opérateur d'affectation « = » qui ne retourne aucune valeur, il faut donc utiliser une procédure de type SUBROUTINE, pour surcharger cet opérateur.

On définit la surcharge d'un opérateur grâce à un bloc INTERFACE.

Remarque : en général, la surcharge est définie dans un module.

Remarque : Certains opérateurs ont déjà fait l'objet d'une surcharge au sein du langage.

VII.D.1 Interface OPERATOR

Pour surcharger un opérateur (autre que l'opérateur d'affectation), on utilisera un bloc d'interface, en lui ajoutant l'instruction OPERATOR. Le bloc interface a la même syntaxe et la même fonction que dans le cas des interfaces explicites. C'est le compilateur qui choisit quelle procédure sera effectivement utilisée lors de tel ou tel appel.

On indique l'opérateur que l'on veut surcharger, entre parenthèses, à la suite du mot clé OPERATOR :

- si, c'est un opérateur existant, on utilise directement son signe ;
- si, on crée un nouvel opérateur, on utilise un identifiant, placé entre les caractères « . ».

Syntaxe générale :

```
interface operator (*)          (ou autre signe)
    function nom_fun (list_arg)
        bloc déclaratif avec intent (in)
    end function nom_fun
    …
end interface
```

ou :

```
interface operator (.nom_op.)
    function nom_fun (list_arg)
        bloc déclaratif avec intent (in)
    end function nom_fun
    …
end interface
```

Remarque : Les arguments de la fonction, associée à un opérateur pour sa sur-définition, doivent avoir l'attribut *intent (in)*.

Exemple VII.9

```
MODULE TYPMAT
    TYPE MAT_INT
        INTEGER :: lin, row
        INTEGER :: DIMENSION(:,:), POINTER :: ptr_mat
    EN TYPE MATRICE
END MODULE TYPMAT
```

```
INTERFACE OPERATOR (*)
    FUNCTION MULT(a, b)
        USE TYPMAT
        TYPE (mat_int), INTENT (IN) :: a, b
        TYPE (mat_int) :: mult
    END FUNCTION MULT
END INTERFACE
```

```
TYPE (mat_int) FUNCTION MULT(a, b)
    USE TYPMAT
    TYPE (mat_int) , INTENT (IN) :: a, b
    INTEGER, TARGET, PRIVATE :: i, j
    INTEGER :: ok

    i = a%lin ; j = b%row
    mult%lin = i ; mult%row = j

    ALLOCATE(mult%ptr_mat(i,j), stat=ok)
    IF (ok > 0) THEN
        PRINT*, "Erreur d'allocation de mult"
        STOP 120
    END IF

    mult%ptr_mat = MATMUL(a%ptr_mat, b%ptr_mat)
END FUNCTION MULT
```

VII.D.2 Interface ASSIGNMENT

Comme nous l'avons mentionné précédemment, l'opérateur d'affectation se comporte différemment des autres opérateurs, car il ne retourne pas une nouvelle valeur. Donc pour surcharger cet opérateur, on utilise l'instruction ASSIGNMENT (=). La syntaxe est identique au bloc INTERFACE

OPERATOR, se reporter donc à l'exemple précédent. Par contre, on utilisera une procédure de type *subroutine* pour sur définir l'affectation.

Syntaxe générale :

```
interface assignment (=)
    subroutine nom_fun (list_arg)
        bloc déclaratif avec intent(out)
                        puis intent (in)
    end subroutine nom_fun
    …
end interface
```

Remarque : Les arguments de la *subroutine*, associée à l'opérateur « = » pour sa sur-définition, doivent avoir respectivement :

- l'attribut *intent (out)* ou *intent (inout)*, pour le premier argument, qui est l'opérande de gauche ;
- l'attribut *intent (in)*, pour le second argument, qui est l'opérande de droite.

