Ecole Normale Supérieure
Cours d'introduction aux éléments finis
Mars 2009
Tahar Zamène BOULMEZAOUD

FORTRAN 90 EN BREF

Sommaire.

Les points essentiellement non traités dans ce résumé sont:

- Les pointeurs.
- La surdéfinition des opérateurs (vers une programmation orientée-objet en fortran 90).

On renvoie pour toutes ces questions à:

Claude DELLANOY, *Programmer en Fortran 90, Guide complet*, Editions Eyrolles, 1993.

1 Généralités.

- Dans tout ce document, une expression est écrite en **gras** quand elle est réservée. Quand une expression est placée entre deux crochets **[ ]**, cela signifie qu'elle est optionnelle (ou facultative).
- Les commentaires en Fortran 90 s'écrivent sous la forme:
  ! *commentaire*
- Le Fortran 90 ne fait pas de différence entre les majuscules et les minuscules.
- En format libre de Fortran 90, et contrairement au Fortran 77 (dit format fixe), une instruction peut commencer à n'importe quel emplacement de la ligne. En outre, une ligne peut avoir une longueur quelconque à concurrence de 132 caractères (il n'y a donc pas de découpage de la ligne en trois zones comme en Fortran 77).
- Plusieurs instructions peuvent être écrites sur une même ligne, à condition d'être séparées par un point virgule **;**.
- Une instruction peut être prolongée sur plusieurs lignes, à condition de mettre le caractère **&** en fin de chacune des lignes incomplètes.

1. LES TYPES ET LES OPÉRATEURS.

2 Les types simples.

LE TYPE ENTIER: **integer**

LE TYPE BOOLÉEN: **logical** (**.true.** or **.false.**).

LES TYPES RÉELS: **real** ou **double precision**.

Une constante réelle doit être écrite de préférence avec un point virgule afin d'éviter qu'elle ne soit considérée comme une variable entière lors des opérations (ex.: 3.20E+5, 1., 3.0, 3.21D12).

La déclaration d'une constante ou d'une variable d'un type simple se fait de la manière suivante:

```
type, parameter ::   nom_constante = valeur  ! decl. d'une constante
type            ::   nom_variable            ! decl. d'une variable
```

LE TYPE COMPLEXE : **complex**.

Il s'agit plutot d'un type structuré. Les deux fonctions **real()** et **aimag()** fournissent respectivement la partie réelle et la partie imaginaire d'un nombre complexe. Les constantes complexes s'écrivent sous la forme (1.0, 2.0) par exemple. La construction d'un nombre complexe à partir de deux variables réelles se fait en utilisant la fonction **cmplx()**; par exemple **cmplx(x, y)**.

3 LES OPÉRATEURS AGISSANT SUR LES TYPES SIMPLES.

liste par ordre de priorité:

****** (puissance), *** /** , **+ -** (unaires), **+ -** (binaires), $==$ **/=** $<$ $>$ $<=$ $>=$ (les quatres derniers opérateurs de comparaîson ne sont pas utilisables avec les complexes), **.not.** , **.and.** , **.or.** , **.eqv. .neqv.** (opérateurs logiques).

Dans les expressions, la conversion forcée (implicite) se fait dans le sens

integer $\rightarrow$ **real** $\rightarrow$ **double precision**

Dans les affectations, le terme à droite est converti au type du terme de gauche. Ainsi, p. ex., lors d'une affectation du genre $x = z$, où z est un complexe, x prendra la partie réelle de z si x est de type réel ou la partie entière de la partie réelle de z s'il est de type entier.

4 Les types structurés.

4.1 Le type tableau.

4.1.1 Déclaration.

On déclare un tableau de la manière suivante:

```
type, dimension([imin_1:] imax_1, [imin_2:] imax_2,...) :: nom_tableau
```

On appelle profil d'un tableau la liste de ses étendues dans chaque direction. Ainsi, dans la déclaration:

```
real, dimension(4)       :: a
real, dimension(8, -2:9) :: t
```

le tableau **a** a comme profil (4) tandis que le tableau **t** a comme profil (8, 12).

