

Langage Fortran

Support de cours

Patrick Corde

Anne Fouilloux

Messagerie : Prenom.Nom@idris.fr

1 – Introduction	8
1.1 – Historique	9
1.2 – bibliographie	12
1.3 – documentation	15
2 – Généralités	18
2.1 – Bases de numération	19
2.2 – Représentation des données	21
2.2.1 – Représentation des entiers	21
2.2.2 – Représentation des réels	22
2.2.3 – Représentation des complexes	24
2.2.4 – Représentation des logiques	25
2.2.5 – Représentation des caractères	26
2.3 – Jeu de caractères	29
2.4 – Notion d'unité de programme	30
2.5 – Éléments syntaxiques	31
2.5.1 – Format libre	31
2.5.2 – Commentaires	33
3 – Déclarations	34
3.1 – Identificateurs	35

3.2 – Différents types	36
3.3 – Syntaxe	38
3.3.1 – Forme générale d’une déclaration . .	38
3.3.2 – Cas particulier : le type CHARACTER	39
3.4 – Instruction IMPLICIT NONE	40
3.5 – Constantes littérales	41
3.5.1 – Constantes entières	41
3.5.2 – Constantes réelles simple précision . .	42
3.5.3 – Constantes réelles double précision . .	43
3.5.4 – Constantes complexes	44
3.5.5 – Constantes chaînes de caractères . . .	45
3.6 – Initialisation	46
3.6.1 – L’instruction DATA	46
3.6.2 – Le symbole ”=”	48
3.7 – Constantes symboliques	49
3.8 – Instruction EQUIVALENCE	50
4 – Opérateurs et expressions	53
4.1 – Opérateurs arithmétiques	54

4.1.1 – Les opérateurs	54
4.1.2 – Les expressions	55
4.1.3 – Conversion implicite	57
4.2 – Opérateurs relationnels	58
4.3 – Opérateurs logiques	59
4.3.1 – Les tables de vérité	60
4.4 – Opérateur de concaténation	61
4.5 – Opérateur d’affectation	62
4.5.1 – syntaxe générale	62
4.5.2 – Règles de typage	62
4.6 – Priorité des Opérateurs	64
5 – Structures de contrôle	66
5.1 – Les tests	67
5.1.1 – Le bloc IF	67
5.1.2 – Le bloc SELECT-CASE	69
5.2 – Les itérations	71
5.2.1 – L’instruction GOTO	71
5.2.2 – Les boucles DO	72
6 – Tableaux	79

6.1 – Déclaration	80
6.2 – Définitions (rang, profil, étendue)	82
6.3 – Initialisation	85
6.3.1 – Le symbole “=”	85
6.3.2 – Le constructeur de vecteurs	86
6.3.3 – L’instruction DATA	88
6.4 – Manipulation de tableaux	89
6.4.1 – Expressions de type tableau	89
6.4.2 – Sections de tableaux	90
7 – Entrées-Sorties	93
7.1 – Introduction	94
7.2 – Accès séquentiel	96
7.2.1 – Fichier binaire séquentiel	97
7.2.2 – Fichier texte séquentiel	99
7.3 – Accès direct	136
7.3.1 – Fichier binaire à accès direct	137
7.3.2 – Fichier texte à accès direct	139
7.4 – Fichier temporaire	144

7.5 – Destruction d'un fichier	145
7.6 – Fichier interne	146
7.7 – Instructions de positionnement	149
7.8 – Instruction INQUIRE	152
7.9 – Remarques	154
8 – Procédures	158
8.1 – Arguments	159
8.2 – Subroutines	161
8.3 – Fonctions	162
8.4 – Arguments de type chaîne de caractères .	164
8.5 – Arguments de type tableau	166
8.6 – Arguments de type procédure	169
8.7 – Procédures internes	171
8.8 – Durée de vie et visibilité des identificateurs	174
8.9 – Procédures intrinsèques	176
9 – Common	177
9.1 – L'instruction COMMON	178
9.2 – Common blanc	179
9.3 – Common étiqueté	181

9.3.1 – Initialisation : BLOCK DATA	182
9.3.2 – Instruction SAVE et COMMON . . .	184
9.4 – Règles et restrictions	186
10 – Include	189
10.1 – La directive INCLUDE	190
11 – Annexe A : entrées-sorties - syntaxes	192
12 – Annexe B : procédures intrinsèques	210
13 – Annexe C : aspects obsolètes	222
14 – Annexe D : système de compilation	226
15 – Annexe E : exercices	230

1 – Introduction

- ➡ 1.1 Historique
- ➡ 1.2 Bibliographie
- ➡ 1.3 Documentation

1.1 – Historique

- ☞ Code machine (notation numérique en octal).
- ☞ Assembleurs de codes mnémoniques.
- ☞ 1954–Projet création du premier langage symbolique par John Backus d'IBM $\Rightarrow$ **FORTRAN** (*Mathematical **FOR**mula **TRAN**slating System*) :
 - Efficacité du code généré (performance).
 - Langage quasi naturel pour scientifiques (productivité, maintenance, lisibilité).
- ☞ 1957–Livraison des premiers compilateurs.
- ☞ 1958–**Fortran II** (IBM) $\Rightarrow$ sous-programmes compilables de façon indépendante.

- ☞ Généralisation aux autres constructeurs mais :
 - ⇒ divergences des extensions $\implies$ nécessité de **normalisation**,
 - ⇒ **ASA** *American Standards Association* ($\implies$ **ANSI** *American Nat. Standards Institute*). Comité chargé du développement d'une norme Fortran $\implies$ 1966.
- ☞ 1966–**Fortran IV** (Fortran 66).
- ☞ Évolution par extensions divergentes...
- ☞ 1977–**Fortran V** (Fortran 77).
Quasi compatible :
aucune itération des boucles *nulles* (DO I=1,0)
 - ⇒ **Nouveautés principales** :
 - ➡ type caractère,
 - ➡ IF-THEN-ELSE,
 - ➡ E/S accès direct et OPEN.

- ✎ Travail des comités X3J3/ANSI et WG5/ISO pour moderniser Fortran 77 :
 - ⇒ Standardisation : inclusion d'extensions.
 - ⇒ Développement : nouveaux concepts déjà exploités par langages plus récents APL, Algol, PASCAL, Ada, ...
 - ⇒ Performances en calcul scientifique
 - ⇒ Totalement compatible avec Fortran 77
- ✎ 1991/1992–Norme ISO et ANSI $\Rightarrow$ **Fortran 90**
- ✎ 1994 – Premiers compilateurs Fortran 90
Cray et IBM
- ✎ 1997 – Norme ISO et ANSI $\Rightarrow$ **Fortran 95**
- ✎ 1999 – Premiers compilateurs Fortran 95 sur
Cray T3E puis IBM RS/6000
- ✎ septembre 2004 – Norme ISO et ANSI $\Rightarrow$
Fortran 2003.

1.2 – bibliographie

- ➡ Adams, Brainerd, Martin, Smith et Wagener, *Fortran 95 Handbook*, MIT Press, 1997, (711 pages), ISBN 0-262-51096-0,
- ➡ Brainerd, Goldberg, Adams, *Programmer's guide to Fortran 90*, 3^eédit. Unicom, 1996, (408 pages), ISBN 0-07-000248-7,
- ➡ Chamberland Luc, *Fortran 90 : A Reference Guide*, Prentice Hall, ISBN 0-13-397332-8,
- ➡ Delannoy Claude, *Programmer en Fortran 90 – Guide complet*, Eyrolles, 1997, (413 pages), ISBN 2-212-08982-1,
- ➡ Dubesset M., Vignes J., *Les spécificités du Fortran 90*, Éditions Technip, 1993, (400 pages), ISBN 2-7108-0652-5,
- ➡ Ellis, Phillips, Lahey, *Fortran 90 Programming*, Addison-Wesley, 1994, (825 pages), ISBN 0-201-54446-6,

- ☞ Hahn B.D., *Fortran 90 for the Scientist & Engineers*, Edward Arnold, London, 1994, (360 pages), ISBN 0-340-60034-9,
- ☞ Kerrigan James F., *Migrating to Fortran 90*, O'Reilly & Associates Inc., 1994, (389 pages), ISBN 1-56592-049-X,
- ☞ Lignelet P., *Fortran 90 : approche par la pratique*, Éditions Studio Image (série informatique), 1993, ISBN 2-909615-01-4,
- ☞ Lignelet P., *Manuel complet du langage Fortran 90 et Fortran 95*, calcul intensif et génie logiciel, Col. Mesures physiques, Masson, 1996, (320 pages), ISBN 2-225-85229-4,
- ☞ Lignelet P., *Structures de données et leurs algorithmes avec Fortran 90 et Fortran 95*, Masson, 1996, (360 pages), ISBN 2-225-85373-8,
- ☞ Morgan and Schoenfelder, *Programming in Fortran 90*, Alfred Waller Ltd., 1993, ISBN 1-872474-06-3,

- ☞ Metcalf M., Reid J.,
 - ⇒ Fortran 90 explained, Science Publications, Oxford, 1994, (294 pages), ISBN 0-19-853772-7,
Traduction française par Pichon B. et Caillat M., Fortran 90 : les concepts fondamentaux, Éditions AFNOR, 1993, ISBN 2-12-486513-7,
 - ⇒ Fortran 90/95 explained, Oxford University Press, 1996, (345 pages), ISBN 0-19-851888-9,
 - ⇒ Fortran 95/2003 explained, Oxford University Press, 2004, (416 pages), ISBN 0-19-852693-8,
- ☞ Olagnon Michel, Traitement de données numériques avec Fortran 90, Masson, 1996, (364 pages), ISBN 2-225-85259-6,
- ☞ Redwine Cooper, *Upgrading to Fortran 90*, Springer, 1995, ISBN 0-387-97995-6,
- ☞ *International Standard ISO/IEC 1539-1 :1997(E) Information technology - Progr. languages - Fortran - Part1 : Base language*. Disponible auprès de l'AFNOR.

1.3 – documentation

👉 Documentation IBM RS/6000 :

- ⇒ XL Fortran Language Reference
- ⇒ XL Fortran USER's Guide
- ⇒ ESSL - Engineering and Scientific Subroutine Library Guide

👉 Disponibles sur le serveur Web IDRIS à l'URL :

`www.idris.fr/data/doc_fournisseur/ibm/index-ibmdoc.html`

👉 Documentation IDRIS RS/6000 :

- ⇒ descriptif matériel et logiciel,
- ⇒ supports de cours,
- ⇒ FAQ,
- ⇒ etc.

👉 Disponibles sur le serveur Web IDRIS à l'URL :

`http ://www.idris.fr/su/Scalaire`

Documentation NEC SX-5

- ⇒ Fortran 90/SX Language Reference Manual -- G1AF06E-7
- ⇒ Fortran 90/SX User's Guide -- G1AF07E-8
- ⇒ Fortran 90/SX Multitasking User's Guide -- G1AF08E-7
- ⇒ SUPER-UX PSUITE User's Guide -- G1AF26E-5
- ⇒ SUPER-UX OpenGL Programmer's Guide -- G1AF24E-1A
- ⇒ SUPER-UX DBX User's Guide -- G1AF19E-5
- ⇒ Serveur Web IDRIS :
 - ➡ doc. NEC en ligne (accès restreint) :
www.idris.fr/users/doc_nec-users/globalcont.html
 - ➡ support de cours « Portage de codes sur NEC SX-5 » :
http://www.idris.fr/su/divers/SX5_p.html
 - ➡ FAQ : <http://www.idris.fr/faqs/nec.html>

Documentation générale

⇒ Supports de cours Fortran 95 IDRIS :

www.idris.fr/data/cours/lang/fortran/choix_doc.html

⇒ Manuel "Fortran 77 pour débutants" (en anglais) :

www.idris.fr/data/cours/lang/fortran/choix_doc.html

⇒ *Fortran Market Index* :

www.swcp.com/~walt/

2 – Généralités

- ➡ 2.1 Bases de numération
- ➡ 2.2 Représentation des données
 - ➡ 2.2.1 Représentation des entiers
 - ➡ 2.2.2 Représentation des réels
 - ➡ 2.2.3 Représentation des complexes
 - ➡ 2.2.4 Représentation des logiques
 - ➡ 2.2.5 Représentation des caractères
- ➡ 2.3 Jeu de caractères
- ➡ 2.4 Notion d'unité de programme
- ➡ 2.5 Éléments syntaxiques
 - ➡ 2.5.1 Format libre
 - ➡ 2.5.2 Commentaires

2.1 – Bases de numération

Soit un nombre n dont l'écriture en base b est de la forme :

$$(u_p u_{p-1} \dots u_1 u_0)_b$$

avec :

$$\forall i \in \{0, 1, \dots, p\} \quad 0 \leq u_i < b$$

La valeur du nombre n en base 10 est :

$$n_{10} = \sum_{i=0}^p u_i b^i$$

Les ordinateurs ne savent calculer qu'en base 2, de ce fait les données stockées dans la mémoire le sont sous la forme d'une suite de chiffres binaires 0 et 1 appelés **bits** abréviation de *binary digits*. Un ensemble de 8 bits s'appelle un **octet**.

L'écriture des données en base 2 se révèle fastidieuse. Par commodité, on adopte plutôt la base 8 (base octale) ou la base 16 (hexadécimale) pour les définir.

L'écriture d'un nombre en octal s'effectuera à l'aide des chiffres de 0 à 7.

L'écriture d'un nombre en hexadécimal s'effectuera à l'aide des chiffres de 0 à 9 auxquels on ajoute les lettres de a à f.

Supposons que l'on dispose de l'écriture d'un nombre en base 2. Sa conversion en octal peut être faite en découpant le motif binaire par tranches de 3 bits en partant de la droite, puis en convertissant en base 10 chaque groupe obtenu.

Sa conversion en hexadécimal pourra s'effectuer de la même manière à l'aide d'un découpage par tranches de 4 bits.

Exemple

$$1001110101_2 = 1 \cdot 2^0 + 1 \cdot 2^2 + 1 \cdot 2^4 + 1 \cdot 2^5 + 1 \cdot 2^6 + 1 \cdot 2^9 \\ = 629_{10}$$

$$1001110101_2 = 1|001|110|101_2 = 1165_8$$

$$1001110101_2 = 10|0111|0101_2 = 275_{16}$$

2.2.2 – Représentation des réels

Un nombre réel ou *flottant* est caractérisé par :

- 👉 son signe,
- 👉 son exposant ou caractéristique,
- 👉 sa mantisse.

Son mode de représentation est un motif binaire respectant la norme IEEE.

Représentation d'un nombre réel sur 32 bits.

Ce type de réel, appelé réel simple précision, admet un motif binaire de la forme :

seeeeeeeem——m

- 👉 **s** : bit de signe,
- 👉 **e** : exposant sur 8 bits à excédent 127,
- 👉 **m** : mantisse sur 23 bits.

Le nombre représenté correspond à :

$$r = s1.m*2^{e-127}$$

Ce type de représentation permet de représenter les nombres :

$$1.2*10^{-38} \leq |r| \leq 3.4*10^{+38}$$

avec 6 chiffres significatifs.

Représentation d'un nombre réel sur 64 bits.

Ce type de réel, appelé réel double précision, admet un motif binaire de la forme :

seeeeeeeeeem——m

☞ s : bit de signe,

☞ e : exposant sur 11 bits à excédent 1023,

☞ m : mantisse sur 52 bits.

Le nombre représenté correspond à :

$$r = s1.m*2^{e-1023}$$

Ce type de représentation permet de représenter les nombres :

$$2.2*10^{-308} \leq |r| \leq 1.8*10^{+308}$$

avec 15 chiffres significatifs.

2.2.3 – Représentation des complexes

Un nombre complexe est une paire de nombres réels, simple ou double précision, correspondant aux parties réelle et imaginaire.

Exemple

Soit le nombre complexe $1.5 - 1.5i$

Sa représentation en simple précision nécessite 2 réels sur 32 bits :

$$0 \ 01111111 \ 1000 \dots 000_2 = 3FC00000_{16} = +1.5_{10}$$

$$1 \ 01111111 \ 1000 \dots 000_2 = BFC00000_{16} = -1.5_{10}$$

2.2.4 – Représentation des logiques

Un logique est une entité qui peut prendre comme valeur :

➡ .TRUE.

➡ .FALSE.

Il est représenté en général sur 32 bits (4 octets). Il peut exister des variantes codées sur 1, 2 voire 8 octets. Tous les bits sont positionnés à 0 sauf le bit le plus à droite qui pour la valeur .TRUE. est positionné à 1.

2.2.5 – Représentation des caractères

Un caractère est codé sur 1 octet. Sa représentation interne respecte un codage appelé codage ASCII.

Il existe 128 caractères différents dont les représentations sont indiquées dans une table dite table ASCII.

Dans cette table les caractères numériques ainsi que les caractères alphabétiques (majuscules et minuscules) sont rangés consécutivement et en ordre croissant.

On appelle chaîne de caractères une suite de caractères rangés de façon consécutive en mémoire.

TAB. 1 – table des codes ASCII des caractères

Caract.	déc.	hex	oct.	Caract.	déc.	hex	oct.
C-@ (NUL)	0	0x00	000	espace	32	0x20	040
C-a (SOH)	1	0x01	001	!	33	0x21	041
C-b (STX)	2	0x02	002	"	34	0x22	042
C-c (ETX)	3	0x03	003	#	35	0x23	043
C-d (EOT)	4	0x04	004	\$	36	0x24	044
C-e (ENQ)	5	0x05	005	%	37	0x25	045
C-f (ACK)	6	0x06	006	&	38	0x26	046
C-g (BEL)	7	0x07	007	,	39	0x27	047
C-h (BS)	8	0x08	010	(	40	0x28	050
C-i (HT)	9	0x09	011	)	41	0x29	051
C-j (LF)	10	0x0a	012	*	42	0x2a	052
C-k (VT)	11	0x0b	013	+	43	0x2b	053
C-l (FF)	12	0x0c	014	,	44	0x2c	054
C-m (CR)	13	0x0d	015	-	45	0x2d	055
C-n (SO)	14	0x0e	016	.	46	0x2e	056
C-o (SI)	15	0x0f	017	/	47	0x2f	057
C-p (DLE)	16	0x10	020	0	48	0x30	060
C-q (DC1)	17	0x11	021	1	49	0x31	061
C-r (DC2)	18	0x12	022	2	50	0x32	062
C-s (DC3)	19	0x13	023	3	51	0x33	063
C-t (DC4)	20	0x14	024	4	52	0x34	064
C-u (NAK)	21	0x15	025	5	53	0x35	065
C-v (SYN)	22	0x16	026	6	54	0x36	066
C-w (ETB)	23	0x17	027	7	55	0x37	067
C-x (CAN)	24	0x18	030	8	56	0x38	070
C-y (EM)	25	0x19	031	9	57	0x39	071
C-z (SUB)	26	0x1a	032	:	58	0x3a	072
C-[(ESC)	27	0x1b	033	;	59	0x3b	073
C-\ (FS)	28	0x1c	034	<	60	0x3c	074
C-] (GS)	29	0x1d	035	=	61	0x3d	075
C-\$ (RS)	30	0x1e	036	>	62	0x3e	076
C-_ (US)	31	0x1f	037	?	63	0x3f	077

Caract.	déc.	hex	oct.	Caract.	déc.	hex	oct.
@	64	0x40	100	'	96	0x60	140
A	65	0x41	101	a	97	0x61	141
B	66	0x42	102	b	98	0x62	142
C	67	0x43	103	c	99	0x63	143
D	68	0x44	104	d	100	0x64	144
E	69	0x45	105	e	101	0x65	145
F	70	0x46	106	f	102	0x66	146
G	71	0x47	107	g	103	0x67	147
H	72	0x48	110	h	104	0x68	150
I	73	0x49	111	i	105	0x69	151
J	74	0x4a	112	j	106	0x6a	152
K	75	0x4b	113	k	107	0x6b	153
L	76	0x4c	114	l	108	0x6c	154
M	77	0x4d	115	m	109	0x6d	155
N	78	0x4e	116	n	110	0x6e	156
O	79	0x4f	117	o	111	0x6f	157
P	80	0x50	120	p	112	0x70	160
Q	81	0x51	121	q	113	0x71	161
R	82	0x52	122	r	114	0x72	162
S	83	0x53	123	s	115	0x73	163
T	84	0x54	124	t	116	0x74	164
U	85	0x55	125	u	117	0x75	165
V	86	0x56	126	v	118	0x76	166
W	87	0x57	127	w	119	0x77	167
X	88	0x58	130	x	120	0x78	170
Y	89	0x59	131	y	121	0x79	171
Z	90	0x5a	132	z	122	0x7a	172
[	91	0x5b	133	{	123	0x7b	173
\	92	0x5c	134		124	0x7c	174
]	93	0x5d	135	}	125	0x7d	175
^	94	0x5e	136	~	126	0x7e	176
_	95	0x5f	137	C-?	127	0x7f	177

2.3 – Jeu de caractères

- ➡ 26 lettres de l'alphabet,
- ➡ chiffres 0 à 9,
- ➡ caractères spéciaux :

!	*	+	"	<
(	=	>	)	;
%	/	-	:	,
?	'	.	&	\$

- ➡ le caractère espace,
- ➡ le caractère `_` (*underscore*).

Remarque :

les caractères minuscules sont convertis en majuscules par le compilateur.

2.4 – Notion d'unité de programme

Un programme source **Fortran** est composé de parties indépendantes appelées **unités de programme** (*scoping unit*).

Chaque partie est compilée de façon indépendante. Chacune admet son propre environnement. Il sera cependant possible que ces parties communiquent entre elles.

Les différentes unités de programme sont :

- 👉 le programme principal,
- 👉 les sous-programmes :
 - ➡ de type *subroutine*,
 - ➡ de type *function*,
- 👉 les modules,
- 👉 les *block data*.

Chaque unité comprend une partie déclarative (déclaration des variables locales, ...) suivie d'une partie comportant des instructions exécutables ; l'instruction **STOP** interrompt le programme.

2.5 – Éléments syntaxiques

2.5.1 – Format libre

Dans le mode « **format libre** » les lignes peuvent être de longueur quelconque à concurrence de **132** caractères.

Il est également possible de coder plusieurs instructions sur une même ligne en les séparant avec le caractère « ; ».

Exemple

```
print *, ' Entrez une valeur :'; read *,n
```

Une instruction peut être codée sur plusieurs lignes : on utilisera alors le caractère « & ».

Exemple

```
print *, 'Montant HT :', montant_ht, &  
      '      TVA :', tva      , &  
      'Montant TTC :', montant_ttc
```

Lors de la coupure d'une chaîne de caractères la suite de la chaîne doit obligatoirement être précédée du caractère « & ».

Exemple

```
print *, 'Entrez un nombre entier &  
&compris entre 100 & 199'
```

Remarque : il existe aussi le « Format fixe », considéré maintenant comme obsolète dont la structure d'une ligne est :

- 👉 zone étiquette (colonnes 1 à 5)
- 👉 zone instruction (colonnes 7 à 72)
- 👉 colonne suite (colonne 6)

2.5.2 – Commentaires

Le caractère « ! » rencontré sur une ligne indique que ce qui suit est un commentaire. On peut évidemment écrire une ligne complète de commentaires : il suffit pour cela que le 1^{er} caractère non blanc soit le caractère « ! »

Exemple

```
if (n < 100 .or. n > 199) ! Test cas d'erreur
    . . . . .
! On lit l'exposant
read *,x
! On lit la base
read *,y
if (y <= 0) then    ! Test cas d'erreur
    print *, ' La base doit être un nombre >0'
else
    z = y**x        ! On calcule la puissance
end if
```

Remarque : en format fixe, les lignes qui commencent par **C**, **c**, ***** ou **!** en colonne 1 sont des commentaires.

3 – Déclarations

- ➡ 3.1 Identificateurs
- ➡ 3.2 Différents types
- ➡ 3.3 Syntaxe
- ➡ 3.4 Instruction IMPLICIT NONE
- ➡ 3.5 Constantes littérales
- ➡ 3.6 Initialisation
- ➡ 3.7 Constantes symboliques
- ➡ 3.8 Instruction EQUIVALENCE

3.1 – Identificateurs

Un identificateur permet de donner un nom à :

- ➡ une **variable**,
- ➡ une **constante**,
- ➡ une **procédure**.

Il est défini par :

- ➡ une suite de caractères **alphanumériques** (lettres non accentuées, chiffres, *underscore*),
- ➡ le premier caractère doit être une **lettre**,
- ➡ la longueur est limitée à **31 caractères**,
- ➡ on ne distingue pas les lettres **majuscules** des **minuscules**.

Exemples d'identificateurs

compteur

Compteur

fin_de_fichier

montant_annee_1993

3.2 – Différents types

Le type d'une variable détermine :

- ➡ le nombre d'octets à réserver en mémoire,
- ➡ un mode de représentation interne,
- ➡ l'ensemble des valeurs admissibles,
- ➡ l'ensemble des opérateurs qui peuvent lui être appliqués.

