Chapter 1 3rd Generation Languages for Statistics (by G. Mélard)

1.0 INTRODUCTION

FORTRAN

History

Fortran IV	1966	change programs if possible
Fortran 77	1978	just read them
Fortran 90	1991	nice
Fortran 95	1996	wait a little bit

Be careful! Fortran 77, 90 ≠ Fortran

It is still possible to write "spaghetti" code but this is true in any language (even Pascal or C)

Since Fortran 77, Fortran is a STRUCTURED PROGRAMMING LANGUAGE (like Pascal)

CONTENTS OF THE CHAPTER

1.1 Basic algorithmic language

1.2 Fortran primer

1.3 Introduction to Fortran 90

1.4 Advanced study of Fortran 90 and elements of Fortran 95

1.5 Instructions of Fortran IV and Fortran 77 to be deciphered

1.6 Use of scientific libraries

1.7 Preparation of test data sets

1.1 ALGORITHMS

Excerpts from the Table of contents of the course of G. Mélard (2ème candidature en sciences économiques)

Introduction

Concept of variables and constants

Expressions and assignment statements

Input and output statements

Sequence or block structure

Conditional structures

Repetitive structures

Use of a counter

Use of an accumulator

Loop with counter

Other repetitive structures

Example

Tables

Modules and parameters

1.2 FORTRAN PRIMER

Types

Constants

Variables

Expressions

Assignment

Data input

Data output

Sequence structure

Branching

Loop

Arrays

Program appearance

Complete example 1

Complete example 2

Complete example 3

Remark 1.

Remark 2.

Remark 3.

Test run

TYPES

REAL	*or* REAL(4)
REAL(8)	*2 times more accurate*
INTEGER	*don't abuse !*
CHARACTER	Character strings (*chaînes de caractères)*

Remarks.

1. Simple precision : 4 bytes => precision of about 7 digits (" chiffres décimaux ").

2. Double precision, generally 8 bytes, about 13 digits.

3. Integer : just for counts (and be careful), often 4 bytes, sometimes just 2 (=> limitation!).

CONSTANTS

Examples

"REAL"	1.0	0.5E-4	0.5D-4
"CHARACTER"	'Moyenne'		
"INTEGER"	10		

VARIABLES

1) name of 1-31 letters/digits (" chiffres ") or character _ (underscore, "souligné") and starting with a letter

2) either upper case (" majuscules ") or lower case (" minuscules ") but unaccentuated letters

3) variables need to be declared

4) for variables of type characters, add the number *n* of characters (LEN=*n*)

Examples

```
REAL :: S1, S2, DONNEE
```

(not *donnée* but well donnee)

```
INTEGER :: N

CHARACTER(LEN=3) :: Answer
```

EXPRESSIONS

1. operators

\+ - * / * * (exponentiation)

Remark.

OPERATIONS AMONG INTEGERS GIVE AN INTEGER RESULT!

Example: 1/2 equals 0

2. logical operators: .AND. .OR. .NOT.

don't forget the dots

3. relational operators

/= > < >= <= ==

(≠) (>) (<) (≥) (≤) (=)

4. functions

ABS( )	absolute value
REAL( )	real conversion
LOG( )	Neperian logarithm
LOG10( )	decimal logarithm
SQRT( )	square root
INT( )	integer conversion
MOD ( )	modulo
NINT( )	integer conversion with rounding (to nearest integer)

Examples of expressions:

```
0,    N,    S1 + DONNEE,    S1/N,
S2 + DONNEE**2,    N <= 0
```

ASSIGNMENT (" INSTRUCTION D'AFFECTATION ")

Syntax

```
variable = expression
```

Remark

! Use the same type or be careful to conversion from real to integer

Examples I = 1.6 => I equals 1 !

S1 = 0.0 N = 0 initialization of variables

DATA INPUT (" INSTRUCTION DE SAISIE ")

```
READ(* ,* ) list of variables
```

Example

```
READ(* ,* ) N
```

Remarks

1. A list of variables consists in variable names separated by comas (,)

2. Stars ("astérique") can be replaced by file name and format specification.

DATA OUTPUT (" INSTRUCTION D'AFFICHAGE ")

```
WRITE(* ,* ) list of expressions
```

Example

```
WRITE ( * , * ) 'MOYENNE = ' , S1/N
```

Remark. Also variables or constants.

SEQUENCE STRUCTURE (" STRUCTURE DE SÉQUENCE ")

Example

```
S1 = S1 + DONNEE

S2 = S2 + DONNEE**2
```

BRANCHING (" STRUCTURE D'ALTERNATIVE ")

Example

```
IF( N > 0 ) THEN
  WRITE ( * , * ) 'Moyenne = ' , S1/N
ELSE
  WRITE ( * , * ) 'Pas de données'
END IF
```

Blocks can have one or more instructions,

or (without ELSE)

```
IF( N > 0 ) THEN
  WRITE ( * , * ) 'MOYENNE = ' , S1/N
END IF
```

or (one line IF)

```
IF (N <= 0) N = 1
```

In the latter case there is only one instruction.

LOOP (" STRUCTURES DE RÉPÉTITIVE SANS COMPTEUR ")

```
DO
  Block (one or more instructions )
IF( condition ) EXIT
  Block (one or more instructions )
END DO
```

Example

```
DO
  READ(* , *) DONNEE
IF( DONNEE == -999 ) EXIT
  N = N + 1
  S1 = S1 + DONNEE
END DO
```

LOOP WITH COUNTER (" STRUCTURES DE RÉPÉTITIVE AVEC COMPTEUR ")

```
DO variable = début, fin
  Loop (one or more instructions )
END DO
```

Example

```
DO  I = 1, N
  READ(* , *) DONNEE
  S1 = S1 + DONNEE
END DO
```

Can be exited with instruction EXIT and reach the end with instruction CYCLE.

ARRAYS (" TABLEAUX ")

```
REAL, DIMENSION(100) :: X
REAL, DIMENSION(12, 12) :: A
REAL, DIMENSION(:) :: Y
```

Data entry in arrays (implicit DO loop):

```
READ(* , * ) (X(I), I = 1, N)
WRITE(* , * ) ( (A(N1, N2), N2 = 1, U), N1 = 1, U )
```

! count the parentheses

PROGRAM APPEARANCE (" DISPOSITION DU PROGRAMME ")

A program can be subdivided in a main program and one or several subprograms (subroutines or functions) and/or modules.

Horizontal disposition

instruction or nothing ! comment indicator
instruction too long to be terminated on a single line &
end of instruction

Vertical disposition

```
     PROGRAM  name                                  only one !
or   SUBROUTINE  name  (arguments)
or   FUNCTION  name  (arguments)
or   MODULE  name
          USE module
          IMPLICIT NONE
          Type declaration (+ dimensions)
               and INTERFACE blocks
          Executable instructions
          STOP              stopping the main program
or        RETURN            return to the calling program
     END PROGRAM name       physical end of main program
or   END SUBROUTINE  name  (arguments)
or   END FUNCTION  name  (arguments)
or   END MODULE  name
```

COMPLETE EXAMPLE 1

Compute the mean and the variance of a series of observations

Not always necessary to put the data in an array but will be done here

Main program ESSVAR1 calls a subroutine called VARCAL, for computing the mean and the variance.

Remark

VARCAL makes use of the "calculator" algorithm consisting to accumulate the data and their squares in variables S1 and S2. Mean = S1/ N, variance = S2/N - Mean^2

```
PROGRAM  ESSVAR1
!-----Essai de l'algorithme pour le calcul de la variance
!-------------------------------------------------------
      IMPLICIT NONE
      INTEGER                :: N, I
      REAL                   :: XBAR, VARI
      REAL, DIMENSION(100) :: X
      INTERFACE
         SUBROUTINE VARCAL( X, N, XB, VARI )
           IMPLICIT NONE
           INTEGER, INTENT(IN)             :: N
           REAL, DIMENSION(N), INTENT(IN) :: X
           REAL, INTENT(OUT)               :: XB, VARI
         END SUBROUTINE VARCAL
      END INTERFACE
!-----I N I T I A L I S A T I O N
      READ(*,*) N
      READ(*,*) (X(I), I = 1, N)
!-----T R A I T E M E N T
      CALL VARCAL( X, N, XBAR, VARI)
!-----C L O T U R E
      WRITE(*,*) 'MOYENNE =  ', XBAR
      WRITE(*,*) 'VARIANCE = ', VARI
      STOP
   END PROGRAM ESSVAR1
```

```
   SUBROUTINE VARCAL( X, N, XB, VARI )
!-----------------------------------------------------
!     algorithme de calcul de la variance
!     par le procédé "calculatrice"
!-----------------------------------------------------
      IMPLICIT NONE
      INTEGER, INTENT(IN)             :: N
      REAL, DIMENSION(N), INTENT(IN) :: X
      REAL, INTENT(OUT)               :: XB, VARI
      INTEGER :: I
      REAL    :: S1, S2
      S1 = 0.0
      S2 = 0.0
      DO  I = 1, N
        S1 = S1 + X(I)
        S2 = S2 + X(I)**2
      END DO
      XB = S1/N
      VARI = S2/N - XB**2
      RETURN
   END SUBROUTINE VARCAL
```

Remark. Not safe ! What if N > 100?

COMPLETE EXAMPLE 2

Better to have a rock solid subprogram by transmitting the dimension to the subroutine (variable NMAX) and checking that N ≤ NMAX

```
SUBROUTINE VARSEC( X, NMAX, N, XB, VARI, IERR )
!-------------------------------------------------------
!    algorithme de calcul de la variance
!    par le procédé "calculatrice" ! (version de VARCAL
sécurisée)
!-------------------------------------------------------
      IMPLICIT NONE
      INTEGER, INTENT(IN)              :: N, NMAX
      REAL, DIMENSION(N), INTENT(IN)   :: X
      INTEGER, INTENT(OUT)             :: IERR
      REAL, INTENT(OUT)                :: XB, VARI
      INTEGER :: I
      REAL    :: S1, S2
      IERR = 0
      IF( N <= 0 ) IERR = 1
      IF( N > NMAX ) IERR = 2
      IF( IERR > 0 ) RETURN
      S1 = 0.0
      S2 = 0.0
      DO  I = 1, N
        S1 = S1 + X(I)
        S2 = S2 + X(I)**2
      END DO
      XB = S1/N
      VARI = S2/N - XB**2
      RETURN
   END SUBROUTINE VARSEC
```

Error code using variable IERR : when leaving VARSEC (new name of VARCAL), a 0 value for IERR indicates that everything's OK. A value 1 or 2 indicates an error (respectively N<1 and N> NMAX). The main program needs to be adapted.

```
PROGRAM  ESSVAR2
!-----Essai de l'algorithme pour le calcul de la variance
!     (version de VARSEC)
!-------------------------------------------------------
      IMPLICIT NONE
      INTEGER                :: N, I, IERR
      REAL                   :: XBAR, VARI
      REAL, DIMENSION(100) :: X
      INTERFACE
         SUBROUTINE VARSEC( X, NMAX, N, XB, VARI, IERR )
           IMPLICIT NONE
           INTEGER, INTENT(IN)              :: N, NMAX
           REAL, DIMENSION(N), INTENT(IN) :: X
           INTEGER, INTENT(OUT)             :: IERR
           REAL, INTENT(OUT)                :: XB, VARI
         END SUBROUTINE VARSEC
      END INTERFACE
!-----I N I T I A L I S A T I O N
      READ(*,*) N
      READ(*,*) (X(I), I = 1, N)
!-----T R A I T E M E N T
      CALL VARSEC( X, 100, N, XBAR, VARI, IERR )
!-----C L O T U R E
      WRITE(*,*) 'Code d''erreur ', IERR
      WRITE(*,*) 'MOYENNE =  ', XBAR
      WRITE(*,*) 'VARIANCE = ', VARI
      STOP
   END PROGRAM ESSVAR2
```