4.1.2 Construction d'un tableau.

Syntaxe:

```
nom_tableau = (/ liste de valeurs /)
```

Exemple:

```
a = (/ 2, -1, 11, 3 /)
a = 2*(/ 3, -4, p, n + m/) + 3          ! n et m sont deux entiers
a = (/ (3*i +1, i = 1, 4, 2), 20, 8 /)  ! En boucle (le 2 etant le pas)
a = (/ ((2*i-j, i = 1, 2), j = 1, 2) /)
```

La référence à un élément d'un tableau se fait en mettant l'indice (ou les indices entre parenthèses):

```
a(1) = 2.0 ; t(7, -1) = 2*p + 3.0; t(2*i, 2*j+1) = i - j
```

4.1.3 Opérations sur les tableaux.

Les opérateurs (unaires ou binaires) qui sont applicables à des variables d'un type donné sont

- applicables aussi aux tableaux (ou sous-tableaux) de *profils identiques* et dont les éléments sont de ce type. Ainsi on peut écrire -a, a + b, a*b, où a et b sont des tableaux de type scalaires et ayant le *même profil*,
- applicables aussi entre une variable et un tableau du même type. Ainsi, on peut écrire $2.0 * a + x$, $a/3$, **sqrt(a)**, où **a** est un tableau de réels par exemple et x une variable de type réel.
  Les règles de priorité et de la conversion implicite sont les mêmes que ceux des types associés.

4.1.4 Les sous-tableaux.

Un sous-tableau est un tableau construit par extraction d'une partie d'élements d'un tableau. On le note en général de la manière suivante:

```
nom_tableau([debut]: [fin] [: pas])
```

Exemple:

`a(2:4), a(1:4:2) ou  a(::), a(:) ou a , t(:,0:8), t(:,:) (=t)`

On peut aussi utiliser un *vecteur d'indices* (tableau d'entiers). La syntaxe est la suivante:

```
nom_tableau(liste_des_indices)
```

Exemple:

```
b = a ((/ 1, 3/)), b = a((/ i/2, i = 2, 8 /))
```

4.1.5 Tableaux dynamiques.

Déclaration :

```
type, dimension(:,:,...), allocatable :: nom_tableau
```

On utilise ensuite l'instruction **allocate()** pour allouer un emplacement mémoire:

```
allocate(nom_tableau(imin:imax,...) [, stat = var_ent])
```

L'attribut **stat** renvoie la valeur entière 0 si l'espace a été effectivement alloué.
Pour libérer l'emplacement occupé par un tableau dynamique, on utilise l'instruction **deallocate()**.
Exemple:

```
integer         :: n, p, reserv
dimension, dimension(:,:), allocatable ::  mat

print *, 'Entrer les tailles de la matrice:'
read *, n, p
allocate(mat(n,p), stat = reserv)  ! reserve un emplacement
if (reserv > 0) deallocate(mat)    ! libere l'emplacement
allocate(mat(n,p))                 ! reserve encore cet emplacement
```

4.1.6 L'instruction where.

Permet d'effectuer un test global sur les tableaux.
Syntaxe:

```
  where (expression_logique_tableau)
     instructions
[ elsewhere
     instructions
]
  end where
```

Exemple:

```
where (a > = 0)
     b = sqrt(a)
elsewhere
     b = sqrt(-a)
end where
```

4.2 Les chaînes de caractères.

4.2.1 Déclaration.

```
character ([len =] nbr_caract) :: nom_chaine [= `` ch_cste ``]
```

Les chaînes peuvent être écrites entre deux apostrophes (' ') ou entre guillemets (" ").