Types prédéfinis ou de bases

INTEGER	: entier
CHARACTER	: caractère
LOGICAL	: deux valeurs .TRUE. / .FALSE.
REAL	: réel simple précision
DOUBLE PRECISION	: réel double précision
COMPLEX	: complexe simple précision

Remarque : la précision d'un réel simple est de 7 chiffres décimaux significatifs alors que celle d'un double est de 15.

Attributs

Chaque type peut être surchargé des attributs suivants :

PARAMETER	: constante symbolique
DIMENSION	: taille d'un tableau
SAVE	: objet statique
EXTERNAL	: procédure externe
INTRINSIC	: procédure intrinsèque

3.3 – Syntaxe

3.3.1 – Forme générale d'une déclaration

type[, liste_attributs ::] liste_identificateurs

Exemple :

```
PROGRAM declaration
```

```
  INTEGER, SAVE      ::  compteur
```

```
  INTEGER            ::  temperature
```

```
  LOGICAL            ::  arret_boucle
```

```
    ...
```

```
END PROGRAM declaration
```

```
PROGRAM declaration
```

```
  INTEGER            indice_boucle
```

```
  SAVE              indice_boucle
```

```
    ...
```

```
END PROGRAM declaration
```

3.3.2 – Cas particulier : le type CHARACTER

Pour déclarer une chaîne de caractères on précise de plus sa longueur. Si elle n'est pas indiquée elle est égale à 1 :

```
CHARACTER(LEN=N) ch_car
```

```
CHARACTER      c
```

Il est possible mais déconseillé d'utiliser la syntaxe suivante :

```
CHARACTER*N ch_car
```

Exemple

```
PROGRAM declaration
  CHARACTER(LEN=11)  chaine1
  CHARACTER*11      chaine2
  ...
END PROGRAM declaration
```

3.4 – Instruction IMPLICIT NONE

Par défaut, les variables dont l'identificateur commence par les caractères I à N sont de type **INTEGER**.

Toutes les autres sont de type **REAL**.

L'instruction **IMPLICIT NONE** change cette règle car elle impose à l'utilisateur la **déclaration** de chaque variable.

👉 elle permet la détection d'un certain nombre d'erreurs à la compilation.

Cette instruction est vivement **recommandée!!!**

👉 **IMPLICIT NONE** se place avant les déclarations des variables,

👉 L'instruction ne s'applique qu'à l'unité de programme qui la contient.

3.5 – Constantes littérales

3.5.1 – Constantes entières

- ☞ une suite de chiffres en base 10,
- ☞ une suite de chiffres en base 2 encadrée par des quotes, le tout précédé du caractère **B**,
- ☞ une suite de chiffres en base 8 encadrée par des quotes, le tout précédé du caractère **O**,
- ☞ une suite de chiffres en base 16 encadrée par des quotes, le tout précédé du caractère **Z**.

Une valeur négative sera précédée du signe -.

Exemple

```
1
123
-28
B'11011011100'
O'3334'
Z'6DC'
```

Remarque : les constantes écrites en base 2, 8 ou 16 s'appellent des constantes **BOZ**. Elles ne peuvent figurer que dans les intructions d'initialisation de type **DATA**.

3.5.2 – Constantes réelles simple précision

Une constante de type **REAL** doit obligatoirement comporter :

- ➡ soit le point décimal, même s'il n'y a pas de chiffres après la virgule,
- ➡ soit le caractère E pour la notation en virgule flottante.

Pour les nombres écrits `0.xxxxx`, on peut omettre le 0 avant le point décimal.

Exemple

```
0.  
1.0  
1.  
3.1415  
31415E-4  
1.6E-19  
1E12  
.001  
-36.
```

3.5.3 – Constantes réelles double précision

Une constante double precision doit obligatoirement être écrite en virgule flottante, le E étant remplacé par un D.

Exemple

```
0D0
0.D0
1.D0
1d0
3.1415d0
31415d-4
1.6D-19
1d12
-36.d0
```

3.5.4 – Constantes complexes

Une constante de type **COMPLEX** est obtenue en combinant deux constantes réelles entre parenthèses séparées par une virgule.

`2.5+i` s'écrira `(2.5,1.)`

Exemple

`(0.,0.)`

`(1.,-1.)`

`(1.34e-7, 4.89e-8)`

3.5.5 – Constantes chaînes de caractères

Une constante chaînes de caractères est une suite de caractères encadrée par le délimiteur ' ou bien ".

Si parmi la suite des caractères figure le caractère délimiteur, il devra être doublé.

Exemple

```
'La somme des n premiers entiers est : '  
'l''étendue désirée est : '  
"l'étendue désirée est : "
```

À partir d'une variable chaîne de caractères on peut extraire une suite de caractères contigus. Pour cela on spécifie le nom de la variable suivi entre parenthèses d'un couple d'entiers *n:m* indiquant les rangs de début et de fin d'extraction.

Exemple

```
CHARACTER(LEN=10) :: ch  
  
ch = "Bonjour"  
ch(4:7) = "soir"
```

3.6 – Initialisation

3.6.1 – L'instruction DATA

DATA liste₁/init₁/[, ..., liste_i/init_i/, ...]

- ➡ liste_i fait référence à une liste de variables à initialiser,
- ➡ init_i indique les valeurs d'initialisation,
- ➡ le type des valeurs d'initialisation doit respecter les règles suivantes :
 - ⇒ pour un objet de type caractère ou logique, la constante d'initialisation doit être de même type,
 - ⇒ pour un objet de type entier, réel ou complexe, la constante d'initialisation peut être de type entier, réel ou complexe.

Exemple

REAL	A, B, C
INTEGER	N, M
LOGICAL	arret
DATA	A, B, N/1.0, 2.0, 17/
DATA	C/2.6/, M/3/
DATA	arret/.FALSE./

Remarques :

- 👉 cette instruction peut apparaître après des instructions exécutables, mais la norme F95 a déclaré cette possibilité comme obsolète,
- 👉 les variables initialisées par ce moyen héritent de l'attribut **SAVE** : elles sont alors permanentes (cf. chapitre Procédures, section Durée de vie des identificateurs),
- 👉 ce type d'initialisation peut être faite directement lors de la déclaration.

Exemple

REAL	A/3.14/, B/2.718/
INTEGER	N/1/, M/4/
LOGICAL	arret/.false./

3.6.2 – Le symbole “=”

Fortran permet d’initialiser une variable lors de sa déclaration à l’aide du symbole “**=**”. Dans ce contexte, les caractères `::` sont obligatoires.

```
TYPE[, attributs] :: v1=c1[, ..., v_i=c_i, ...]
```

où v_i est le nom de la variable à initialiser et c_i sa valeur.

Exemple

```
PROGRAM initialisation
  INTEGER  :: debut   = 100
  REAL     :: valeur  = 76.3
  LOGICAL  :: drapeau = .TRUE.
              ...
END PROGRAM initialisation
```

3.7 – Constantes symboliques

L'attribut **PARAMETER** permet de donner un nom symbolique à une constante littérale :

TYPE, **PARAMETER** :: $n_1 = c_1$ [, ..., $n_i = c_i$, ...]

où n_i est le nom donné à une constante et c_i sa valeur.

La notation suivante est aussi utilisable :

PARAMETER ($n_1 = c_1$ [, ..., $n_i = c_i$, ...])

Exemple

```
PROGRAM constante
  LOGICAL, PARAMETER :: VRAI=.TRUE., FAUX=.FALSE.

  DOUBLE PRECISION    :: PI, RTOD
  PARAMETER (PI=3.14159265d0, RTOD=180.d0/PI)
      ...
END PROGRAM constante
```

3.8 – Instruction EQUIVALENCE

- ➡ L'instruction **EQUIVALENCE** permet à des variables de partager la même zone mémoire au sein d'une unité de programme,
- ➡ il n'y a pas de conversion de type,
- ➡ chaque variable garde les propriétés de son type.
- ➡ le type **CHARACTER** ne peut pas être associé à d'autres types.

Syntaxe générale

EQUIVALENCE (v_1 , v_2) [, ..., (v_{i-1} , v_i), ...]

où les v_i sont des scalaires (variables simples ou éléments de tableaux).

Exemple-1

```

PROGRAM correspondance
  COMPLEX      cmplx(2)
  REAL         temp(4)
  EQUIVALENCE  (temp(1), cmplx(1))
               ...
END PROGRAM correspondance

```

On a alors en mémoire :

```

|-----cmplx(1)-----|-----cmplx(2)-----| | |
|---|---|---|---|
|--temp(1)--|--temp(2)--|--temp(3)--|--temp(4)--|

```

Exemple-2

```
PROGRAM correspondance
  CHARACTER(LEN=4)                :: A, B
  CHARACTER(LEN=3)                :: C(2)
  CHARACTER(LEN=10)               :: chaine
  CHARACTER(LEN=1), DIMENSION(10) :: tab_car
  EQUIVALENCE (A,C(1)), (B,C(2))
  EQUIVALENCE (chaine,tab_car(1))
  ...
END PROGRAM correspondance
```

On a alors en mémoire :

```
| 01 | 02 | 03 | 04 | 05 | 06 | 07 |
|-----A-----|
|----C(1)----|----C(2)----|
                |-----B-----|

| 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 |
|----- chaine -----|
|                                |
|--> tab_car(1)                |--> tab_car(7)
```

4 – Opérateurs et expressions

- ➡ 4.1 Opérateurs arithmétiques
- ➡ 4.2 Opérateurs relationnels
- ➡ 4.3 Opérateurs logiques
- ➡ 4.4 Opérateur de concaténation
- ➡ 4.5 Opérateur d'affectation
- ➡ 4.6 Priorité des opérateurs

4.1 – Opérateurs arithmétiques

4.1.1 – Les opérateurs

Opérateur	Opération
+	addition (opérateur binaire)
+	identité (opérateur unaire)
-	soustraction (opérateur binaire)
-	opposé (opérateur unaire)
*	multiplication
/	division
**	puissance

4.1.2 – Les expressions

Règle d'usage	Interprétation
$\mathbf{o_1} \mathbf{+} \mathbf{o_2}$	ajoute $\mathbf{o_2}$ à $\mathbf{o_1}$
$\mathbf{+} \mathbf{o_1}$	égal à $\mathbf{o_1}$
$\mathbf{o_1} \mathbf{-} \mathbf{o_2}$	soustrait $\mathbf{o_2}$ à $\mathbf{o_1}$
$\mathbf{-} \mathbf{o_1}$	inverse le signe de $\mathbf{o_1}$
$\mathbf{o_1} \mathbf{*} \mathbf{o_2}$	multiplie $\mathbf{o_1}$ par $\mathbf{o_2}$
$\mathbf{o_1} \mathbf{/} \mathbf{o_2}$	$\mathbf{o_1}$ divisé par $\mathbf{o_2}$
$\mathbf{o_1} \mathbf{**} \mathbf{o_2}$	élève $\mathbf{o_1}$ à la puissance $\mathbf{o_2}$

Les opérandes $\mathbf{o_1}$ et $\mathbf{o_2}$ peuvent être :

- 👉 une constante numérique,
- 👉 une variable numérique, précédée ou non d'un opérateur unaire (+ ou -),
- 👉 une expression arithmétique entre parenthèses.

Exemples d'expressions

3.14159

K

(A + B) * (C + D)

-1.0 / X + Y / Z ** 2

-2.0 * 3.14159 * RADIUS

4.1.3 – Conversion implicite

Le type d'une expression arithmétique dépend des types de ses opérandes.

Dans le cas d'opérateurs binaires :

- ❶ si les 2 opérandes sont du même type alors l'expression arithmétique résultante sera du même type.
- ❷ si un des 2 opérandes est de type **INTEGER** et l'autre de type **REAL**, alors l'opérande de type **INTEGER** est préalablement converti en **REAL** et l'opération effectuée en mode **REAL**.

Exemples

Expression	Valeur
99/100	0
7/3	2
(100*9)/5	180
(9/5)*100	100

4.2 – Opérateurs relationnels

Opérateur	Opération
<code>.LT.</code> ou <code><</code>	strictement plus petit
<code>.LE.</code> ou <code><=</code>	inférieur ou égal
<code>.EQ.</code> ou <code>==</code>	égal
<code>.NE.</code> ou <code>/=</code>	non égal
<code>.GT.</code> ou <code>></code>	strictement plus grand
<code>.GE.</code> ou <code>>=</code>	supérieur ou égal

Ces opérateurs admettent des opérandes de type INTEGER, REAL ou CHARACTER. Seuls les opérateurs `==`, `/=` peuvent s'appliquer à des expressions de type COMPLEX.

Exemples

```
N .GE. 0
X .LT. Y
Z /= 3.7
(B**2 - 4*A*C) .GT. 0.
```

4.3 – Opérateurs logiques

Opérateur	Opération
.NOT.	négation logique
.AND.	conjonction logique
.OR.	disjonction inclusive
.EQV.	équivalence logique
.NEQV.	non-équivalence logique

Les opérandes doivent être des expressions de type LOGICAL.

4.3.1 – Les tables de vérité

Opérateur de négation :

l	$.NOT.l$
<code>.true.</code>	<code>.false.</code>
<code>.false.</code>	<code>.true.</code>

Autres opérateurs :

l_1	l_2	$l_1 .AND. l_2$	$l_1 .OR. l_2$
<code>.true.</code>	<code>.true.</code>	<code>.true.</code>	<code>.true.</code>
<code>.true.</code>	<code>.false.</code>	<code>.false.</code>	<code>.true.</code>
<code>.false.</code>	<code>.true.</code>	<code>.false.</code>	<code>.true.</code>
<code>.false.</code>	<code>.false.</code>	<code>.false.</code>	<code>.false.</code>

l_1	l_2	$l_1 .EQV. l_2$	$l_1 .NEQV. l_2$
<code>.true.</code>	<code>.true.</code>	<code>.true.</code>	<code>.false.</code>
<code>.true.</code>	<code>.false.</code>	<code>.false.</code>	<code>.true.</code>
<code>.false.</code>	<code>.true.</code>	<code>.false.</code>	<code>.true.</code>
<code>.false.</code>	<code>.false.</code>	<code>.true.</code>	<code>.false.</code>

4.4 – Opérateur de concaténation

L'opérateur de concaténation n'admet que des expressions de type **CHARACTER**.

Expression	Interprétation
$c_1 \text{ // } c_2$	concatène c_1 avec c_2

Exemple

'BON' // 'JOUR' --> 'BONJOUR'

CHARACTER(LEN=10) :: ch = 'BON'

ch = ch // 'JOUR' ! <-- INVALIDE !!!

ch = TRIM(ch) // 'JOUR' ! <-- VALIDE

4.5 – Opérateur d'affectation

4.5.1 – syntaxe générale

`variable` **=** `expression`

où `expression` est une expression arithmétique, logique ou relationnelle.

4.5.2 – Règles de typage

- ❶ une valeur de type **CHARACTER** ne peut pas être affectée à une variable numérique ou *vice-versa*,
- ❷ une valeur de type **INTEGER** peut être affectée à une variable de type **REAL**,
- ❸ une valeur de type **REAL** peut également être affectée à une variable de type **INTEGER**. Mais dans ce cas, la valeur est alors tronquée en supprimant la partie fractionnaire.

Exemples

Expression	Interprétation
x  5	x  5.0
N  0.9999	N  0
M  -1.9999	M  -1

4.6 – Priorité des Opérateurs

Dans le tableau suivant, les opérateurs sont donnés par ordre de priorité décroissante :

Opérateur	Associativité
**	$D \rightarrow G$
* et /	$G \rightarrow D$
+ et -	$G \rightarrow D$
//	$G \rightarrow D$
<, <=, == / =, >, >=	$G \rightarrow D$
.NOT.	$G \rightarrow D$
.AND.	$G \rightarrow D$
.OR.	$G \rightarrow D$
.EQV. et .NEQV.	$G \rightarrow D$

Exemples

```
REAL a,b,c,d
```

```
LOGICAL e, f, g
```

Expression

Interprétation

```
2**3**2
```

```
2**(3**2) = 512
```

```
5.+4.*9.**2
```

```
5.+(4.*(9.**2)) = 329
```

```
e.OR.f.AND.g
```

```
e.OR.(f.AND.g)
```

```
a**b+c.GT.d.AND.e
```

```
((a**b)+c).GT.d).AND.e
```

5 – Structures de contrôle

5.1 Les tests

5.1.1 Le bloc IF

5.1.2 Le bloc SELECT-CASE

5.2 Les itérations

5.2.1 L'instruction GOTO

5.2.2 Les boucles DO

5.1 – Les tests

5.1.1 – Le bloc IF

```
[nom_bloc: ] IF( exp1 ) THEN  
    bloc1  
[ELSE IF( exp2 ) THEN [nom_bloc]  
    bloc2  
    ...  
[ELSE [nom_bloc]  
    blocn]]  
END IF [nom_bloc]
```

- 👉 nom_bloc une étiquette,
- 👉 exp_i une expression de type **LOGICAL**,
- 👉 bloc_i une suite d'instructions **Fortran**.

En l'absence de clause ELSE lorsque bloc₁ est réduit à une seule instruction, la structure IF se simplifie en :

```
IF (exp) instruction
```

Exemples

```
PROGRAM structure_if
  REAL A,B,SUM
  ...
  IF (A.LT.B) THEN
    SUM = SUM + A
    IF (SUM > 0.) PRINT *, SUM
  END IF
  ...
END PROGRAM structure_if
```

```
PROGRAM structure_if
  REAL A,HRS
  ...
  IF (HRS.LE.40.0) THEN
    A = HRS*150.0
  ELSE IF (HRS.LE.50.) THEN
    A = (HRS-40.0)*150.0*1.5
  ELSE
    A = (HRS-50.0)*150.0*2.0
  END IF
  ...
END PROGRAM structure_if
```

5.1.2 – Le bloc SELECT-CASE

L'instruction **SELECT CASE** permet des branchements multiples qui dépendent de la valeur d'une expression scalaire de type entier, logique ou chaîne de caractères.

```
[ nom_bloc: ] SELECT CASE(expression)
    [ CASE(liste) [ nom_bloc ]
        bloc1
        ...
    [ CASE DEFAULT [ nom_bloc ]
        blocn ]
END SELECT [ nom_bloc ]
```

- 👉 `nom_bloc` est une étiquette,
- 👉 `expression` est une expression de type **INTEGER**, **LOGICAL** ou **CHARACTER**,
- 👉 `liste` est une liste de constantes du même type que `expression`,
- 👉 `bloci` est une suite d'instructions **Fortran**.

Exemples

```
PROGRAM structure_case
  integer :: mois, nb_jours
  logical :: annee_bissext
  ...
  SELECT CASE (mois)
    CASE (4, 6, 9, 11)
      nb_jours = 30
    CASE (1, 3, 5, 7:8, 10, 12)
      nb_jours = 31
    CASE (2)
      !-----
      fevrier: select case (annee_bissext)
        case (.true.)
          nb_jours = 29
        case (.false.)
          nb_jours = 28
      end select fevrier
      !-----
    CASE DEFAULT
      print *, ' Numéro de mois invalide'
  END SELECT
END PROGRAM structure_case
```

5.2 – Les itérations

5.2.1 – L'instruction GOTO

L'instruction **GOTO** permet d'effectuer un branchement à un endroit particulier du code :

GOTO étiquette

Cette instruction est à éviter car elle peut générer des programmes **illisibles** et **difficiles à corriger**.

Exemple

```
PROGRAM iteration_goto
  REAL diviseur, valeur, facteur
  ...
  valeur = 0. ; diviseur = 360.
69  IF (diviseur .NE. 0.) THEN
    valeur = valeur + facteur / diviseur
    diviseur = diviseur - 10.
    GOTO 69
  END IF
  ...
END PROGRAM iteration_goto
```

Cet exemple peut être remplacé par une boucle itérative de type **DO WHILE**.

5.2.2 – Les boucles DO

Il existe plusieurs types de boucles itératives qui sont toutes de la forme :

```
[ nom_bloc: ] DO [contrôle_de_boucle]
                bloc
                END DO [ nom_bloc ]
```

- 👉 `nom_bloc` est une étiquette,
- 👉 `contrôle_de_boucle` définit les conditions d'exécution et d'arrêt de la boucle,
- 👉 `bloc` est une suite d'instructions **Fortran**.

1^{re} forme : DO indéré

contrôle_de_boucle est de la forme :

variable = expr₁, expr₂ [,expr₃]

avec :

- 👉 variable est une variable de type **INTEGER**,
- 👉 expr₁, expr₂ et expr₃ sont des expressions arithmétiques de type **INTEGER**.

Le nombre d'itérations est évalué avant le démarrage de la boucle.

Exemple

```
PROGRAM iteration_do
    INTEGER i, somme, n
        ...
! affectation de n
    somme=0
    DO i=1,n,2
        somme=somme+i
    END DO
        ...
END PROGRAM iteration_do
```

2^e forme : DO WHILE

`contrôle_de_boucle` est de la forme :

WHILE (`expression`)

avec `expression` de type scalaire logique.

Le corps de la boucle est exécuté tant que l'expression est vraie.

Remarque : pour pouvoir sortir de la boucle, il faut que `expression` puisse prendre la valeur `.FALSE.` dans le bloc.

Exemple

Sommation de la série $\sum_{n \geq 1} 1/n^2$ jusqu'à ce que le terme général soit inférieur à ϵ fois la somme partielle courante :

```
PROGRAM iteration_while
  INTEGER                                :: n
  DOUBLE PRECISION                       :: somme
  DOUBLE PRECISION, PARAMETER            :: epsilon = 1.d-3
  LOGICAL                                :: fini

! Initialisation
n=0
somme=0.d0
fini=.FALSE.
DO WHILE (.not. fini)
  n=n+1
  somme=somme + 1d0/n**2
  fini=(1d0/n**2 .LT. epsilon*somme)
END DO
print *, "Nombre d'itérations : ", n
print *, "Somme = ", somme
END PROGRAM iteration_while
```

3^e forme : DO

Ce sont des boucles **DO** sans contrôle de boucle. Pour en sortir, on utilise une instruction conditionnelle avec une instruction **EXIT**.

bloc est de la forme :

```
bloc1  
  IF (expression) EXIT  
bloc2
```

avec :

- 👉 **expression** une expression de type **LOGICAL**,
- 👉 **bloc_i** des séquences de code **Fortran**.

Notons que la condition **IF** peut être remplacée par une instruction de type **SELECT CASE**.

Exemple

```
PROGRAM iteration_exit
  REAL          :: valeur
  REAL          :: x, xlast
  REAL, PARAMETER :: tolerance = 1.0e-6

  valeur = 50.

  x = 1.0          ! valeur initiale (diff. 0)

  DO
    xlast = x
    x = 0.5 * (xlast + valeur/xlast)
    IF (ABS(x-xlast)/x < tolerance) EXIT
  END DO
END PROGRAM iteration_exit
```

Instruction CYCLE

bloc_i peut aussi contenir une instruction **CYCLE** :

IF (expression) **CYCLE**

CYCLE permet d'abandonner le traitement de l'itération courante et de passer à la suivante.

Là aussi, l'instruction **IF** peut être remplacée par une instruction de type **SELECT CASE**.

Exemple

```
PROGRAM iteration_cycle
  INTEGER :: annee

  DO
    READ(*,*) annee
    IF (annee .LE. 0) EXIT
    ! On élimine les années bissextiles.
    IF( ((annee/4*4 .EQ. annee) .AND. &
        (annee/100*100 .NE. annee)) .OR. &
        (annee/400*400 .EQ. annee) ) CYCLE
    PRINT*, 'Traitement des années non-bissextiles'
    ...
  END DO
END PROGRAM iteration_cycle
```

6 – Tableaux

- ➡ 6.1 Déclaration
- ➡ 6.2 Définitions (rang, profil, étendue)
- ➡ 6.3 Initialisation
 - ➡ 6.3.1 Le symbole ”=”
 - ➡ 6.3.2 Le constructeur de vecteurs
 - ➡ 6.3.3 L’instruction DATA
- ➡ 6.4 Manipulation de tableaux

6.1 – Déclaration

Un tableau est un ensemble d'éléments de même type contigus en mémoire.

Pour déclarer un tableau, il est recommandé d'utiliser l'attribut **DIMENSION** :

TYPE, **DIMENSION**($\text{expr}_1, \dots, \text{expr}_n$) :: liste_tab

avec :

- ➡ $n \leq 7$ i.e un tableau peut avoir jusqu'à 7 dimensions
- ➡ expr_i sert à indiquer l'**étendue** dans la dimension correspondante. C'est une expression qui peut être spécifiée à l'aide :
 - ➡ d'une constante entière (littérale ou symbolique) ; dans ce cas, la borne inférieure du tableau est 1,
 - ➡ d'une expression de la forme **cste₁ : cste₂** avec **cste₁**, **cste₂** des constantes entières telles que **cste₁ ≤ cste₂**,
- ➡ liste_tab est une liste de tableaux.