COMPLETE EXAMPLE 3

Main inconvenience is when several data sets are used. The following programs allows for as many data sets as requested

```
   PROGRAM  ESSVAR3
!-----Essai de l'algorithme pour le calcul de la variance
!     (version de VARCAL sécurisée) pour plusieurs essais
!-------------------------------------------------------
      IMPLICIT NONE
      INTEGER               :: N, I, IERR
      REAL                  :: XBAR, VARI
      REAL, DIMENSION(100)  :: X
      INTERFACE
         SUBROUTINE VARSEC( X, NMAX, N, XB, VARI, IERR )
           IMPLICIT NONE
           INTEGER, INTENT(IN)             :: N, NMAX
           REAL, DIMENSION(N), INTENT(IN)  :: X
           INTEGER, INTENT(OUT)            :: IERR
           REAL, INTENT(OUT)               :: XB, VARI
         END SUBROUTINE VARSEC
      END INTERFACE
      DO
!-------I N I T I A L I S A T I O N
        READ(*,*) N
      IF( N == 0 ) EXIT
        READ(*,*) (X(I), I = 1, N)
!-----T R A I T E M E N T
        CALL VARSEC( X, 100, N, XBAR, VARI, IERR )
!-----C L O T U R E
        WRITE(*,*) 'Code d''erreur ', IERR
        WRITE(*,*) 'MOYENNE =  ', XBAR
        WRITE(*,*) 'VARIANCE = ', VARI
      END DO
      WRITE(*,*) 'Fin des jeux d''essai'
      STOP
   END PROGRAM ESSVAR3
```

REMARKS

1. Different version of subroutine VARSEC using another algorithm called the "corrected 2 passes algorithm" (to be seen later)

```
   SUBROUTINE VARCOR( X, NMAX, N, XB, VARI, IERR )
!-------------------------------------------------------
!     algorithme de calcul de la variance
!     en 2 passages et corrigé (version sécurisée)
!-------------------------------------------------------
      IMPLICIT NONE
      INTEGER, INTENT(IN)              :: N, NMAX
      REAL, DIMENSION(N), INTENT(IN)   :: X
      INTEGER, INTENT(OUT)             :: IERR
      REAL, INTENT(OUT)                :: XB, VARI
      INTEGER :: I
      REAL    :: S1, S2
      IERR = 0
      IF( N <= 0 ) IERR = 1
      IF( N > NMAX ) IERR = 2
      IF( IERR > 0 ) RETURN
      S1 = 0
      DO  I = 1, N
        S1 = S1 + X(I)
      END DO
      XB = S1/N
      S1 = 0.0
      S2 = 0.0
      DO  I = 1, N
        S1 = S1 + (X(I) - XB)
        S2 = S2 + (X(I) - XB)**2
      END DO
      VARI = S2/N - (S1/N)**2          ! au lieu de S2/N
      RETURN
   END SUBROUTINE VARCOR
```

2. Possible to rewrite the main program using advanced Fortran 90 (to be used with the first version VARCAL (no problem with *N*).

```
   PROGRAM  ESSVAR5
!-----Essai de l'algorithme pour le calcul de la variance
!     (version de VARCAL non sécurisée) pour plusieurs essais
!     avec gestion dynamique de la mémoire
!-------------------------------------------------------
      IMPLICIT NONE
      INTEGER              :: N, I
      REAL                 :: XBAR, VARI
      REAL, DIMENSION(:), ALLOCATABLE :: X
      INTERFACE
         SUBROUTINE VARCAL( X, N, XB, VARI )
           IMPLICIT NONE
           INTEGER, INTENT(IN)             :: N
           REAL, DIMENSION(N), INTENT(IN) :: X
           REAL, INTENT(OUT)               :: XB, VARI
         END SUBROUTINE VARCAL
      END INTERFACE
      DO
!-------I N I T I A L I S A T I O N
        READ(*,*) N
      IF( N == 0 ) EXIT
        ALLOCATE( X(N) )
        READ(*,*) (X(I), I = 1, N)
!-----T R A I T E M E N T
        CALL VARCAL( X, N, XBAR, VARI ) ! sécurité inutile
!-----C L O T U R E
        WRITE(*,*) 'MOYENNE = ', XBAR
        WRITE(*,*) 'VARIANCE = ', VARI
        DEALLOCATE( X )
      END DO
      WRITE(*,*) 'Fin des jeux d''essai'
      STOP
   END PROGRAM ESSVAR5
```

3. In Fortran 77

- No dynamic allocation
- No END DO and EXIT (to be replaced, respectively, by CONTINUE and GO TO)
- With a label ("étiquette") at the end of the loop, to be specified in instruction DO
- Comments preceded by * instead of!
- Operators <= & > to be replaced by **.LE.** (less or equal) and **.GT.** (greater than) (with dots!)
- Instructions of end of unit: END (without anything else)
- Rigorous presentation in columns:

```
      PROGRAM  ESSVAR
*---- - Essai de l'algorithme pour le calcul de la
      variance
*---- - ---------------------------------------------
      REAL X(100)
*---- - I N I T I A L I S A T I O N
      READ(*,*) N
      READ(*,*) (X(I), I = 1, N)
*---- - T R A I T E M E N T
      CALL VARSEC( X, 100, N, XBAR, VARI, IERR )
*---- - C L O T U R E
      WRITE(*,*) 'CODE D''ERREUR', IERR
      WRITE(*,*) 'MOYENNE =  ', XBAR
      WRITE(*,*) 'VARIANCE = ', VARI
      STOP
      END
```

With the subprogram:

```
      SUBROUTINE VARSEC( X, NMAX, N, XB, VARI, IERR )
*---- - ---------------------------------------------
      ----
*     algorithme de calcul de la variance
*     par le procédé "calculatrice"
*---- - ---------------------------------------------
      ----
      REAL X(NMAX)
      IERR = 0
      IF( N .LE. 0 ) IERR = 1
      IF( N .GT. NMAX ) IERR = 2
      IF( IERR .GT. 0 ) RETURN
      S1 = 0.0
      S2 = 0.0
      DO 100 I = 1, N
         S1 = S1 + X(I)
         S2 = S2 + X(I)**2
  100 CONTINUE
      XB = S1/N
      VARI = S2/N - XB**2
      RETURN
      END
```

Also:

- Variables don’t need to be declared
- Interfaces don’t exist
- Declarative instructions without '::' and DIMENSION or INTENT clauses

- In Fortran 77, there is an implicit declaration based on the initial letter

I - N : "INTEGER"

A - H , O - Z : "REAL"

TEST RUN

Using 3 numbers 10 000 001, 10 000 002, 10 000 003, where the variance should be 2/3 = 0.66666...

A silly result is obtained.

Ill-conditioned algorithm:

use at least S2 as a double precision (64-bit) variable

```
   SUBROUTINE DVARCAL( X, N, XB, VARI)
!-------------------------------------------------------
!      algorithme de calcul de la variance
!      par le procédé "calculatrice" ! (double précision)
!-------------------------------------------------------
       IMPLICIT NONE
       INTEGER, INTENT(IN)              :: N
       REAL, DIMENSION(N), INTENT(IN) :: X
       REAL, INTENT(OUT)                :: XB, VARI
       INTEGER :: I
       REAL(8) :: S1, S2          ! double précision
       S1 = 0.0
       S2 = 0.0
       DO  I = 1, N
         S1 = S1 + X(I)
         S2 = S2 + X(I)**2
       END DO
       XB = S1/N
       VARI = S2/N - XB**2
       RETURN
   END SUBROUTINE DVARCAL
```

Better: Reading data and computing squares in double precision.

1.3 INTRODUCTION TO FORTRAN

CONTENTS

VARIABLE ⎫
⎬ + TYPE + NAME
CONSTANT ⎭

EXPRESSIONS numerical ⎫
character, logical ⎬ Attention!
ASSIGNMENT Instruction ⎭

INPUT instruction ⎫
⎬ + format specification
OUTPUT instruction ⎭

SEQUENCE structure
+ physical presentation of program
+ free source mode presentation

ALTERNATIVE structures

REPETITIVE structures Attention !

ARRAYS : declaration, use of elements, use of arrays

PROGRAM UNITS AND PARAMETERS

types, parameter transmission, interface Attention !

LOCAL AND GLOBAL VARIABLES Attention !

FILES (" FICHIERS ") : types, identification, input, output

CONCEPT OF VARIABLE AND OF CONSTANT

NAMES (OF VARIABLES, PROGRAM UNITS, ETC.)

From 1 to 31 letters/digits or underline _

Letters from a to z or A to Z

Accentuated characters not accepted in names (but well within strings)

Names must begin with a letter

TYPES

Numerical	"REAL" or "REAL(4)"	real on 4 bytes (7-8 significant digits)
	"INTEGER"	integer (7-8 significant digits)
	"COMPLEX"	Complex (7-8 significant digits)
	"REAL(8)"	Double precision (often 13-14 significant digits)
Logical	"LOGICAL"	or Boolean
Character	"CHARACTER"	length needs to be specified
Pointer	"POINTER"	Pointer (new in Fortran 90)

CONSTANTS

"REAL"	`1.0 -0.5E-4`	Note the decimal point
"INTEGER"	`1 -9999`	
"COMPLEX"	`(0.5, 0.5)`	

"REAL(8)"	`1.0D0 1.0D-6`	Note D instead of E
"LOGICAL"	`.TRUE. .FALSE.`	
"CHARACTER"	`"Bonjour" 'L''ami' ''`	double apostrophe

Note : apostrophe (') or quotation marks (") can be used but in pairs!

VARIABLES

1. Ancient versions of Fortran have allowed not to declare explicitly variables : automatic or implicit declaration. To avoid this, use :

```
IMPLICIT NONE
```

2. Example of explicit declaration of variables

Example

```
REAL :: MONTAN, TAUX
LOGICAL :: TROUVE
INTEGER :: ENTIER
```

3. Furthermore, the length of character variables must be declared (a number of characters)

Example

```
CHARACTER(LEN=20) :: NOM or
CHARACTER, DIMENSION(20) :: NOM
CHARACTER(LEN=80) :: TITRE
CHARACTER(LEN=10) :: MOT1, MOT2
```

NAMED CONSTANT

Using the PARAMETER clause in a declaration

```
type, PARAMETER :: nom = valeur
```

Example

```
INTEGER, PARAMETER :: N = 100
REAL(8), PARAMETER :: PI = 3.141592653
```

EXPRESSIONS

Let us distinguish numerical, character and logical expressions

NUMERICAL EXPRESSIONS

1. Operators:

\+ – * / * * (exponentiation)

2. Mixing "INTEGER" and "REAL" types

BUT

3. An expression involving **integers** has an integer result

Example

If	J = 2		and	A = 2.0	
4/J	equals	2	4/A	equals	2.0
1/J	**equals**	**0**	1/A	equals	0.5
1.0 /J	equals	0.5	1.0/A	equals	0.5
2.0*(1/J)	**equals**	**0.0**	2.0*(1/A)	equals	1.0

CHARACTER EXPRESSIONS

1. Operator: // (concatenation)

Example 'BON'//'JOUR' equals 'BONJOUR'

2. Substring

Example

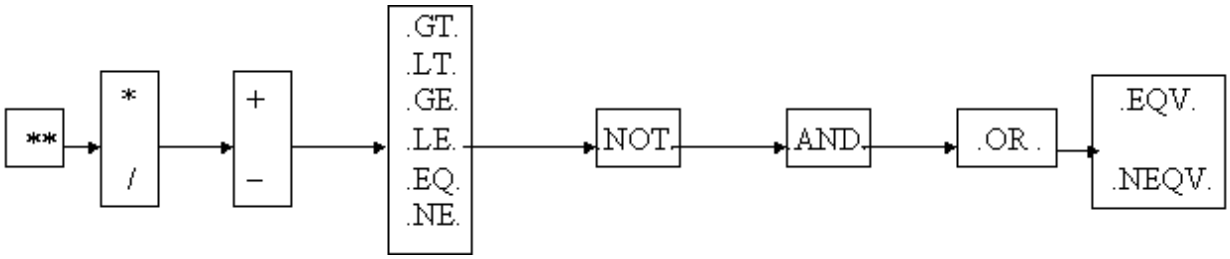
If MOT equals `'BONJOUR'`
MOT(4:6) equals `'JOU'`

LOGICAL EXPRESSIONS

1. Logical operators: .AND. .OR. .NOT. .EQV. .NEQV.
   et ou non equivalence non equivalence

2. Relational operators: > < >= <= == /=
   or .GT. .LT. .GE. .LE. .EQ. .NE.
   means > < ≥ ≤ = ≠

PRIORITIES



FUNCTIONS

ABS( )	absolute value (modulus of a complex number)
MOD(I, J)	remaining of integer division of I by J
MAX(I, J)	maximum of I and J
MIN(I, J)	minimum of I and J
REAL( )	conversion to a real number
CMPLX( )	conversion to a complex number (1 or 2 arguments)
AIMAG( )	imaginary part of a complex number

Examples

1. NINT(X*100)/100.0

   rounding the real to 0,01

2. MOD(I + J ,60)

addition in the Euclidian group $Z_{60,+}$

3. ABS(X) < 1.0E-7 true if $|X| < 10^{-7}$, false otherwise

Remark. If X = 1.0 and Y = X/10.0, then expression

Y == 0.10

is generally false because of rounding

Divisions by 2 are exact but not divisions by 5 nor by 10

(numbers are encoded in binary form)

ASSIGNMENT INSTRUCTION

SYNTAX

```
variable = expression
```

EVALUATION RULES

1. If variable and expression have the same type, no problem
2. If variable and expression have the different numerical types, then **type conversion**

	Expression	
Variable	"REAL"	"INTEGER"
"REAL"	Unchanged	**`REAL(expression)`**
"INTEGER"	**`INT(expression)`**	Unchanged

Examples (supposing A is a real variable and I is an integer variable)

A = 1.0	A takes the real value 1.0
A = 1	A takes the real value 1.0
I = 1	I takes the integer value 1
I = 1.5	I takes the integer value INT(1.5) = 1 !!!