4.2.2 Les sous-chaînes de caractères.

Syntaxe :

```
nom_chaine([debut] : [fin ])
```

Exemple:

```
character (20)        ::  prenom
character (len = 5)   ::  abrege
prenom = ``Stephane''
abrege  = premier(1:5)
```

4.2.3 L'opérateur de concaténation.

Il permet de concaténer deux chaînes (sans enlever le caractère vide).
Syntaxe :

```
chaine_1 // chaine_2
```

On peut enlever le caractère vide d'une chaîne en utilisant la fonction **trim**().
La référence à l'élément d'une chaîne se fait de la même manière que pour un tableau.

4.3 Type structure.

Il permet de regrouper un ensemble d'éléments de types différents dans un seul objet. Un élément de la structure est appellé "champ".
Déclaration du type de la structure:

```
type nom_struct
     type_1 :: nom_champ1
     type_2 :: nom_champ2
       :
       :
 end type nom
```

Déclaration d'une variable de type structure:

```
type (nom_struct) :: nom_variable [ = structure_constante ]
```

Initialisation d'une variable de type structure:

```
nom_variable = nom_struct(liste_des_valeurs)
```

Exemple:

```
type produit
    integer :: numero
    integer :: quantite
    real    :: prix
end type produit

type(produit)  :: tomates, lait, cafe, the
 lait = produit(214, 45, 4.25)
 tomate%prix     = 8.40
 tomate%numero   = 30
 tomate%quantite = 70
 print *, 'Entrer le numero, la quantite et le prix du the :'
 read *, the
 cafe = the
```

2. LES INSTRUCTIONS DE CONTRÔLE.

5 L'instruction If.

Syntaxe:

```
[nom:]  if (expression_booleenne_1) then
                  :
            instructions
                  :
        [ else if  (expression_booleenne_2) then
                  :
            instructions
                  :                                ]
        [ else
                  :
            instructions
                  :                                ]
          endif [nom]
```

Dans le cas d'une seule instruction (sans else), on peut écrire

```
          if (expression_booleenne) instruction
```

6 L'instruction Select case.

Syntaxe:

```
[nom:]  select case (var_scalaire)
         [ case(valeurs_1)
                     :
                instructions
                     :
           case(valeurs_2)
                     :
                instructions
                     :
            case default
                     :
                instrcutions  ]
            end select [nom]
```

7 L'instruction de boucle avec compteur.

Syntaxe:

```
[nom:]  do var = debut, fin [, pas]
                    :
                instructions
                    :
        end do [nom]
```

8 L'instruction de boucle sans compteur.

Syntaxe:

```
[nom:]  do
                    :
                instructions
                    :
        end do [nom]
```

9 Les instructions Exit et Cycle.

L'instruction **exit**, placée dans une boucle, permet de sortir de la boucle (et donc d'interrompre son déroulement).
L'instruction **cycle** permet quant à elle de passer prématurément au tour suivant de la boucle.
Syntaxe:

```
exit [nom_de_la_boucle]
cycle  [nom_de_la_boucle]
```

10 L'instruction Do while (boucle conditionnelle).

Syntaxe:

```
[nom:]  do while (expression_booleenne)
                    :
                instructions
                    :
        end do [nom]
```

11 Les instructions Goto et stop.

L'instruction go to permet de sauter directement vers une instruction exécutable repérée par une *étiquette* (nombre entier non nul comportant au maximum 5 chiffres et placé devant l'instruction sur laquelle on souhaite se brancher).
Quant à l'instruction **stop**, elle permet d'arréter un programme.
Syntaxe:

```
go to etiquette
stop
```

3. LES SUBROUTINES.

On distingue essentiellement deux types de subroutines: internes et externes.