Exemples

```

INTEGER, PARAMETER                :: lda = 6
REAL, DIMENSION(0:lda-1)          :: Y
REAL, DIMENSION(1+lda*lda,10)     :: Z

REAL, DIMENSION(100)              :: R
REAL, DIMENSION(1:5,1:5, 1:5)     :: S
REAL, DIMENSION(-10:-1)           :: X
    
```

Exemple

Remarque : les dimensions d'un tableau peuvent aussi être spécifiées sans l'attribut **DIMENSION** :

```

REAL :: T(10,10), U(4,2), G(-1:10,4:9,1:3)
    
```

Attention, cette notation n'est pas recommandée !

6.2 – Définitions (rang, profil, étendue)

- ➡ Le **rang** (*rank*) d'un tableau est son nombre de dimensions.
- ➡ Le nombre d'éléments dans une dimension s'appelle l'**étendue** (*extent*) du tableau dans cette dimension.
- ➡ Le **profil** (*shape*) d'un tableau est un vecteur dont chaque élément est l'**étendue** du tableau dans la dimension correspondante.
- ➡ La **taille** (*size*) d'un tableau est le produit des éléments du vecteur correspondant à son **profil**.
- ➡ Deux tableaux sont dits **conformants** s'ils ont le même **profil**.

Exemples

```
REAL, DIMENSION(15)      :: X  
REAL, DIMENSION(1:5,1:3) :: Y, Z
```

- ➡ Le tableau X est de rang 1, Y et Z sont de rang 2.
- ➡ L'étendue de X est 15, Y et Z ont une étendue de 5 et 3.
- ➡ Le profil de X est le vecteur (/ 15 /), celui de Y et Z est le vecteur (/ 5,3 /)
- ➡ La taille des tableaux X, Y et Z est 15.
- ➡ Les tableaux Y et Z sont conformants.

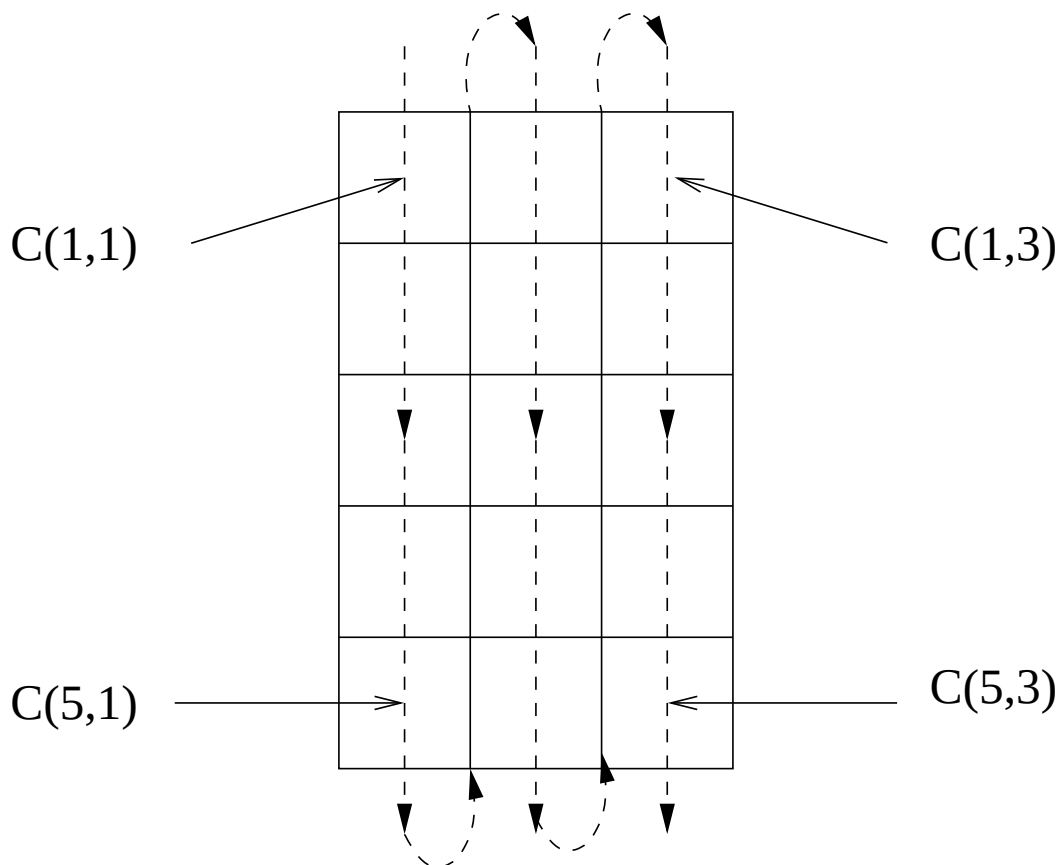
Ordre des éléments

En mémoire la notion de tableau n'existe pas : les éléments sont rangés les uns à la suite des autres.

Pour accéder à ces éléments, dans l'ordre mémoire, Fortran fait d'abord varier le premier indice, puis le second et ainsi de suite.

Par exemple, les éléments d'un tableau à deux dimensions sont ordonnés comme suit :

```
REAL, DIMENSION(5,3) :: C  
C(1,1), C(2,1), ..., C(5,1), C(1,2), C(2,2), ..., C(5,3)
```



6.3 – Initialisation

6.3.1 – Le symbole “=”

Fortran permet de manipuler globalement l’ensemble des éléments d’un tableau.

On pourra alors utiliser le symbole “=” comme pour l’initialisation d’une variable scalaire.

Exemple

Pour initialiser à 3 l’ensemble d’un vecteur :

```
REAL, DIMENSION(100) :: X = 3.
```

6.3.2 – Le constructeur de vecteurs

Un constructeur de vecteur est un **vecteur de scalaires** dont les valeurs sont encadrées par les caractères (/ et /) :

$$\text{tableau} = (/ \text{expr}_1, \text{expr}_2, \dots, \text{expr}_n /)$$

👉 tableau est un tableau de **rang 1**,

👉 expr_i est :

⇒ un scalaire,

⇒ une boucle **DO** implicite de la forme
($\text{expr_scalaire}, \text{variable} = m_1, m_2 [, m_3]$) avec **variable** une variable **INTEGER** correspondant à l'indice de cette boucle et m_1, m_2, m_3 des constantes entières délimitant la boucle (voir boucle **DO**).

👉 Le constructeur et le tableau **tableau** doivent être conformants.

Exemple

```

IMPLICIT NONE

REAL, DIMENSION(4)          :: heights =      &
    (/ 5.10, 5.6, 4.0, 3.6 /)

CHARACTER(len=5), DIMENSION(3) :: colours =      &
    (/ 'RED  ', 'GREEN', 'BLUE ' /)

INTEGER                      :: i
INTEGER, DIMENSION(10)      :: ints      =      &
    (/ 100, (i, i=1,8), 100 /)
    
```

6.3.3 – L'instruction DATA

Comme pour les variables simples, on peut utiliser l'instruction **DATA** pour initialiser les tableaux lors de leur déclaration. Elle permet d'initialiser tout ou partie de tableaux à l'aide d'une liste de constantes encadrée par le caractère / (la notation **n*valeur** peut être employée pour répéter une même valeur).

Les éléments à initialiser peuvent être spécifiés au moyen d'une boucle **DO implicite** : (tab(i), i = m₁,m₂[,m₃]).

Exemple

```
IMPLICIT NONE
INTEGER                                :: i, j

REAL,    DIMENSION(20)                :: A, B
INTEGER, DIMENSION(10)                :: ints
REAL,    DIMENSION(2,3)               :: MAT

DATA A/20*7.0/, B/10., 3.87, 10.5/
DATA (ints(i),i=1,4)/4*6.0/, A(10)/-1.0/
DATA MAT/1., 2., 3., 4., 5., 6./
DATA ((MAT(i,j),j=1,3),i=1,2)/1., 2., 3., &
                                     4., 5., 6./
```

6.4 – Manipulation de tableaux

6.4.1 – Expressions de type tableau

Les tableaux peuvent être utilisés en tant qu'opérandes dans une expression :

- ➡ les opérateurs intrinsèques sont applicables à des tableaux conformants,
- ➡ les fonctions élémentaires s'appliquent à l'ensemble du tableau.

Dans ces cas, les fonctions ou opérateurs sont appliqués à chaque élément du tableau (`log`, `sqrt`, ...).

Exemple

```
REAL, DIMENSION(-4:0,0:2) :: B
REAL, DIMENSION(5,3)      :: C
REAL, DIMENSION(0:4,0:2)  :: D
...
B = C * D - B**2
B = SIN(C)+COS(D)
```

6.4.2 – Sections de tableaux

Les **sections régulières** de tableaux sont décrites à l'aide d'un triplet :

$$[\text{limite}_1] : [\text{limite}_2] [: \text{pas}]$$

- ➡ cette notation est équivalente à une pseudo-boucle,
- ➡ une section de tableau est aussi un tableau,
- ➡ le **rang** d'une section de tableau est inférieur ou égal à celui du tableau global,
- ➡ la section démarre à `limite1` et se termine à ou avant `limite2`,
- ➡ `pas` est l'incrément des indices.

Exemples

```
A(:)      ! Le tableau global
A(3:9)    ! A(3) à A(9) par pas de 1
A(3:9:1)  ! Idem
A(m:n)    ! A(m) à A(n)
A(m:n:k)  ! A(m) à A(n) par pas de k
A(8:3:-1) ! A(8) à A(3) par pas de -1
A(8:3)    ! A(8) à A(3), pas de 1 => taille nulle
A(m:)     ! de A(m) à la borne supérieure de A
A(:n)     ! de la borne inférieure de A à A(n)
A(:,2)    ! borne inf. de A à borne sup., pas de 2
A(m:m)    ! section constituée de 1 élément
           ! (ce n'est pas un scalaire !)
A(m)      ! section équivalente à un scalaire
```

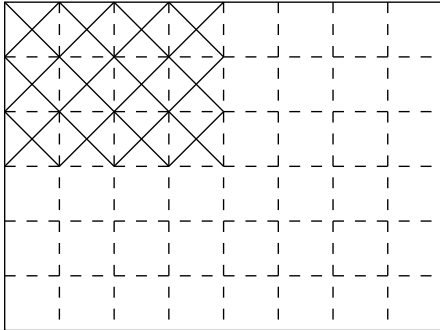
Seules les opérations sur des sections conformantes sont valides :

```
REAL, DIMENSION(1:6,1:8) :: P
```

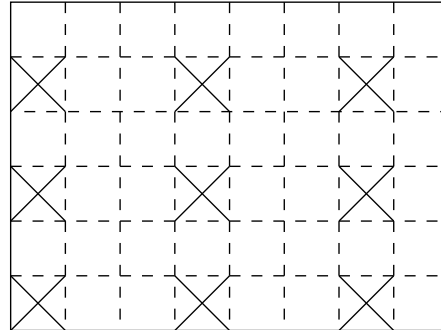
```
P(1:3, 1:4)      = P(1:6:2, 1:8:2) ! VALIDE
P(2:8:2, 1:7:3)  = P(1:3, 1:4)     ! INVALIDE
P(2:6:2, 1:7:3)  = P(2:5, 7)       ! INVALIDE
```

Exemples

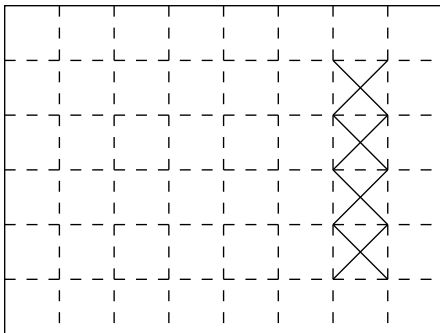
```
REAL, DIMENSION(1:6,1:8) :: P
```



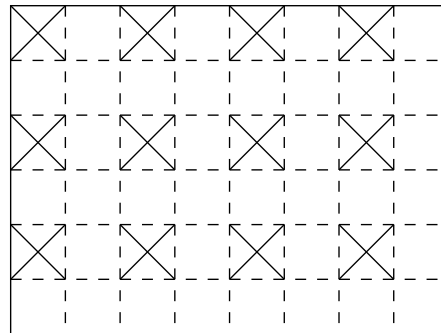
P(1:3,1:4)



P(2:6:2,1:7:3)



P(2:5,7), P(2:5,7:7)



P(1:6:2,1:8:2)

Attention, $P(2:5,7)$ est une section 1D tandis que $P(2:5,7:7)$ est une section 2D : ces 2 tableaux ne sont donc pas conformants.

7 – Entrées-Sorties

- ➡ 7.1 - Introduction
- ➡ 7.2 - Accès séquentiel
 - ➡ 7.2.1 - Fichier binaire séquentiel
 - ➡ 7.2.2 - Fichier texte séquentiel
 - ➡ 7.2.2.1 - Format d'édition
 - ➡ 7.2.2.2 - *namelist*
- ➡ 7.3 - Accès direct
 - ➡ 7.3.1 - Fichier binaire à accès direct
 - ➡ 7.3.2 - Fichier texte à accès direct
- ➡ 7.4 - Fichier temporaire
- ➡ 7.5 - Destruction d'un fichier
- ➡ 7.6 - Fichier interne
- ➡ 7.7 - Instructions de positionnement
- ➡ 7.8 - L'instruction INQUIRE
- ➡ 7.9 - Remarques
- ➡ 7.10 - Syntaxes

7.1 – Introduction

On appelle *entrée-sortie*, un transfert d'informations entre la mémoire de l'ordinateur et l'un de ses périphériques (un disque le plus souvent).

Une *entrée* se traduit par une *lecture* d'informations du périphérique vers la mémoire, tandis qu'une *sortie* implique une écriture de la mémoire vers le périphérique.

Ces informations sont stockées dans un fichier qui possède un nom.

L'unité de transmission entre la mémoire et le périphérique s'appelle le **bloc**. Il permet d'effectuer le traitement en passant par une zone intermédiaire dite *zone tampon* (*buffer*) permettant ainsi de limiter le nombre de transferts entre la mémoire et le périphérique : opération coûteuse.

L'unité de traitement est l'enregistrement logique : il correspond à la longueur des données traitées lors d'une opération de lecture-écriture.

L'exploitation d'un fichier au sein d'un programme nécessite au préalable son ouverture qui, en Fortran, est faite au moyen de l'instruction **OPEN**.

Cette instruction permet notamment :

- ➡ de connecter le fichier à un numéro d'*unité logique* : c'est celui-ci que l'on indiquera par la suite pour toute opération de lecture-écriture,
- ➡ de spécifier le mode désiré : lecture, écriture ou lecture-écriture,
- ➡ d'indiquer le mode de transfert : avec ou sans conversion en caractères,
- ➡ d'indiquer le mode d'accès au fichier : séquentiel ou direct.

Si l'ouverture du fichier est fructueuse, des lectures-écritures pourront être lancées à l'aide des instructions **READ/WRITE** par l'intermédiaire du numéro d'*unité logique*.

Une fois le traitement du fichier terminé, on le fermera au moyen de l'instruction **CLOSE**.

7.2 – Accès séquentiel

On dit qu'un fichier est séquentiel lorsqu'il est nécessaire d'avoir traité les enregistrements précédant celui auquel on désire accéder.

Pour un fichier en lecture le paramètre IOSTAT de l'instruction READ permet de gérer la fin de fichier ; celui-ci fait référence à une variable entière qui est valorisée à l'issue de la lecture comme suit :

- ➡ à 0 si la lecture s'est bien déroulée,
- ➡ à une valeur positive si une erreur s'est produite,
- ➡ à une valeur négative si la fin de fichier ou une fin d'enregistrement à été rencontrée.

On prendra soin de tester la valeur de cette variable **immédiatement après chaque lecture.**

7.2.1 – Fichier binaire séquentiel

On appelle fichier *binaire* un fichier dans lequel on stocke les informations telles qu'elles sont représentées en mémoire.

C'est au moment de l'ouverture du fichier que l'on indique le type de fichier à traiter.

```
real, dimension(100) :: tab
integer                :: i
real                  :: r
integer                :: ios

OPEN( UNIT=1,          &
      FILE="data_bin_seq", &
      FORM="unformatted", &
      ACCESS="sequential", &
      ACTION="read",      &
      POSITION="rewind",   &
      IOSTAT=ios )
if ( ios /= 0 ) stop "Problème à l'ouverture"
READ( UNIT=1, IOSTAT=ios ) tab, i, r
do while ( ios == 0 )
    ...
    READ( UNIT=1, IOSTAT=ios ) tab, i, r
end do
CLOSE( UNIT=1 )
```

On demande l'ouverture du fichier dont le nom est `data_bin_seq`. C'est un fichier binaire séquentiel (`unformatted, sequential`) que l'on désire lire depuis le début (`rewind`).

Ce fichier est connecté à l'unité logique dont le numéro est 1. C'est ce numéro que l'on indique au moment de la lecture des variables `tab`, `i`, `r`, ainsi qu'à la fermeture du fichier.

L'entier `ios` contient le code retour de l'`OPEN` : il est nul si tout s'est bien passé, non nul sinon. Il est vivement conseillé de le tester avant toute opération de lecture ou d'écriture.

Ce même entier est utilisé au sein de l'instruction `READ` afin de connaître l'état de la lecture.

7.2.2 – Fichier texte séquentiel

Dans un fichier texte les données sont stockées sous forme de caractères. De ce fait :

- ➡ lors d'une lecture, elles sont converties en binaire avant d'être rangées en mémoire,
- ➡ lors d'une écriture, elles sont converties en caractères avant d'être écrites dans le fichier.

Cette opération de conversion est signalée au sein des instructions READ/WRITE :

- ➡ à l'aide d'une chaîne de caractères appelée **format d'édition** (paramètre FMT=),
- ➡ ou bien en utilisant un nom de liste (NAMELIST) regroupant les variables que l'on désire exploiter (paramètre NML=).

Formats d'édition

Pour que la conversion puisse être faite, il est nécessaire de connaître le type de la donnée à convertir.

Pour cela le **format d'édition** contient des **descripteurs** :

- ☞ descripteur I pour le type INTEGER,
- ☞ descripteurs F, E pour le type REAL,
- ☞ descripteur L pour le type LOGICAL,
- ☞ descripteur A pour le type CHARACTER.

Exemple

```

PROGRAM texte_sequentiel
  real, dimension(10) :: tab
  integer              :: i
  real                 :: r
  integer              :: ios

  OPEN( UNIT=1,          &
        FILE="data_txt_seq", &
        FORM="formatted",  &
        ACCESS="sequential", &
        STATUS="old",      &
        ACTION="write",    &
        POSITION="rewind",  &
        IOSTAT=ios )
  if ( ios /= 0 ) then ! Problème à l'ouverture
    ...
  else
    WRITE( UNIT=1, &
           FMT='(10F8.4,I3,F6.3)') tab, i, r
    ...
  endif
  ...
  CLOSE( UNIT=1 )
END PROGRAM texte_sequentiel

```

Dans cet exemple, on demande l'ouverture du fichier dont le nom est `data_txt_seq`. C'est un fichier texte séquentiel (`formatted, sequential`) existant (`old`) que l'on désire écraser (`rewind`).

Comme précédemment, à l'issue de l'`OPEN` on teste la valeur de retour contenue dans l'entier `ios`.

Si l'ouverture s'est bien passée on lance, à l'aide de l'instruction `WRITE`, l'écriture en caractères d'un enregistrement comportant un tableau de réels (`tab`) suivi d'un entier puis d'un réel (`i, r`).

Le format d'édition spécifié sous la forme d'une constante chaîne de caractères (`'(10F8.4,I3,F6.3)'`) permet de convertir en caractères les variables ci-dessus :

- 👉 `10F8.4` : écriture des 10 éléments du tableau `tab`. Chacun a un gabarit de 8 caractères avec 4 chiffres en partie décimale,
- 👉 `I3` : écriture de l'entier `i` sur 3 caractères,
- 👉 `F6.3` : écriture du réel `r` sur 6 caractères avec 3 chiffres en partie décimale.

Formats d'édition en lecture

- ➡ Iw permet la conversion des w caractères suivants dans le type INTEGER,
- ➡ $Fw.d$: permet la conversion des w caractères suivants dans le type REAL. Si le point décimal n'est pas présent alors les d derniers caractères désignent la partie fractionnaire,
- ➡ $Ew.d$: permet la conversion des w caractères suivants (interprétés comme un nombre réel en notation exponentielle) dans le type REAL,
- ➡ Lw : permet la conversion des w caractères suivants dans le type LOGICAL,
- ➡ $A[w]$: permet de lire des caractères.

Exemples

Dans ce qui suit le caractère ^ représente l'espace. Les différentes entrées présentées sont supposées figurer dans un fichier texte séquentiel connecté à l'unité 1 après un OPEN.

Format d'édition I en lecture

```
INTEGER i, j
...
READ( UNIT=1, FMT='(I5,I4)' ) i, j
...
```

Entrées	Affectations
^^45^^^9^	i=45, j=9
^-24^10^^	i=-24, j=10

À noter : dans un champ en entrée, l'espace est ignoré.

Formats d'édition F, E et D en lecture

Ce format à la forme générale : $Fw.d$, $Ew.d$ ou bien $Dw.d$.

Le nombre réel à lire peut être soit en notation virgule fixe, soit exponentielle avec, dans ce dernier cas, l'exposant préfixé de la lettre E ou D .

Le point décimal peut ne pas figurer :

- ☞ s'il est spécifié alors le nombre indiqué par d est ignoré, c'est le nombre de décimales figurant en entrée qui est pris en compte,
- ☞ s'il est omis c'est le nombre indiqué par d qui est considéré.

```
REAL x, y
...
READ( UNIT=1, FMT='(F4.1,F6.2)' ) x, y
...
```

Entrées	Affectations
$^3.1-3.141$	$x=3.1, y=-3.141$
$^123^^5678$	$x=12.3, y=56.78$

```
REAL x
...
READ( UNIT=1, FMT='(E12.6)' ) x
! ou bien
READ( UNIT=1, FMT='(F12.6)' ) x
...
```

Entrées	Affectations
2.718281 ^{^^^}	x=2.718281
2718281 ^{^^^^}	x=2.718281
27.18281d-1 [^]	x=2.718281
.2718281e+1 [^]	x=2.718281
.2718281 [^] e [^] 1	x=2.718281
^{^^^^} 2718281	x=2.718281
^{^^^^^^} 271	x=0.000271

Format d'édition L en lecture

Ce format à la forme générale : Lw

Ce type de format permet la lecture de valeurs logiques.

Le caractère w indique comme précédemment la largeur du champ en entrée. Celui-ci doit comporter comme premier caractère non blanc l'une des lettres F, f, T ou t éventuellement précédée du caractère '.'. N'importe quels caractères peuvent compléter le champ.

```
LOGICAL 11, 12
...
READ( UNIT=1, FMT='(L6,L7)' ) 11, 12
...
```

Entrées	Affectations
.true..false.	11=.TRUE., 12=.FALSE.
^^^^t..FACILE	11=.TRUE., 12=.FALSE.
t^^^^^^F^^^^^^	11=.TRUE., 12=.FALSE.

Format d'édition A en lecture

Ce format à la forme générale : $A[w]$

Ce type de format permet la lecture de caractères. La valeur w indique le nombre de caractères que l'on désire traiter en entrée.

- ➡ si la valeur de w est plus grande que la longueur l de la chaîne réceptrice, ce sont les l caractères les plus à droite qui seront lus,
- ➡ si elle est plus petite, w caractères seront lus et stockées dans la chaîne réceptrice qui sera complétée à droite par des blancs,
- ➡ si elle n'est pas précisée, c'est la longueur spécifiée lors de la déclaration de la chaîne de caractères qui indiquera le nombre de caractères à lire. Si la fin de l'enregistrement est atteinte avant la fin de la lecture, la chaîne est complétée par des blancs.

Exemples

```
CHARACTER(len=7) :: ch1, ch2
...
READ( UNIT=1, FMT='(A6,A8)' ) ch1, ch2
READ( UNIT=1, FMT='(A6,A8)' ) ch1, ch2
READ( UNIT=1, FMT='(A,A)' ) ch1, ch2
...
```

Entrées	Affectations
BACH^^^^BACH^^	ch1="BACH^^^^", ch2="^BACH^^"
MOZARTHAENDEL^	ch1="MOZART^", ch2="AENDEL^"
MOZARTHAENDEL^	ch1="MOZARTH", ch2="AENDEL^"

Formats d'édition en écriture

- ➡ $Iw[.d]$ permet l'édition d'une variable de type INTEGER sur w caractères. S'il est présent d indique le nombre minimum de chiffres édités : si nécessaire des 0 apparaîtront en tête du nombre,
- ➡ $Fw.d$: permet l'édition d'une variable de type REAL sur w caractères comprenant le point décimal suivi de d chiffres pour la partie fractionnaire,
- ➡ $Ew.d$: idem format F mais la sortie est faite en notation exponentielle,
- ➡ Lw : permet l'édition d'une variable de type LOGICAL sur w caractères,
- ➡ $A[w]$: permet l'édition d'une variable de type CHARACTER.