INPUT INSTRUCTION

Equivalent of "Saisir" using the keyboard

SYNTAX

```
READ(*,*) list of variables
```

or

```
READ(*,format) list of variables
```

Example. READ(*,*) A, B, C

```
1.2 3.4 -5.6
```

Example. READ(*,'(A3)') REPONS

where '(A3)' is a constant of type character (format specification) and REPONS is declared of type CHARACTER(LEN=3).

```
OUI
```

FORMAT SPECIFICATIONS

Syntax: '(specifications)'

The main specifications are:

*n*X	*n* spaces
A*n*	*n* characters
I*n*	integer with *n* characters
F*n*.*d*	real with *n* characters and *d* decimal digits

Example. READ(*,'(3F5.1)') A, B, C

```
  1.23.4   -5.6
<---><---><--->
  5    5    5
```

Remark. The decimal point has priority on the specification of *d*.

OUTPUT INSTRUCTION

Equivalent of "Afficher" on the screen or "Imprimer" on paper.

SYNTAX:

```
WRITE(*,*) list of expressions
```

ou

```
WRITE(*,format) list of expressions
```

Example. WRITE(*,*) 'Entrez', N, ' nombres'

```
Entrez 100 nombres
```

Example. WRITE(*, '(1X, A10, F5.2)') 'Somme = ', S

```
 Somme =   -2.34
1<---10---><-5->
```

Example. WRITE(*,'("'Données:"', 3F7.3, " - Résultat:", F7.3)') A, B, C, A + B + C

```
Données:  1.000  2.000  3.000 - Résultat:  6.000
<------><--7--><--7--><--7--><----------><--7-->
```

SEQUENCE STRUCTURE

Example

```
S1 = S1 + X
S2 = S2 + X**2
S3 = S3 + X**3
S4 = S4 + X**4
```

Remark. Vertical alignment is not (unfortunately) compulsory. But highly recommended.

PHYSICAL PRESENTATION OF THE PROGRAM

RULES

A program is composed of one or several program units

If there is only one, it is the main program

If there are several program units, one and only one is the main program

The other program units are subprogram or modules

Subprograms are called by one or several program units

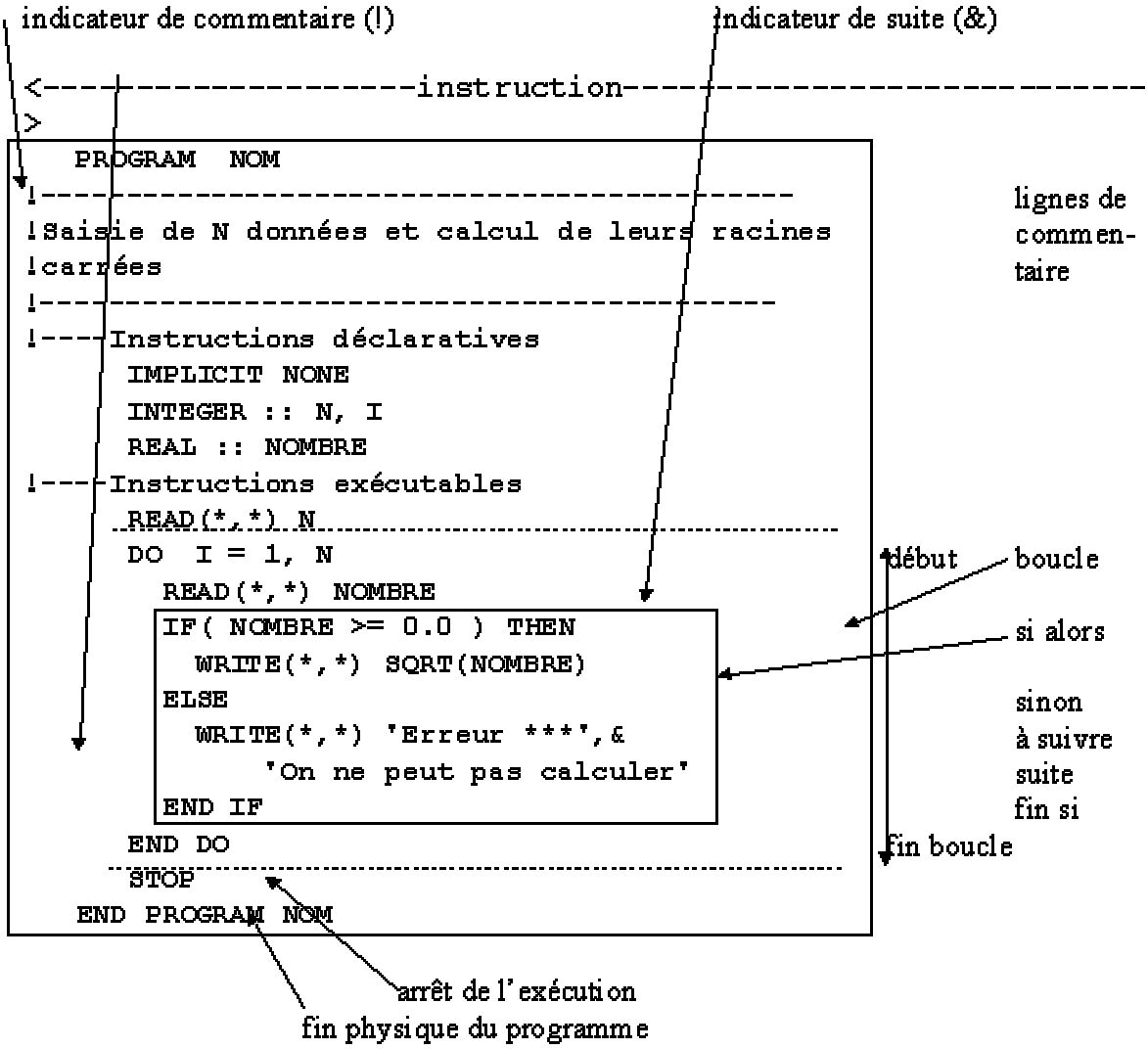
Note : Blank lines are comment lines

Remarks.

1. Program units can be in one or several files. The order of program units is arbitrary except for modules (which must be before the other program units)
2. Program files are treated by a special application called a compiler, which transform them in object programs. Object programs can be stored in libraries (allowing to use them even without the source code)
3. Besides external subprograms, we'll examine modules later, a new concept in Fortran 90. They collect object definitions and internal subprograms for manipulating these objects.

FREE SOURCE MODE PRESENTATION

Présentation Fortran 90 en mode source libre ou Fortran 95



STRUCTURES D'ALTERNATIVE

ALTERNATIVE STRUCTURES

IF STRUCTURE

```
IF(expression logique) THEN
  bloc 'alors'
ELSE
  bloc 'sinon'
END IF
```

Remarks

1. Instructions IF( ) THEN, ELSE et END IF are distinct
2. THEN must be in the IF instruction
3. ELSE and END IF must be separated from the blocks
4. The blocks are sequence structures

Example

```
IF( REALIS >= 0.0 ) THEN
  WRITE(*,*) 'Solutions réelles'
ELSE
  WRITE(*,*) 'Solutions complexes'
END IF
```

SYNTAX WITHOUT ELSE

```
IF(expression logique) THEN
  bloc 'alors'
END IF
```

Example

```
IF( REALIS >= 0.0 .AND. ABS(A) > 1.0E-7 ) THEN
  X1 = (-B + SQRT(REALIS))/(2*A)
  X2 = (-B - SQRT(REALIS))/(2*A)
END IF
```

SIMPLIFIED (ONE LINE) SYNTAX

```
IF( expression logique ) instruction
```

Example.

```
IF( REALIS < 0.0 ) STOP
IF( REALIS > 0.0 ) WRITE(*,*) '2 solutions'
```

GENERALISED SYNTAX CASE OF ("SELON QUE")

```
IF( expression logique1 ) THEN
  bloc1
ELSE IF( expression logique2 ) THEN
  bloc2
ELSE
  bloc3
END IF
```

Equivalent to 2 embedded IFs

```
IF( expression logique1 ) THEN
  bloc1
ELSE
  IF( expression logique2 ) THEN
      bloc2
  ELSE
      bloc3
  END IF
END IF
```

Example

```
IF( MENU == 1 ) THEN
  READ(*,*) A, B
ELSE IF( MENU == 2 ) THEN
  WRITE(*,*) 'Somme :', A + B
ELSE IF( MENU == 3 ) THEN
  WRITE(*,*) 'Différence :', A - B
ELSE
  WRITE(*,*) 'Erreur dans le menu'
END IF
```

REPETITIVE STRUCTURE

Structures "Itérer" as well as Do while (" Tantque ... faire") and Repeat until ("Répéter...jusqu'à"), and For ...endfor can be simulated using the loop with counter and the **loop without counter**, respectively.

SYNTAX OF THE LOOP WITHOUT COUNTER (" ITÉRER ")

```
DO
        bloc1
IF( condition ) EXIT
        bloc2
END DO
```

Example. Data entry with a sentinel equal to 0 and sum of the data

```
S = 0.0
DO
  WRITE(*,*) 'Entrez une donnée'
  READ(*,*) DONNEE
IF( DONNEE == 0.0 ) EXIT
  S = S + DONNEE
END DO
WRITE(*,*) 'Somme = ', S
```

EXIT is necessary to avoid an infinite loop

SYNTAX OF THE LOOP WITH COUNTER (" POUR ")

```
DO compteur = début, fin
        bloc = boucle
END DO
```

RULES

Counter = variable of type INTEGER

Not subject of an assignment in the loop

début, fin : expressions of type INTEGER

if début > fin nothing happens (the loop is not executed)

Example.

```
S = 0.0
DO  I = 1, N
  READ(*,*) NOMBRE
  S = S + NOMBRE
END DO
WRITE(*,*) 'SOMME DES ', N, ' NOMBRES:', S
```

Remark. It is also possible to use a step pas instead of 1, and that step can be strictly negative (for example -1 for a reverse loop):

```
DO compteur = début, fin, pas
```

Example.

```
DO CAR = LEN(MOT), 1, -1
```

Loops with and without counters can be combined : loop with a counter but also EXIT

Example.

```
S = 0.0
DO  I = 1, N
  READ(*,*) NOMBRE
IF( NOMBRE <= 0 ) EXIT
  S = S + LOG(NOMBRE)
END DO
WRITE(*,*) 'SOMME DES LOGARITHMES DE ', I - 1, ' NOMBRES:', S
```

Remark

Repetitive structures can be embedded one in the other *but it is necessary to change the counter variable*

ARRAYS

TYPE AND SIZE DECLARATION

Declare type and size together

Example

```
REAL, DIMENSION(5) :: TAUX
REAL, DIMENSION(5,3) :: MAT
```

Remark. 2-dimensional arrays are stored in memory column by column (not row by row)

USE OF THE ELEMENTS OF AN ARRAY

Everywhere a variable is accepted, and also in so-called implicit DO loops, especially but not alone in input and output instructions. Besides isolated (element by element) assignment, it is also possible to assign several elements in a single instruction, using the pair (/ ... /):

Examples

1.

```
TAUX(1) = 0.06
TAUX(2) = 0.21
```

2.

```
TAUTVA = (/ 0.06, 0.21 /)
INDICE = (/ (I, I = 1, 10) /)
```

3.

```
X = SQRT(MAT(I + J - 1, I - J + 1))
```

4.

```
DO K = 1, 5
   READ(*,*) TAUX(K)
END DO
```

5.

```
READ(*,*) (TAUX(K), K = 1, 5)
```

In the latter case, data can be on one or several data lines. With the explicit DO loop, as in Example 4, data items should be disposed on separated lines (each execution of READ reading a single line).