12 Subroutines externes.

Une subroutine externe est une “unité de compilation séparée”. Elle est donc indépendante à priori des autres sous-programmes. Ses variables locales ne sont donc reconnues nulle part en dehors d’elle-même. Les variables locales du programme appellant ne sont pas reconnues en conséquence à l’intérieur de la subroutine (à moins qu’elles ne soient partagèes à travers un module).
Syntaxe:

```
subroutine nom_subroutine(liste_es_parametres formels)

  Declarations des parametres formels

  Declarations des variables locales
      :
  Instructions
      :
end  subroutine nom_subroutine
```

L’appel d’une subroutine externe par un autre sous programme se fait en utilisant l’instruction **call** et cela de la manière suivante:

```
call nom_subroutine(liste_des_parametres_effectifs)
```

On peut aussi effectuer un appel avec des arguments à mots clés, c’est-à-dire qu’on précise le nom de chaque paramètre formel devant sa valeur:

```
call nom_subroutine(nom_par_1 = valeur_1, nom_par_2 = valeur_2,...)
```

13 Les subroutines internes.

Une subroutine interne est une subroutine qui fait partie d’un sous programme (elle est donc compilée avec ce sous-programme). Elle n’est pas utilisable en dehors de ce sous-programme (même en faisant une interface). Les déclaration et les corps des subroutines et fonctions internes se font à la fin du sous-programme qui les accueille après l’instruction **contains**.
Syntaxe:

```
debut_sous_programme_accueillant
      :
      :
      :
      :
 contains
  corps des fonctions et subroutines internes
 end contains
end sous_programme_accueillant
```

Les variables locales du sous-programme père sont alors des variables *globales* par rapport aux subroutines et fonctions internes et y sont donc reconnues. À l’intérieur d’une subroutine interne, une variable globale (déclarée dans le programme père) deviendrait masquée (et donc n’existe plus) si un paramètre formel ou une variable locale à la subroutine interne porte le même nom que cette variable globale.

14 Déclaration d’une subroutrine externe en interface.

Elle sert à prédéclarer une subroutine ou une fonction externe dans le sous-programme appellant (programme principale, subroutine ou fonction externes: unité de compilation séparée de celle de la subroutine).
Syntaxe:

```
program nom
 implicite none
  INTERFACE
     subroutine nom(parametres_formels)
       declaration des parametres_formels
     end  subroutine nom
  END INTERFACE
  :
  :
end
```

4. LES FONCTIONS.

On distingue aussi les externes et les internes (de manière identiques aux subroutines). Notons que tout ce qu'on a dit concernant les interfaces d'une subroutine reste valable pour une fonction.

15 Déclaration d'une fonction.

```
function nom_funct(parametres_formels) [result (nom_var)]
   declaration des parametres formels
   type_du_resltat      :: nom_funct [ nom_var]
    :
    :
end function nom_funct
```

La variable nom_var est destinée à contenir le résultat. En cas où **result** n'est pas utilisé, c'est le nom de la fonction **nom_fonct** qui sert à stocker le résultat.
Quand une fonction *externe* n'est pas déclarée en interface, sa déclaration dans le programme appellant est nécessaire et cela se fait en utilisant la commande **external** de la manière suivante

```
type_du_resultat  ::  nom_funct
external  nom_funct
```

16 Fonctions récursives.

Une fonction est dite récursive si elle appelle elle même (directement ou de manière croisée). *Il est nécessaire qu'elle soit prédéclarée en interface du programme appellant.* L'attribut **result** est par ailleurs nécessaire dans ce cas.
Syntaxe:

```
recursive function nom_funct(parametres_formels) result (nom_var)
    declaration des parametres formels
    type_du_resltat      :: nom_var
          :
          :
 end function nom_var
```

17 Fonctions renvoyant un tableau.

Il est possible en Fortran 90 qu'une fonction renvoie un tableau. Lors de la déclaration de cette fonction, le tableau peut-être déclaré de manière rigide (profil connu) ou alors de manière adjustable en précisant le profil à partir des paramètres formels de la fonction elle-même.

Exemple:

```
function moyenne(n, note)
   integer                     :: n
   integer, dimension(3, 120) :: note
   integer, dimension(120)    :: moyenne ! profil rigide
      :
      :
end function moyenne

function moyenne2(n, note)
   integer                   :: n
   integer, dimension(3, n) :: note
   integer, dimension(n)    :: moyenne2  ! profil adjustable
      :
      :
end function moyenne2
```

18 Les fonctions et les subroutines génériques.

Une subroutine (ou une fonction) générique est une subroutine qui a un seul nom mais qui correspond en fait à plusieurs subroutines (ou fonctions). Il s'agit donc de regrouper sous un seul nom toute une famille de subroutines. Pour définir une fonction ou une subroutine générique, il suffit de:

- définir "classiquement" les différentes subroutines de la famille,
- écrire ensuite une interface qui spécifie le nom générique et les interfaces des différentes subroutines de la famille.