VIII LES ENTRÉES / SORTIES

Le langage Fortran dispose d'un ensemble de possibilités en matière d'entrée / sortie (E/S) particulièrement riches.

Les nouvelles caractéristiques significatives apportées par la norme Fortran 90 sont:

- les E/S sans déplacement (nom-advancing I/O),
- les listes nommées (NAMELIST),
- quelques nouveaux spécificateurs des instructions OPEN et INQUIRE.
- quelques nouveaux descripteurs d'édition et une généralisation du descripteur d'édition G,

VIII.A INSTRUCTIONS GÉNÉRALES

Les instructions générales d'entrée / sortie sont les instructions de lecture et d'écriture : READ, WRITE et PRINT. On distingue les entrées / sortie à accès directes des entrées / sortie à séquentielles.

- L'accès séquentiel permet de traiter les enregistrements dans l'ordre dans lequel ils apparaissent ; ils sont donc repérés par la position du pointeur.
- Tandis que l'accès direct gère des enregistrements de taille unique dans un ordre quelconque ; ils sont donc repérés par un numéro d'enregistrement (REC).

Syntaxe générale accès séquentiel :

```
read ([unit=]unit, [fmt=]format [, iostat] [, err]
      [, end] [, advance] [, size] [, eor]) list_var
write ([unit=] unit, [fmt=]format [, iostat] [, err]
      [, advance]) list_var
print fmt, list_var
```

Syntaxe générale accès direct:

```
read ([unit=]unit, [fmt=]format [, rec] [, iostat]
      [, err] [, end]) list_var
write ([unit=]unit, [fmt=] format [, rec] [, iostat]
      [, err]) list_var
```

Les paramètres sont les suivants :

- *unit* désigne l'unité dans laquelle on veut lire ou écrire ;
- *fmt* est l'indicateur de format ;
- *rec* désigne le numéro de l'enregistrement ;
- *iostat* indicateur d'erreurs ;
- *err* renvoie à l'étiquette d'instruction en cas d'erreurs ;
- *end* renvoie à l'étiquette d'instruction lorsque la fin de fichier est rencontrée ;

- *advance* précise si le pointeur avance ou pas à la fin de l'enregistrement ;
- *size* permet de récupérer la longueur d'enregistrement ;
- *eor* renvoie à l'étiquette d'instruction lorsque la fin d'enregistrement est rencontrée.

VIII.A.1 Description des paramètres de lecture / écriture

a) l'unité

Lorsqu'on veut accéder à un fichier, on donne au paramètre *unit* la valeur entière qui repère le fichier. Le numéro d'identification du fichier est défini à son ouverture (voir l'instruction OPEN ci-après).

Lorsqu'on veut accéder à une variable du type chaîne de caractères, on affecte au paramètre *unit* le nom de la variable.

Exemple VIII.1

```
INTEGER :: n, k
CHARACTER (LEN=10), PRIVATE :: name = 'graphX.out'
INTEGER, PRIVATE :: iname = 0

DO k = 1, n
    iname = iname + 1
    WRITE(name(6:6),'(i1)') iname
EN DO
```

b) le format

Pour écrire un format d'enregistrement, il faut connaître quelques règles : on précise le type d'enregistrement, le nombre total de caractères, le nombre de chiffres significatifs pour les nombres décimaux..., puis on peut affiner le format avec des espaces…

Les descripteurs de types :

- a : pour une chaîne de caractères ;
- l : pour une valeur logique ;
- i : pour un nombre entier ;
- f : pour un nombre décimal ;
- e : pour un réel en notation scientifique ;

Ensuite, on donne le nombre de caractères directement : a3, par exemple.