MANIPULATION OF ARRAYS

They can be used directly in instructions. The operators (+, *, ...) and most intrinsic functions (logarithms, square root, ...) act on arrays element by element. Several intrinsic functions allow for other operations (for example, MATMUL to multiply matrices).

Example

```
REAL :: LAMBDA
REAL, DIMENSION(10,10) :: A, B, C
READ(*,*) ((A(I,J), J = 1, 10), I = 1, 10)
READ(*,*) LAMBDA
B = 0                    ! met des zéros dans B
DO  I = 1, 10
  B(I,I) = 1             ! pas de moyen de faire autrement
END DO
C = B - LAMBDA*A         ! instruction de tableau
```

PROGRAM UNITS AND PARAMETERS

DIFFERENT TYPES OF PROGRAM UNITS

- The main program :unique
- Subprograms : written by the user or belonging to a library
- Functions : the same
- Modules : the same

In principle, functions return a unique result by their name (that result can be an array).

Modules aimed to define objects to be used in several program units and can contain subprograms to manipulate these objects (object-oriented programming).

Remark. Old versions of Fortran allow side effects (modification of the parameters of the function, modification of global variables, input and output). It is recommended to avoid them (they prevent optimization and parallelization by the compiler) particularly in functions.

PARAMETER TRANSMISSION

Transmission of information between program units can be done in two different ways:

1. by transmission at calling time, through the parameters: let us distinguish the formal parameters of the subprogram (subroutine or function) and the effective parameters of the calling program
2. by the use of modules (see later).

EXAMPLE : MANIMA

```
    PROGRAMME MANIMA
       IMPLICIT NONE
       REAL :: A(10, 5), B(5, 8), C(10, 8)
       INTERFACE
       SUBROUTINE LECMAT( X, M, N )
          IMPLICIT NONE
          INTEGER, INTENT(IN) :: M, N
          REAL, DIMENSION(M, N), INTENT(OUT) :: X
       END SUBROUTINE LECMAT
       END INTERFACE
       INTERFACE
       SUBROUTINE ECRMAT( X, M, N )
          IMPLICIT NONE
          INTEGER, INTENT(IN) :: M, N
          REAL, DIMENSION(M, N), INTENT(IN) :: X
       END SUBROUTINE ECRMAT
       END INTERFACE
       INTERFACE
       SUBROUTINE PROMAT( X, Y, M, N, L, Z )
          IMPLICIT NONE
          INTEGER, INTENT(IN) :: M, N, L
          REAL, INTENT(IN)  :: X(M, N), Y(N, L)
          REAL, INTENT(OUT) :: Z(M,L)
       END SUBROUTINE PROMAT
       END INTERFACE
 !-----Lecture des matrices A et B
       CALL LECMAT( A, 10, 5 )
       CALL LECMAT( B, 5, 8 )
 !-----Calcul du produit de A et de B dans C
       CALL PROMAT( A, B, 10, 5, 8, C )
 !-----Ecriture du produit
       CALL ECRMAT( C, 10, 8 )
       STOP
    END PROGRAM MANIMA
```

Note that A, 10 and 5 are effective parameters when calling LECMAT the first time. They correspond, in the order, to the formal parameters in the definition below, X, M and N. The names can of course be different but the nature (type and size) should be the same. Here X is replaced by A, M by 10 and N by 5. Let us insist that no type conversion is done. A real should correspond to a real, an integer to an integer. Let us notice the interface blocks. Compare, for example, the first interface block relative to subroutine LECMAT with the code of LECMAT below.

```
   SUBROUTINE LECMAT( X, M, N )
      IMPLICIT NONE
      INTEGER, INTENT(IN) :: M, N
      REAL, DIMENSION(M, N), INTENT(OUT) :: X
      INTEGER :: I, J
      DO  I = 1, M
        READ(*,*) (X(I, J), J = 1, N)
      END DO
      RETURN
   END SUBROUTINE LECMAT
```

Instruction RETURN returns to the calling program unit, here the main program MANIMA.

More precisely, execution continues at the instruction following the call. In the example, it is a second call to LECMAT.

Subroutine ECRMAT is not given. It resembles LECMAT except that WRITE is used instead of READ and INTENT(IN) instead of INTENT(OUT) for matrix X.

Subroutine PROMAT can be written

```
SUBROUTINE PROMAT( X, Y, M, N, L, Z )
   IMPLICIT NONE
   INTEGER, INTENT(IN) :: M, N, L
   REAL, INTENT(IN)  :: X(M, N), Y(N, L)
   REAL, INTENT(OUT) :: Z(M,L)
   INTEGER :: I, J, K
   DO  I = 1, M
     DO  J = 1, L
       Z(I,J) = 0.0
       DO  K = 1, N
         Z(I, J) = Z(I, J) + X(I, K)*Y(K, J)
       END DO
     END DO
   END DO
   RETURN
END SUBROUTINE PROMAT
```

Remark. The call to PROMAT can be replaced by the following instruction:

```
C = MATMUL(A, B)
```

RULES

1. Correspondence between lists of formal parameters and effective parameters:

   same number of parameters

   identical types of parameters

   same size

2. The most important: the way by which the effective parameters are transmitted to the subprogram at calling time: the memory address of the effective parameters is transmitted to the

subprogram: transmission by reference

Remark : type and size are also transmitted

3. Let us distinguish the parameters used as input of the subprogram (INTENT(IN)), as output (INTENT(OUT)) or both (INTENT(IN OUT)). A constant can also be used as effective parameter as input but not as output

4. With the exception of subprograms contained in modules, subprogram should be defined in interface blocks. Their syntax is as follows:

```
INTERFACE
  SUBROUTINE nom(...) or FUNCTION nom(...)
      IMPLICIT NONE
      Declarative instructions
  END SUBROUTINE nom or END FUNCTION nom
END INTERFACE
```

It collects the declarative instructions relative to the parameters (not the other variables).

EXAMPLES.

They show errors with respect to PROMAT given above. They are trapped by the pure Fortran 90 execution time routines:

1)

```
CALL PROMAT( A, B, C, 8, 5, 9 )
```

parameters not in the right order

2)

```
CALL PROMAT( A, B, 10, 5, 9, C )
```

Outside memory space (9th column of B don't exist; worse, a 9th column of C that doesn't exist is created,

destroying information in memory)

3)

```
CALL PROMAT( A, B, 10, 5, 8.0, C )
```

The 5th effective parameters is real 8.0. The formal integer parameter L will take a value with a real representation, surely a big number.

4)

```
CALL PROMAT( A, B, 10, 5, 9, 1.0 )
```

The 6th parameter is real 1.0. Bad situation. Instead of transmitting an array where to place the results of PROMAT, we have transmitted a real parameter 1.0. Not only there will be memory leakage, but constant 1.0 will receive another value!

Remark. These errors are not normally detected in Fortran 77 !

LOCAL AND GLOBAL VARIABLES

Unless otherwise indicated, variables are local in the program unit where they are declared. In Fortran 90, global variables are defined in modules, which are isolated program units.

SYNTAX

```
MODULE nom de module
       declarative instructions
END MODULE nom de module
```

RULES

1. Variables defined in a module are global in all program units where the module is ***used*** (meaning, see 2). Variable names can be different from one use to the other (this is not recommended, of course).

2. The use of a module is declared by the declarative instruction (one name of module per instruction):

```
USE nom de module
```

or

```
USE nom de module, nouveau => ancien
```

(where *nouveau* is the new name of a variable used instead of the old name *ancien*, the one defined in the module)

3. USE instructions must come before all other declarative instructions (even IMPLICIT NONE)

4. Modules can be located in the same file as the main program or/and subprograms that use. In that case, they must precede them in the file. Modules can also be in one or several distinct files, which must have been compiled before or at least simultaneously.

5. Interface blocks can be given in modules, which avoid putting them in each program units.

6. Modules can contain subprograms (but not other modules) with the following syntax:

```
MODULE nom de module
  instructions déclaratives
CONTAINS
   sous-programme
   ...                                  } as many times as required
   fin de sous-programme
END MODULE nom de module
```

EXAMPLE FACNC

Here is a main program for issuing invoices and credit notes and 2 modules are used, one with a contained subprogram:

```
MODULE COMSOC
  IMPLICIT NONE
  CHARACTER(LEN = 30) :: NOMSOC
END MODULE COMSOC
MODULE CTAUX
  IMPLICIT NONE
  INTEGER, PARAMETER :: NTAUX = 2
  REAL, DIMENSION(NTAUX) :: TAUTVA
CONTAINS
    SUBROUTINE INITVA( )
      TAUTVA(1) = 0.06
      TAUTVA(2) = 0.21
      RETURN
    END SUBROUTINE INITVA
END MODULE CTAUX
```

The name of the firm is defined in module COMSOC and VAT rates are declared in module CTAUX. Besides, initialization instructions of VAT rates are placed in a subprogram in module CTAUX.

```
PROGRAM FACNC
  USE COMSOC  ! définition de NOMSOC
  USE CTAUX   ! définition TAUTVA
  IMPLICIT NONE
  INTEGER  :: N, I, LECHOI
  INTERFACE
    SUBROUTINE FACTUR( NUMERO )
    IMPLICIT NONE
    INTEGER, INTENT(IN) :: NUMERO
    END SUBROUTINE FACTUR
  END INTERFACE
  ...! de même pour NOTCRE
  READ(*, '(A30)') NOMSOC
  READ(*,*) N
!---- - Initialisation des taux de TVA
  CALL INITVA( )
!---- - Réalisation de N factures/notes de crédit
  DO  I = 1, N
    READ(*,*) LECHOI
    IF( LECHOI == 1 ) CALL FACTUR( I )
    IF( LECHOI == 2 ) CALL NOTCRE( I )
  END DO
  STOP
END PROGRAM FACNC
```

Subroutine FACTUR is as follows:

```
SUBROUTINE FACTUR( NUMERO )
  USE COMSOC, TITRE => NOMSOC ! on a envie de l'appeler TITRE
  USE CTAUX                   ! defini NTAUX et TAUTVA
  IMPLICIT NONE
  INTEGER, INTENT(IN) :: NUMERO
  INTEGER :: NUMTAU
  REAL :: PRIX  READ(*,*) PRIX, NUMTAU
  WRITE(*,*) TITRE, 'Facture ', NUMERO
  IF( NUMTAU >= 1 .AND. NUMTAU <= NTAUX ) &
      WRITE(*,*) PRIX*(1.0 + TAUTVA(NUMTAU))
  RETURN
END SUBROUTINE FACTUR
```

Remark. The name of the firm NOMSOC defined in module COMSOC is recovered in subroutine FACTUR under the name TITRE. Similarly, the VAT rates initialized in module CTAUX are recovered in module FACTUR. The principle is the same for subroutine NUMCRE (which could have been merged with FACTUR, by transmitting either 'Facture' or 'Note de crédit').

FILES

TYPES OF FILES

Several criteria :

1. Text files with respect to binary files (more efficient because requiring no conversion but unreadable by humans and sometimes not portable from one system to another, PCs to Unix systems, for example). We restrict ourselves to text files.

2. Sequential access files with respect to direct access files. The former are written and read sequentially. The records of the latter can be accessed in an arbitrary order, which is of prime importance for management applications.