Illutration par un exemple:

Imaginons qu'on a déjà écrit trois subroutines **aff_ent()**, **aff_reel()** et **aff_tab()**, qui affichent respectivement un entier donné en paramètre, un réel ou un tableau de réels. Pour en faire une subroutine générique qui s'appelle **affiche()**, il suffit de créer la bloc d'interface approprié:

```
 interface affiche       ! le nom generique est ``affiche''
   subroutine aff_ent(n)
         integer :: n
   end subroutine aff_ent

   subroutine aff_real(x)
         integer :: x
   end subroutine aff_real

   subroutine aff_tab(t)
         real, dimension(:) :: t
   end subroutine aff_ent
 end interface affiche
```

À la rencontre d'un appel **call affiche()**, le compilateur se sert du bloc d'interface correspondant pour déterminer automatiquement quel est le sous programme (aff_ent(), aff_real() ou aff_tab()) à appeller effectivement.

5. LES PARAMÈTRES DES SUBROUTINES ET FONCTIONS.

19 Les trois sortes de paramètres formels.

On distingue essentiellement trois types:

- Argument d'entrée: cela signifie que la valeur du paramètre ne peut-être "changée" à l'intérieur de la subroutine. Cela se fait en rajoutant l'attribut **intent(in)** lors de la déclaration du paramètre.

  Exemple:

```
integer, intent(in) :: p
```

- Argument de sortie: cela signifie que la valeur du paramètre ne peut-être "utilisée" à l'intérieur de la subroutine. Cela se fait en rajoutant l'attribut **intent(out)** lors de la déclaration du paramètre.
- Argument d'entrée-sortie: cela signifie que la valeur du paramètre peut-être "utilisée" et "changée" à l'intérieur de la subroutine. Cela se fait en rajoutant l'attribut **intent(inout)** lors de la déclaration du paramètre.

20 Paramètres optionnels.

Un argument est dit "optionnel" s'il est autorisé à être abscent lors de l'appel effectif de la subroutine. Cela signifie qu'il est prévu à l'intérieur de la subroutine un traitement de ce cas. Lors de la déclaration du paramètre formel en question, on rajoute le qualificatif **optional**.

Exemple:

```
integer, optional :: p
```

Le test de présence ou de non présence d'un paramètre optionnel se fait à l'intérieur de la subroutine en utilisant la fonction booléenne **present()** (exemple: present(p)) qui renvoie la valeur vraie si le paramètre est présent lors de l'appel de la subroutine.

21 Paramètres statiques.

À priori, entre deux appels succéssifs d'une même subroutine, les valeurs des variables locales ne sont pas conservées. Ce sont des variables automatiques. Pour rendre une variable locale statique (ce qui lui permet en particulier de conserver sa valeur entre deux appels) on utilise le qualificatif **save**.

```
integer, save :: x
```

22 Les tableaux en arguments.

Quand l'un ou plusieurs des paramètres formels d'une subroutine (ou une fonction) est un tableau, on est confronté au problème de déclaration du profil du tableau. Il y'a trois manières de le faire:

- Déclaration rigide (dans ce cas la taille est connue lors de l'écriture de la subroutine).

  Exemple:

```
subroutine moyenne(note)
  integer, dimension(1:120) :: note
      :
      :
```

- Déclaration ajustable avec le profil en paramètre aussi. Dans ce cas, la taille du tableau n'est pas fixe, mais dépend d'un autre (ou de plusieurs) paramètre formel.