Lors de l'utilisation de formats identiques, les uns à la suite des autres, on peut multiplier les formats directement avec un nombre de fois : 4a3, par exemple, pour écrire 4 fois une chaîne de 3 caractères. Pour des format plus complexes, on se sert de parenthèses et on opère une distribution comme pour la multiplication. Par exemple : 4(a3, i2)

On peut sauter des caractères, avec l'identifiant x : 4(a3, 1x, i2). Ainsi, on saute 1 caractère entre la chaîne de 3 caractères et l'entier à 2 chiffres ; le tout est répété 4 fois. En écriture, cela revient à laisser un espace.

On peut se positionner dans la mémoire tampon, avec le descripteur t. Pour cela, on indique aussi l'emplacement sur lequel on veut pointer. Par exemple : t20, veut dire se placer sur le $20^{ième}$ caractère de la ligne.

Le caractère « / » indique d'aller à une nouvelle ligne.

On peut aussi appliquer un facteur d'échelle, sur les nombres réels, à l'aide de l'identifiant p. Par exemple, 3p devant les autres descripteurs, multiplie par 10^3 la mantisse et diminue l'exposant de 3.

Ces descripteurs de format sont les plus couramment utilisés, mais la liste des descripteurs n'est pas, ici, exhaustive, ni même leurs effets. Pour plus de détails, il faut se référer à un manuel complet du langage Fortran.

Exemple VIII.2

```
FORMAT ("val1 = ", 1p, f7.4, 1x, "val2 = ", i3)
FORMAT ("Les resultats sont : ", 3(e12.4, 2x, f8.3))
FORMAT (t12, 3(e12.4, 2x, f8.3), 3x, i20)
```

c) le numéro de l'enregistrement

Ce paramètre (*rec*) ne sert que dans les fichiers à accès direct et repère le numéro de l'enregistrement. C'est donc une valeur entière.

d) l'indicateur d'erreurs

On utilise le paramètre *iostat*, pour gérer les erreurs d'enregistrement. Ce paramètre peut prendre plusieurs valeurs, de type entier :

- il est nul, lorsqu'il n'y a aucune erreur ;
- il est négatif lorsque l'erreur est générée par un fin :
  - de fichier (*end*),
  - ou d'enregistrement (eor), dans les accès séquentiels formatés, sans avancement du pointeur (advance = 'no') ;
- il est positif dans les autres cas d'erreurs.

e) *err* renvoie à l'étiquette d'instruction en cas d'erreurs

Les trois paramètres *err*, *end* et *eor* reçoivent chacun une étiquette, qui renvoie à l'instruction correspondante, en fonction des besoins. Ainsi, le programmateur peut décider de la suite du programme, lorsque des erreurs surgissent dans la lecture ou l'écriture d'enregistrement, lorsqu'une fin de fichier, ou bien une fin d'enregistrement, sont respectivement rencontrées.

Exemple VIII.3 : étiquettes d'instruction

```
…
CHARACTER (LEN=4) :: nom
DO
    READ (15, 'i4', err=105, end=106) nom
…
END DO
…
105 PRINT*,'erreur a la lecture' ; STOP 222
106 PRINT*,'on a fini !'
```

Note : Le paramètre *iostat* permet aussi d'orienter le programme si une erreur apparaît, ou si une fin de fichier est détectée. L'intérêt est de permettre une programmation plus structurée que celle de l'exemple précédent.

Exemple VIII.4

```
…
CHARACTER (LEN=4) :: nom
INTEGER :: dem
DO
    READ (15, 'i4', iostat=dem) nom
    …
    IF (dem > 0) THEN
        PRINT*,'erreur a la lecture' ; STOP 222
    ELSE
        PRINT*,'on a fini !' ; EXIT
    END IF
END DO
```

f) avancement ou non du pointeur

Par défaut, lors de l'exécution d'une instruction d'entrée / sortie, le pointeur d'un fichier passe à l'enregistrement suivant. Toutefois, on peut préciser que le pointeur reste sur le dernier enregistrement, avec le paramètre *advance*.

- *advance = 'no'* spécifie que le pointeur reste sur place ;
- *advance = 'yes'* spécifie que le pointeur avance à l'enregistrement suivant (c'est le défaut).