IDENTIFICATION OF A FILE

This is done by a ***unit number***, an integer from 0 to 999. For the OS, it is also identified by a name, generally completed by a volume (or drive unit) and a directory (" répertoire ") (they depend on the OS). The association unit number-name is done by the OPEN instruction.

```
OPEN( unité, FILE = 'nom de fichier' )
```

Remarks

1. Data entry by the keyboard and display on the screen correspond respectively to input from and output to unit *.

2. For historical reasons, there is a frequent association of unit 5 to input and 6 to output.

3. It is allowed to input the name of a file!

Examples

On a PC:

```
OPEN( 1, FILE = 'C:\USERS\MELARD\ESSAI.DTA' )
```

To enter the name of a file at execution time:

```
CHARACTER(LEN = 80) :: NOMFIC
...
READ(*, '(A)') NOMFIC
OPEN( 1, FILE = NOMFIC )
```

On a Unix system (lower and upper case is not the same in file names and / is used instead of \ for defining a path of directories)

```
OPEN( 1, FILE = '/usr/melardg/essai.dta' )
```

INPUT

```
READ(unité,*) liste des variables
```

for free format input

```
READ(unité,format) liste des variables
```

for fixed format input

OUTPUT

```
WRITE(unité,*) liste des expressions
```

for free format output

```
WRITE(unité,format) liste des expressions
```

for fixed format output

REWINDING A FILE

```
REWIND(unité)
```

1.4 ADVANCED STUDY OF FORTRAN 90 AND ELEMENTS OF FORTRAN 95

INTRODUCTION

COMPLEMENTS OF FORTRAN 90

Initialisation of a variable

Handling of character strings

Generalized alternative structure

Instruction DO WHILE

Complements on the declaration of arrays

Dynamic arrays

Manipulation of arrays

Complements on subprograms

Complements on output and input

Complements on specification formats

Other features of Fortran 90

INTRODUCTION TO FORTRAN 95

INTRODUCTION

- Still a part of the features of Fortran 90 : only those useful in statistical applications (but not elaborated data structure and program structures)
- A few features of Fortran 95 (just to provide indication)
- In the next section: additional instructions coming from Fortran 77 (to be able to read existing programs)

Remark. Some Fortran 77 compilers such as MIPS f77 or Microsoft Fortran 5.1 accept some features of Fortran 90.

COMPLEMENTS OF FORTRAN 90

INITIALIZATION OF A VARIABLE

A variable can be initialized *at program loading* in a type declarative instruction. That variable gets a SAVE attribute, which means that its value will be saved when getting out of the subprogram where the value is given.

```
type :: nom de variable = valeur
```

Example

```
INTEGER :: N = 100
REAL(8) :: ONE, ZERO, TWO = (/1.0D0, 0.0D0, 2.0D0 /)
REAL, DIMENSION(4) :: TAUTVA =(/ 6.0, 21.0, &
                                 25.0, 33.0 /)
```

The variable can then receive another value, following the execution of an assignment statement, for example. Let us emphasize the fact that initialization takes place only once, before the execution of the program. This type of initialization does not replace the assignment statements used usually to initialize a counter or an accumulator, in particular inside a part of the program that can be carried out several times (loops, subroutine calls).

When only part of a table must be initialized, or when more flexibility is wished (use of a list of variables, use of counters, use of implicit DO loops), the instruction DATA from Fortran 77 must be employed.

In the example of program FACNC, since INITVA does not carry out that initializations one could have removed this subroutine and have carried out them directly in the declarative instructions, as follows

```
MODULE CTAUX
  INTEGER, PARAMETER :: NTAUX = 2
  REAL, DIMENSION(NTAUX) :: TAUTVA = (/ 0.06, 0.21 /)
END MODULE CTAUX
```

HANDLING OF CHARACTER STRINGS

Extraction of substrings of characters, conversions from numerical to characters and some intrinsic functions allow a handling of the character strings, in addition to the assignment instruction and the operator of concatenation // already mentioned.

Example

```
CHARACTER(LEN = 10) :: CH

...

CH = 'I'//'9310xxxxx'
```

1. If CH is a string of 10 characters, CH(2:5) is the substring located in positions 2 to 5, by counting that the position on the left carries number 1, in the example '9310'. Note that CH(:5) equals CH(1:5) and CH(5:) equals CH(5:10). The positions can of course be variables.

2. An alternative form of instructions READ and WRITE makes it possible to carry out conversions numerical-characters. If positions 2 to 5 of CH are supposed to contain an integer, the instruction

```
READ(CH,'(1X,I4,5X)') ENTIER
```

allows to extract it (in the example, the integer constant 9310). Conversely, the instruction

```
WRITE(CH,'(1X,I4,5X)') ENTIER + 1
```

allows to add 1 to the integer placed in positions 2 to 5 of CH,

giving

```
I9311xxxxx
```

1. The following functions are useful to handle characters

LEN(CH)	length of the string (in the example: 10)
INDEX(CH, 'xx')	position of the first occurrence of ' xx' in string CH (in the example: 6), 0 in the case of absence
IACHAR('I')	position of the character ' I' in the ASCII sequence.

Example

Logical expression to express that ABREV is an abbreviation of at least 3 characters of string MOT:

```
LEN(ABREV) >= 3 .AND. INDEX( MOT, ABREV) == 1
```

Example

```
NCAR = LEN(MOT)
DO  CAR = LEN(MOT), 1, -1
  IF( MOT(CAR:CAR) == ' ' ) NCAR = CAR - 1
END DO
```

In this example, NCAR will be the number of characters before the first space, equivalent to the effect of the intrinsic function

```
INDEX(MOT,' ')-1.
```

4. It is possible to handle strings of variable length, defined in declarative instructions like

```
CHARACTER(LEN = *) :: CH2, CH3
```

GENERALIZED ALTERNATIVE STRUCTURE

Instead of the structure IF THEN ELSE END IF with ELSE IF, one can use the following structure SELECT CASE.

Example

```
SELECT CASE( MENU )
  CASE( 1 )
    CALL LECTURE( )
  CASE( 2 )
    CALL TRAITEMENT( )
  CASE( 3 )
    CALL ECRITURE( )
  CASE DEFAULT
    WRITE(*,*) 'Erreur dans le menu'
END SELECT
```

In instruction SELECT CASE an integer expression (like here) or an expression of type CHARACTER(LEN = 1) can be used. In instruction CASE, an integer expression can be used like here or an interval of the form "beginning : end", for example 1:5 or 'A':'Z'. A value can appear in only one instruction CASE of a structure SELECT CASE.

INSTRUCTION DO WHILE

In addition to the loop with counter, the "Répéter jusqu'à" structure can also be reproduced.

```
DO WHILE expression logique
  bloc
END DO
```

COMPLEMENTS ON THE DECLARATION OF ARRAYS

1. The number of dimensions can be between 1 and 7. An element of a table is thus defined by a number of indices equal to the number of dimensions. Unless contrary indication, the lower limit of the indices is equal to 1. To specify that an index varies between -36 and 36 can be indicated in the declarative instruction as in the following example.

Example

```
REAL, DIMENSION(-36:36) :: AUTOC
```

2. One can use expressions employing named constants (by means of the PARAMETER attribute) in the declarative instructions. It is even strongly recommended rather than using constants that can be changed, sometimes in an incoherent way.

Example

```
PROGRAM PRINCIP
   IMPLICIT NONE
   INTEGER, PARAMETER :: NBMOIS = 12
   REAL, DIMENSION (0:3*NBMOIS) :: AUTOC
   ...
   CALL SUBROU( AUTOC, NBMOIS)
   ...
END PROGRAM PRINCIP
```

3. In a subroutine, for arrays that are formal arguments (other than arrays defined locally or defined globally), integer variables can also be used in the specification of size. These variables need then to appear among the formal arguments. In the three cases, expressions (with integer value, of course) can be used.

Example (suite)

```
SUBROUTINE SUBROU( VECT, LARG )
   IMPLICIT NONE
   INTEGER, INTENT(IN) :: LARG
   REAL, INTENT(IN OUT), DIMENSION(0:3*LARG) :: VECT
   INTEGER :: I
   DO  I = 0, 3*LARG
     VECT(I) = I*VECT(I)
   ...
```

4. It is not necessary to transmit the dimensions of arrays, the subroutines being able to find them. Let us take again the end of the example of point 1 with a modified version of SUBROU. Notice here the use of an intrinsic function SIZE, which gives dimensions of an array.

Example

```
   ...
   CALL SUBROU( AUTOC)
   ...
END PROGRAM PRINCIP
SUBROUTINE SUBROU( VECT )
   IMPLICIT NONE
   REAL, INTENT(IN OUT), DIMENSION(0: ) :: VECT
   INTEGER :: I, LARG3
   LARG3 = SIZE(VECT) - 1
   DO  I = 0, LARG3
     VECT(I) = I*VECT(I)
   ...
```

By rewriting program MANIMA of paragraph 1.3, all the integer arguments are removed from subroutines LECMAT, ECRMAT and PROMAT.

5. It is also possible to use automatic arrays, for example on a temporary basis. Notice here another use of SIZE, which gives the extent of the dimensions of a table. Caution: the memory capacity is not recovered at the exit (this is an error corrected in FORTRAN 95).

Example

```
SUBROUTINE ECHANGE( A, B )
   IMPLICIT NONE
   REAL, INTENT(IN OUT), DIMENSION(:,:) :: A, B
   REAL, DIMENSION(SIZE(A,1), SIZE(A,2)) :: TEMP
   TEMP = A
   A    = B
   B    = TEMP
   RETURN
END SUBROUTINE ECHANGE
```

DYNAMIC ARRAYS

Contrarily to the arrays of FORTRAN 77, which are static i.e. of size specified at the compilation of the program, FORTRAN 90 makes it possible to use dynamic arrays by declaring them with the ALLOCATABLE attribute and by allocating the memory at execution time (supposing it is available!).

Example

```
PROGRAM GEANT
  IMPLICIT NONE
  INTEGER :: I, J, N, M, IERR
  REAL(8) :: S
  REAL, DIMENSION(:,:), ALLOCATABLE :: X
  READ(*,*) N, M
  ALLOCATE( X(N, M), STAT = IERR )
  IF( IERR == 0 ) THEN
    S = 0.0
    DO  I = 1, N
      DO  J = 1, M
        X(I, J) = I + J
        S = S + X(I, J)
      END DO
    END DO
    WRITE(*,*) 'Sommme = ', S
    DEALLOCATE( X )
  ELSE
    WRITE(*,*) 'La matrice est trop grande'
  END IF
  STOP
END PROGRAM GEANT
```

Remarks.

1. Allocated arrays lose their allocation by the DEALLOCATE instruction.

2. Dynamic allocation makes it possible to deal with problems whose dimensions are known only at execution time, including the use of temporary variables. Two matrices $N \times N$ called *A1* and *A2*, and three vectors of size *N*, called *V1*, *V2* and *V3* can be created, provided that N is not too large. One proceeds as follows.

```
PROGRAM DYNAM
  IMPLICIT NONE
  INTEGER :: N
  REAL, DIMENSION(:,:), ALLOCATABLE :: A1, A2      ! matrices
  REAL, DIMENSION(:), ALLOCATABLE :: V1, V2, V3    ! vecteurs
  READ(*,*) N
  ALLOCATE( A1(N, N), A2(N, N), V1(N), V2(N), V3(N) )
  CALL CALCUL( N, A1, A2, V1, V2, V3 )
...
END PROGRAM DYNAM
SUBROUTINE CALCUL( N, A1, A2, V1, V2, V3 )
  IMPLICIT NONE
  REAL :: A1(N, N), A2(N, N), V1(N), V2(N), V3(N)
...
```

For subroutine CALCUL, space necessary is allocated for the two matrices and the three vectors.