Format d'édition I en écriture

```
INTEGER i, j, k
```

```
i = -125
```

```
j = 1756
```

```
k = 1791
```

```
WRITE( UNIT=1, FMT='(I4,I4,I4)' ) i, j, k
```

```
WRITE( UNIT=1, FMT='(I5,I6,I6)' ) i, j, k
```

```
WRITE( UNIT=1, FMT='(I6.5,I6.6,I6.6)' ) i, j, k
```

Sorties
-12517561791
^-125^^1756^^1791
-00125001756001791

Format d'édition F en écriture

```
REAL x, y, z
x = 3.14159
y = -15.137
z = 799.7432
WRITE( UNIT=1, &
      FMT='(F7.5,F8.3,F9.4)' ) x, y, z
WRITE( UNIT=1, &
      FMT='(F6.2,F9.4,F10.5)' ) x, y, z
```

Sorties
3.14159^-15.137^799.7432
^^3.14^-15.1370^799.74323

Remarque

En ce qui concerne les formats^a d'écriture I, B, O, Z et F, lorsque le gabarit de la zone réceptrice est insuffisant celle-ci est remplie par le caractère *. Depuis la norme Fortran 95 il est possible d'éviter cela en indiquant 0 comme largeur de zone.

Exemples

```
PROGRAM gabarit
  INTEGER I
  REAL    R

  I = 129876
  R = -2345.78

  WRITE( UNIT=1, FMT='(I4, F6.3)' ) I, R
  WRITE( UNIT=1, FMT='(IO, F0.3)' ) I, R
END PROGRAM gabarit
```

Sorties

129876-2345.780

^aformats B, O, Z : c.f. Remarques en fin de chapitre.

Format d'édition E et D en écriture

Avec les format $Ew.d$, $Dw.d$ on obtiendra en sortie le motif :

```
SO.XXXXXXXESXX, SO.XXXXXXXDSXX
  <--d-->      <--d-->
<-----w-----> <-----w----->
```

Le caractère S indique une position pour le signe.

Un facteur d'échelle peut précéder ce type de format. Il s'écrit sous la forme kP et permet d'éditer le nombre avec k chiffres avant le point décimal (modifie en conséquence la valeur de l'exposant).

Si $-d < k \leq 0$, la partie décimale sera constituée de $|k|$ zéros suivis de $d - |k|$ chiffres significatifs.

Si $0 < k < d + 2$, le nombre en sortie sera constitué de k chiffres significatifs avant le point décimal et de $d - k + 1$ chiffres significatifs en partie décimale.

Toute autre valeur de k est invalide : dans ce cas, la zone en sortie sera remplie par le caractère *.

Celui-ci s'applique à tous les formats E qui suivent. Pour retrouver le comportement par défaut il suffit de préciser le facteur 0P.

Exemples

```

REAL x, y, z
DOUBLE PRECISION xd, yd, zd
x = 3.14159;    xd = 3.14159d0
y = -15.137;   yd = -15.137d0
z = 799.74328; zd = 799.74328d0
WRITE( UNIT=1, &
       FMT='(D12.6,E13.5,E15.8)' ) x, y, z
WRITE( UNIT=1, &
       FMT='(4P,D12.6,E13.5,0P,E10.3)' ) x, y, z
WRITE( UNIT=1, &
       FMT='(D12.6,E13.5,E15.8)' ) xd, yd, zd
WRITE( UNIT=1, &
       FMT='(4P,D12.6,E13.5,0P,E15.8)' ) xd, yd, zd
    
```

Sorties

0.314159D+01^-0.15137E+02^0.79974329E+03

3141.590D-03^-1513.70E-02^0.800E+03

0.314159D+01^-0.15137E+02^0.79974328E+03

3141.590D-03^-1513.70E-02^0.79974328E+03

Il existe deux variantes du format E qui correspondent aux descripteurs EN et ES. Le facteur d'échelle kP ne peut être appliqué dans ce contexte.

Le descripteur EN permet la sortie d'un réel en notation dite *ingénieure*. Dans ce type de représentation, la valeur absolue de la partie entière est supérieure ou égale à 1 et inférieure à 1000 et l'exposant est divisible par 3.

```
REAL x, y, z, t
x = 6.421
y = -.5
z = .00217
t = 4721.3
WRITE( UNIT=1, &
      FMT='(EN12.3,EN12.3,EN12.3,EN12.3)' ) x, y, z, t
```

Sorties

```
^^^6.421E+00-500.000E-03^^^2.170E-03^^^4.721E+03
```

Le descripteur **ES** permet la sortie d'un réel en notation dite *scientifique*. Dans ce type de représentation, la valeur absolue de la partie entière est supérieure ou égale à 1 et inférieure à 10.

```
REAL x, y, z, t
x = 6.421
y = -.5
z = .00217
t = 4721.3
WRITE( UNIT=1, &
      FMT='(ES12.3,ES12.3,ES12.3,ES12.3)' ) x, y, z, t
```

Sorties

```
^^^6.421E+00^^-5.000E-01^^^2.170E-03^^^4.721E+03
```

Par défaut, la présence du signe + pour les données numériques positives dépend du compilateur. Il existe des descripteurs permettant de forcer sa présence ou bien de le supprimer :

- 👉 SS (*sign suppress*) : le signe + n'apparaît pas,
- 👉 SP (*sign print*) : le signe + précède la donnée,
- 👉 S : restauration du mode par défaut.

```
INTEGER i, j
REAL x, y
i = 1756
j = -69
x = 6.421
y = .00217
WRITE( UNIT=1, &
      FMT='(SP,F7.3,SS,ES12.3,I4,SP,I6)' ) x, y, j, i
```

Sorties

```
^+6.421^^^2.170E-03^-69^+1756
```

Format d'édition L en écriture

Ce format à la forme générale : Lw

Ce type de format permet l'écriture de valeurs logiques.

En sortie on obtiendra $w-1$ blancs suivis de la lettre T pour une valeur `.true.` et F pour une valeur `.false.`.

```
LOGICAL l1/.true./, l2/.false./  
...  
WRITE( UNIT=1, FMT='(L6,L7)' ) l1, l2  
...
```

Sorties
^^^^T^^^^F

Format d'édition A en écriture

Le format $A[w]$ permet la sortie de chaînes de caractères. La valeur w est facultative. Si elle est précisée, elle indique la largeur de la zone réceptrice.

- ➡ si la valeur de w est plus grande que la longueur 1 de la chaîne, en sortie celle-ci apparaîtra précédée de $w-1$ blancs,
- ➡ si elle est plus petite, seuls les w caractères les plus à gauche de la chaîne seront écrits,
- ➡ si la valeur w est absente, c'est la longueur de la chaîne spécifiée à la déclaration qui indique la largeur du champ en sortie.

Exemples

```

CHARACTER(len=9) :: ch1, ch2, ch3
...
ch1 = "BEETHOVEN"
ch2 = "PUCCINI"
ch3 = "VERDI"
WRITE( UNIT=1, FMT='(A9,A8,A6,A)' ) ch1, &
                                     ch2, &
                                     ch3, &
                                     ch3

WRITE( UNIT=1, FMT='(A10)' ) ch3
ch1 = "Ludwig"
ch2 = " Van"
ch3 = "BEETHOVEN"
WRITE( UNIT=1, FMT='(A,A,A,A)' ) trim(ch1), &
                                     trim(ch2), &
                                     ' ', &
                                     ch3
...

```

Sorties

BEETHOVENPUCCINI^VERDI^VERDI^^^^

^VERDI^^^^

Ludwig^Van^BEETHOVEN

Format d'édition : *Literal string*

Si une constante littérale de type chaîne de caractères est spécifiée dans un format, celle-ci est reproduite telle quelle en sortie.

```
CHARACTER(len=9) :: ch1, ch2, ch3
...
ch1 = "Ludwig"
ch2 = " Van"
ch3 = "BEETHOVEN"
WRITE( UNIT=1, &
      FMT='("NOM : ",A," PRÉNOM : ",A,A)' ) &
      ch3, trim(ch1), trim(ch2)
...
```

Sortie

NOM^: ^BEETHOVEN, ^PRÉNOM^: ^Ludwig^Van

Descripteurs de contrôle

☞ descripteurs de positionnement :

- ⇒ nX : ignore (en entrée), saute (en sortie) les n caractères suivants,
- ⇒ Tc : permet de se positionner au caractère de rang c ,
- ⇒ TLn : permet de se positionner au caractère situé n positions à gauche par rapport à la position courante,
- ⇒ TRn : permet de se positionner au caractère situé n positions à droite par rapport à la position courante.

☞ descripteurs de gestion des blancs dans les champs numériques en entrée :

- ⇒ BN (*Blank Null*) : ignore les blancs,
- ⇒ BZ (*Blank Zero*) : interprète le caractère blanc comme un 0.

```

INTEGER i, j, k, l
...
READ( UNIT=1, &
      FMT='(I4,3X,I2,T12,I3,TL4,I1)' ) i, j, k, l
...
    
```

Entrées	Affectations
1901^1254^4361	i=1901,j=54,k=361,l=4

```

INTEGER i, j, k
...
READ( UNIT=1, &
      FMT='(I3,BZ,I2,BN,I3)' ) i, j, k
...
    
```

Entrées	Affectations
^8^2^4^^	i=8,j=20,k=4

Un autre descripteur, /, provoque le passage à l'enregistrement suivant :

☞ en entrée : abandon de l'enregistrement courant et positionnement au début du suivant,

```
INTEGER i, j
READ( UNIT=1, FMT='(I4,/,I4)') i, j
```

Entrées	Affectations
1756^1254 1791	i=1756,j=1791

☞ en sortie : écriture du caractère *newline*.

```
CHARACTER(len=9) :: ch1, ch2, ch3
ch1 = "Ludwig"
ch2 = " Van"
ch3 = "BEETHOVEN"
WRITE( UNIT=1, &
      FMT='("NOM      : ",A,/, "PRÉNOM : ",A,A)' ) &
ch3, trim(ch1), trim(ch2)
```

Sortie
NOM^^^^: ^BEETHOVEN
PRÉNOM^: ^Ludwig^Van

Facteur de répétition

Lorsqu'une liste de descripteurs identiques figure dans un format il est possible de les regrouper au moyen d'un facteur de répétition indiqué sous la forme d'une constante littérale entière.

```

INTEGER i, j, k
INTEGER t(3)
...
READ( UNIT=1, FMT='(I4,I4,I4)' ) i, j, k
READ( UNIT=1, FMT='(3I4)' ) t
...
WRITE(6, '(3(1X,I4))') t

```

Entrées	Affectations
^^45^^^9^^10	i=45, j=9, k=10
^-24^10^^^99	t(1)=-24, t(2)=10, t(3)=99

Sortie

```
^^-24^^^10^^^99
```

Réexploration d'un format

L'ensemble des variables spécifiées dans une instruction READ/WRITE s'appelle la liste d'entrée-sortie. Chaque élément de cette liste est associé à un descripteur du format.

- ➡ si le nombre de descripteurs correspond au nombre de variables de la liste, ceux-ci s'appliquent aux éléments successifs de cette liste,
- ➡ s'il est plus grand, les suivants sont ignorés (le format est abandonné),
- ➡ s'il est plus petit, le format est réexploré. Cette réexploration entraîne le passage à l'enregistrement suivant.

Conclusion : en Fortran la liste d'entrée-sortie est toujours satisfaite.

La règle de réexploration est la suivante :

- ☞ si le format ne contient aucun groupe de descripteurs entre parenthèses, alors il est réexploré depuis son début,
- ☞ sinon, la réexploration est faite à partir du groupe de niveau 1 le plus à droite. S'il est précédé d'un facteur de répétition, il est pris en compte.

Exemples

Le caractère | indique l'endroit à partir duquel la réexploration est effectuée.

```
'( I6, 10X,I5, 3F10.2 )'
```

|-----

```
'( I6, 10X,I5, (3F10.2) )'
```

|-----

```
'( I6,(10X,I5), 3F10.2 )'
```

|-----

```
'( F6.2, (2F4.1,2X,I4, 4(I7,F7.2)) )'
```

|-----

```
'( F6.2, 2(2F4.1,2X,I4), 4(I7,F7.2) )'
```

|-----

```
'( F6.2,(2(2F4.1,2X,I4), 4(I7,F7.2)) )'
```

|-----

Exemples

```

INTEGER i/100/, &
        j/200/, &
        k/300/, &
        l/400/
REAL    t(3,4)
DATA    t/  1.,   4.,   9., &
        16.,  25.,  36., &
        49.,  64.,  81., &
        100., 121., 144. /
WRITE(6, '( 4i5, (t25,4f8.2) )') &
        i, j, k, l,
        ((t(i,j),j=1,4),i=1,3)
    
```

Sorties

```

~~100~~200~~300~~400~~~~~1.00~~~16.00~~~49.00~~100.00
~~~~~4.00~~~25.00~~~64.00~~121.00
~~~~~9.00~~~36.00~~~81.00~~144.00
    
```

Format libre

En Fortran il existe un format implicite appelé **format libre** (*list-directed formatting*).

Dans l'instruction READ/WRITE, on spécifie alors le caractère * à la place du format.

Dans ce contexte, les enregistrements sont interprétés comme une suite de valeurs séparées par des caractères appelés **séparateurs**. C'est le type des variables auxquelles ces valeurs vont être affectées qui détermine la conversion à effectuer.

Les caractères interprétés comme des séparateurs sont :

- 👉 la virgule (,),
- 👉 le blanc (espace).

Une chaîne de caractères contenant un caractère séparateur doit être délimitée soit par des quotes (') soit par des guillemets (").

En entrée, plusieurs valeurs identiques peuvent être regroupées à l'aide d'un facteur de répétition sous la forme **n*valeur**.

Exemples

```

INTEGER i/100/, &
        j/200/, &
        k/300/
REAL    t(3)
COMPLEX c
CHARACTER(len=8) ch
...
READ ( UNIT=1, FMT=* ) i, j, k, t, c, ch
WRITE( UNIT=2, FMT=* ) i, j, k, c, ch
WRITE( UNIT=2, FMT=* ) t
    
```

Entrées	Affectations
150 260,, 3*2.718 (^2. ,^3.^)^^^'Wolfgang Amadeus Mozart'	i=150 j=260 t=(/ 2.718,2.718,2.718 /) c=(2.,3.) ch='Wolfgang'

Sorties
^150^260^300^(2.000000000,3.000000000)^Wolfgang
^2.717999935^2.717999935^2.717999935

namelist

On peut regrouper les variables que l'on désire lire ou écrire au sein d'une liste à laquelle on donne un nom.

Ce regroupement s'effectue au moyen de l'instruction **NAMelist** :

```
NAMelist/nom_liste/liste_variables
```

- 👉 `nom_liste` est le nom de la **NAMelist**,
- 👉 `liste_variables` est une liste de variables précédemment déclarées.

Au niveau de l'instruction **READ/WRITE** la *namelist* remplace le format ainsi que la liste de variables qui, dans ce contexte, devient inutile.

l'enregistrement correspondant (lu ou écrit) respecte le format suivant :

```
&nom_liste liste d'affectations /
```

La liste d'affectations concerne les variables de la *namelist* qui peuvent apparaître dans n'importe quel ordre, certaines pouvant être omises. Les différentes affectations sont séparées par des caractères séparateurs (, ou blancs).

Le caractère / indique la fin de la *namelist*. En entrée, les éventuels caractères qui suivent sont ignorés.

En entrée les chaînes de caractères doivent être délimitées à l'aide du caractère ' ou ".

En sortie celles-ci apparaissent par défaut sans délimiteur. C'est le paramètre DELIM= de l'instruction OPEN qui permet de le définir.

Dans l'instruction READ/WRITE, la *namelist* est indiquée à l'aide du paramètre NML= (à la place de FMT=).

L'utilisation des *namelist* est un moyen très commode d'effectuer des entrées-sorties sans avoir à définir de format.

Exemples

```
INTEGER i/100/, j/200/, k/300/
INTEGER t(3)
CHARACTER(len=11) ch
NAMELIST/LISTE/i,j,k,t,ch
...
READ ( UNIT=1, NML=liste )
WRITE( UNIT=2, NML=liste )
...
```

Entrées

```
&LISTE t=3*2,i=1, k=4 ch="Rythm&Blues" /
```

Affectations

```
i=1,k=4,t(1)=t(2)=t(3)=2,ch="Rythm&Blues"
```

Sorties

```
^&LISTE
```

```
^I=1,^J=200,^K=4,^T=2,^2,^2,^CH=Rythm&Blues
```

```
^/
```

7.3 – Accès direct

À la différence d'un fichier séquentiel, il est possible d'accéder à un enregistrement d'un fichier à accès direct sans avoir traité les précédents.

Chaque enregistrement est repéré par un numéro qui est son rang dans le fichier. Leur taille est fixe.

Au sein de l'instruction OPEN :

- ➡ le paramètre RECL= est obligatoire, sa valeur indique la taille des enregistrements (en caractères pour les fichiers textes, dépend du processeur pour les fichiers binaires),
- ➡ le paramètre POSITION= est invalide,
- ➡ si le paramètre FORM n'est pas précisé, c'est la valeur *unformatted* qui est prise en compte.

Le rang de l'enregistrement que l'on désire traiter doit être spécifié à l'aide du paramètre REC= de l'instruction READ/WRITE. Un enregistrement ne peut pas être détruit mais par contre il peut être réécrit. Dans ce contexte, les *namelist* ainsi que le format libre sont interdits.

7.3.1 – Fichier binaire à accès direct

```
real, dimension(100) :: tab
integer ios, n
OPEN( UNIT=1,                                &
      FILE="data_bin_direct", &
      ACCESS="direct",        &
      ACTION="read",          &
      STATUS="old",           &
      RECL=400,               &
      IOSTAT=ios )
if ( ios /= 0 ) then ! Problème à l'ouverture
  ...
else
  OPEN( UNIT=2, FILE="data_txt_seq", &
        ACTION="read", STATUS="old", &
        IOSTAT=ios )
  ...
  READ( UNIT=2, FMT=* ) n
  READ( UNIT=1, REC=n, IOSTAT=ios ) tab
  if ( ios > 0 ) stop "erreur à la lecture."
  ...
  CLOSE( UNIT=2 )
end if
CLOSE( UNIT=1 )
```

Le fichier dont le nom est `data_bin_direct` est connecté à l'unité logique numéro 1. C'est un fichier binaire à accès direct (`ACCESS="direct"` et paramètre `FORM` absent donc considéré égal à *unformatted*). Chaque enregistrement fait 400 octets (`RECL=400`).

On accède à l'enregistrement de rang `n`, valeur préalablement lue dans le fichier texte séquentiel de nom `data_txt_seq` connecté à l'unité logique numéro 2. Le paramètre `IOSTAT` de l'instruction `READ` permet de récupérer l'état de la lecture dans l'entier `ios` : une valeur non nulle positive signale une erreur du type enregistrement inexistant par exemple.

7.3.2 – Fichier texte à accès direct

```

real, dimension(100) :: tab
integer ios, n

OPEN( UNIT=1,                                &
      FILE="data_txt_direct", &
      FORM="formatted"           &
      ACCESS="direct",           &
      ACTION="read",             &
      STATUS="old",              &
      RECL=800,                  &
      IOSTAT=ios )
if ( ios /= 0 ) stop "Problème à l'ouverture!"
OPEN( UNIT=2,                                &
      FILE="data_txt_seq", &
      ACTION="read",           &
      STATUS="old",            &
      IOSTAT=ios )
...
READ( UNIT=2, FMT=* ) n
READ( UNIT=1, REC=n, &
      IOSTAT=ios, FMT='(100F8.4)' ) tab
if ( ios > 0 ) stop "erreur à la lecture."
...
CLOSE( UNIT=2 )
CLOSE( UNIT=1 )

```

On a adapté l'exemple précédent à un fichier texte à accès direct : le paramètre `FORM="formatted"` a donc été précisé.

La valeur du paramètre `RECL` correspond à la taille en caractères de chacun des enregistrements qui correspond ici au format indiqué au niveau de l'instruction `READ` ($100 \times 8 = 800$).

Notes :

- ✎ il n'est pas permis de lire un fichier texte à accès direct au moyen d'un format libre,
- ✎ un fichier dans lequel on écrit un seul enregistrement de rang `n`, contiendra en réalité `n` enregistrements avec les `n-1` premiers ayant un contenu indéterminé. Lors de la lecture d'un tel fichier, se pose le problème de la reconnaissance de la nature de celui-ci. On pourra résoudre ce problème en adjoignant aux données une marque sous la forme d'une variable logique par exemple.

Dans l'exemple suivant des enregistrements de rang aléatoire sont créés. Pour les repérer on ajoute en tête des données une variable logique *flag*.

```
program dir
  implicit none
  integer, parameter    :: n = 100, m = 1000
  real, dimension(n,m) :: matrice
  real, dimension(n)    :: vec
  integer i,j, rang, ios, size
  logical flag

  inquire( iolength=size ) flag, matrice(:,1)
  open( unit=1,      &
        file="direct.file", &
        form="unformatted", &
        access="direct", &
        action="readwrite", &
        status="unknown", &
        recl=size, &
        iostat=ios )
  if ( ios /= 0 ) stop "Erreur à l'ouverture"

  flag = .false.
  do j=1,m
    write( unit=1, rec=j ) flag
  end do
```

```

call random_number( matrice )
flag = .true.
do j=1,m
  if ( matrice(1,j) > matrice(n,j) ) &
    write( unit=1, rec=j ) flag, matrice(:,j)
end do

do
  print *, "Entrez un rang : "
  read( *, *, iostat=ios ) rang
  if ( ios < 0 ) exit
  if ( ios > 0 ) then
    print *, "Erreur de saisie. Veuillez recommencer."
    cycle
  end if
  read( unit=1, rec=rang, iostat=ios ) flag, vec
  if ( ios > 0 ) then
    print *, "Erreur lors de la lecture."; cycle
  end if
  if ( flag ) then
    print *, "vec(1) = ", vec(1), "vec(n) = ", vec(n)
  else
    print *, "Enregistrement ", rang, " indéfini."
  end if
end do
close(1)
end program dir

```

Notes :

- 👉 l'instruction `inquire( iolength=size ) ...` permet de récupérer dans l'entier *size* la taille de la liste de variables fournies qui servira à renseigner, par la suite, le paramètre RECL de l'instruction OPEN. Comme cette valeur est exprimée dans une unité dépendant du compilateur (pour les fichiers binaires en accès direct), cette nouvelle forme de l'instruction `inquirea`, introduite par la norme Fortran 90, permet de la calculer en s'affranchissant de l'unité,
- 👉 l'instruction `call random_number( matrice )` fait appel à la procédure intrinsèque Fortran 90 RANDOM_NUMBER qui valorise le tableau transmis à l'aide de nombres aléatoires générés dans l'intervalle [0., 1.].

^acette instruction est détaillée plus loin.

7.4 – Fichier temporaire

Si à l'ouverture d'un fichier on positionne le mot-clé **STATUS** à la valeur "**scratch**" alors celui-ci sera temporaire et détruit à sa fermeture. Un tel fichier est anonyme : le paramètre **FILE** de l'instruction **OPEN** ne doit donc pas être spécifié dans ce contexte.

```
real, dimension(100) :: tab
integer ios

OPEN( UNIT=1,                                &
      FORM="formatted"                        &
      ACCESS="direct",                       &
      ACTION="write",                        &
      STATUS="scratch",                      &
      RECL=1200,                             &
      IOSTAT=ios )
if ( ios /= 0 ) then ! Problème à l'ouverture
  ...
end if
...
CLOSE( UNIT=1 )
```

7.5 – Destruction d'un fichier

L'instruction `CLOSE` admet le paramètre à mot-clé `STATUS` qui par défaut (pour un fichier non temporaire) prend la valeur `KEEP` permettant ainsi de conserver le fichier après fermeture. Si on désire le supprimer il suffira de préciser la valeur `DELETE` pour ce paramètre.

```
real, dimension(100) :: tab
integer ios
OPEN( UNIT=1,                                &
      FILE="data_seq",                        &
      ACTION="read",                          &
      STATUS="old",                           &
      IOSTAT=ios )
if ( ios /= 0 ) then ! Problème à l'ouverture
  ...
end if
...
if ( condition ) then
  CLOSE( UNIT=1, STATUS="delete" )
else
  CLOSE( UNIT=1 )
end if
```

7.6 – Fichier interne

On appelle fichier interne un fichier dont les enregistrements sont en mémoire. Ce type de fichier induit des échanges entre zones de la mémoire et non plus entre un support externe et la mémoire.