Dans le cas d'enregistrement statique (*advance = 'no'*), le format doit toujours être fixé.

g) longueur d'enregistrement

Dans le cas où une fin d'enregistrement est détectée, le paramètre *size* permet de récupérer la longueur de l'enregistrement. Ce paramètre ne s'applique qu' à une instruction de lecture statique (advance = 'no'), et donc formatées.

```
READ (12, fmt=405, advance = 'no', size = l, eor=106) var
…
106 PRINT*, 'longueur ', l
```

VIII.B OUVERTURE DE FICHIERS

L'instruction OPEN permet d'ouvrir un fichier, la syntaxe générale est la suivante :

```
open (unit, file, status, action, access, iostat, err,
position, form, recl, blank, pad, delim)
```

- *unit* : numéro d'identification de l'unité logique que l'on manipule ;
- *file* : nom du fichier que l'on ouvre ;
- *status* :statut du fichier (nouveau, ancien…) ;
- *action* : précise ce qu'on veut faire de ce fichier (lecture, écriture) ;
- access : précise l'accès direct ou séquentiel à un fichier ;
- *iostat* : sert à mentionner s'il y a des erreurs à l'exécution ;
- *err* : sert à donner une instruction en cas d'erreurs ;
- *position* : indique la position du pointeur du fichier ;
- *form* : précise si un fichier est formaté ou non ;
- *recl* : précise la longueur d'un enregistrement ;
- *blank* : précise la gestion des espaces lors de la lecture ;
- *pad* : sert pour compléter ou non avec des blancs les enregistrements ;
- *delim* : sert à délimiter des chaînes de caractères.

VIII.B.1 Description des paramètres d'ouverture

a) l'unité logique

On repère un fichier par un numéro, en affectant une valeur au paramètre *unit*. Tant que le fichier reste ouvert, il est identifié par ce numéro dans les instructions d'entrée / sortie du reste programme.

Exemple VIII.5

```
OPEN (unit=20)
READ(20, fmt=12) var
```

b) nom du fichier

On affecte, au paramètre file, une chaîne de caractères qui est le nom du fichier à ouvrir.

Exemple VIII.6

```
OPEN (unit=20, file='fich_res')
```

c) status

Ce paramètre peut recevoir une de ces cinq affectations : ***new, old, unknown, replace, scratch***.

- *new* : si le fichier doit être créé.
- *old* : si le fichier doit déjà exister.
- *unknown* : statut inconnu qui dépend de l'environnement.
- *replace* : si le fichier existe déjà lors de l'ouverture, il sera détruit et dans tous les cas un nouveau fichier sera créé.
- *scratch* : indique qu'un fichier anonyme sera créé et connecté à l'unité spécifiée via le paramètre *unit*. La durée de vie de ce fichier sera soit le temps d'exécution du programme, soit le temps s'écoulant jusqu'à la fermeture de cette unité.

d) vocation du fichier

Le paramètre *action* peut recevoir les attributs ***read, write, readwrite***.

- *read* : toute tentative d'écriture est interdite.
- *write* : toute tentative de lecture est interdite.
- *readwrite* : les opérations de lecture et d'écriture sont autorisées.

e) accès direct ou séquentiel

Pour préciser le type d'accès à un fichier, on affecte la paramètre *access* des attributs ***sequentiel*** ou ***direct***. L'accès séquentiel étant l'accès par défaut.

Exemple VIII.7

```
OPEN (unit=20, file='fich_res', action='write',&
      acces='direct')
```

f) détection d'erreurs

On se sert du paramètre *iostat* pour détecter les erreurs lors de l'ouverture :

- *iostat* = 0 : il n'y a pas d'erreurs :
- *iostat* > 0 : des erreurs sont survenues.

g) instruction en cas d'erreurs

On affecte une étiquette au paramètre err. En cas d'erreur détectées, le programme exécute les instructions contenues à partir de l'étiquette. (Voir Exemple VIII.3 : étiquettes d'instruction.)

h) position du pointeur du fichier

Le paramètre *position* peut recevoir les attributs ***rewind, append, asis***.