MANIPULATION OF ARRAYS

Sections of arrays can be used, for example:

`A(1:2, : )` rows 1 and 2 of A

`B( :, (/ 3, 6, 9/) )` columns 3, 6 and 9 of B

Among the new functions intended to act on arrays, let us mention (for the examples, C is a matrix dimensioned (2,3) with 1, 2, 3 on the 1st line and 4, 5, 6 on the 2nd line):

SIZE(array)	number of dimensions (Example: 2)
SIZE(array, dim)	extent along dimension *dim* (Example: SIZE(C,2)=3)
SHAPE(array)	vector of dimensions (Example: SHAPE(C)=(/2, 3/))
RESHAPE(initial, form)	array with dimensions according to vector *form*, with the elements of the *initial* array taken in natural order

RESHAPE(initial, forme,0,permutation)	Array with dimensions according to vector *forme*, with *permutation* of the indices of the *initial* array
COUNT(array)	Number of values true in the logical *array*
COUNT(array, dim)	Array without dimension *dim* containing the number of values true in the logical *array* in dimension *dim*
SUM(array)	sum of the elements of the *array* (Example: SUM(C)=21)
SUM(array, dim)	Array without dimension *dim* containing the marginal sums of the elements along dimension *dim* (Example: SUM(A, 1)=(/ 5, 7, 9 /))
DOT_PRODUCT(vec A, vecB)	Scalar product of two vectors (if vecA is complex, it is conjugated)
MATMUL(matA, matB)	Matrix product of *matA* and *matB* (at least one of both must be with 2 dimensions)
TRANSPOSE(matA)	Transpose of a matrix *matA*

Remark: Notice absence of any numerical procedure, like matrix inversion

Example. Function RESHAPE allows, in particular in the declarative instructions or arguments of subroutine, to transform the number of dimensions or the order of dimensions of an array.

```
REAL, DIMENSION(5, 2) :: DONNEE = &
           RESHAPE( (/1,2,2,4,6,17,11,23,19,30/), (/5,2/) )
```

COMPLEMENTS ON SUBPROGRAMS

1. The subroutines not contained in a module require an interface (see the example of program MANIMA in paragraph 1.3).

2. A name of a subprogram unit can be transmitted as an actual parameter when calling another subprogram. Function SUMSQ calculates an objective function. The optimization subroutine OPTIM is written for any function FUNCT working in the same way:

Example

```
SUBROUTINE OPTIM( FUNCT, VAR )
  IMPLICIT NONE
  REAL, DIMENSION(:), INTENT(IN OUT) :: VAR
  REAL :: OBJ
  INTERFACE
    FUNCTION FUNCT(X)      ! fonction générique à optimiser
      IMPLICIT NONE
      REAL :: FUNCT
      REAL, INTENT(IN), DIMENSION(:) :: X
    END FUNCTION FUNCT
  END INTERFACE
  NVAR = SIZE(VAR)
  ...! instructions choisissant les bonnes valeurs pour VAR
  OBJ = FUNCT( VAR )
  ...
END SUBROUTINE OPTIM
```

OPTIM declares FUNCT whose skeleton is defined by the INTERFACE block. Typically, we should use for OPTIM a library program of which the source code will not be modified (especially if we don't have it!) to change the name of the function to be optimized nor to transmit the data that need to be transmitted. The following call carries out the optimization of a specific function SUMSQ by using the data declared in module CDON. The interface block for SUMSQ can be noticed in the main program and that for OPTIM, which contains the interface block of FUNCT.

```
MODULE CDON
   IMPLICIT NONE
   REAL, DIMENSION(100) :: DONNEE
END MODULE CDON

PROGRAM OPTIDEMO
  USE CDON
  IMPLICIT NONE
  INTERFACE                    ! bloc d'interface pour SUMSQ
    FUNCTION SUMSQ(X)
      IMPLICIT NONE
      REAL :: SUMSQ
      REAL, INTENT(IN), DIMENSION(:) :: X
    END FUNCTION SUMSQ
  END INTERFACE
  INTERFACE                    ! bloc d'interface pour OPTIM
    SUBROUTINE OPTIM( FUNCT, VAR, NVAR )
    IMPLICIT NONE
    INTEGER, INTENT(IN) :: NVAR
    REAL, DIMENSION(NVAR), INTENT(IN OUT) :: VAR
    INTERFACE       ! bloc d'interface pour FUNCT dans OPTIM
      FUNCTION FUNCT(X)
        IMPLICIT NONE
        REAL :: FUNCT
        REAL, INTENT(IN), DIMENSION(:) :: X
      END FUNCTION FUNCT
    END INTERFACE
    END SUBROUTINE OPTIM
  END INTERFACE
  REAL, DIMENSION(2) :: PAR
  ...
  READ(*,*) DONNEE(1:100)    ! lecture des sonnées
  READ(*,*) PAR(1:2)         ! lecture des valeurs initiales
  CALL OPTIM( SUMSQ, PAR )   ! optimisation
  WRITE(*,*) PAR(1:2)        ! écriture des valeurs optimales
  ...
END PROGRAM OPTIDEMO

FUNCTION SUMSQ( PAR ) !satisfait à l'interface de FUNCT
  USE CDON  ! accès à DONNEE(100)
  IMPLICIT NONE
  REAL :: SUMSQ
  REAL, DIMENSION(:), INTENT(IN) :: PAR
  INTEGER :: I
  SUMSQ = 0.0
  DO I = 1, 100
    SUMSQ = PAR(1)*DONNEE(I)**PAR(2) ! mauvais exemple court
  END DO
  RETURN
END FUNCTION SUMSQ
```

1. The subroutines contained in a module share the variables declared in that module (see the example of program FACNC in paragraph 1.3). They do not require an interface block. If we could put SUMSQ but also OPTIM (what is not very realistic) in module CDON, here would be the result:

```
MODULE CDON
  IMPLICIT NONE
  REAL, DIMENSION(100) :: DONNEE
CONTAINS
FUNCTION SUMSQ( PAR ) !satisfait à l'interface de FUNCT
  REAL :: SUMSQ  ! IMPLICIT NONE et USE CDON pas nécessaire
  REAL, DIMENSION(:), INTENT(IN) :: PAR
  INTEGER :: I
  SUMSQ = 0.0
  DO I = 1, 100
    SUMSQ = PAR(1)*DONNEE(I)**PAR(2) ! mauvais exemple court
  END DO
  RETURN
END FUNCTION SUMSQ
SUBROUTINE OPTIM( FUNCT, VAR )
!                                  IMPLICIT NONE pas nécessaire
  REAL, DIMENSION(:), INTENT(IN OUT) :: VAR
  REAL :: OBJ
  INTERFACE                ! interface nécessaire!!!
    FUNCTION FUNCT(X)      ! fonction générique à optimiser
      IMPLICIT NONE
      REAL :: FUNCT
      REAL, INTENT(IN), DIMENSION(:) :: X
    END FUNCTION FUNCT
  END INTERFACE
  NVAR = SIZE(VAR)
  ...! instructions choisissant les bonnes valeurs pour VAR
  OBJ = FUNCT( VAR )
  ...
END SUBROUTINE OPTIM
END MODULE CDON

 PROGRAM OPTIDEMO
   USE CDON                      ! nécessaire
   IMPLICIT NONE                 ! nécessaire
 !                             blocs d'interface pas nécessaires
   REAL, DIMENSION(2) :: PAR
   ...
   READ(*,*) DONNEE(1:100)     ! lecture des sonnées
   READ(*,*) PAR(1:2)          ! lecture des valeurs initiales
   CALL OPTIM( SUMSQ, PAR )    ! optimisation
   WRITE(*,*) PAR(1:2)         ! écriture des valeurs optimales
   ...
 END PROGRAM OPTIDEMO
```

4. When calling a subprogram, it is equivalent to use the name of an array or its first element but another element can also be used. For example, the following calls calculate several sums of products, while making slip the shift between the two vectors.

```
 SUBROUTINE AUTOCOR( X, N, IRETMX, AUTOC )
   IMPLICIT NONE
   INTEGER, INTENT(IN) :: N, IRETMX
   REAL, INTENT(IN), DIMENSION(N) :: X
   REAL, INTENT(OUT), DIMENSION(IRETMX) :: AUTOC
   INTEGER :: IRET
   INTERFACE
   FUNCTION PRODSC( X, Y, M )
     IMPLICIT NONE
     REAL :: PRODSC
     INTEGER, INTENT(IN) :: M
     REAL, INTENT(IN), DIMENSION(M) :: X, Y
   END FUNCTION PRODSC
   END INTERFACE
   DO  IRET = 0, IRETMX
     AUTOC(IRET) = PRODSC( X, X(1 + IRET), N - IRET )
   END DO
   STOP
 END SUBROUTINE AUTOCOR

FUNCTION PRODSC( X, Y, M )
  IMPLICIT NONE
  REAL :: PRODSC
  INTEGER, INTENT(IN) :: M
  REAL, INTENT(IN), DIMENSION(M) :: X, Y
  INTEGER :: I
  PRODSC = 0.0
  DO  I = 1, M
    PRODSC = PRODSC + X(I)*Y(I)
  END DO
  RETURN
END FUNCTION PRODSC
```

It is obviously preferable to use arrays when they are expected, for example

```
AUTOC(IRET) = PRODSC(X(1:N - IRET), X(1 + IRET:N), N - IRET
)
```

and also by using the RESHAPE function.

5. Errors related to the order of the arguments of a subroutine or a function can be avoided by communicating the names of the formal arguments, as follows

```
CALL PROMAT( X = A, Y = B, Z = C, M = 8, N = 5, L = 9 )
```

An argument can be omitted (by the use of the OPTIONAL attribute in the declarative for the formal argument).

Function

PRESENT(...) makes it possible to test for the presence of an argument.

2. A subroutine or a function can be called themselves, either directly or indirectly. This is called recursivity. It must then be declared as recursive and to comprise a result variable (clause RESULT):

```
RECURSIVE FUNCTION FACTORIAL( N ) RESULT(RES)
  IMPLICIT NONE
  INTEGER, INTENT(IN) :: N
  INTEGER :: RES
  IF( N == 1 ) THEN
    RES = 1
  ELSE
    RES = N*FACTORIAL( N - 1 )
  END IF
  RETURN
END FUNCTION FACTORIAL
```

COMPLEMENTS ON OUTPUT AND INPUT

1. Input instructions must be protected against user errors. This is done with a complementary clause in the input instruction.

```
READ(*,*,IOSTAT = variable) list of variables
IF( variable /= 0 ) THEN
  WRITE(*,*) 'Erreur de lecture'
END IF
```

where the variable receives a null value if everything happened well, an error code in the contrary case.

2. This safety measure can be added to other input-output instructions (OPEN, WRITE) as well as to conversions from numeric to characters.

COMPLEMENTS ON SPECIFICATION FORMATS

1. In addition to the use of a format specification in the form of a constant, it is also possible to use a variable and (survival of FORTRAN 66) a label. More precisely, the three following writings are equivalent.

```
!cas1
      CHARACTER(50) FMT11
      ...
      WRITE(*, '(3F10.3)') A, B, C
!cas2
      FMT11 = '(3F10.3)'
      WRITE(*, FMT11) A, B, C
!cas3
      WRITE(*, 11) A, B, C
   11 FORMAT( 3F10.3 )
```

2. In addition to the already specified specifications, one can in particular use the repeaters (as the 3 that precedes F10.3 in the example above, meaning that three numbers are to be placed on a line in the same specification F10.3) as well as the following specifications:

En.d	scientific notation with *n* digits and *d* significant digits, $n > d + 6$
Dn.d	the same for the double precision numbers
Gn.d	equivalent to Fn.d if the number is neither too large nor too small, to En.d, if the scientific notation imposes itself
‘x...x’	characters to be reproduced without change (on exit only)
/	jump to the following line

OTHER FEATURES OF FORTRAN 90

Among the possibilities, let us mention :

- almost complete compatibility with FORTRAN 77 (see

paragraph 1.5);

- concept of the kind of a number (" kind "), more general for example that the distinction made here between REAL(4) and REAL(8), allowing (with some limitations related to the processor used: no quadruple precision on an Intel Pentium processor, for example) to specify the interval of variation and the number of significant digits;
- to test the number of binary significant figures (function DIGITS) or decimal (PRECISION), to find the smallest positive number that, added to 1, is equal to 1 (EPSILON), the greatest number that can be represented (HUGE) or the smallest one (TINY)
- structures of data richer than arrays (records, objects) called derived types (declarative TYPE), with the possibility of defining operations on these objects and of even extending the action of the existing operational symbols (+, .AND., etc.), with access to the components of the object (objet%composant);
- structures of data such as lists or trees can be represented by using pointers;
- use of pointers for dynamic arrays;
- masks on operations on arrays (instruction and structure WHERE);
- use of interface blocks to call upon (similarly to intrinsic functions) under only one generic name several program units acting on arguments of the different types (for example, a subroutine acting on real numbers and another on double-precision numbers);
- private or public attributes for the data and the subroutines of the modules;

- functions with values of array form;
- intrinsic functions that act element by element on the argument of array type (for example SQRT); ·
- intrinsic functions for the handling of character strings (ADJUSTL, ADJUSTR, REPEAT, TRIM, etc), bits handling, reduction and handling of arrays, transfers between different types of data (TRANSFER), time and dates (DATE_AND_TIME, SYSTEM_CLOCK), pseudo-random numbers (RANDOM_NUMBER, RANDOM_SEED);
- instructions of input-output without advance;
- new format descriptors.