Ces fichiers sont pré-connectés : il n'y a donc aucune ouverture ni fermeture à effectuer.

Dans les instructions READ/WRITE, à la place du numéro d'unité logique on indique une variable de type chaîne de caractères. C'est celle-ci qui fait référence à l'enregistrement en mémoire.

Seul l'accès séquentiel formaté est permis dans ce cas. Les *namelist* sont interdites.

Lors d'une écriture il faut s'assurer que la chaîne de caractères réceptrice est de taille suffisante.

Lors d'une lecture, la fin de fichier est atteinte lorsqu'on essaie d'accéder aux caractères situés au-delà de la chaîne qui fait référence à l'enregistrement.

Exemples

```
INTEGER, PARAMETER      :: n = 4, m = 6
REAL, DIMENSION(n,m)    :: tab
CHARACTER(LEN=8)         :: fmt = "( F8.3)"
INTEGER                  :: i, j, ios

WRITE( fmt(2:3), '(I2)' ) n ! fichier interne
OPEN( UNIT=1,             &
      FILE="data_txt_seq", &
      POSITION="rewind",    &
      ACTION="write",      &
      STATUS="new",        &
      IOSTAT=ios )
if ( ios /= 0 ) then ! Problème à l'ouverture
  ...
else
  WRITE( UNIT=1, FMT=fmt ) &
    ((tab(i,j),i=1,n),j=1,m)
end if
CLOSE( UNIT=1 )
```

Dans un format le facteur de répétition doit obligatoirement être précisé à l'aide d'une constante littérale. Cet exemple montre comment le générer dynamiquement en utilisant un fichier interne.

```

PROGRAM fichier_interne
  CHARACTER(len=80) enreg
  INTEGER          ios
  REAL             x, y, z
  NAMELIST/liste/x, y, z

  OPEN( UNIT=1,          FILE="data_txt_seq", &
        FORM="formatted", ACTION="read",      &
        STATUS="old",    POSITION="rewind",    &
        IOSTAT=ios )
  IF ( ios /= 0 ) STOP 4
  READ( UNIT=1, FMT='(a)', IOSTAT=ios ) enreg
  DO WHILE( ios == 0 )
    IF ( VERIFY( enreg, &
                 " ,+-0123456789.eEdD" ) == 0 ) THEN
      READ( enreg, FMT=*, iostat=ios ) x, y, z
      !-----
      WRITE( UNIT=*, NML=liste )
    END IF
    READ( UNIT=1, FMT='(a)', iostat=ios ) enreg
  END DO
  CLOSE( UNIT=1 )
END PROGRAM fichier_interne

```

Dans cet exemple on lit un fichier en ne traitant que les enregistrements constitués de réels et en ignorant tous les autres.

7.7 – Instructions de positionnement

Toute opération de lecture-écriture dans un fichier est effectuée par rapport à la position courante dans ce fichier. À l'ouverture celle-ci peut être précisée à l'aide du paramètre `POSITION`. Dans un fichier séquentiel toute lecture-écriture d'un enregistrement de rang n implique le positionnement à l'enregistrement de rang $n+1$.

Trois instructions `BACKSPACE`, `REWIND` et `ENDFILE` permettent de modifier la position :

- ➡ `BACKSPACE` force la position au début de l'enregistrement précédent,
- ➡ `REWIND` force la position au début du fichier,
- ➡ `ENDFILE` écrit un enregistrement de type fin de fichier. Il est alors nécessaire d'exécuter ensuite l'une des deux instructions précédentes.

Ces instructions admettent en paramètre le numéro de l'unité logique auquel le fichier est connecté.

Exemple : troncature contrôlée d'un fichier

```
program troncature
  REAL, dimension(100) :: tab
  INTEGER ios
  LOGICAL flag/.false./
  ...
  OPEN( UNIT=1,           &
        FILE="data_txt_seq", &
        ACTION="readwrite", &
        POSITION="append",   &
        STATUS="old",       &
        IOSTAT=ios )
  IF ( ios /= 0 ) then ! Problème à l'ouverture
    ...
  ELSE
    tab(:) = acos(-1.)
    WRITE( UNIT=1, FMT='(100F6.3)' ) tab
    REWIND( UNIT=1 )
  END IF
```

Le fichier dont le nom est `data_txt_seq` est ouvert avec un positionnement en fin de fichier (`POSITION="append"`). Après écriture d'un enregistrement, on se repositionne en tête (`REWIND`).

```
READ( UNIT=1, FMT='(100F6.3)', IOSTAT=ios ) tab
DO WHILE( ios == 0 )
  if ( flag ) then
    BACKSPACE( UNIT=1 )
    ENDFILE( UNIT=1 )
    BACKSPACE( UNIT=1 )
  END IF
  ...
  READ( UNIT=1, FMT='(100F6.3)', IOSTAT=ios ) tab
END DO
CLOSE( UNIT=1 )
end program troncature
```

Ensuite on relit le fichier et si la variable `flag` contient la valeur `.TRUE.` on le tronque avant le dernier enregistrement lu. (Instructions `BACKSPACE` et `ENDFILE`).

7.8 – Instruction INQUIRE

L'instruction d'interrogation INQUIRE permet de récupérer un certain nombre d'informations concernant un fichier ou un numéro d'unité logique.

Elle permet par exemple de tester si un fichier existe, s'il est connecté et dans ce cas de connaître les valeurs des paramètres positionnés lors de son ouverture via OPEN.

Cette interrogation peut être faite en indiquant soit le numéro d'unité logique soit le nom du fichier.

```
program inquire
  LOGICAL existe
  INTEGER ios
  CHARACTER(len=3)  :: form
  CHARACTER(len=10) :: acces

  INQUIRE( FILE="data_txt_seq", EXIST=existe )
```

```
if ( existe ) then
  OPEN( UNIT=1,           &
        FILE="data_txt_seq", &
        POSITION="rewind",   &
        ACTION="read",     &
        IOSTAT=ios )
  if ( ios /= 0 ) then ! erreur à l'ouverture
    ...
  else
    INQUIRE( UNIT=1,      &
              FORMATTED=form, &
              ACCESS=acces )
  end if
  ...
  CLOSE( UNIT=1 )
end if
end program inquire
```

Dans les variables caractères `form` et `acces` on récupère respectivement les valeurs YES et SEQUENTIAL (si le fichier `data_txt_seq` existe).

7.9 – Remarques

Les spécificateurs de format Bw[.d] , 0w[.d] et Zw[.d] permettent la conversion de données entières sous forme binaire, octale et hexadécimale respectivement.

```
PROGRAM boz
  INTEGER I, J, K

  I = 1415; J = 1515; K = 1715
  WRITE( UNIT = 1, &
        FMT = "(B32.32,'|',011.11,'|',Z8.8)" &
        ) I, J, K
  I = -1415; J = -1515; K = -1715
  WRITE( UNIT = 1, FMT='(B32.32)' ) I
  WRITE( UNIT = 1, FMT='(011.11)' ) J
  WRITE( UNIT = 1, FMT='(Z8.8)' ) K
END PROGRAM boz
```

Sorties

000000000000000000000000010110000111 00000002753 000006B3
11111111111111111111111101001111001
37777775025
FFFFFF94D

Les fichiers associés au clavier et à l'écran d'une session interactive sont pré-connectés en général aux numéros d'unités logiques 5 et 6 respectivement : en lecture pour le premier, en écriture pour le second.

Dans un souci de portabilité, on préférera utiliser dans les instructions READ/WRITE le caractère * à la place du numéro de l'unité logique pour référencer l'*entrée standard* (READ) ou la *sortie standard* (WRITE). C'est la valeur par défaut du paramètre UNIT. L'instruction PRINT remplace l'instruction WRITE dans le cas où celui-ci n'est pas précisé.

Formes équivalentes

```
CHARACTER(LEN=8)      :: fmt = "(F8.3)"  
READ( UNIT=5, FMT=fmt) ...  
READ( UNIT=*, FMT=fmt) ...  
READ fmt, ...
```

```
WRITE( UNIT=6, FMT=fmt) ...  
WRITE( UNIT=*, FMT=fmt) ...  
PRINT fmt, ...
```

Le format d'édition peut être défini en dehors des instructions d'entrées-sorties `READ/WRITE`. Dans ce cas le paramètre `FMT=` est positionné à un numéro (étiquette) renvoyant à une instruction de définition de format (`FORMAT`).

```
REAL, DIMENSION(5,6) :: tab
INTEGER n, i
CHARACTER(len=10) :: ch
...
PRINT '( I4,A,(T20,F8.3) )', &
      n, ch, (tab(i,:),i=1,5)
PRINT 100, n, ch, (tab(i,:),i=1,5)
100 FORMAT( I4,A,(T20,F8.3) )
```

En Fortran l'ouverture d'un fichier séquentiel est facultative. À défaut, l'ouverture du fichier est faite implicitement lors de l'exécution de la première instruction d'entrée-sortie. Le compilateur attribue au fichier un nom de la forme `fort.i` (i étant le numéro de l'unité logique indiqué dans l'instruction `READ/WRITE`). L'ouverture est faite en mode *formatted* ou *unformatted* suivant qu'un format a été ou non précisé.

Le paramètre **END** de l'instruction **READ** offre un autre moyen de tester une fin de fichier pour un accès séquentiel. On lui indique le numéro (étiquette) de l'instruction à laquelle on désire poursuivre le traitement. Ce paramètre n'est pas valide dans le cas d'un fichier à accès direct.

De même, le paramètre **ERR** permet de se débrancher à une instruction dans le cas d'une erreur de lecture (problème de format pour un fichier formaté, enregistrement inexistant pour un fichier à accès direct ...).

```
INTEGER :: date

do
  print *, "Saisie d'une date :"
  read( *, '(i4)', END=1, ERR=2 ) date
  print '(i4)', date
  cycle
2  print *, "Saisie invalide. Veuillez recommencer."
end do
1  print *, "Arrêt de la saisie."
  ...
```

Remarque : au clavier, la saisie du caractère **Ctrl-D** après le caractère *newline* (touche *Enter*) indique une fin de fichier.

8 – Procédures

- ➡ 8.1 - Arguments
- ➡ 8.2 - Subroutines
- ➡ 8.3 - Fonctions
- ➡ 8.4 - Arguments de type chaîne de caractères
- ➡ 8.5 - Arguments de type tableau
- ➡ 8.6 - Arguments de type procédure
- ➡ 8.7 - Procédures internes
- ➡ 8.8 - Durée de vie des identificateurs
- ➡ 8.9 - Procédures intrinsèques

8.1 – Arguments

Très souvent, dans un programme, on a besoin d'effectuer un même traitement plusieurs fois avec des valeurs différentes. La solution est de définir ce traitement une seule fois à l'aide d'une unité de programme de type procédure (SUBROUTINE ou FUNCTION).

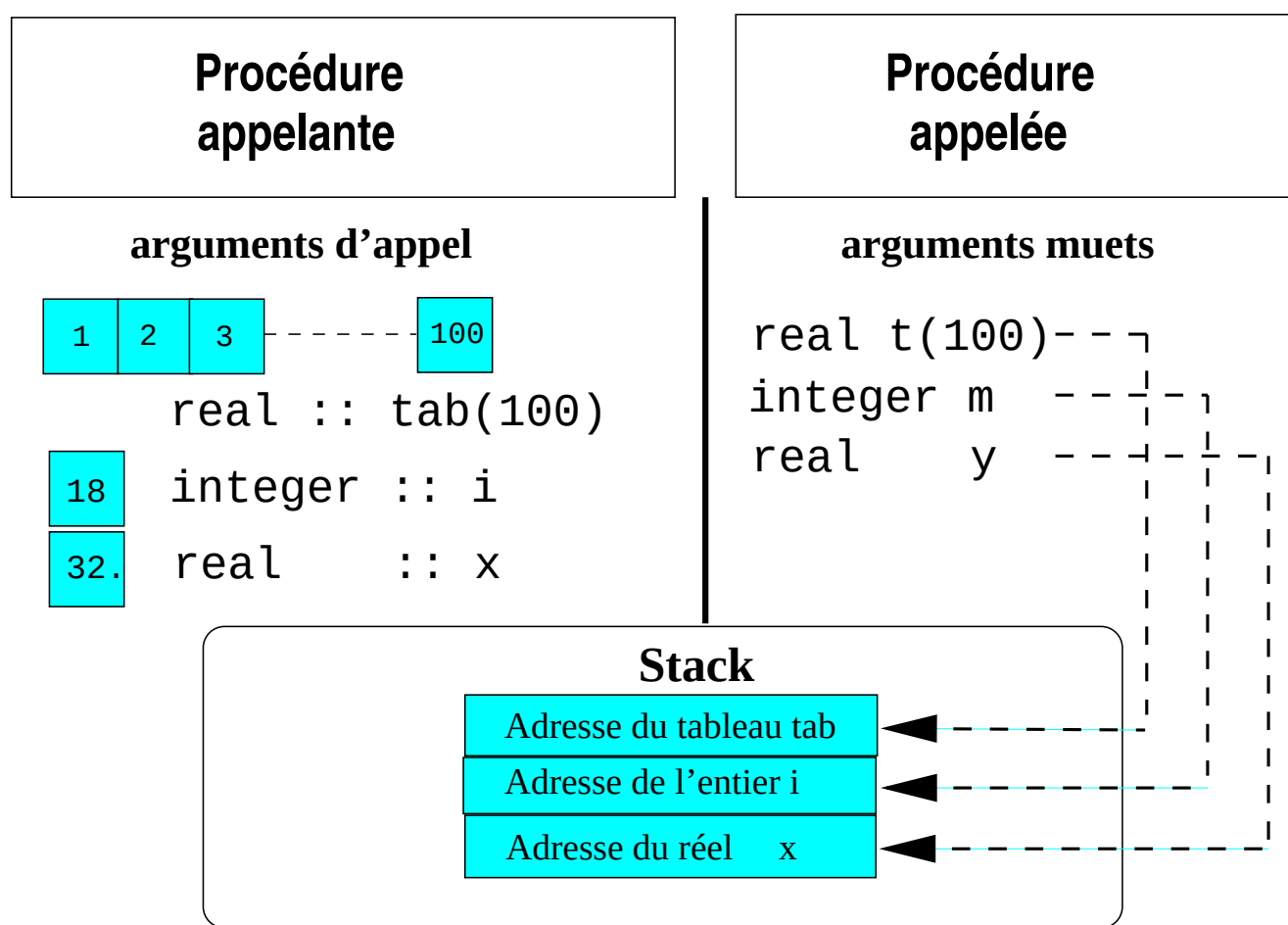
Les unités de programmes désirant effectuer ce traitement feront appel à cette procédure en lui transmettant des valeurs via des variables appelées **arguments d'appel** (*actual-arguments*). La procédure appelée récupère les valeurs qu'on lui a transmises via des variables appelées **arguments muets** (*dummy-arguments*).

En Fortran le passage de ces valeurs s'effectue par **référence** ;

- 👉 les adresses des **arguments d'appel** sont transmises à la procédure appelée,
- 👉 dans la procédure appelée, les **arguments muets** sont des alias des **arguments d'appel**.

Schéma passage arguments

FIG. 1 – Schéma passage arguments



8.2 – Subroutines

L'appel d'une procédure de type SUBROUTINE s'effectue à l'aide de l'instruction `CALL` suivie du nom de la procédure à appeler avec la liste des arguments d'appels entre parenthèses.

Exemple

```
REAL, DIMENSION(100) :: tab
REAL                  :: moyenne, maximum
...
CALL SP( tab, moyenne, maximum )
PRINT *,moyenne, maximum
END
SUBROUTINE SP( t, moy, max )
  REAL, DIMENSION(100) :: t
  REAL                  :: moy, max
  INTEGER                :: i
  max = t(1); moy = t(1)
  DO i=2,100
    IF ( t(i) > max ) max = t(i)
    moy = moy + t(i)
  END DO
  moy = moy/100
END SUBROUTINE SP
```

8.3 – Fonctions

Un autre moyen de transmettre des valeurs à une unité de programme est l'utilisation d'une procédure de type **FUNCTION**.

À la différence d'une **SUBROUTINE**, une **FUNCTION** retourne une valeur ; celle-ci est donc typée. De plus, son appel s'effectue en indiquant uniquement son nom suivi entre parenthèses de la liste des arguments d'appels.

Au sein de la fonction l'instruction **return** sert à transmettre à la procédure appelante la valeur à retourner. Celle-ci n'est nécessaire que dans le cas où on désire effectuer ce retour avant la fin de la définition de la fonction.

Dans la procédure appelante l'expression correspondant à l'appel de la fonction est remplacée par la valeur retournée.

Exemple

```
REAL, DIMENSION(100) :: tab
REAL                  :: moyenne, maximum
REAL                  :: maxi

...
maximum = maxi( tab, moyenne )
PRINT *,moyenne, maximum

...
END

FUNCTION maxi( t, moy )
  REAL, DIMENSION(100) :: t
  REAL                  :: moy, maxi
  INTEGER                :: i
  maxi = t(1); moy = t(1)
  DO i=2,100
    IF ( t(i) > maxi ) maxi = t(i)
    moy = moy + t(i)
  END DO
  moy = moy/100
END FUNCTION maxi
```

8.4 – Arguments de type chaîne de caractères

Lorsqu'une chaîne de caractères est transmise en argument, **Fortran** passe également sa longueur de façon implicite.

Dans la procédure appelée, celle-ci peut être récupérée à l'aide de la fonction intrinsèque **LEN**.

La déclaration de la chaîne de caractères au sein de la procédure appelée est faite en spécifiant le caractère ***** à la place de la longueur.

La procédure appelée fait alors référence à une chaîne de caractères à taille implicite (*assumed-size string*).

Dans l'exemple ci-dessous les fonctions ICHAR/ACHAR permettent de mettre en relation un caractère et son rang dans la table des caractères ASCII.

Exemple

```

program arg_chaine
  implicit none
  character(len=10) :: ch

  read '(a)',ch
  call conv( ch )
  print *,ch
end program arg_chaine
subroutine conv( chaine )
  implicit none
  character(len=*) :: chaine
  integer i, j

  do i=1,len(chaine)
    if( ichar( chaine(i:i) ) < 97 .or. &
       ichar( chaine(i:i) ) > 122 ) cycle
    j = ichar( chaine(i:i) ) - 32
    chaine(i:i) = achar( j )
  end do
end subroutine conv

```

8.5 – Arguments de type tableau

Lorsque l'on transmet un tableau en argument il est commode de transmettre également ses dimensions afin de pouvoir déclarer l'argument muet correspondant au sein de la procédure appelée à l'aide de celles-ci; de ce fait le tableau est ajustable.

Exemple

```
program arg_tab
  implicit none
  integer, parameter    :: n = 3, m = 2
  real, dimension(n,m) :: tab
  real                  :: somme
  read *,tab; print *,somme( tab, n, m )
end program arg_tab

real function somme( t, n, m )
  integer          :: n,m,i,j
  real, dimension(n,m) :: t
  somme = 0.
  do i=1,n
    do j=1,m
      somme = somme + t(i,j)
    end do
  end do
end function somme
```

Lorsqu'un tableau est passé en argument c'est l'adresse de son premier élément qui est transmise.

La procédure appelée doit posséder les informations lui permettant d'adresser les différents éléments de ce tableau.

De façon générale, supposons que l'on dispose d'un tableau `tab` à 2 dimensions constitué de `n` lignes et `m` colonnes. L'adresse de l'élément `tab(i,j)` est :

$$@tab(i,j) = @tab(1,1) + [n*(j-1)+(i-1)]*taille(élt)$$

Le nombre de colonnes `m` n'intervient pas dans ce calcul.

Souvent en **Fortran**, lors de l'appel d'une procédure seule la première dimension d'un tableau à 2 dimensions est transmise.

Dans la procédure appelée celui-ci est déclaré en indiquant le caractère `*` à la place de la deuxième dimension. On fait alors référence à un tableau à **taille implicite** (*assumed-size array*).

Dans un tel cas, il faut faire preuve d'une certaine prudence car dans la procédure appelée on ne maîtrise pas l'espace mémoire total occupé par le tableau.

Exemple

```
program taille_implicit
  implicit none
  integer, parameter    :: n = 5, m = 6
  real, dimension(n,m) :: tab
  real                  :: somme, som

  read *, tab
  somme = som( tab, n )
  print *,somme
end program taille_implicit

real function som( t, lda )
  implicit none
  real, dimension(lda,*) :: t
  integer                :: lda
  integer                :: i,j

  som = 0.
  do i=1,lda
    do j=1,lda
      som = som + t(i,j)
    end do
  end do
end function som
```

8.6 – Arguments de type procédure

Une procédure peut être transmise à une autre procédure. Il est nécessaire de la déclarer dans la procédure appelante avec l'attribut **EXTERNAL** ou **INTRINSIC** si elle est intrinsèque.

Exemple

```
program arg_proc
  implicit none
  double precision b_inf, b_sup, aire
  double precision integrale
  integer pas
  double precision, external :: carre

  b_inf = 1.
  b_sup = 6.
  pas = 200000
  aire = integrale( b_inf, b_sup, pas, carre )
  print '("Aire : ", f11.6)', aire
end program arg_proc
```

```
function integrale( borne_i, borne_s, pas, f )  
  implicit none  
  double precision borne_i, borne_s  
  double precision integrale  
  integer          pas, i  
  double precision h, f  
  
  h = (borne_s - borne_i)/pas  
  integrale = 0.  
  do i=0, pas-1  
    integrale = integrale + h*f(borne_i+i*h)  
  end do  
end function integrale  
  
function carre( x )  
  implicit none  
  double precision x  
  double precision carre  
  
  carre = x*x  
end function carre
```

8.7 – Procédures internes

En Fortran une procédure peut en contenir d'autres. Ces procédures sont appelées **procédures internes**. Elles ne peuvent être appelées que depuis la procédure les contenant.

Les définitions des procédures internes sont faites dans la procédure les incluant après l'instruction CONTAINS.

Il n'y a pas d'imbrications possibles : une procédure interne ne peut pas elle-même en contenir.

Exemple

```
program proc_interne
  implicit none
  character(len=10) :: ch

  read '(a)',ch
  call conversion( ch )
  print *,ch
end program proc_interne
```

```

subroutine conversion( chaine )
  implicit none
  character(len=*) :: chaine
  integer i

  do i=1,len(chaine)
    if( ichar( chaine(i:i) ) < 97 .or.
       ichar( chaine(i:i) ) > 122 ) cycle
    chaine(i:i) = car_majuscule( chaine(i:i) )
  end do
CONTAINS
  function car_majuscule( c )
    character(len=1) :: c, car_majuscule
    integer          :: i

    i = ichar( c ) - (ichar('a') - ichar( 'A' ))
    car_majuscule = achar( i )
  end function car_majuscule
end subroutine conversion

```

Dans une procédure interne, toute variable déclarée dans l'unité de programme qui la contient est accessible, à moins qu'elle n'ait fait l'objet d'une redéclaration.

Exemple

```
program visibilite
  implicit none
  integer i, j
  i = 10; j = 20
  call sub
  print '("i =",i3,"", j =",i3)', i, j
contains
  subroutine sub
    integer j      ! masque le "j" de l'appelant
    j = 100; i = i + 1
  end subroutine sub
end program visibilite
```

Sortie

i[^]=11, j[^]=20

8.8 – Durée de vie et visibilité des identificateurs

On appelle durée de vie d'un identificateur le temps pendant lequel il existe en mémoire. Il est visible s'il existe en mémoire et est accessible, car il peut exister mais être masqué par un autre de même nom (c.f. procédure interne).

- ➡ Par défaut, une variable a une durée de vie limitée à celle de l'unité de programme dans laquelle elle a été définie,
- ➡ l'attribut **SAVE** permet de prolonger la durée de vie à celle de l'exécutable : on parle alors de variable permanente ou statique,
- ➡ dans une unité de programme l'instruction **SAVE** sans spécification de liste de variables indique que toutes les variables de cette unité sont permanentes,
- ➡ une compilation effectuée en mode *static* force la présence de l'instruction **SAVE** dans toutes les unités de programme, ce qui implique que toutes les variables sont permanentes,
- ➡ par contre si elle est faite en mode *stack*, les variables permanentes sont :
 - ➡ celles pour lesquelles l'attribut **SAVE** a été précisé,
 - ➡ celles initialisées à la déclaration (via l'instruction **DATA** ou à l'aide du signe =).

program newton

```
double precision :: valeur = 50.d0
double precision :: tolerance
double precision :: racine, x

tolerance = 1.0d-1
x = racine( valeur, tolerance )
print '("Racine de ",f5.1," = ", d16.8)', valeur, x
tolerance = 1.0d-6
x = racine( valeur, tolerance )
print '("Racine de ",f5.1," = ", d16.8)', valeur, x
```

end program newton**function racine**(valeur, tol)

```
double precision :: valeur, tol
double precision :: racine
double precision :: x = 1.0d0, x_prev
integer          :: nb_iterations

nb_iterations = 0
do
    nb_iterations = nb_iterations + 1
    x_prev = x
    x = 0.5 * (x_prev + valeur/x_prev)
    if ( abs(x-x_prev)/x < tol ) exit
end do
print *, "Nombre d'itérations = ", nb_iterations
racine = x
```

end function racine

8.9 – Procédures intrinsèques

Le compilateur Fortran dispose d'une bibliothèque de procédures couvrant différents domaines : mathématique, conversion de type, manipulation de chaînes de caractères, comparaison de chaînes de caractères, ...