  Exemple:

```
subroutine moyenne(note, p, n)
  integer    :: n, p
  integer, dimension(p:n) :: note
      :
      :
```

- Déclaration ajustable sans profil. La subroutine ne dispose d'aucune information sur le profil du tableau (mais le rang doit être connu) et suppose donc que lors de l'appel de cette subroutine, le tableau existe déjà et est alloué. Exemple:

```
subroutine moyenne(ecrit, oral )
  integer, dimension(:,:) :: ecrit
  integer, dimension(:)   :: oral
      :
      :
```

23 Transmission d'une subroutine en paramètre.

Il est possible d'avoir comme paramètre formel une subroutine. Il suffit pour ce faire de de déclarer une interface correpondant à cette subroutine au début de la subroutine dont elle est paramètre.
Exemple:

```
subroutine integrale(a, b, fct, resultat)
 implicit none
 real   :: a, b, resultat
 interface
   function fct(x)
      real, intent(in) :: x
      real             :: fct
   end  function fct
 end interface

      :
      :
end subroutine integrale
```

24 Le format libre

Il s'agit du format *.

Le format libre en lecture: On dipose d'une grande liberté pour entrer les données en format libre en lecture (avec `read *`). Ainsi par exemple on peut taper **389.45, 3.8945e2** ou **0.38945e3** pour une variable réelle qui vaut 389.45 et **T, t, TRUE, true, .true., .T.** ou **.t.** pour variable de type logical qui vaut "vrai".
La séparation de données entrées se fait avec **des séparateurs** qui sont *la virgule* (**,**), *l'espace vide* et *le retour à la ligne*. Quant on doit entrer la même donnée plusieurs fois, on peut la mettre en facteur sous la forme

```
n*valeur
```

ce qui d'évite d'entrer *valeur* au clavier `n` fois (voir exemple 1 tableau I ci-dessous).
Si on veut éviter d'entrer une valeur (qui va donc rester inchangée), on le fait en ne mettant rien entre *deux virgules* (voir exemple 2, tableau I). De même, le caractère **/** en fin de ligne signifie que tous les données suivantes et non entrées encore restent inchangées dans le programme (voir exemple 3, Tableau I).

Le format libre en écriture: L'écriture d'une variable en format libre est laissé à la liberté de la machine qui le fait en fonction du type de la variable ou de l'expression affichés. Dans ce cas, la présentation du résultat affiché n'est pas maîtrisée, ce qui n'est pas très commode lorsque on affiche des tableaux ou beaucoup de variables. La solution pour une bonne présentation est d'utiliser un format avec descripteurs (voir ci-dessous).

25 Les formats avec descripteurs

Un format en général est une liste de descripteurs mis sous la forme

```
'(descpt_1, descpt_2,...,descpt_n)'
```

et placé derrière `print` ou `read` comme suit

```
print '(descpt_1, descpt_2,...,descpt_n)', exp_1,...,exp_2
```

où `descpt_1, descpt_2,...,descpt_n` sont les descripteurs et `exp_1,...,exp_2` sont les expressions (ou les variables) à afficher selon la présentation indiquée par ces descripteurs.

Les principaux descripteurs: Ici, n et m désignent deux constantes entières nont signées. On a les descripteurs suivants pour afficher les types simples et dit descripteurs *actif* (car ils aggissent sur l'affichage ou la lecture des expressions ou variables associés):

i*n* : affiche (ou lit) la variable *entière* correspondante sur *n* caractères.

E*n.m* : affiche (ou lit) la variable *entière* correspondante sur *n* caractères.

6. LES MODULES

Un module est une unité de compilation séparée comportant un ensemble de déclarations susceptibles d'être partagées par plusieurs sous-programmes. Ces déclarations sont en général des déclarations de variables, de types, de subroutines et fonctions et d'interfaces. L'intérêt d'un module réside donc dans la possibilité de remplacer ces déclarations dans le début des sous programmes qui les utilisent par un simple appel au module (ce qui revient donc à expliciter le contenu du module). Il est préférable de mettre un module dans un fichier séparé.