- rewind : le pointeur du fichier sera positionné au début du fichier.

- asis : permet de conserver la position du pointeur de fichier. Il est utile lorsqu'on veut modifier certaines caractéristiques du fichier, tout en restant positionné au même endroit.
- append : le pointeur du fichier sera positionné à la fin du fichier.

i) type formaté ou non

On spécifie le type des enregistrements, contenus dans un fichier, à l'aide du paramètre *form*, qui reçoit ainsi les attributs ***formatted*** ou ***unformatted***. Par défaut, les enregistrements sont formatés.

j) longueur d'un enregistrement

La paramètre *recl* spécifie la longueur d'un enregistrement pour l'accès direct, sinon précise la longueur maximale d'un enregistrement pour l'accès séquentiel.

k) blank

Ce paramètre ne s'utilise que dans le cas d'enregistrements formatés. Il peut prendre les attributs ***null***, ou ***zero***.

- *null* : on ne tient pas compte des espaces, comme si il n'existaient pas.
- *zero* : on remplace les espaces par des 0.

l) pad

Ce paramètre reçoit les attributs ***yes***, ou ***no***.

- *yes* : un enregistrement lu avec format est complété par des blancs, si nécessaire.
- *no* : pas de complément avec des blancs.

m) délimitation des chaînes de caractères

Le paramètre *delim* peut recevoir les attributs ***apostrophe, quote, none***.

- *apostrophe* : indique que l'apostrophe sera utilisé pour délimiter les chaînes de caractères lors d'une écriture de type *namelist*.
- *quote* : indique que l'apostrophe sera utilisée pour délimiter les chaînes de caractères lors d'une écriture de type *namelist*.
- *none* : indique qu'aucun délimiteur ne sera utilisé (valeur par défaut)

VIII.C L'INSTRUCTION INQUIRE

C'est un instruction d'interrogation sur les paramètres d'un fichiers. Elle renvoie donc une valeur logique « .TRUE. » ou « .FALSE ». Elle possède des paramètres très proches de ceux de l'instruction *open* : ***position, delim, action, pad*** ont la même signification et les mêmes valeurs que pour l'instruction *open*.

Syntaxe générale :

```
inquire (unit, file, iostat, err,
opened, exist, number, named, name,
read, write, readwrite, action, access, direct,
position, form, recl, nextrec, blank, pad, delim)
```

a) read, write, readwrite

Ces paramètres reçoivent chacun les attributs ***yes, no*** ou ***unknown***. ; ils indiquent respectivement si la lecture, l'écriture et la lecture et l'écriture, dans un fichier, sont autorisées ou non, ou indéterminées.

VIII.C.2 Par liste de sortie

Une variante de l'instruction INQUIRE permet de se renseigner sur la longueur d'une liste de sortie lorsque l'écriture d'enregistrements non formatés est souhaitée. Cette variante est connue sous le nom "INQUIRE par liste de sortie", elle a la syntaxe suivante :

```
inquire (iolength = longueur) out_list
```

où longueur est une variable scalaire de type entier par défaut, utilisée pour déterminer la longueur de la liste de sortie non formatée *out_list*.

Note : cette longueur peut être utilisée pour déterminer la valeur à fournir au spécificateur *recl* lors d'une prochaine instruction *open.*

Exemple VIII.8

```
INTEGER   :: longueur
...
INQUIRE (IOLENGTH=longueur) nom, adresse, tel
...
OPEN (1, FILE='repertoire', RECL=longueur, &
        FORM='unformatted')
...
WRITE(1) nom, adresse, tel
```

VIII.D L'INSTRUCTION NAMELIST

L'instruction **namelist** permet de lier un ensemble de variables dans un seul groupe, afin de faciliter les opérations d'entrée /sortie.