Pour une procédure donnée, le nom d'appel diffère suivant le type des arguments transmis. Un nom générique permet de s'affranchir de ces types : c'est la solution fortement conseillée car elle facilite la portabilité.

Par exemple un appel à la procédure générique ABS, retournant la valeur absolue de son argument, est transformé par le compilateur en un appel à la procédure :

- 👉 IABS pour un argument entier,
- 👉 ABS pour un argument réel simple précision,
- 👉 DABS pour un argument réel double précision,
- 👉 CABS pour un argument complexe.

Une liste des procédures intrinsèques est fournie en annexe B.

9 – Common

- ☞ 9.1 - L'instruction Common
- ☞ 9.2 - Common blanc
- ☞ 9.3 - Common étiqueté
 - ⇒ 9.3.1 - Initialisation : BLOCK DATA
 - ⇒ 9.3.2 - Instruction SAVE et COMMON
- ☞ 9.4 - Règles et restrictions

9.1 – L’instruction COMMON

L’instruction **COMMON** permet le regroupement de zones mémoires pouvant être partagées par différentes unités de programme (fonctions, procédures).

La syntaxe d’une instruction **COMMON** est la suivante :

```
COMMON [/ [nom_common] /] liste_variables
```

Le **COMMON** est dit étiqueté si `nom_common` est précisé. S’il n’a pas de nom on l’appelle **COMMON blanc**.

Les différentes zones regroupées au sein du bloc **COMMON** sont adressées via des variables dont les noms sont indiqués dans la partie `liste_variables` de l’instruction **COMMON**. Pour qu’une unité de programme ait accès à ces zones, il est nécessaire qu’elle contienne l’instruction **COMMON** les référant.

9.2 – Common blanc

Lors de la déclaration d'un **COMMON** blanc `nom_common` est omis et les deux *slashes* `"/` sont alors optionnels.

Ses particularités sont :

- ☞ un **COMMON** blanc est permanent (il hérite de l'attribut **SAVE**),
- ☞ les variables apparaissant dans un **COMMON** blanc ne peuvent pas être initialisées lors de leur déclaration. Ce type de bloc est initialement constitué de *bits* à 0. En conséquence :
 - ⇒ les données numériques sont initialisées à 0,
 - ⇒ les données logiques sont initialisées à la valeur `.FALSE.`,
 - ⇒ les données de type chaîne de caractères représentent des chaînes vides.
- ☞ un **COMMON** blanc peut ne pas avoir la même taille d'une unité de programme à une autre, c'est alors la taille maximum qui sera retenue pour l'ensemble du programme.

Exemples

PROGRAM common_blanc

```
INTEGER                :: i
INTEGER, DIMENSION(6) :: itab
LOGICAL, DIMENSION(3) :: ltab
COMMON                 itab,ltab
```

```
DO i=1,6
    itab(i) = i
END DO
ltab(1) = .true.
ltab(2) = .false.
ltab(3) = .true.
CALL sub
```

END PROGRAM common_blanc

```
!*****
```

SUBROUTINE sub

```
INTEGER                :: i
INTEGER, DIMENSION(6) :: itab
LOGICAL, DIMENSION(3) :: ltab
COMMON                 itab,ltab
```

```
PRINT*, 'Tableau entier = ', itab
PRINT*, 'Tableau logique = ', ltab
```

END SUBROUTINE sub

9.3 – Common étiqueté

Un **COMMON** est dit étiqueté lorsque `nom_common` est précisé entre caractères `/`.

Ses particularités sont :

- ➡ il peut être initialisé par l'intermédiaire d'une unité de programme de type **BLOCK DATA**,
- ➡ un bloc **COMMON** défini dans le programme principal reçoit implicitement l'attribut **SAVE**. S'il ne l'est pas, l'attribut **SAVE** est nécessaire si on désire le rendre permanent.

Exemples

```
INTEGER, DIMENSION(6)  :: itab  
REAL,    DIMENSION(12) :: rtab  
  
COMMON /TAB/itab, rtab
```

9.3.1 – Initialisation : BLOCK DATA

BLOCK DATA est une unité de programme qui permet d'initialiser des objets déclarés dans des **COMMON**s étiquetés :

```
BLOCK DATA [nom_block_data]
```

```
[bloc_init]
```

```
END BLOCK DATA [nom_block_data]
```

👉 `nom_block_data` est le nom du **BLOCK DATA**

👉 `bloc_init` est une suite :

- ⇒ de déclarations de type (`INTEGER`, `REAL`, ...),
- ⇒ de déclarations de zones communes (**COMMON**),
- ⇒ d'initialisations statiques (`DATA`).

Un bloc **COMMON** ne peut apparaître que dans un seul **BLOCK DATA**.

On peut se définir plusieurs unités **BLOCK DATA**, chacune regroupant les **COMMON** qui ont un lien logique entre eux.

Exemples

```
BLOCK DATA INIT
```

```
REAL                                :: A, B, C, D
```

```
REAL, DIMENSION(10,15) :: MATRICE
```

```
INTEGER, DIMENSION(20) :: VECTEUR
```

```
COMMON /BLOC1/ MATRICE, A, B
```

```
COMMON /BLOC2/ VECTEUR, C, D
```

```
DATA A /-1./, B /3.14/, C /0./, D /7.1/
```

```
DATA MATRICE /150 * 50.0/, VECTEUR /20 * 4/
```

```
END BLOCK DATA INIT
```

9.3.2 – Instruction SAVE et COMMON

Les valeurs des variables d'un **COMMON** étiqueté deviennent indéfinies quand une procédure se termine (retour à l'unité appelante) sauf s'il existe une autre unité de programme active le référençant.

Le cas échéant, on lui appliquera l'instruction **SAVE** pour conserver son contenu :

SAVE /nom_common/

- ➡ Un **COMMON** qui reçoit l'attribut **SAVE** dans une fonction ou procédure devra toujours être déclaré avec ce même attribut dans toutes les autres unités de programme,
- ➡ Il est inutile de spécifier l'attribut **SAVE** si le **COMMON** a été déclaré dans le programme principal.

Exemple

```

PROGRAM common_save
  call first
  call second
END PROGRAM common_save

!*****

SUBROUTINE first
  REAL, DIMENSION(6)      :: rtab
  LOGICAL                  :: drapeau
  COMMON/BLOC/ rtab,drapeau
  SAVE /BLOC/

  CALL random_number(rtab)
  PRINT*, 'Afficher le tableau (true/false)'
  READ(*,*) drapeau
END SUBROUTINE first

!*****

SUBROUTINE second
  REAL, DIMENSION(6)      :: rtab
  LOGICAL                  :: drapeau
  COMMON /BLOC/           rtab,drapeau
  SAVE /BLOC/

  IF (drapeau) PRINT*, 'Tableau de réels = ', rtab
END SUBROUTINE second

```

9.4 – Règles et restrictions

- ➡ Un **COMMON** ne peut pas contenir :
 - ➡ les noms de procédures (sous-programmes, fonctions),
 - ➡ les arguments de procédures,
 - ➡ les constantes symboliques (ayant l'attribut **PARAMETER**).
- ➡ une même variable ne peut pas apparaître dans 2 **COMMON**s de noms différents,
- ➡ la taille d'un bloc **COMMON** étiqueté doit être la même dans chaque unité de programme le référénçant,
- ➡ Fortran 90 permet désormais le mélange de données numériques et caractères au sein d'un bloc **COMMON**,
- ➡ d'une unité de programme à une autre, les variables de la liste peuvent porter des noms différents,
- ➡ au sein d'une même unité de programme, un bloc **COMMON** (étiqueté ou non) peut être référencé plusieurs fois : les différentes listes de variables sont alors ajoutées les unes aux autres.

- ☞ un bloc **COMMON** peut être découpé différemment d'une unité de programme à une autre, c'est-à-dire référencé à l'aide de variables de types différents à condition que ce découpage soit cohérent (les zones numériques doivent correspondre à des zones numériques et de même pour les zones caractères),
- ☞ une instruction **EQUIVALENCE** ne peut pas :
 - ⇒ associer des variables déclarées dans des blocs **COMMON** différents,
 - ⇒ avoir pour effet de prolonger le **COMMON** à sa gauche. Dans l'exemple suivant, on ne peut pas associer la variable scalaire A au scalaire B(2) :

```
REAL                :: A
REAL, DIMENSION(2) :: B

COMMON /X/A
EQUIVALENCE (A,B(2))    ! INVALIDE
EQUIVALENCE (A,B(1))    ! VALIDE
```

Exemple

```
program common_mixte
  complex, dimension(10) :: c
  character(len=100)      :: chaine
  COMMON/MIXTE/c, chaine
  . . .
  call sub
end program common_mixte

subroutine sub
  real, dimension(20)      :: tab_reels
  character(len=1), dimension(100) :: tab_car
  COMMON/MIXTE/tab_reels, tab_car
  . . .
  ! impression des parties réelles
  print *,tab_reels(1:20:2)
  print *,tab_car(1), tab_car(10)
end subroutine sub
```

Recommandations :

1. il est préférable de déclarer un bloc **COMMON** de la même façon dans les différentes unités de programme et donc d'employer la technique de l'exemple précédent avec modération,
2. l'instruction **INCLUDE** ci-après favorise le bon emploi des blocs **COMMON**.

10 – Include

10.1 – La directive INCLUDE

Elle demande au compilateur d'inclure un fichier contenant des instructions Fortran :

```
INCLUDE 'nom_de_fichier'
```

Exemple

```
PROGRAM inclusion
  IMPLICIT NONE
  INTEGER :: i
  INCLUDE 'inclusion.inc'

  DO i=1,6
    itab(i) = PI
  END DO
  ltab(1) = .true.
  ltab(2) = .false.
  ltab(3) = .true.
  CALL sub
END PROGRAM inclusion
```

```
SUBROUTINE sub
  IMPLICIT NONE
  INCLUDE 'inclusion.inc'

  PRINT*, 'Tableau réels    = ', rtab
  PRINT*, 'Tableau logique = ', ltab
END SUBROUTINE sub
```

Le fichier `inclusion.inc` contient les déclarations de COMMONs et de paramètres :

```
DOUBLE PRECISION, PARAMETER      :: PI=3.14159265d0
DOUBLE PRECISION, DIMENSION(6)  :: rtab
LOGICAL, DIMENSION(3)           :: ltab
COMMON rtab, ltab
```

Remarques :

- 👉 L'inclusion du contenu du fichier se fait là où est placée la directive `INCLUDE`,
- 👉 Il est possible d'imbriquer plusieurs directives `INCLUDE`. Le nombre d'imbrications possible dépend du compilateur.
- 👉 La localisation du fichier à inclure peut être précisée à l'aide de l'option `-I` du système de compilation.

11 – Annexe A : entrées-sorties - syntaxes

➡ Entrées-Sorties : syntaxes

Note : les valeurs par défaut sont soulignées.

Instruction OPEN

```
OPEN( [UNIT=]u,      &      ! entier
      IOSTAT=ios,    &      ! entier
      FILE=filename, &      ! chaîne de caractères
      STATUS=st,     &      ! chaîne de caractères
      ACCESS=acc,    &      ! chaîne de caractères
      FORM=form,     &      ! chaîne de caractères
      RECL=recl,     &      ! entier
      POSITION=pos,   &      ! chaîne de caractères
      ACTION=act,    &      ! chaîne de caractères
      BLANK=blk,     &      ! chaîne de caractères
      DELIM=del,     &      ! chaîne de caractères
      PAD=pad )      ! chaîne de caractères
```

- 👉 UNIT : numéro de l'unité logique sur laquelle on désire connecter le fichier,
- 👉 IOSTAT : entier valorisé une fois l'ouverture effectuée. Il est nul si l'opération s'est bien passée, non nul sinon,
- 👉 FILE : nom du fichier à ouvrir,

- 👉 **STATUS** : état du fichier avant l'ouverture,
 - ⇒ **OLD** : le fichier doit exister,
 - ⇒ **NEW** : le fichier ne doit pas exister, il sera créé,
 - ⇒ **UNKNOWN** : état dépendant du compilateur,
 - ⇒ **REPLACE** : si le fichier n'existe pas, il est créé et hérite de l'état "OLD", sinon, il est détruit et un nouveau fichier est créé,
 - ⇒ **SCRATCH** : un fichier temporaire anonyme est créé, il sera détruit à la fin du programme ou au moment du **CLOSE**.
- 👉 **ACCESS** : mode d'accès au fichier,
 - ⇒ **DIRECT** : accès direct,
 - ⇒ **SEQUENTIAL** : accès séquentiel.

- ☞ **FORM** : type du transfert,
 - ⇒ **FORMATTED** : mode caractère (avec format),
 - ⇒ **UNFORMATTED** : mode binaire (sans format).
- ☞ **RECL** : longueur des enregistrements logiques en accès direct ou de l'enregistrement maximum en accès séquentiel. L'unité de longueur est le caractère pour les fichiers textes et dépend du processeur pour les fichiers binaires,
- ☞ **POSITION** : positionnement dans le fichier après ouverture,
 - ⇒ **ASIS** : positionnement inchangé si le fichier est déjà connecté, indéfini sinon,
 - ⇒ **REWIND** : positionnement en tête de fichier,
 - ⇒ **APPEND** : positionnement en fin de fichier pour extension,
- ☞ **ACTION** : mode d'ouverture.
 - ⇒ **READ** : seule la lecture est possible,
 - ⇒ **WRITE** : seule l'écriture est possible,
 - ⇒ **READWRITE** : la lecture ainsi que l'écriture sont permises,

- ☞ **BLANK** : indique la façon dont les blancs sont interprétés lors d'une opération d'entrée-sortie,
 - ⇒ **NULL** : ils sont ignorés,
 - ⇒ **ZERO** : ils sont interprétés comme des zéros,
- ☞ **DELIM** : indique le caractère délimiteur utilisé pour les constantes chaînes de caractères écrites en format libre ou via une **NAMelist**,
 - ⇒ **APOSTROPHE** : délimiteur → `"`,
 - ⇒ **QUOTE** : délimiteur → `'`,
 - ⇒ **NONE** : aucun délimiteur,
- ☞ **PAD** : permet de gérer le *padding* lors d'une lecture,
 - ⇒ **YES** : la liste de variables spécifiée dans l'instruction de lecture peut être plus grande que l'enregistrement logique, les variables non valorisées lors de la lecture sont remplies par des zéros pour les variables numériques ou logiques, par des blancs pour les variables de type chaîne de caractères,
 - ⇒ **NO** : aucun *padding* n'est autorisé. La taille de l'enregistrement logique doit être suffisante pour permettre la valorisation des variables de la liste.

Instruction READ

```
READ( [UNIT=]u,      &      ! entier
      [FMT=]format,  &      ! chaîne de caractères
      [NML=]namelist, &      ! chaîne de caractères
      ADVANCE=ad,    &      ! chaîne de caractères
      END=label,     &      ! label
      EOR=label,     &      ! label
      ERR=label,     &      ! label
      IOSTAT=st,     &      ! entier
      REC=n,         &      ! entier
      SIZE=n )       ! entier
```

- 👉 UNIT : numéro de l'unité logique sur laquelle le fichier est connecté,
- 👉 FMT : format de conversion pour les fichiers textes spécifié sous forme :
 - ⇒ d'une constante chaîne de caractères,
 - ⇒ d'une variable chaîne de caractères,
 - ⇒ d'une étiquette d'instruction FORMAT.
- 👉 NML : nom d'une *namelist*,

- ➡ **ADVANCE** : positionnement à partir duquel s'effectue l'entrée-sortie suivante :
 - ➡ **YES** → enregistrement suivant,
 - ➡ **NO** → suite de l'enregistrement courant,
- ➡ **END** : étiquette de l'instruction à exécuter en cas de fin de fichier,
- ➡ **EOR** : étiquette de l'instruction à exécuter en cas de fin d'enregistrement,
- ➡ **ERR** : étiquette de l'instruction à exécuter en cas d'erreur,
- ➡ **IOSTAT** : entier valorisé une fois l'entrée-sortie effectuée. Il est nul si l'opération s'est bien passée. Une valeur positive indique une erreur. Une valeur négative signale une fin de fichier dans le cas d'un accès séquentiel et l'absence d'enregistrement du rang spécifié si l'accès est direct,
- ➡ **REC** : numéro de l'enregistrement à traiter pour un fichier à accès direct,
- ➡ **SIZE** : entier récupérant le nombre de caractères traités si la fin d'enregistrement a été atteinte.

Instruction WRITE

```
WRITE( [UNIT=]u,      &      ! entier
      [FMT=]format,   &      ! chaîne de caractères
      [NML=]namelist, &      ! chaîne de caractères
      ADVANCE=ad,     &      ! chaîne de caractères
      ERR=label,      &      ! label
      IOSTAT=st,      &      ! entier
      REC=n )         ! entier
```

- ☞ UNIT : numéro de l'unité logique sur laquelle le fichier est connecté,
- ☞ FMT : format de conversion pour les fichiers textes spécifié sous forme :
 - ⇒ d'une constante chaîne de caractères,
 - ⇒ d'une variable chaîne de caractères,
 - ⇒ d'une étiquette d'instruction FORMAT.
- ☞ NML : nom d'une *namelist*,

- ➡ **ADVANCE** : positionnement à partir duquel s'effectue l'entrée-sortie suivante :
 - ➡ **YES** → enregistrement suivant,
 - ➡ **NO** → suite de l'enregistrement courant,
- ➡ **ERR** : étiquette de l'instruction à exécuter en cas d'erreur,
- ➡ **IOSTAT** : entier valorisé une fois l'entrée-sortie effectuée. Il est nul si l'opération s'est bien passée, non nul sinon,
- ➡ **REC** : numéro de l'enregistrement à traiter pour un fichier à accès direct.

Instruction INQUIRE

```
INQUIRE( [UNIT=]u,      &    ! entier
          ACCESS=acc,    &    ! chaîne de caractères
          ACTION=act,    &    ! chaîne de caractères
          BLANK=bl,      &    ! chaîne de caractères
          DELIM=del,     &    ! chaîne de caractères
          DIRECT=dir,    &    ! chaîne de caractères
          ERR=label,     &    ! label
          EXIST=ex,      &    ! logique
          FILE=file,     &    ! chaîne de caractères
          FORM=form,     &    ! chaîne de caractères
          FORMATTED=form, &    ! chaîne de caractères
          IOSTAT=ios,    &    ! entier
          NAME=name,     &    ! chaîne de caractères
          NAMED=named,   &    ! logique
          NEXTREC=next,  &    ! entier
          NUMBER=num,    &    ! entier
          OPENED=op,     &    ! logique
          PAD=pad,       &    ! chaîne de caractères
          POSITION=pos,   &    ! entier
          READ=r,        &    ! chaîne de caractères
          READWRITE=rw,  &    ! chaîne de caractères
          RECL=n,        &    ! entier
          SEQUENTIAL=seq, &    ! chaîne de caractères
          UNFORMATTED=un, &    ! chaîne de caractères
          WRITE=wr )     ! chaîne de caractères
```

☞ **UNIT** : numéro de l'unité logique sur laquelle le fichier est connecté,

☞ **ACCESS** : méthode d'accès

⇒ **SEQUENTIAL** si fichier connecté pour un accès séquentiel,

⇒ **DIRECT** si fichier connecté pour un accès direct,

⇒ **UNDEFINED** si fichier non connecté,

☞ **ACTION** : type d'accès

⇒ **READ** si fichier connecté en lecture,

⇒ **WRITE** si fichier connecté en écriture,

⇒ **READWRITE** si fichier connecté en lecture/écriture,

⇒ **UNDEFINED** si fichier non connecté,

☞ **BLANK** :

⇒ **NULL** si les blancs sont ignorés,

⇒ **ZERO** si les blancs sont interprétés comme des 0,

⇒ **UNDEFINED** si le fichier n'est pas connecté en mode formaté ou bien n'est pas connecté du tout,

- ☞ DELIM : délimiteur pour les chaînes de caractères en sortie en format libre ou via *namelist*
 - ⇒ APOSTROPHE délimiteur → "
 - ⇒ QUOTE délimiteur → '
 - ⇒ UNDEFINED si le fichier n'est pas connecté en mode formaté ou bien n'est pas connecté du tout,
- ☞ DIRECT : accès direct
 - ⇒ YES l'accès direct est un mode d'accès permis,
 - ⇒ NO l'accès direct n'est pas un mode d'accès permis,
 - ⇒ UNKNOWN le compilateur ne sait pas si l'accès direct est permis ou non,
- ☞ ERR : étiquette de l'instruction à exécuter en cas d'erreur,
- ☞ EXIST :
 - ⇒ **.true.** si le fichier ou l'unité logique existe,
 - ⇒ **.false.** si le fichier ou l'unité logique n'existe pas,
- ☞ FILE : nom du fichier pour lequel on souhaite avoir des informations,

👉 **FORM** : type d'accès

⇒ **FORMATTED** si le fichier est connecté en mode formaté,

⇒ **UNFORMATTED** si le fichier est connecté en mode binaire,

⇒ **UNDEFINED** si le fichier n'est pas connecté,

👉 **FORMATTED** :

⇒ **YES** si le traitement du fichier en mode formaté est valide

⇒ **NO** si le traitement du fichier en mode formaté n'est pas valide,

⇒ **UNKNOWN** lorsque le compilateur ne sait pas si le traitement du fichier en mode formaté est permis ou non,

👉 **IOSTAT** : valeur de retour

⇒ > 0 si une erreur s'est produite,

⇒ $= 0$ si aucune erreur ne s'est produite,

👉 **NAME** : nom du fichier connecté s'il a un nom, sinon valeur indéfinie,

👉 NAMED :

- ⇒ `.true.` si le fichier a un nom,
- ⇒ `.false.` si le fichier est anonyme,

👉 NEXTREC :

- ⇒ renvoie le numéro du prochain enregistrement à traiter si le fichier est ouvert en accès direct (1 si aucun enregistrement n'a déjà été traité,
- ⇒ renvoie une valeur indéfinie si le fichier n'a pas été ouvert en accès direct,

👉 NUMBER : renvoie le numéro de l'unité logique sur laquelle le fichier est connecté (-1 s'il ne l'est pas),

👉 OPENED :

- ⇒ `.true.` si le fichier est ouvert,
- ⇒ `.false.` si le fichier n'est pas ouvert,

👉 PAD :

- ⇒ NO le fichier a été ouvert avec le paramètre PAD=NO,
- ⇒ YES le fichier a été ouvert avec le paramètre PAD=YES ou bien le fichier n'est pas connecté,

 POSITION :

- ⇒ REWIND si fichier ouvert avec un positionnement en tête,
- ⇒ APPEND si fichier ouvert avec un positionnement en fin,
- ⇒ ASIS si fichier ouvert sans changement de la position,
- ⇒ UNDEFINED si fichier non connecté ou bien connecté en accès direct,

 READ :

- ⇒ YES un accès en lecture est permis,
- ⇒ NO un accès en lecture n'est pas permis,
- ⇒ UNKNOWN le compilateur ne sait pas si un accès en lecture est permis ou non,

 READWRITE :

- ⇒ YES un accès en lecture/écriture est permis,
- ⇒ NO un accès en lecture/écriture n'est pas permis,
- ⇒ UNKNOWN le compilateur ne sait pas si un accès en lecture/écriture est permis ou non,

 RECL : renvoie la taille de l'enregistrement logique maximum ou une valeur indéfinie si le fichier n'existe pas,

 SEQUENTIAL :

- ⇒ YES l'accès séquentiel est un mode d'accès permis,
- ⇒ NO l'accès séquentiel n'est pas un mode d'accès permis,
- ⇒ UNKNOWN le compilateur ne sait pas si l'accès séquentiel est permis ou non,

 UNFORMATTED :

- ⇒ YES si le traitement du fichier en mode binaire est valide
- ⇒ NO si le traitement du fichier en mode binaire n'est pas valide,
- ⇒ UNKNOWN lorsque le compilateur ne sait pas si le traitement du fichier en mode binaire est permis ou non,

 WRITE :

- ⇒ YES un accès en écriture est permis,
- ⇒ NO un accès en écriture n'est pas permis,
- ⇒ UNKNOWN le compilateur ne sait pas si un accès en écriture est permis ou non.