26 Structure d'un module.

```
module nom_module
        :
  declarations
        :
end module nom_module
```

SYNTAXE D'UN APPEL: on place l'expression

```
use nom_module
```

au début du sous-programme qui s'en sert.

7.LES ENTRÉES-SORTIES STANDARDS

Le but de ce paragraphe est de préciser comment maîtriser la lecture et l'écriture standards (c'est à dire lecture à partir du clavier et affichae au moniteur).

La syntaxe général d'une instruction d'entree ou de sortie standard est la suivante:

```
READ fmt, liste_d_elements          PRINT fmt, liste_d_elements
```

où `fmt` est le format qui figure sous l'une des quatres formes suivantes:

***** : c'est le format *libre*,
une chaîne de cacartères : elle comporte une liste de *descripteurs*
nom d'une variable de type chaîne de cacartères
une etiquette: qui est nombre entier non nul comportant au maximum cinq chiffres,

Le paramètre `liste_d_elements` contient quant à lui la liste des élements à lire ou á écrire. Par le mot *élément* on entend soit une expression, soit une variable soit une liste avec boucle implicite (de la forme `(liste_elements, i = debut, fin [, pas])`.

8. LES FICHIERS

9. QUELQUES FONCTIONS INTERINSÈQUES À FORTRAN 90.

27 Fonctions arithmétiques:

Abs(x) (valeur absolue), **Acos**(x), **Asin**(x), **Atan**(x), **Atan2**(x, y) (argument dans $]-\pi, \pi]$ du nombre complexe $z = x + iy$), **sin**(x), **cos**(x), **exp**(x), **sqr**(x)(carré de x), **sqrt**(x) (racine de x), **log**(x), **log10**(x), **sinh**(x), **cosh**(x), **tan**(x), **tanh**(x), **sqrt**(x), **conjg**(z), **dim**(x, y) (fournit $max(x-y, 0)$), **max**(*x1, x2,...*), **min**(*x1, x2,...*), **mod**(a, p) (reste de la division de a par p), **floor**(x) (partie entière de x), **aint**(x) (partie fractionnaire de x), **nint**(x) (l'entier le plus proche de x), **ceiling**(x) (fournit l'entier immédiatement superieur à x).

28 Fonctions relatives aux chaînes de caractères:

achar(i): fournit une chaîne de longueur 1 correpondant au i-ème caractère de la table ASCII.
iachar(c): fournit le numéro du caractère c dans la table ASCII.
len(*chaine* **)**: longueur d'une chaîne,
trim(*chaine* **)**: founit une chaîne en enlevant les blancs de *chaine*,
repeat(*chaine, n* **)**: founit une chaîne obtenue en concaténant n fois *chaine*,
len_trim(*chaine* **)**: longueur d'une chaîne sans compter les espaces vides,
lge(*chaine1, chaine2* **)**: = vrai si chaine1 $\geq$ chaine2.
lle(*chaine1, chaine2* **)**: = vrai si chaine1 $\leq$ chaine2.
lgt(*chaine1, chaine2* **)**: = vrai si chaine1 $>$ chaine2.
llt(*chaine1, chaine2* **)**: = vrai si chaine1 $<$ chaine2.

29 Fonctions relatives aux tableaux.

all(a): fournit la valeur vraie si tous les éléments du tableau logique a sont vrais,
any(a): fournit la valeur vraie si l'un des éléments du tableau logique a est vrai,
count(a): fournit le nombre des valeurs vraies du tableau logique a,
dot_product(a, b): produit scalaire de deux tableaux a et b de rang 1,
MatMul (*m1, m2*): produit de deux matrices $m1$ et $m2$,
Maxval(a): fournit la plus grande valeur d'un tableau a,
Minval(a): fournit la plus petite valeur d'un tableau a,
Product(a): fournit le produit des éléments d'un tableau a,
Size(*a[, dim]*): fournit la taille de a. Si dim est présent, il fournit l'étendue dans la direction dim,
Sum(a): fournit la somme des éléments d'un tableau a,
Transpose(m): fournit la transposée d'une matrice m.