VIII.D.1 Déclaration

La déclaration d'un tel groupe se fait par une instruction de spécification **namelist** avant toute instruction exécutable, dont la forme est :

```
namelist /nom_namelist/ liste_var
```

Remarques :

- Il est possible de compléter l'énumération de la liste dans la même unité de portée.
- Si une variable a l'attribut PUBLIC, aucune variable dans la liste nommée ne peut avoir l'attribut PRIVATE.
- Les champs doivent être des variables définies à la compilation, donc pas de paramètres de tableaux ou de chaînes, d'objets automatiques ou pointers…

VIII.D.2 Usage

Dans une instruction READ ou WRITE, la liste nommée sera désignée par le spécificateur *nml*.

Exemple VIII.9

```
INTEGER, DIMENSION(5) :: age = (/6, 7, 8, 9, 10/)
CHARACTER (LEN=3) :: classe(5)=(/'cp','ce1','ce2',
                                 'cm1','cm2'/)
CHARACTER (LEN=15) :: maitre(5) = ''
NAMELIST /ecole/ age, classe, maitre
...
READ(*, nml=ecole)
! lit des données sous la forme suivante:
&ECOLE maitre(4) = 'Mme Grosso' /
&ECOLE maitre(2) = 'M. Chaduiron' /
...
WRITE(*, nml= ecole)
! génère les lignes suivantes :
&ECOLE AGE = 6, 7, 8, 9, 10, CLASSE =  cpce1ce2cm1cm2
MAITRE =              M. Chaduiron              Mme Grosso
             /
```

IX ANNEXE

IX.A QUELQUES FONCTIONS INTRINSÈQUES

IX.A.1 Manipulation de bits

Les variables i, j, n, m, utilisées sont toutes des valeurs entières. On rappelle que les bits d'un nombre sont numérotés de gauche à droite et comment avec le bit 0.

iand (i, j)
ieor (i, j)
ior (i, j)

Ces trois fonctions renvoient respectivement un entier du même nombre de bits que i, formé par le résultat des trois opérations logiques : ET, OU exclusif, OU inclusif, effectuées bit à bit entre les entiers i et j.

ishft (i, shift [, size])
ishftc (i, shift [, size])

Ces fonctions procèdent à un décalage vers la gauche, ou vers la droite, du nombre de bits indiqué par *shift*. Lorsque *shift* est positif, on décale vers la gauche, lorsque *shift* est négatif, on décale vers la droite. La fonction *ishft* remplie les bits manquants par des 0, tandis que la fonction *ishftc* opère une permutation circulaire. Elles renvoient toutes les deux un entier du même nombre de bits que l'entier i.

Exemple IX.1

```
ishft(120,2)          donne 480
ishftc(134678, 8)     donne 34477568
```

not (i)

Cette fonction fait le complément à 1 (ou complément logique) des bits de l'entier i ; elle retourne un entier du même nombre de bits que i.

btest (i, n)

Cette fonction renvoie « .TRUE. » si le nième bit du nombre entier i est à 1, sinon « .FALSE. » si il est à 0.

```
ibset (i, n)
ibclr (i, n)
```

Ces deux fonctions retournent un entier identique à i, avec respectivement le nième bit à 1 et le nième bit à 0.

```
ibits (i, n, m)
```

Cette fonction extrait m bits de l'entier i, à partir du nième bit ; puis les décale vers la droite pour former un entier du même type que i (les autres bits de gauche sont mis à 0).

```
mvbits (i, n, m, j, nj)
```

mvbits est un sous-programme, dons on l'appelle avec un *call*), il est construit sur le même principe que la fonction *ibits*. Il extrait m bits de l'entier i, à partir du nième bit ; puis les place à la « njième » position de l'entier j.