INTRODUCTION TO FORTRAN 95

FORTRAN 95 is a light update compared to FORTRAN 90. The innovations are relative to:

- so-called pure procedures (subroutines and functions) i.e. declared without side effects, allowing by this fact additional possibilities of optimization by the compiler in a parallel processor environment, for example;
- the user can write procedures that act element by element on arrays;
- automatic desallocation of arrays allocated at the exit of a subroutine;
- instruction FORALL, for example

```
FORALL(I = 1 : N, J = 1 : M, Y(I,J) /= 0.) &
                                    A(I,J) = X(I)/Y(I,J)
```

and the FORALL structure;

- some new functions among which CPU_TIME

1.5 INSTRUCTIONS OF FORTRAN IV AND FORTRAN 77 TO BE DECIPHERED

BASIC INSTRUCTIONS

Names of variables and program units

Types

Variables

Named constant

First initialisation of a variable

Branching instructions

Repetitive instruction

Fortran 77 or fixed mode source Fortran 90 presentation

ARRAYS

Type and size declaration

Handling of arrays

COMPLEMENTS ON PROGRAM UNITS

LOCAL AND GLOBAL VARIABLES

OUTPUT AND INPUT

USE OF FORTRAN 77 PROGRAMS WITH FORTRAN 90 AND FORTRAN 95 COMPILERS

BASIC INSTRUCTIONS

NAMES OF VARIABLES AND PROGRAM UNITS

Maximum 6 characters (most compilers accepted more)

TYPES

Type REAL(4) is called REAL (sometimes REAL*4)

Type REAL(8) is called DOUBLE PRECISION (sometimes REAL*8).

VARIABLES

Declaratives instructions of Fortran 77 don't accept "::". IMPLICIT NONE do not exist.

Variables don't need to be declared (except arrays).

The implicit declarative rules are as follows :

1. The default implicit type is based on the ***initial*** of the name:

"INTEGER" for names with initials I, J, K, L, M, N

"REAL" for all other initials

Example

NBART	integer
X	real

2. By default, logical and character variables need to be declared (explicitly) as well as those variables for which we don't want implicit declaration based on the initial.

3. It is possible to change the default implicit declaration and circumvent the selection of initials mentioned above.

Example

```
IMPLICIT DOUBLE PRECISION (A-H, O-Z )
```

Declarative instructions of Section 1.4 should be read in Fortran 77:

```
REAL MONTAN, TAUX

LOGICAL TROUVE

INTEGER ENTIER

CHARACTER NOM* 20, TITRE* 80

CHARACTER* 10 MOT1, MOT2
```

NAMED CONSTANT

By means of the declarative instruction PARAMETER it is possible to give a name to a constant :

```
PARAMETER (nom = valeur)
```

Example

```
PARAMETER (N = 100, PI = 3.14159D0 )
```

FIRST INITIALISATION OF A VARIABLE

A variable can be initialised *at program loading time* thanks to instruction DATA. Note this is an executable instruction, not a declarative instruction.

```
DATA variable /valeur/
```

Example

```
DATA N /100/

DATA ONE, ZERO, TWO /1.0D0,0.0D0,2.0D0/

DATA (TAUTVA(I), I = 1, 4) /6.,20.5,25.,33./
```

A variable can then receive another value, because of the execution of an assignment statement, for example. It is necessary to insist on the fact that initialization takes place only once, before the execution of the program. This type of initialization does not replace the assignment statements used

usually to initialize a counter or an accumulator, in particular inside a part of program that can be carried out several times (loop, subroutine).

Remark. It is not allowed to initialize by DATA a formal argument of a subroutine. It is not allowed either to initialize by DATA a variable that belongs to a COMMON block, except inside a BLOCK DATA subprogram (see later).

BRANCHING INSTRUCTIONS

These instructions do exist mainly in Fortran 66 programs. They should be read correctly, never written.

1. The unconditional branching instruction (cause of 'spaghetti' style programs)

   ```
   GO TO étiquette
   ```

   Remark. The label must be in the same program unit (hopefully!).

2. The conditional branching instruction or computed GO TO

   ```
   GO TO (ét1, ét2,..., étn) expression entière
   ```

Branches to label *ét1* if the expression equals 1, *ét2* if it equals 2, etc.

3. The arithmetic branching instruction

   ```
   IF(expression numérique) étnég, étnul, étpos
   ```

It sends to label *étnég* if the expression has a strictly negative value, to label *étnul*, if it is equal to zero, and to label *étpos*, if it is strictly positive.

REPETITIVE INSTRUCTION

Fortran 77 has only one repetitive structure, the loop with counter, formed with the DO instruction

```
        DO    étiquette    variable  =  début,  fin
        Boucle  (une  ou  plusieurs  instructions )
étiquette CONTINUE
```

Example.

1 5	6	7 72	73...

		S = 0.0
		DO 100 I = 1, N
		  READ(*,*) NOMBRE
		  S = S + NOMBRE	

100		CONTINUE	

RULES

Etiquette : label , unsigned integer number of 5 digits maximum that must be in columns 1 to 5, and be unique in each program unit

Variable : counter of type INTEGER that cannot be the object of an assignment in the loop

Début, Fin : respectively, starting and ending value of *variable*, expressions of type INTEGER ; if *Début* > *Fin* , it does not occur anything (the loop is not carried out). One can nevertheless simulate an ITERER structure, while proceeding as follows, while employing for *infini* a very large finite number:

```
        DO    étiquette0    variable  =  1, infini
             bloc1  d'instructions
        IF( expression logique ) GO TO étiquette1
             bloc2  d'instructions
étiquette0 CONTINUE
étiquette1 . . .
```

Remark. It should be noticed that *étiquette1* follows immediately instruction CONTINUE. Notice also the use of a GO TO instruction in order to simulate sortir si. In structured programming, it is the one and only use of instruction GO TO that will be allowed for.

Example. Data entry with a sentinel equal to 0 and computation of the sum of the data

1 5	6	7 72	73...

		`S = 0.0`
		`DO 200  J = 1, 9999`
		`  WRITE(*,*) 'Entrez une donnée'`
		`  READ(*,*) DONNEE`
		`IF( DONNEE .EQ. 0.0 ) GO TO 201`
		`  S = S + DONNEE`
		`200 CONTINUE`
		`201 WRITE(*,*) 'Somme = ', S`	

Remarks

1. Note that the number of data items is limited to 9999, which means a sufficiently large number.
2. Note that the labels 200 and 201 are used one and only once in columns 1 to 5 and once starting with column 7.
3. In order to ensure readability of the program, it is recommended to put the labels in natural order.
4. Several loops can be embedded. N.B. In FORTRAN 90, it is necessary that the CONTINUE instructions are distinct.

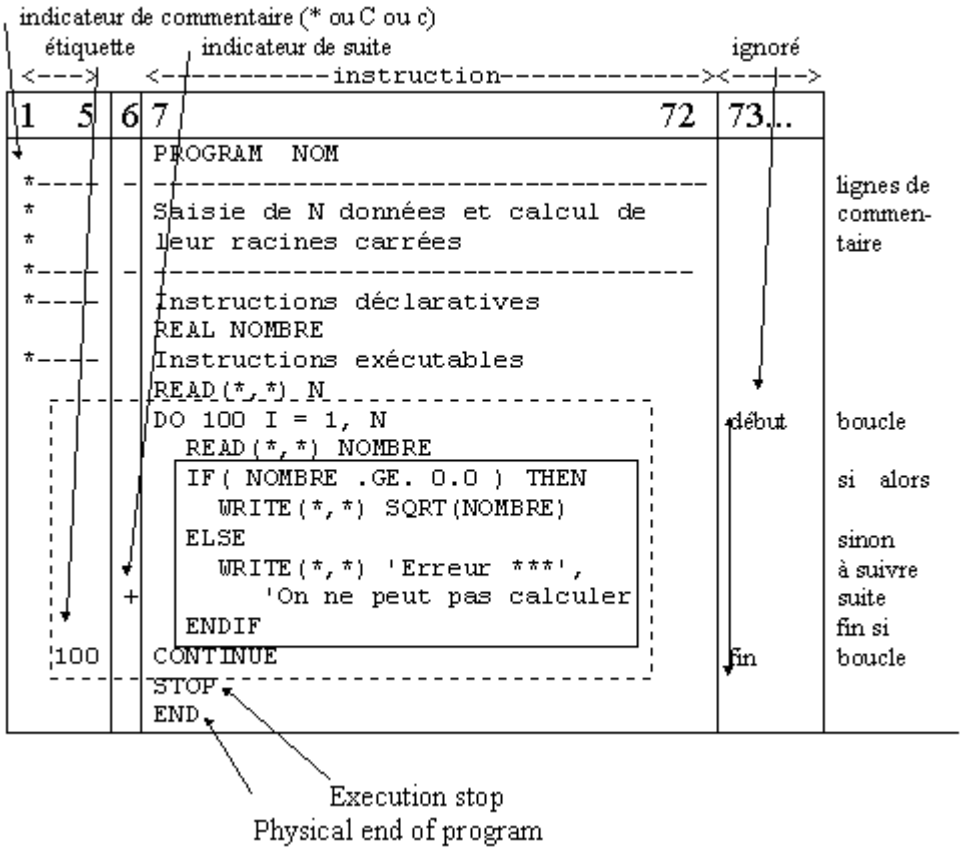
```
DO étiquette compteur = début, fin, pas
```

FORTRAN 77 OR FIXED MODE SOURCE FORTRAN 90 PRESENTATION

In addition to the fixed mode source presentation of the instructions between columns 6 to 72 (see paragraph 1.2), it is necessary to give the following details:

1. The continuation lines are indicated by a character in column 6, other that a space but also other than zero.
2. In FORTRAN 77, the lines with *, C or C in the first column are comment lines.

Compare the following figure with that of paragraph 1.3:


indicateur de commentaire (* ou C ou c)
étiquette
indicateur de suite
ignoré
<---><----------instruction----------><---->
1 5 6 7 72 73...
PROGRAM NOM
*---- - ----------
* Saisie de N données et calcul de
* leur racines carrées
*---- - ----------
*---- Instructions déclaratives
REAL NOMBRE
*---- Instructions exécutables
READ(*,*) N
DO 100 I = 1, N
début
READ(*,*) NOMBRE
IF(NOMBRE .GE. 0.0) THEN
WRITE(*,*) SQRT(NOMBRE)
ELSE
WRITE(*,*) 'Erreur ***',
+ 'On ne peut pas calculer
ENDIF
100 CONTINUE
fin
STOP
END
lignes de commen-taire
boucle
si alors
sinon
à suivre
suite
fin si
boucle
Execution stop
Physical end of program

ARRAYS

TYPE AND SIZE DECLARATION

Example

Either only one declarative instruction:

```
REAL TAUX(5)

REAL MAT(5,3)
```

But only size can be declared (either using another declarative for declaring type, or assuming implicit declaration)

```
DIMENSION TAUX(5)

DIMENSION MAT(5,3)
```

In the absence of dynamic arrays, a simple method to manage here a form of dynamic dimension in FORTRAN 77. If WKSP is an array of dimension 1000, one can place there two matrices $N \times N$ called *A1* and *A2*, and three vectors of size *N*, called *V1*, *V2* and *V3*, provided that *N* is not too large. One proceeds as follows.