Instruction CLOSE

```
CLOSE( [UNIT=]u,      &  
        IOSTAT=ios,   &  
        STATUS=st )
```

- ➡ UNIT : numéro de l'unité logique du fichier à fermer,
- ➡ IOSTAT : entier valorisé une fois la fermeture effectuée. Il est nul si l'opération s'est bien passée, non nul sinon,
- ➡ STATUS : état du fichier après fermeture,
 - ➡ DELETE : le fichier est détruit après fermeture. C'est la valeur par défaut si l'ouverture a été faite avec le paramètre STATUS="scratch",
 - ➡ KEEP : le fichier est conservé après fermeture. C'est la valeur par défaut si l'ouverture a été faite avec le paramètre STATUS différent de "scratch".

Exemples

```
integer ios
...
OPEN( UNIT=1,           &
      FILE="donnees",   &
      FORM="unformatted", &
      ACTION="read",     &
      POSITION="rewind",  &
      IOSTAT=ios )
if ( ios /= 0 ) ! Problème à l'ouverture
...
CLOSE( UNIT=1 )

OPEN( UNIT=2,           &
      FORM="formatted",  &
      ACCESS="direct",   &
      STATUS="scratch",  &
      ACTION="write",    &
      LRECL=9600,        &
      IOSTAT=ios )
if ( ios /= 0 ) ! Problème à l'ouverture
...
CLOSE( UNIT=2 )
```

12 – Annexe B : procédures intrinsèques

➡ Principales procédures intrinsèques

Principales procédures intrinsèques élémentaires

➡ **ABS** : retourne la valeur absolue de son argument.
Pour un complexe, retourne sa norme : $\sqrt{x^2 + y^2}$.

`ABS(-1) = 1; ABS(-1.5) = 1.5; ABS((3.,4.)) = 5.0`

➡ **ACHAR** : retourne le caractère de la table ASCII dont le rang est transmis en argument.

`ACHAR(88) = 'X'; ACHAR(42) = '*'`

➡ **ACOS** : retourne l'arc cosinus en radians de son argument réel.

`ACOS(0.54030231) = 1.0`

➡ **ADJUSTL** : cadre à gauche la chaîne passée en argument : supprime les blancs en tête ; complète à droite par des blancs.

`ADJUSTL('^^Fortran') = 'Fortran^^'`

➡ **ADJUSTR** : cadre à droite la chaîne passée en argument : supprime les blancs en fin ; complète à gauche par des blancs.

`ADJUSTR('Fortran^^') = '^^Fortran'`

➡ **AIMAG** : retourne la partie imaginaire du complexe passé en argument.

`AIMAG((2.,3.)) == 3.0`

➡ **AINT** : tronque le réel passé en argument.

`AINT(2.783) = 2.0; AINT(-2.783) = -2.0`

➡ **ANINT** : retourne, sous forme d'un réel, l'entier le plus proche du réel transmis.

`ANINT(2.783) = 3.0; ANINT(-2.783) = -3.0`

➡ **ASIN** : retourne l'arc sinus en radians de son argument réel.

`ASIN(0.84147098) = 1.0`

➡ **ATAN** : retourne l'arc tangente en radians de son argument réel.

`ATAN(1.5574077) = 1.0`

➡ **BIT_SIZE** : retourne le nombre de bits utilisés pour la représentation de l'entier passé en argument.

`BIT_SIZE(1) = 32`

👉 **BTEST** : permet de tester la valeur d'un bit d'un entier : l'entier ainsi que la position du bit à tester sont passés en argument.

`BTEST(8,3) = .true.; BTEST(10,2) = .false.`

👉 **CEILING** : retourne l'entier immédiatement supérieur au réel transmis en argument.

`CEILING(3.7) = 4, CEILING(-3.7) = -3`

👉 **CMPLX** : retourne un complexe dont les parties réelle et imaginaire sont transmises en argument.

`CMPLX(-3.) = -3.0+0.i; CMPLX(2,4.) = 2.0+4.0i`

👉 **CONJG** : retourne le complexe conjugué de celui passé en argument.

`CONJG((-3.0,4.0)) = -3.0-4.0i`

👉 **COS** : retourne le cosinus de l'angle passé en argument (exprimé en radians).

`COS(1.0) = 0.54030231`

👉 **COSH** : retourne le cosinus hyperbolique.

`COSH(1.0) = 1.5430806`

➡ **DBLE** : convertit en double précision l'argument transmis.

➡ **EXP** : retourne l'exponentiel de l'argument transmis.

$\text{EXP}(1.0) = 2.7182818$

➡ **FLOOR** : retourne l'entier immédiatement inférieur au réel transmis en argument.

$\text{FLOOR}(3.7) = 3, \text{FLOOR}(-3.7) = -4$

➡ **IACHAR** : retourne le rang dans la table ASCII du caractère transmis en argument.

$\text{IACHAR}('X') = 88; \text{IACHAR}('*') = 42$

➡ **IAND** : retourne l'entier dont la représentation binaire est obtenue en combinant à l'aide d'un "et logique" les bits des deux entiers transmis en argument.

$\text{IAND}(1,3) = 1; \text{IAND}(10,10) = 10$

➡ **IBCLR** : permet de forcer à zéro un bit d'un entier : l'entier ainsi que la position du bit à forcer sont passés en argument.

$\text{IBCLR}(14,1) = 12$

$\text{IBCLR}((/ 1,2,3,4 /), 31) = (/ 29,27,23,15 /)$

👉 **IBITS** : permet l'extraction d'une séquence de bits d'un entier. L'entier suivi de la position ainsi que la longueur pour effectuer l'extraction sont passés en argument.

`IBITS(14,1,3) = 7`

👉 **IBSET** : permet de forcer à 1 un bit d'un entier : l'entier ainsi que la position du bit à forcer sont passés en argument.

`IBSET(12,1) = 14`

`IBSET((/ 1,2,3,4 /), 0) = (/ 1,3,3,5 /)`

👉 **IEOR** : retourne l'entier dont la représentation binaire est obtenue en combinant à l'aide d'un "ou exclusif" les bits des deux entiers transmis en argument.

`IEOR(1,3) = 2; IEOR(10,10) = 0`

👉 **INDEX** : retourne la position d'une sous-chaîne dans une chaîne. La chaîne suivie de la sous-chaîne et du sens de la recherche sont fournis en argument.

`INDEX('FORTRAN','R') = 3`

`INDEX('FORTRAN','R',BACK=.true.) = 5`

👉 INT : convertit en entier l'argument transmis.

$\text{INT}(-3.7) = -3$; $\text{INT}(9.1/4.0) = 2$

👉 IOR : retourne l'entier dont la représentation binaire est obtenue en combinant à l'aide d'un "ou logique" les bits des deux entiers transmis en argument.

$\text{IOR}(1,3) = 3$

$\text{IOR}((/ 3,2 /), (/ 1,10 /)) = (/ 3,10 /)$

👉 ISHFT : permet d'effectuer un décalage des bits de l'entier passé en premier argument. Le deuxième argument indique le nombre de bits à décaler : son signe indique le sens du décalage (positif = gauche, négatif = droite). Les bits sortants sont perdus, les positions vacantes sont mises à zéro.

$\text{ISHFT}(3,1) = 6$; $\text{ISHFT}(3,-1) = 1$

👉 ISHFTC : idem ISHFT à la différence que le décalage est circulaire et s'effectue sur les n bits de droite de l'entier, n étant fourni en troisième argument (s'il est absent il est considéré égal au nombre de bits de l'entier).

$\text{ISHFT}(3,2,3) = 5$; $\text{ISHFT}(3,-2) = -1073741824$

- 👉 **LEN** : retourne la longueur de la chaîne de caractères transmise en argument.

```
CHARACTER(len=10) CH; LEN(CH) = 10
```

- 👉 **LEN_TRIM** : retourne la longueur de la chaîne de caractères transmise en argument sans considérer les blancs de fin.

```
LEN_TRIM('^^FORTAN^^') = 9; LEN_TRIM('^^^') = 0
```

- 👉 **LGE** : compare les deux chaînes de caractères transmises en argument : retourne **.true.** si la première chaîne est supérieure ou égale à la deuxième, **.false.** sinon.

```
LGE('MANET', 'MONET') = .false.
```

```
LGE('MANET Edouard', 'MANET') = .true.
```

- 👉 **LGT** : compare les deux chaînes de caractères transmises en argument : retourne **.true.** si la première chaîne est supérieure strictement à la deuxième, **.false.** sinon.

```
LGT('MANET', 'MANET') = .false.
```

👉 LLE : compare les deux chaînes de caractères transmises en argument : retourne `.true.` si la première chaîne est inférieure ou égale à la deuxième, `.false.` sinon.

```
LLE('MANET','MONET') = .true.
```

```
LLE('MANET','MANET') = .true.
```

👉 LLT : compare les deux chaînes de caractères transmises en argument : retourne `.true.` si la première chaîne est inférieure strictement à la deuxième, `.false.` sinon.

```
LLT('MANET','MANET') = .false.
```

👉 LOG : retourne le logarithme népérien de l'argument transmis.

```
LOG(2.7182818) = 1.0; LOG(10.0) = 2.3025851
```

👉 LOG10 : retourne le logarithme décimal de l'argument transmis.

```
LOG10(10.0) = 1.0; LOG10(10.E10) = 11.0
```

👉 MAX : retourne le maximum des nombres passés en argument.

```
MAX(-9.0,7.0,2.0) = 7.0
```

➡ **MIN** : retourne le minimum des nombres passés en argument.

`MIN(-9.0,7.0,2.0) = -9.0`

➡ **MOD** : retourne le reste de la division effectuée à l'aide des deux arguments fournis.

`MOD(3.0,2.0) = 1.0; MOD(-8,5) = -3`

➡ **NOT** : retourne l'entier dont la représentation binaire est obtenue en inversant les bits de l'entier transmis en argument.

`NOT(10) = -11`

➡ **REAL** : convertit en réel l'argument transmis.

`REAL(3) = 3.0`

➡ **REPEAT** : permet de concaténer **n** fois une chaîne de caractères.

`REPEAT('A',10) = 'AAAAAAAAAA'`

➡ **SCAN** : retourne la position du premier caractère d'une chaîne figurant parmi un ensemble de caractères donné. La recherche peut être faite dans les deux sens.

`SCAN('RENOIR','OI') = 4`

`SCAN('RENOIR','OI',BACK=.true.) = 5`

➡ **SIGN** : retourne le nombre dont la valeur absolue est celle du premier argument et le signe celui du deuxième.

`SIGN(-3.0,2.0) = 3.0`

➡ **SIN** : retourne le sinus de l'angle passé en argument (exprimé en radians).

`SIN(1.0) = 0.84147098`

➡ **SINH** : retourne le sinus hyperbolique.

`SINH(1.0) = 1.1752012`

➡ **SQRT** : retourne la racine carré de son argument.

`SQRT(5.0) = 2.236068010`

➡ **TAN** : retourne la tangente de l'angle passé en argument (exprimé en radians).

`TAN(1.0) = 1.5574077`

➡ **TANH** : retourne la tangente hyperbolique.

`TANH(1.0) = 0.76159416`

➡ **TRIM** : retourne la chaîne de caractères transmise débarrassée de ses blancs de fin.

`TRIM('PICASSO^^^^')` = 'PICASSO'

👉 **VERIFY** : retourne la position du premier caractère d'une chaîne ne figurant pas parmi un ensemble de caractères donné. La recherche peut être faite dans les deux sens.

```
VERIFY('RENOIR','OI') = 1
```

```
VERIFY('RENOIR','OI',BACK=.true.) = 6
```

13 – Annexe C : aspects obsolètes

➡ Aspects obsolètes

Aspects obsolètes

1. **IF arithmétique** : IF (ITEST) 10,11,12
==> IF--THEN--ELSE IF--ELSE--ENDIF
2. Branchement au END IF depuis l'extérieur (*)
==> se brancher à l'instruction suivante.
3. Boucles DO pilotées par réels : DO 10 R=1., 5.7, 1.3 (*)
4. Partage d'une instruction de fin de boucle :

```
DO 1 I=1,N
  DO 1 J=1,N
    A(I,J)=A(I,J)+C(J,I)
  1 CONTINUE
```

==> autant de CONTINUE que de boucles.

5. Fins de boucles autres que CONTINUE ou END DO
6. **ASSIGN** et le **GO TO** assigné : (*)

```
ASSIGN 10 TO intvar
....
ASSIGN 20 TO intvar
....
GO TO intvar
```

==> SELECT CASE ou IF/THEN/ELSE

(*) : aspects obsolètes déclarés hors norme par Fortran 95

7. **ASSIGN** d'une étiquette de FORMAT : (*)

ASSIGN 2 TO NF		CHARACTER(7),DIMENSION(4)::C
2 FORMAT (F9.2)	==>	I = 2; C(2) = '(F9.2)'
PRINT NF,TRUC		PRINT C(I),TRUC

8. **RETURN multiples** :

```
CALL SP1(X,Y,*10,*20)
...
10    ...
...
20    ...
...
SUBROUTINE SP1(X1,Y1,*,*)
...
RETURN 1
...
RETURN 2
...
```

==> SELECT CASE sur la valeur d'un argument retourné

9. **PAUSE** 'Montez la bande 102423 SVP' (*)

==> READ qui attend les données

10. **FORMAT(9H A éviter)** (*)

==> Constante littérale : FORMAT(' Recommandé')

(*) : aspects obsolètes déclarés hors norme par Fortran 95

Aspects obsolètes introduits par Fortran 95 :

1. Le "format fixe" du source
==> "format libre".
2. Le GO TO calculé (GO TO (10,11,12,...), *int_expr*)
==> select case.
3. L'instruction DATA placée **au sein** des instructions exécutables
==> **avant** les instructions exécutables.
4. *Statement functions* (sin_deg(x)=sin(x*3.14/180.))
==> procédures internes.
5. Le type CHARACTER*... dans les déclarations
==> CHARACTER(LEN=...)
6. Le type CHARACTER(LEN=*) de longueur implicite en retour d'une fonction
==> CHARACTER(LEN=len(str)).

14 – Annexe D : système de compilation

➡ Système de compilation

Système de compilation

La commande **f90** permet de générer un exécutable à partir de fichiers sources **Fortran**.

Celle-ci appelle un système de compilation faisant successivement appel à :

- ➡ un préprocesseur,
- ➡ un compilateur,
- ➡ un *loader* ou éditeur de liens.

La composante préprocesseur transforme le source **Fortran** en entrée au moyen de directives.

La composante compilateur analyse le source **Fortran** fourni (éventuellement transformé à l'étape précédente) avec :

- ➡ détection des erreurs de syntaxe,
- ➡ traduction du source en langage machine plus ou moins optimisé,
- ➡ production d'un module objet.

Enfin la dernière composante fait appel au *loader* qui récupère les modules objets précédemment créés et les regroupe pour produire un module exécutable.

Les différentes unités de programme constituant une application **Fortran** peuvent figurer dans un même fichier ou bien être réparties dans plusieurs fichiers. Ceux-ci doivent être suffixés par **.f** ou **.f90**.

Le compilateur suppose que le source est écrit avec le format fixe si le fichier est suffixé par **.f** et avec le format libre si le fichier est suffixé par **.f90**. Ce format d'écriture peut être explicité au moyen d'une option indépendamment du suffixe employé.

Les fichiers correspondant aux modules objets sont suffixés par **.o**.

Par défaut le module exécutable est stocké dans un fichier de nom **a.out** qu'il est possible de renommer à l'aide de l'option **-o nom_exécutable_désiré**.

L'option **-c** permet de conserver le ou les modules objets produits par le compilateur et d'inhiber l'étape du *loader*.

C'est la technique employée par l'utilitaire **make** qui, automatiquement, lance les compilations des différents fichiers source constituant l'application. Les modules objets obtenus sont *in fine* fournis au système de compilation pour la production du module exécutable.

Exemple

```
$ f90 -c source1.f90
$ f90 -c source2.f90
...
$ f90 -c sourcen.f90
$ f90 *.o -o a.exe
```

Il est possible d'archiver les fichiers `*.o` à l'aide de l'utilitaire `ar` dans le but de les regrouper dans un seul fichier dont le nom est de la forme `libxxx.a`.

Cela permet la constitution de bibliothèques de modules objets lesquelles sont transmises à la composante *loader* à l'aide des options `-L` et `-l` permettant de spécifier l'endroit où celles-ci sont stockées et leur noms.

Exemple

```
$ f90 -c source1.f90
$ f90 -c source2.f90
...
$ f90 -c sourcen.f90
$ ar -rv libexemple.a *.o
$ mv libexemple.a $HOME/lib
$ f90 -L$HOME/lib -lexemple -o a.exe
```

15 — Annexe E : exercices

- ➡ E.1 Exercices : énoncés
- ➡ E.2 Exercices : corrigés

Exercice 1

Écrire un programme permettant de résoudre le système de 2 équations à 2 inconnues :

$$\begin{cases} u_1x + v_1y = w_1 \\ u_2x + v_2y = w_2 \end{cases}$$

On pourra imprimer les solutions à l'aide de l'instruction :

```
PRINT *, 'X = ', X, ', Y = ', Y
```

Exercice 2

Écrire un programme permettant de calculer les racines du trinôme du 2^{nd} degré : $ax^2 + bx + c$. On s'assurera que a est non nul. Les racines, si elles existent, pourront être imprimées à l'aide de l'instruction :

```
PRINT *, 'X1 = ', X1, ', X2 = ', X2
```

Exercice 3

Écrire un programme calculant le nombre d'Or. Celui-ci peut être obtenu à partir de la suite de Fibonacci u_n définie par :

$$u_0 = 1$$

$$u_1 = 1$$

...

$$u_{n+1} = u_n + u_{n-1}$$

La suite $(\frac{u_{n+1}}{u_n})$ converge vers le nombre d'Or.

Exercice 4

Écrire un programme permettant de déterminer les nombres premiers dans l'intervalle $[1, n]$ à l'aide du crible d'Ératosthène. Il consiste à former une table avec tous les entiers naturels compris entre **2** et **n** et à rayer (mise à zéro), les uns après les autres, les entiers qui ne sont pas premiers de la manière suivante : dès que l'on trouve un entier qui n'a pas encore été rayé, il est déclaré premier, et on raye tous les multiples de celui-ci.

À la fin du procédé, les nombres non barrés sont des nombres premiers.

On tiendra compte du fait qu'un nombre donné peut déjà avoir été éliminé en tant que multiple de nombres précédents déjà testés.

Par ailleurs, on sait que l'on peut réduire la recherche aux nombres de **2** à $\sqrt{n}$ (si un entier non premier est strictement supérieur à $\sqrt{n}$ alors il a au moins un diviseur inférieur à $\sqrt{n}$ et aura donc déjà été rayé).

Exercice 5

Écrire un programme permettant de trier un vecteur de nombres en ordre croissant puis décroissant. On s'appuiera sur l'algorithme appelé *tri à bulle* qui consiste à comparer 2 éléments consécutifs et à les intervertir si nécessaire.

Si après avoir terminé l'exploration du tableau au moins une interversion a été effectuée, on renouvelle l'exploration, sinon le tri est terminé.

Exercice 6

Écrire un programme permettant d'effectuer le produit de 2 matrices **A** et **B**. Leurs profils seront définis à l'aide de constantes symboliques. La matrice résultat **C** sera imprimée à l'écran ligne par ligne avec l'instruction **PRINT** puis stockée dans un fichier binaire que l'on nommera « **exo6.matrice** ».

Exercice 7

Le fichier texte séquentiel « **musiciens** » est constitué de plusieurs enregistrements, chacun contenant un nom de musicien suivi de ses années de naissance et de mort.

Écrire un programme dont le but est de lire le fichier « **musiciens** » et de stocker les enregistrements lus dans un fichier binaire à accès direct que l'on nommera « **musiciens.bin** ».

Exercice 8

Imprimer l'enregistrement du fichier « **musiciens** » dont le rang est entré au clavier. Son extraction sera effectuée à partir d'un fichier temporaire à accès direct, image du précédent.

On permettra la saisie de plusieurs rangs.

Exercice 9

Les enregistrements des fichiers séquentiels « `index_naissance.dat` » et « `index_deces.dat` » sont constitués d'une date de naissance (ou de décès) d'un musicien suivi de son rang dans le fichier « `musiciens.bin` » créé à l'exercice 7.

Écrire un programme permettant d'imprimer le ou les musiciens dont la date de naissance ou de mort est saisie au clavier. Le type de date désirée sera préalablement déterminé.

La sélection des enregistrements répondant aux choix spécifiés, s'effectuera par l'intermédiaire du fichier d'index correspondant au type de date.

On offrira la possibilité d'effectuer plusieurs recherches.

Exercice 10

Le but de cet exercice est de transformer la matrice stockée dans le fichier binaire « `exo6.matrice` ». Cette transformation consiste à modifier chaque élément à l'aide d'une fonction paramétrable de la forme $y = f(x)$.

On définira plusieurs fonctions de ce type. La valeur d'un entier lu dans une *namelist* indiquera la fonction à transmettre en argument de la procédure chargée d'effectuer la transformation.

Exercice 11

Trier les vecteurs lignes puis les vecteurs colonnes d'une matrice en utilisant l'algorithme *tri à bulle* et la matrice stockée dans le fichier binaire « `exo6.matrice` ».

On se définira une procédure effectuant le tri (croissant ou décroissant) des différents vecteurs au moyen d'une procédure interne.