IX.A.2 Précision et codage numérique

```
digit(x)
```

Cette fonction admet un argument x entier ou réel et renvoie un entier égal au nombre de chiffres significatifs de x:

- le nombre de chiffres binaires de la mantisse si x est réel
- le nombre de chiffres binaires de stockage (hors bit de signe) si x est un entier.

```
epsilon (x)
```

Cette fonction renvoie un réel avec même paramètre de sous-type que x qui est presque négligeable comparé à 1.

```
huge (x)
```

Cette fonction admet un paramètre x entier ou réel et renvoie la plus grande valeur représentable dans le sous-type de x (limite d'overflow).

```
tiny(x)
```

Cette fonction admet un paramètre x réel et renvoie la plus petite valeur positive non nulle représentable dans le sous-type de x (limite d'underflow).

```
maxexponent (x)
```

Cette fonction admet un paramètre x réel et renvoie l'entier représentant le plus grand exposant possible dans le sous-type de x.

`minexponent (x)`

Cette fonction admet un paramètre x réel et renvoie l'entier représentant le plus petit exposant possible dans le sous-type de x.

`exponent (x)`

Cette fonction renvoie un entier égal à l'exposant de la représentation numérique de x (0 si x est égal à 0)

`fraction (x)`

Cette fonction renvoie un réel qui est égal à la mantisse de la représentation numérique de x.

`nearest(x, s)`

Cette fonction renvoie un réel ayant le même paramètre de sous-type que x, égal au nombre exactement représentable en machine le plus proche de x dans la direction indiquée par l'argument s.

`spacing (x)`

Cette fonction renvoie un réel ayant le même paramètre de sous-type que x, égal à l'espacement absolu des nombres de même paramètre de sous-type que x dans le voisinage de x.

`range (x) et precision (x)`

Ces fonctions sont décrites au chapitre II.C.4c) range et precision.

IX.A.3 Fonctions numériques élémentaires

a) Avec conversion

`ceiling (a)`

Cette fonction renvoie le plus petit entier du type par défaut qui est supérieur ou égal à a.

`floor (a)`

Cette fonction renvoie le plus grand entier du type par défaut qui est inférieur ou égal à a.

b) Sans conversion

`modulo (a,p)`

Cette fonction renvoie a modulo p pour a et p soit tous les deux réels soit tous les deux entiers.

IX.A.4 Fonctions élémentaires de type caractère

a) Conversion entier-caractère

`achar (i)`

Cette fonction renvoie le caractère dont le code ASCII est spécifié par l'argument i.

`iachar (c)`

Cette fonction renvoie le code ASCII correspondant au caractère c.

`ichar (c)`

Cette fonction renvoie un entier qui représente le numéro du caractère c dans la table à laquelle appartient c (ASCII/EBCDIC)

b) Manipulation de chaîne de caractères

`adjustl (string)`

Cette fonction cale à gauche la chaîne de caractère STRING.

`adjustr (string)`

Cette fonction cale à droite la chaîne de caractère STRING.

`len_trim`

Cette fonction renvoie un entier qui représente la longueur de chaîne de caractère STRING (sans compter les caractères blancs à la fin).

`scan (string, set [,back])`

Cette fonction renvoie un entier qui représente le numéro du premier caractère de *string* où apparaît l'un des caractères de *set* ou 0 sinon. *back* est optionnel et permet de trouver la dernière occurrence.

`verify (string, set [,back]):`

Cette fonction renvoie un entier qui représente le numéro du premier caractère de *string* ne figurant pas dans set ou 0 sinon. *back* est optionnel, s'il est égal à « .TRUE. », la recherche se fait depuis la fin de *string*.

c) Fonctions de comparaison lexicale

```
lge (stringa, stringb)
lle (stringa, stringb)
```

Ces fonctions renvoient la valeur « .TRUE. », respectivement si *stringa* est après ou avant *stringb* (ou au même niveau) dans la table ASCII, sinon « .FALSE. ».

```
lgt (stringa, stringb)
llt (stringa, stringb)
```

Ces fonctions renvoient la valeur « .TRUE. », respectivement si *stringa* est strictement après ou avant *stringb* dans la table ASCII, sinon « .FALSE. ».