Example

```
PROGRAM DYNAM
PARAMETER (LWKSP = 1000)
DIMENSION WKSP(LWKSP)
READ(*,*) N
IA1 = 1
IA2 = IA1 + N**2
IV1 = IA2 + N**2
IV2 = IV1 + N
IV3 = IV2 + N
IF( IV3 + N .GT. LWKSP ) THEN
  WRITE(*,*) 'WKSP n''est pas assez grand'
  STOP
END IF
CALL CALCUL( N, WKSP(IA1), WKSP(IA2), WKSP(IV1), WKSP(IV2), WKSP(IV3)
)
...
END
SUBROUTINE CALCUL( N, A1, A2, V1, V2, V3 )
DIMENSION A1(N, *), A2(N, *), V1(*), V2(*), V3(*)
...
```

For subprogram CALCUL, the necessary space is allocated for

the two matrices and the three vectors.

In subprograms it is possible to handle strings of variable length, defined in declarative instructions of type

Example

```
CHARACTER CH1*(*)

CHARACTER*(*) CH2, CH3
```

and arrays with the same specification '*' for the last dimension of an array (which plays no role, given that the elements are located according to the order defined by their dimensions.

Example

```
PROGRAM PRINCIP
PARAMETER (NBMOIS = 12, NBAN = 20)
DIMENSION AUTOC(0:3*NBMOIS), DONNEE(NBAN, NBMOIS)
...
CALL SUBROU( AUTOC, NBMOIS, DONNEE, NBAN )
...
END
SUBROUTINE SUBROU( VECT, LARG, TABL, LONG )
DIMENSION VECT(0:*), TABL(LONG, *)
...
```

HANDLING OF ARRAYS

Arrays are handled element by element. In Fortran 77, the only case where the name of an array is employed is in a list of parameters of a subprogram.

Example

```
      SUBROUTINE PROMAT( X, Y, M, N, L, Z )
      DIMENSION X(M, N), Y(N, L), Z(M, L)
      DO 300  I = 1, M
        DO 200  J = 1, L
          Z(I,J) = 0.0
          DO 100  K = 1, N
            Z(I, J) = Z(I, J) + X(I, K)*Y(K, J)
  100     CONTINUE
  200   CONTINUE
  300 CONTINUE
      RETURN
      END
```

COMPLEMENTS ON PROGRAM UNITS

No attributes are allowed in declaratives of Fortran 77, including INTENT. Modules and recursive functions don’t exist either. The last instruction of a program unit is END without anything else (not END SUBROUTINE name). Dimensions of arrays are not transmitted automatically. They must be transmitted to subprograms by the parameter list. In addition to the program units PROGRAM, SUBROUTINE and FUNCTION, there is a subprogram BLOCK DATA that is used only to initialize variables included in common blocks at loading time, using DATA instructions. The BLOCK DATA subprogram cannot contain executable instructions other than DATA.

Example

```
BLOCK DATA INITIAL
REAL TAUTVA(4)
COMMON /CTAUX/ NBTAUX, TAUTVA
DATA NBTAUX / 2 /
DATA (TAUTVA(I), I = 1, 2) / 6.0, 21.0 /
END
```

LOCAL AND GLOBAL VARIABLES

All variables are local to the program unit in which they are defined. Global variables of Fortran 77 are defined by the declarative instruction COMMON. They are global in all program units where they appear in a COMMON instruction but not elsewhere. The names of the variables can be different from one subprogram module to another (but it is not recommended).

SYNTAX

```
COMMON /nom de bloc/ liste de variables
```

EXAMPLE.

Here is the main program of emission of invoices and credit notes where the name of the company is placed in a COMMON block called COMSOC.

```
1    5  6 7                                                72  73...
      PROGRAM FACNC
      CHARACTER*30 NOMSOC
      COMMON /COMSOC/ NOMSOC
*
      READ(*, '(A30)') NOMSOC
      READ(*,*) N
*---- - Initialisation des taux de TVA
      CALL INITVA
*---- - Réalisation de N factures/notes de crédit
      DO 10  I = 1, N
        READ(*,*) LECHOI
        IF( LECHOI .EQ. 1 ) CALL FACTUR( I )
        IF( LECHOI .EQ. 2 ) CALL NOTCRE( I )
   10 CONTINUE
      STOP
      END
```

```
      SUBROUTINE INITVA
      REAL TAUTVA(4)
      COMMON /CTAUX/ NTAUX, TAUTVA
      NTAUX = 2
      TAUTVA(1) = 0.06
      TAUTVA(2) = 0.21
      RETURN
      END
```

```
      SUBROUTINE FACTUR( NUMERO )
      CHARACTER*30 TITRE
      COMMON /COMSOC/ TITRE
      REAL TAUTVA(4)
      COMMON /CTAUX/ NTAUX, TAUTVA
      READ(*,*) PRIX, NUMTAU
      WRITE(*,*) TITRE, 'Facture ', NUMERO
      IF( NUMTAU .GE. 1 .AND. NUMTAU .LE. NTAUX )
      THEN
     +   WRITE(*,*) PRIX*(1.0 + TAUTVA(NUMTAU))
      END IF
      RETURN
      END
```

REMARK

1. The name of the company NOMSOC placed in the common block is recovered in subprogram FACTUR. In the same way the VAT rates initialized in subprogram INITVA are recovered in subprogram FACTUR.
2. One can only use arrays with fixed dimensions in a common block.
3. The character variables cannot be in same common block as the other types.

OUTPUT AND INPUT

The following instructions are equivalent in Fortran 77 to READ(*,*) et WRITE(*,*), respectively :

```
READ*, liste de variables
PRINT*, liste d'expressions
```

USE OF FORTRAN 77 PROGRAMS WITH FORTRAN 90 AND FORTRAN 95 COMPILERS

Fortran 77 programs can be compiled without modification in most Fortran 90 compilers. This is not true for programs using instructions of Fortran IV. The following extensions of Fortran are likely to pose problems:

- type specifications INTEGER*2, REAL*4, REAL*8, COMPLEX*16
- functions DREAL, DIMAG, DCMPLX for COMPLEX*16 numbers
- use of the different types in the sequence of formal parameters and the sequence of effective parameters arguments
- character strings of form *n*H

Fortran 77 programs cannot be compiled anymore by Fortran 95 compilers without modification, not more than programs written with instructions of Fortran IV. Indeed are removed :

- loop indices of DO loops of real or double precision type
- ASSIGN and assigned GO TO instructions
- branching towards instruction END IF
- the PAUSE instruction
- the descriptor of format *n*H.

Are also to be avoided, because being perhaps removed in the next version (FORTRAN 2000?):

- the INCLUDE instruction for including a file (use modules instead)
- the DO WHILE instruction (use DO instead)

- fixed source presentation and thus spaces in constants and names (use the free source presentation)
- instructions computed GO TO (use SELECT CASE)
- the Fortran 77 notation CHARACTER*... instead of CHARACTER(LEN=...)
- the instructions DATA among executable instructions
- the statement function (use an external functions, a subroutine or a module)
- call without parameter list (…) (use the empty list ( ) instead
- results of type character with a variable length
- declarations of the DOUBLE PRECISION type, instead of REAL(8)
- use of specific intrinsic functions names (FLOAT instead of REAL, AMAX1 or MAX0 instead of MAX, IABS instead of ABS, etc).

Finally, it is not recommended to use the following instructions that are redundant and are thus likely to disappear in the future:

- the arithmetic IF instruction (use IF THEN / ELSE IF / ELSE / END IF)
- the alternate RETURN
- ENTRY instructions (multiple entry to a subprogram)
- the COMMON blocks and the instructions EQUIVALENCE, DATA and subprograms BLOCK DATA,

and in a more general way all that was introduced into the

present section.

1.6 USE OF SCIENTIFIC LIBRARIES

The richness of FORTRAN is provided by its modular character and the existence of free domain scientific libraries, in particular for linear algebra (LINPACK) and statistics (Journal of the Royal Statistical Society Series C Applied Statistics), and commercial libraries (book of Press *et al.*, NAG, IMSL). We illustrate the case of NAG here.

HOW TO USE LIBRAIRIES

RULES

1. Seek the subroutines in the index, the table of contents or with the assistance of suitable software.
2. Choose the most adapted subroutines (precision, reputation, facility of use...).
3. Read the documentation of the subroutine chosen (and keep a copy for later reference to it)
4. Insert very carefully the calling instructions of the subroutine, included management of the error code.

EXAMPLE

One wants to calculate the probability of significance p associated with the traditional test for the mean, of the hypothesis H_0: $m = 0$, with respect to the alternative hypothesis H_1: $m < 0$, by considering the asymptotic distribution of the statistic.

$$z = \frac{\bar{x}}{s / \sqrt{n-1}}$$

where $\bar{x}$ is the sample mean and s, the standard deviation of the sample of size n. We need the distribution function Φ of the unit normal law $N(0,1)$ to be evaluated at point z: $p = \Phi(z)$. Suppose we decide for example to use the NAG on system **aster** of the "Centre de Calcul ULB-VUB".

1. We seek in the index (volumes in the principal libraries) or with the assistance of suitable software.
2. The module S15ABF is found and it appears to be a function.

3. An excerpt of the documentation relative to module S15ABF is enclosed in the appendix.
4. The instruction for calling S15ABF are followed; note that there is no error code in this case but we have nevertheless written the appropriate instruction.

Remark. In fact, documentation indicates that IFAIL is never different from zero at the exit of S15ABF, because a data error cannot happen, since *z*, the only input argument, can be arbitrary.

1 5	6	7 72	73...

		`Z = XBAR/SQRT( VAR/(N - 1) )`
		`IFAIL = 0`
		`P = S15ABF( Z, IFAIL )`
		`IF( IFAIL .GT. 0 ) THEN`
		`  WRITE(*,*) 'Erreur fatale dans S15ABF'`
		`  STOP`
		`ELSE`
		`  WRITE(*,*) 'Probabilité de signification = ',`
		`P`
		`END IF`	

1.7 PREPARATION OF TEST DATA SETS

RULES

1. Start from the principle (true in 99,99% of cases) that your program is BAD; without that, it is psychologically very difficult to find errors, a fortiori all the errors.

2. Always test the program on examples where the results are known. They can be obtained using another confirmed method or computed separately (using SAS, GAUSS, Mathematica, Matlab or Excel for that purpose).

3. Select test data sets that explore most frequent paths in the program, but also exceptions. The program should work even with bad data, produce error messages and stop.

4. Keep the data sets and their results so that it is possible to run again the test data sets every time the program is modified in order to be sure that it is still good (compare each time with the good previous results).

5. A short and clear program is will be easier to read and correct. It will contain fewer errors. Subdivide a program in modules (taking care of the fact that most errors will be bad calls of these modules) and use confirmed algorithmic structures.

6. If the program becomes confusing, rewrite it from scratch; compare the results with those of the previous version on all test data sets, correcting the mistakes in both versions. Keep the two versions in the same main program as long as possible. If some parts become useless (display of temporary results, call of checking subprograms), remove them logically, not physically (putting them in a conditional structure) or transform them in comments if they are really not necessary. Keep a copy of the full complete version, just in case.

EXAMPLE

We provide here some examples of data sets for the programs ESSVARx (see Section 1.2 Fortran primer)

1.

```
12
1 2 3 4 5 6 7 8 9 10 11 12
```

should give an average of 6.5 and a variance $(n^2 - 1)/12$ = 11,91666..., hence $p = 1$.

2.

```
3
1000001 1000002 1000003
```

should have an average equal to 1000001 and a variance of 2/3 = 0,6666..., hence $p = 1$.
Check carefully! What happened if one adds one or two zeros?

3.

```
1
1000
```

will provide an average equal to 1 and variance 0, hence z is infinite and p is undetermined.
The program need to be corrected to avoid that case.

4.

```
12
-5 -4 -3 -2 -1 0 1 2 3 4 5 6
```

should give an average of 0.5 and variance $(n^2 - 1)/12$ = 11,91666..., hence $z = 0.48038446141$ and $p = 0.6845229846$.

5.

```
12
```

```
-2 -1 0 1 2 3 4 5 6 7 8 9
```

should give as average 3.5 and as variance $(n^2 - 1)/12$ = 11,91666..., hence z = 3.362691229 and p = 0.9996140667.