Corrigé de l'exercice 1

```
1  program systeme
2      implicit none
3      real u1,u2
4      real v1,v2
5      real w1,w2
6      real delta, delta_x, delta_y
7      real x,y
8
9          ! Valorisation des coefficients.
10     u1 = 2; u2 = 4
11     v1 = 5; v2 = 11
12     w1 = 7; w2 = 6
13
14     ! Calcul du déterminant principal.
15     delta = u1*v2 - u2*v1
16     if ( delta < 1e-6 ) then
17         print *, "Le système n'a pas de solution unique."
18         stop 4
19     end if
20
21     ! Calcul du déterminant en x.
22     delta_x = w1*v2 - w2*v1
23
24     ! Calcul du déterminant en y.
25     delta_y = u1*w2 - u2*w1
26
27     ! calcul des solutions.
28     x = delta_x/delta
29     y = delta_y/delta
30
31     ! Impression des solutions.
32     print *, "x = ", x, ", y = ", y
33
34 end program systeme
```

Corrigé de l'exercice 2

```
1 program trinome
2   implicit none
3   real, parameter :: epsilon = 1e-6
4   real a, b, c
5   real delta, r_delta, x1, x2
6
7   ! Valorisation des coefficients.
8   a = 3.; b = 7.; c = -11.
9
10  ! a doit être non nul.
11  if ( a > -epsilon .and. a < epsilon ) &
12    stop "a doit être non nul."
13
14  ! calcul du déterminant.
15  delta = b*b - 4*a*c
16  ! cas du déterminant négatif.
17  if( delta < -epsilon ) stop "Pas de racine réelle."
18
19  ! cas du déterminant nul.
20  if ( delta > -epsilon .and. delta < epsilon ) then
21    x1 = -b/(2*a); x2 = x1
22  else ! cas du déterminant positif.
23    r_delta = sqrt( delta )
24    x1 = (-b - r_delta)/(2*a); x2 = (-b + r_delta)/(2*a)
25  end if
26
27  ! Impression des racines.
28  print *, "x1 = ", x1, ", x2 = ", x2
29 end program trinome
```

Corrigé de l'exercice 3

```
1  program nombre_dor
2      implicit none
3      real, parameter    :: epsilon = 1.e-5
4      real                :: u_prec, u_cour
5      real                :: v_prec, v_cour
6      real                :: somme
7      real                :: nombre_or
8
9      nombre_or = (1. + sqrt(5.))/2.
10
11     u_prec = 1.; u_cour = 1.
12     do
13         v_prec = u_cour/u_prec
14         somme   = u_cour + u_prec
15         u_prec = u_cour
16         u_cour = somme
17         v_cour = u_cour/u_prec
18         if ( abs( (v_cour-v_prec)/v_prec ) < epsilon ) exit
19     end do
20
21     print*, "Limite de la suite (vn) : ", v_cour, &
22           "Nombre d'or : ", nombre_or
23 end program nombre_dor
```

Corrigé de l'exercice 4

```
1  program eratosthene
2      implicit none
3      integer, parameter      :: n = 1000
4      integer, dimension(n) :: tab_nombres
5      integer                  :: imax
6      integer i, j
7
8      do i=2,n
9          tab_nombres(i) = i
10     end do
11
12     imax = int(sqrt(real(n)))
13     do i=2,imax
14         if( tab_nombres(i) /= 0 ) then
15             do j=i+1,n
16                 if ( tab_nombres(j) /= 0 .and. &
17                     mod( tab_nombres(j), i ) == 0 ) &
18                     tab_nombres(j) = 0
19             end do
20         end if
21     end do
22
23     print *, "Les nombres premiers entre 1 et ", n, " sont :"
24     do i=2,n
25         if ( tab_nombres(i) /= 0 ) print *, tab_nombres(i)
26     end do
27  end program eratosthene
```

Corrigé de l'exercice 5

```
1  program triabulle
2      implicit none
3      integer, parameter :: croissant=1, decroissant=2, n=10
4      real, dimension(n) :: tab
5      real                :: temp
6      logical             :: tri_termine, expr1, expr2
7      integer             :: sens, i
8
9      ! Valorisation du vecteur
10     data tab/0.76, 0.38, 0.42, 0.91, 0.25, &
11           0.13, 0.52, 0.69, 0.76, 0.98/
12     do sens=croissant, decroissant      ! Sens du tri
13         do                               ! Tri
14             tri_termine = .true.
15             do i=2,n
16                 expr1 = sens == croissant .and. tab(i-1) > tab(i)
17                 expr2 = sens == decroissant .and. tab(i-1) < tab(i)
18                 if (expr1 .or. expr2) then
19                     tri_termine = .false.
20                     temp = tab(i-1); tab(i-1) = tab(i); tab(i) = temp
21                 end if
22             end do
23             if (tri_termine) exit
24         end do
25         ! Impression du vecteur trié
26         if (sens == croissant) print*, "Tri croissant "
27         if (sens == decroissant) print*, "Tri décroissant "
28         print*, tab
29     end do
30 end program triabulle
```

Corrigé de l'exercice 6

```
1  program produit_matrice
2      implicit none
3      integer, parameter    :: n = 10, m = 5, p = 3
4      real, dimension(n,m) :: a
5      real, dimension(m,p) :: b
6      real, dimension(n,p) :: c
7      integer               :: i,j,k
8
9      ! Valorisation des matrices A et B
10     data      ((a(i,j),j=1,m),i=1,n)/ &
11         0.00, 0.38, 0.42, 0.91, 0.25, &
12         0.13, 0.52, 0.69, 0.76, 0.98, &
13         0.76, 0.83, 0.59, 0.26, 0.72, &
14         0.46, 0.03, 0.93, 0.05, 0.75, &
15         0.53, 0.05, 0.85, 0.74, 0.65, &
16         0.22, 0.53, 0.53, 0.33, 0.07, &
17         0.05, 0.67, 0.09, 0.63, 0.63, &
18         0.68, 0.01, 0.65, 0.76, 0.88, &
19         0.68, 0.38, 0.42, 0.99, 0.27, &
20         0.93, 0.07, 0.70 ,0.37, 0.44/
21
22     data ((b(i,j),j=1,p),i=1,m)/ &
23         0.76, 0.16, 0.9047, &
24         0.47, 0.48, 0.5045, &
25         0.23, 0.89, 0.5163, &
26         0.27, 0.90, 0.3190, &
27         0.35, 0.06, 0.9866/
```

```
28      ! Produit de matrice.
29      do i=1,n
30          do j=1,p
31              c(i,j) = 0.
32              do k=1,m
33                  c(i,j) = c(i,j) + a(i,k) * b(k,j)
34              end do
35          end do
36      end do
37
38      ! Impression de la matrice c.
39      do i=1,n
40          print *, c(i,:)
41      end do
42
43      ! Écriture de la matrice c dans un fichier.
44      open( unit=1,          file="exo6.matrice", &
45            status="replace", form="unformatted", &
46            action="write" )
47      write( 1 ) c
48      close( unit = 1)
49      end program produit_matrice
```

Corrigé de l'exercice 7

```
1  program ecriture_musiciens
2      character(len=80) :: mus
3      integer          :: ios_mus, val_recl
4      integer          :: numrec
5
6      ! Ouverture du fichier des musiciens
7      ! ainsi que d'un fichier en écriture
8      ! à accès direct dans lequel on
9      ! va recopier le fichier précédent.
10     open( unit=1,          file="musiciens",      &
11           form="formatted", status="old",         &
12           action="read",   position="rewind" )
13     ! Calcul de la taille des enregistrements du fichier
14     ! binaire en sortie. (dépend du compilateur).
15     inquire( iolength=val_recl ) mus
16     open( unit=2,          file="musiciens.bin", &
17           status="replace",                                &
18           form="unformatted", access="direct",           &
19           action="write",   recl=val_recl )
20
21     ! On effectue la copie.
22     numrec = 0
23     read( unit=1, fmt='(a)', iostat=ios_mus ) mus
24     do while ( ios_mus == 0 )
25         numrec = numrec + 1
26         write( unit=2, rec=numrec) mus
27         read( unit=1, fmt='(a)', iostat=ios_mus ) mus
28     end do
29     close( unit=1 )
30     close( unit=2 )
31 end program ecriture_musiciens
```

Corrigé de l'exercice 8

```
1 program musiciens
2   implicit none
3   character(len=80) :: mus
4   integer           :: ios_mus, ios_stdin
5   integer           :: numrec, rang
6
7   ! Ouverture du fichier des musiciens
8   ! ainsi que d'un fichier temporaire
9   ! à accès direct dans lequel on
10  ! va recopier le fichier précédent.
11  open( unit=1,          file="musiciens",    &
12        form="formatted", status="old",      &
13        action="read",   position="rewind" )
14  open( unit=2,          status="scratch",    &
15        form="formatted", access="direct",    &
16        action="readwrite", recl=80 )
17
18  ! On effectue la copie.
19  numrec = 0
20  read( unit=1, fmt='(a)', iostat=ios_mus ) mus
21  do while ( ios_mus == 0 )
22    numrec = numrec + 1
23    write( unit=2, rec=numrec, fmt='(a)' ) mus
24    read( unit=1, fmt='(a)', iostat=ios_mus ) mus
25  end do
26  close( unit=1 )
```

```
27  ! On demande un rang de musicien.
28
29  print *, "Entrez le rang d'un musicien : "
30  read( unit=*, &
31        fmt=*, &
32        iostat=ios_stdin ) rang
33  do while ( ios_stdin == 0 )
34    read( unit=2, &
35          rec=rang, &
36          fmt='(a)', &
37          iostat=ios_mus ) mus
38    if ( ios_mus /= 0 ) then
39      print *, "Le musicien de rang ", &
40              rang, " n'existe pas"
41    else
42      print '("musicien de rang",i3," ==> ", a)', &
43              rang, trim(mus)
44    end if
45    print '(/, "Entrez le rang d''un musicien :")'
46    read( unit=*, fmt=*, iostat=ios_stdin ) rang
47  end do
48  close( unit=2 )
49  end program musiciens
```

Corrigé de l'exercice 9

```
1 program sequentiel_indexe
2   implicit none
3   character(len=19), dimension(2), parameter :: f_index = &
4       (/ "index_naissance.dat", "index_deces.dat"  /)
5   character(len=80) :: mus
6   character(len=50) :: prompt_date
7   integer           :: ios_u1, ios_u2, ios_u3
8   integer           :: numrec, ios_index, ios_stdin
9   integer           :: date_saisie, date_lue
10  integer           :: critere, val_rec
11  logical            :: trouve
12
13  ! Ouverture du fichier des musiciens à accès direct en lecture
14  ! et des fichiers d'index.
15  open ( unit=1,          file = f_index(1),          &
16         status="old",    form="formatted", action="read", &
17         iostat=ios_u1 )
18  open ( unit=2,          file = trim(f_index(2)),    &
19         status="old",    form="formatted", action="read", &
20         iostat=ios_u2 )
21  ! Calcul de la taille des enregistrements du fichier
22  ! créé à l'exercice 7. (dépend du compilateur).
23  inquire( iolength=val_rec ) mus
24  open ( unit=3,          file="musiciens.bin",        &
25         status="old",    form="unformatted",         &
26         access="direct", action="read", recl=val_rec, &
27         iostat=ios_u3 )
28  if ( ios_u1 /= 0 .OR. ios_u2 /= 0 .OR. ios_u3 /= 0 ) &
29      stop "Erreur à l'ouverture des fichiers"
```

```
30 trait: &
31   do
32     do
33       print*, '-----'
34       print*, 'Choix du critère de recherche : '
35       print*, '- par date de naissance (1)'
36       print*, '- par date de décès      (2)'
37       print*, '- QUITTER                (3)'
38       print*, '-----'
39       read(*, *, IOSTAT=ios_stdin) critere
40       if ( ios_stdin < 0 ) exit trait
41       if ( ios_stdin > 0 ) then
42         print '(/,a,/)', "Erreur dans la saisie"
43       else
44         exit
45       end if
46     end do
47     select case (critere)
48     case(1) ! Recherche par date de naissance.
49       prompt_date = &
50         "Entrer une date de naissance d'un musicien"
51       rewind( unit=critere )
52     case(2) ! Recherche par date de décès.
53       prompt_date = &
54         "Entrer une date de décès d'un musicien"
55       rewind( unit=critere )
56     case default ! Quitter
57       exit
58     end select
```

```
59 ! Lecture d'une date.
60   do
61     print *, trim(prompt_date)
62     read(*, *, IOSTAT=ios_stdin) date_saisie
63     if( ios_stdin < 0 ) exit trait
64     if( ios_stdin > 0 ) then
65       print '(/,a,/)', "Date erronée!"
66     else
67       exit
68     end if
69   end do
70   ! Recherche de la date saisie dans le fichier d'index.
71   trouve = .false.
72   read( unit=critere, fmt=*,                               &
73         iostat=ios_index ) date_lue, numrec
74   do while( ios_index == 0 )
75     if ( date_lue == date_saisie ) then
76       ! On lit l'enregistrement correspondant.
77       trouve = .true.
78       read( unit=3, rec=numrec ) mus
79       print *,trim(mus)
80     end if
81     read( unit=critere, fmt=*,                               &
82           iostat=ios_index ) date_lue, numrec
83   end do
84   if ( .not. trouve ) &
85     print *, "Aucun musicien ne répond au critère indiqué."
86   print '(/)'
87 end do trait
88 close( unit=1 ); close( unit=2 ); close( unit=3 )
89 end program sequentiel_indexe
```

Corrigé de l'exercice 10

```

1  program mat_transf
2      implicit none
3      integer, parameter    :: n = 10, m = 3
4      real, dimension(n,m) :: mat
5      integer               :: choix_methode, ios, num_ligne
6      real, external        :: carre, identite, logarithme
7      real, intrinsic       :: sqrt
8      namelist/methode/choix_methode
9
10     ! Ouverture du fichier contenant la matrice.
11     open( unit=1,          file="exo6.matrice", &
12           form="unformatted", action="read",      &
13           status="old",    position="rewind",    &
14           iostat=ios )
15     if (ios /= 0) &
16         stop 'Erreur à l''ouverture du fichier "exo6.matrice"'
17     ! Lecture de la matrice.
18     read(1) mat
19     close(1)
20     ! Ouverture du fichier contenant
21     ! la namelist "methode".
22     open( unit=1,          file="exo10.namelist", &
23           form="formatted", action="read",      &
24           status="old",    position="rewind",    &
25           iostat=ios )
26     if (ios /= 0) &
27         stop 'Erreur à l''ouverture du fichier "exo10.namelist"'
28     read( unit=1, nml=methode )
29     close( unit=1 )

```

```
30      ! Transformation de la matrice à l'aide
31      ! de la méthode choisie.
32
33      select case( choix_methode )
34      case (1)
35          call transform( mat, n, m, identite )
36      case (2)
37          call transform( mat, n, m, carre )
38      case (3)
39          call transform( mat, n, m, sqrt )
40      case (4)
41          call transform( mat, n, m, logarithme )
42      end select
43
44      ! Sauvegarde de la matrice transformée dans
45      ! le fichier "exo6_matrice_transf".
46
47      open( unit=1,          file="exo6_matrice_transf", &
48            form="formatted", action="write",          &
49            status="replace", iostat=ios )
50
51      if ( ios /= 0 ) &
52          stop "Erreur lors de l'ouverture &
53              &du fichier ""exo6_matrice_transf""
54
55      do num_ligne=1,n
56          write( unit=1, fmt='(3f10.6)' ) mat(num_ligne,:)
57      end do
58      close( unit=1 )
59      end program mat_transf
```

```
60      ! Procédure de transformation.
61  subroutine transform( t, n, m, f )
62      implicit none
63      integer          :: n, m, i, j
64      real, dimension(n,m) :: t
65      real              :: f
66
67      do i=1,n
68          do j=1,m
69              t(i,j) = f(t(i,j))
70          end do
71      end do
72  end subroutine transform
73
74      ! Définitions des fonctions de transformation.
75  function identite(x)
76      implicit none
77      real x, identite
78      identite = x
79  end function identite
80
81  function carre(x)
82      implicit none
83      real x, carre
84      carre = x*x
85  end function carre
86
87  function logarithme(x)
88      implicit none
89      real x, logarithme
90      logarithme = log(x)
91  end function logarithme
```

Corrigé de l'exercice 11

```
1 program tri_matrice
2   implicit none
3   integer, parameter :: n=10, m=3
4   real, dimension(n,m) :: mat
5   integer :: ios
6   integer :: i, j
7       ! Lecture de la matrice à trier.
8   open( unit=1, &
9       file="exo6.matrice", &
10      form="unformatted", &
11      status="old", &
12      action="read", &
13      position="rewind", &
14      iostat=ios )
15   if ( ios /= 0 ) stop "Erreur à l'ouverture du fichier &
16       &"exo6.matrice""
17   read( unit=1 ) mat; close( unit=1 )
18   call tri( mat, n, m ) ! Tri de la matrice lue.
19       ! Écriture de la matrice triée.
20   open( unit=1, file="exo11.matrice_triee", &
21       form="formatted", status="replace", &
22       action="write", position="rewind", &
23       iostat=ios )
24   if ( ios /= 0 ) stop "Erreur à l'ouverture du fichier &
25       &"exo11.matrice_triee""
26   do i=1,n
27       write( unit=1, fmt='(3F7.3)' ) mat(i,:)
28   end do
29   close( unit=1 )
30 end program tri_matrice
```

```
31      ! Procédure de tri.
32  subroutine tri( mat, n, m )
33      implicit none
34      integer          :: n, m
35      real, dimension(n,m) :: mat
36      integer          :: ligne, col
37
38      do ligne=1,n      ! Tri des lignes.
39          call tri_vecteur( mat(ligne,:), m )
40      end do
41      do col=1,m        ! Tri des colonnes.
42          call tri_vecteur( mat(:,col), n )
43      end do
44      contains
45      ! Procédure de tri d'un vecteur.
46  subroutine tri_vecteur( v, n )
47      integer          :: n, i
48      real, dimension(n) :: v
49      logical          :: tri_termine
50      do
51          tri_termine = .true.
52          do i=2,n
53              if ( v(i) > v(i-1) ) then
54                  tri_termine = .false.
55                  ! Utilisation de sections régulières pour
56                  ! effectuer l'interversion.
57                  v(i-1:i) = v(i:i-1:-1)
58              end if
59          end do
60          if (tri_termine) exit
61      end do
62  end subroutine tri_vecteur
63  end subroutine tri
```

Index

– Symboles –

*	54, 55
**	54, 55
*.o	229
+	54, 55
-	54, 55
-L	229
-c	228
-l	229
-o nom_exécutable_désiré	228
.AND.	59, 60
.EQ.	58
.EQV.	59, 60
.GE.	58
.GT.	58
.LE.	58
.LT.	58
.NE.	58
.NEQV.	59, 60
.NOT.	59
.NOT.l	60
.OR.	59, 60
.f	227, 228
.f90	227, 228
.o	228
/	54, 55
//	61
/=	58
<	58
<=	58
=	48, 49, 62, 85
==	58
>	58
>=	58
Éléments syntaxiques	31
Énoncés	231
CHARACTER(LEN=*)	225
GO TO calculé	225
RETURN multiples	224

– A –

ABS	211
accès direct	136–139
ACCESS	136, 137
ACCESS : INQUIRE	202
ACCESS : OPEN	194
ACHAR	165, 211
ACOS	211
ACTION : INQUIRE	202
ACTION : OPEN	195
ADJUSTL	211

ADJUSTR	211
ADVANCE : READ	198
ADVANCE : WRITE	200
AIMAG	212
AIN'T	212
ANINT	212
ANSI	10
ar	229
argument procédure	169, 170
argument tableau	166–168
arguments d'appel	159, 161, 162
arguments muets	159
ASA	10
ASCII - table	27
ASIN	212
ASSIGN	223, 224
assign	223
assumed-size array	167, 168
assumed-size string	164, 165
ATAN	212

– B –

BACKSPACE	151
BACSPACE	149
Bases de numération	19
bibliographie	14
bibliothèque	229
BIT_SIZE	212
BLANK : INQUIRE	202
BLANK : OPEN	196
bloc	94
BLOCK DATA	181, 182
BTEST	213
buffer	94

– C –

CALL	161
CASE	69
CASE DEFAULT	69
CEILING	213
CHARACTER	37, 39, 50, 61, 62, 69
CHARACTER*	225
CLOSE	95, 209
CLOSE : IOSTAT	208
CLOSE : STATUS	208
CLOSE : UNIT	208
CMPLX	213
commentaires	32
COMMON	178, 179, 181, 182, 184, 186–188

COMMON blanc	178
compatibilité	11
compilateur	227
COMPLEX	37, 44
CONJG	213
Constantes chaînes de caractères	
45	
Constantes complexes	44
Constantes entières	41
Constantes littérales	41
Constantes réelles : double	
précision	43
Constantes réelles : simple	
précision	42
Constantes symboliques	49
CONTAINS	171–173
Corrigés	236
COS	213
COSH	213
CYCLE	78

– D –

Déclaration	80
Déclarations	38–40, 49, 50
Définitions (rang, profil, étendue)	
82	

DATA	46, 88, 225
DBLE	214
DELIM	134
DELIM : INQUIRE	203
DELIM : OPEN	196
descripteur /	125
descripteur de format	110
descripteur de format : <i>Literal</i>	
<i>string</i>	122
descripteur de format A	108, 109,
120, 121	
descripteur de format E	106, 115
descripteur de format E, D	114
descripteur de format EN	116
descripteur de format ES	117
descripteur de format F	105, 112
descripteur de format I	104, 111
descripteur de format L	107, 119
descripteurs	100
descripteurs de contrôle	123, 124
descripteurs de format	103
descripteurs de format SS,SP,S	118
destruction fichier	145
Différents types	36
DIMENSION	37, 80, 81
direct	95
DIRECT : INQUIRE	203
DO	76, 86, 88
documentation	15, 17

DOUBLE PRECISION	37
Durée de vie des identificateurs	174

– E –

ELSE	67
END	67
END : READ	198
END BLOCK DATA	182
END SELECT	69
ENDFILE	149, 151
enregistrement logique	94
entrée standard	155
EOR : READ	198
EQUIVALENCE	50, 187
ERR : INQUIRE	203
ERR : READ	198
ERR : WRITE	200
Exercices	231, 236
EXIST : INQUIRE	203
EXIT	76
EXP	214
EXTERNAL	37, 169

– F –

f90	227
facteur de répétition	126
fichier : positionnement	149–151
fichier binaire	137
fichier binaire séquentiel	97, 98
fichier destruction	145
fichier interne	146–148
fichier temporaire	144
fichier texte	139
fichier texte séquentiel	99, 102
FILE : INQUIRE	203
FILE : OPEN	193
FLOOR	214
FMT	99
FMT : READ	197
FMT : WRITE	199
fonction	162
fonction : statement function	225
FORM : INQUIRE	204
FORM : OPEN	195
format : facteur de répétition	126
FORMAT : instruction	156
format : réexploration	127–129
format d'édition	100
format fixe	32, 225
format libre	130–132, 225
FORMAT(9H A éviter)	224
FORMATTED : INQUIRE	204
Fortran 95	14, 223–225
Fortran Market	17

FUNCTION 162, 163

– G –

Généralités 19, 21, 29–31
GO TO 223
GO TO calculé 225
GOTO 71

– I –

IACHAR 214
IAND 214
IBCLR 214
IBITS 215
IBSET 215
ICHAR 165
Identificateurs 35
IEOR 215
IF 67, 76, 78
IF arithmétique 223
IMPLICIT NONE 40
INCLUDE 188, 190
INDEX 215
Initialisation 46, 85
INQUIRE 152, 153, 201
INQUIRE : ACCESS 202
INQUIRE : ACTION 202
INQUIRE : BLANK 202
INQUIRE : DELIM 203
INQUIRE : DIRECT 203
INQUIRE : ERR 203
INQUIRE : EXIST 203
INQUIRE : FILE 203
INQUIRE : FORM 204
INQUIRE : FORMATTED 204
INQUIRE : IOSTAT 204
INQUIRE : NAME 204
INQUIRE : NAMED 205
INQUIRE : NEXTREC 205
INQUIRE : NUMBER 205
INQUIRE : OPENED 205
INQUIRE : PAD 205
INQUIRE : POSITION 206
INQUIRE : READ 206
INQUIRE : READWRITE 206
INQUIRE : RECL 206
INQUIRE : SEQUENTIAL 207
INQUIRE : UNFORMATTED 207
INQUIRE : UNIT 202
INQUIRE : WRITE 207
INT 216
INTEGER . 37, 40, 57, 62, 69, 73,
86
INTRINSIC 37, 169
IOR 216

IOSTAT 137
IOSTAT : CLOSE 208
IOSTAT : INQUIRE 204
IOSTAT : OPEN 193
IOSTAT : READ 198
IOSTAT : WRITE 200
ISHFT 216
ISHFTC 216
ISO 11

– J –

Jeu de caractères 29

– L –

LEN 164, 217
LEN_TRIM 217
Les itérations 71
Les tests 67
LGE 217
LGT 217
libxxx.a 229
LLE 218
LLT 218
loader 227
LOG 218
LOG10 218
LOGICAL 37, 67, 69, 76

– M –

make 228
Manipulation de tableaux 89
MAX 218
MIN 219
MOD 219
module exécutable 227
module objet 227
mosaic 17

– N –

NAME : INQUIRE 204
NAMED : INQUIRE 205
NAMELIST 99, 133–135
netscape 17
NEXTREC : INQUIRE 205
NML 99, 134
NML : READ 197
NML : WRITE 199
normalisation 10
norme 10
NOT 219
NUMBER : INQUIRE 205

— O —

Opérateur d'affectation	62
Opérateur de concaténation ...	61
Opérateurs arithmétiques	54
Opérateurs et expressions .	54, 58, 59, 61, 62, 64
Opérateurs logiques	59
Opérateurs relationnels	58
OPEN	95, 97, 193, 209
OPEN : ACCESS	194
OPEN : ACTION	195
OPEN : BLANK	196
OPEN : DELIM	196
OPEN : FILE	193
OPEN : FORM	195
OPEN : IOSTAT	193
OPEN : PAD	196
OPEN : POSITION	195
OPEN : RECL	195
OPEN : STATUS	194
OPEN : UNIT	193
OPENED : INQUIRE	205

— P —

PAD : INQUIRE	205
PAD : OPEN	196
PARAMETER	37, 49, 186
PAUSE	224
POSITION	136, 137
POSITION : INQUIRE	206
POSITION : OPEN	195
positionnement dans un fichier	149–151
préprocesseur	227
PRINT	155
Priorité des opérateurs	64
procédure interne	171–173
procédures intrinsèques	176

— R —

READ : ADVANCE	198
READ : END	198
READ : EOR	198
READ : ERR	198
READ : FMT	197
READ : INQUIRE	206
READ : IOSTAT	198
READ : NML	197
READ : REC	198
READ : SIZE	198
READ : UNIT	197
READWRITE : INQUIRE	206
REAL	37, 40, 42, 57, 62, 219

REC	137
REC : READ	198
REC : WRITE	200
RECL	136, 137
RECL : INQUIRE	206
RECL : OPEN	195
REPEAT	219
Représentation des données	21
return	162
REWIND	149, 150
RS/6000	15

— S —

séquentiel	95
SAVE	37, 174, 179, 181, 184
SCAN	219
scratch	144
SELECT CASE	69, 76, 78
SEQUENTIAL : INQUIRE ...	207
SIGN	220
SIN	220
SINH	220
SIZE : READ	198
sortie standard	155
SQRT	220
Statement functions	225
STATUS : CLOSE	145, 208
STATUS : OPEN	194
STOP	30
Structures de contrôle	67, 71
subroutine	161
Syntaxe	38

— T —

table ASCII	27
Tableaux	80, 82, 85, 89
tampon	94
TAN	220
TANH	220
THEN	67
TRIM	220

— U —

UNFORMATTED : INQUIRE	207
UNIT : CLOSE	208
UNIT : INQUIRE	202
UNIT : OPEN	193
UNIT : READ	197
UNIT : WRITE	199
UNIT=*	155
Unité de programme	30
unité logique	95

— V —	
variable	36
VERIFY	221

— W —	
WHILE	74
WRITE : ADVANCE	200

WRITE : ERR	200
WRITE : FMT	199
WRITE : INQUIRE	207
WRITE : IOSTAT	200
WRITE : NML	199
WRITE : REC	200
WRITE : UNIT	199
WWW	17