Remarque : en cas d'inégalité de longueur, la chaîne la plus courte sera complétée à blanc sur sa droite.

```
repeat (string, n)
```

Cette fonction permet la formation d'une chaîne de caractères par concaténation de n fois la chaîne *string*.

```
trim(string)
```

Cette fonction renvoie la chaîne de caractères *string* sans les blancs éventuels situés à sa fin.

IX.B FONCTIONS DE GÉNÉRATION DE NOMBRES ALÉATOIRES

```
random_number (harvest)
```

Ce sous-programme génère dans *harvest* un nombre (ou des nombres) pseudo aléatoire, dans l'intervalle [0,1[; *harvest* doit être de type réel et peut être un tableau.

```
random_seed ([size] [, put] [, get])
```

Ce sous-programme interroge ou modifie le générateur de nombres aléatoires. Il n'admet qu'un seul des trois arguments à la fois.

- *size* : type entier d'attribut OUT ; le sous-programme retourne alors la taille du tableau d'entiers utilisés comme germe pour la génération des nombres aléatoires.
- *put* : type entier d'attribut IN ; le sous-programme utilise alors ce tableau d'entiers et de rang 1 comme germe pour la génération des nombres aléatoires.
- *get* : type entier d'attribut OUT ; le sous-programme retourne alors le tableau d'entiers et de rang 1 utilisé comme germe pour la génération des nombres aléatoires.

IX.C HORLOGE EN TEMPS RÉEL

`date_and_time (date, time, zone, values)`

Ce sous-programme retourne dans les variables caractères date et time la date et l'heure en temps d'horloge murale. L'écart par rapport au temps universel est fourni optionnellement par zone. *values* est un vecteur d'entiers récupérant les informations précédentes sous forme d'entiers.

`system_clock (count, count_rate, count_max)`

Ce sous-programme retourne dans des variables entières la valeur du compteur de période d'horloge : *count*, le nombre de périodes par seconde : *count_rate* et la valeur maximale du compteur de période : *count_max*. On peut, par exemple, évaluer le temps CPU consommé par une portion de programme.

IX.D SITES INTÉRESSANTS

FORTRAN 90 - procédures prédéfinies : http://www.univ-lille1.fr/~eudil/pyrz/home.htm

Le Programmeur : Fortran 90 : http://www.stcommunications.ca/~mboisso/Fortran_90

Bibliothèque Fortran-90 orientée Objet : http://www.math.u-psud.fr/~laminie/F90_lib

Cours IDRIS Fortran90 / 95 : http://www.idris.fr/data/cours/lang/f90

http://consult.cern.ch/cnl/215/node28.html

Fortran 90/ 95 - Le site Francophone : http://www2.cnam.fr/~lignelet/ftn.htmlx

http://cch.loria.fr/documentation/documents/F95

IX.E BIBLIOGRAPHIE

Manuel Complet du Langage FORTRAN 90 et Langage FORTRAN 95

Calcul intensif et Génie Logiciel - Auteur : Patrice Lignelet

MASSON, 1996 , 314 pages (ISBN 2-225-85229-4)

FORTRAN 90 : approche par la pratique

Auteur : Patrice Lignelet

Série Informatique, édition Menton, 1993 , 240 pages (ISBN 2-909615-01-4)

Traitement De Données Numériques Avec Fortran 90

Traitement de données numériques avec Fortran 90.

1996, 264 pages, Masson, Auteur M. Olagnon

Programmer en Fortran 90 - Guide Complet

Auteur : Claude Delannoy

2000, 413 pages, Édition Eyrolles (ISBN 2-212-08982-1)