Bernard PICHON (*)
Département Cassiopée à Nice
UMR Xyzt du CNRS
Observatoire de la Côte d'Azur
BP 4229
06304 Nice Cedex 04
Courriel : Bernard.Pichon@obs-nice.fr

Bientôt disponible sur : http://www.obs-nice.fr/pichon/

FORTRAN 2003 : QUELLES NOUVEAUTÉS ?

Nouvelle Norme ISO : Langage de base :1539-1:2004

mais aussi des Normes collatérales : 1539-2 : 2000 dite ISO_VARYING_STRING
1539-3 préprocesseur Fortran dit aussi CoCo

Mais aussi une norme vivante : Caractéristiques *dépréciées*, obsolètes ou détruites

Remarque : Disponibilité de (*bons*) compilateurs, de bibliothèques (e.g. Lapack95).

Références à d'autres Normes (fun !) :

ISO/IEC 646:1991 is the international equivalent of ANSI X3.4-1986, commonly known as ASCII.

ISO 8601:1988 , Data elements and interchange formats— Information interchange--Representation of dates and times.

ISO/IEC 9899:1999 , Information technology--Programming languages--C.

ISO/IEC 10646-1:2000 , Information technology--Universal multiple-octet coded character set (UCS)— Part 1: Architecture and basic multilingual plane.

IEC 60559 (1989-01) , Binary floating-point arithmetic for microprocessor systems.
Because IEC 60559 (1989-01) was originally IEEE 754-1985 , Standard for binary floating-point arith-metic, and is widely known by this name, this standard refers to it as the IEEE standard.

(*) : Un des 3 membres représentant la France (et le seul du CNRS !)
du groupe de travail de l'ISO-IEC/JTC1/SC22/WG5 (Fortran)

0 Rappel sur Fortran 90/95

Format libre (jusqu'à 132 caractères par ligne !)
Caractère de continuation en fin de ligne : `&`
Caractère de commentaire (de fin et de début de ligne) : `!`
Rôle significatif du blanc
Délimiteurs de chaînes de caractères : `'` ou `"` `( 'a',"C'est l'été" )`
Noms de variables pouvant atteindre 31 caractères (dont _)
Les minuscules font partie du standard et sont identiques aux majuscules
Séparateurs d'instructions : `;`
Écriture des opérateurs logiques : `<` , `>` , `<=` , `>=` , `==` , `/=`
Directive `INCLUDE`
Quelques caractères d'usage *réservé* (voir par la suite) : `(/` , `/)` , `%` , `::`

Instruction : `Implicit None`
Possibilité de nommer l'instruction END associée à une unité de programme :
Possibilité de donner des noms aux constructions IF et DO :
Plus de possibilités pour définir des constantes nommées, e.g. pour des tableaux :
Liste d'attributs par objet et initialisation des variables :
Remarque : Attribut `Save` (*très*) recommandé

Constructions DO :
`DO`, `END DO`, `CYCLE` et `EXIT`
Construction CASE :
`SELECT CASE` , `END SELECT` , `CASE` et `CASE DEFAULT`
Construction WHERE :
`WHERE` , `ELSEWHERE` , `END WHERE`
Construction FORALL: `FORALL` et `END FORALL`

Amélioration des E/S avec des fonctionnalités supplémentaires :
E/S par listes nommées : `NAMELIST`
`Advance="No"` , `Status="Replace"`
`Action="Read"` , `"Write"` , `"ReadWrite"`
`Position="Rewind"` , `"Append"` , `"AsIs"`
voir aussi : `Blank=` , `Delim=` , `Pad=`
Nouvelle instruction du type 'Inquire' : Instruction `Inquire` par liste
Amélioration de quelques formats :
Rappel (F77) : formats dynamiques, descripteur " `:` "
Descripteur d'édition pour des entiers : `B` , `O` , `Z`
Descripteur d'édition pour des réels : `EN` , `ES`
Descripteur de contrôle : `TR` , `TL` et aussi `BN` et `BZ`
Format de sortie minimal (F95) *e.g.* `I0`

Quelques fonctions intrinsèques supplémentaires :

pour des nombres : `CEILING` et `FLOOR , MODULO`

pour des caractères : `ACHAR` et `IACHAR`

pour des chaînes de caractères :

`LEN_TRIM , TRIM , ADJUSTL , ADJUSTR`

aussi : `VERIFY , SCAN , REPEAT`

pour des appels systèmes :

`RANDOM_NUMBER , RANDOM_SEED ,`
`DATE_AND_TIME , SYSTEM_CLOCK , CPU_TIME`

pour des manipulation de bits :

`BTEST , IBCLR , IBSET , IAND , IEOR , IOR ,`
`NOT , ISHIFT , ISHITC`

aussi `BIT_SIZE , IBITS , MVBITS`

pour manipuler directement dans la mémoire : `TRANSFER`

La grande nouveauté : LA MANIPULATION DES TABLEAUX

Les procédures élémentaires :

Les fonctions (mathématiques) de base sont devenues élémentaires

On peut aussi créer des fonctions : parallèlisables (`PURE`)
élémentaires (`ELEMENTAL`)

Constantes de type tableau, Constructeurs de vecteurs :

Sections de tableaux (*i.e.* sous-tableaux) y compris adressage indirect :

Calculs sur des tableaux complets ou des sections :

Allocation dynamique :

Attribut `ALLOCATABLE`

Instructions `ALLOCATE` et `DEALLOCATE`

Fonction d'interrogation `ALLOCATED`

Objets automatiques : Très utile pour des tableaux de travail

Nouvelles procédures intrinsèques pour utiliser des tableaux :

`LBOUND , UBOUND , SIZE , SHAPE`
`MINVAL , MAXVAL , MINLOC , MAXLOC`
`SUM , PRODUCT` (arguments optionnels : `DIM` et `MASK`)
`DOT_PRODUCT , MATMUL , TRANSPOSE`
`RESHAPE`
`ALL , ANY , COUNT`
`CSHIFT , EOSHIFT`
`MERGE , PACK , UNPACK , SPREAD`

Fortran , génie logiciel et POO

Arguments nommés (nom-clé d'arguments) et arguments optionnels :
Attribut `OPTIONAL` , Fonction d'interrogation `PRESENT`
Mode d'association des arguments (vocation) :
Attribut (ou instruction) `INTENT` de valeur `In` , `Out` ou `InOut`
Interfaces explicites, visibles, implicites (USE), sans objet (F77) :
Déclaration `INTERFACE` et `END INTERFACE`

Création de module : Instructions `MODULE` et `END MODULE`
Importation de ressources avec `USE` avec, évt., la clause `ONLY`

Procédures internes : `CONTAINS`

Clause `RESULT` (écriture des fonctions et récursivité).

Il existe aussi des pointeurs : Attribut `POINTER` et `TARGET`

Définition de type dérivé : Instruction `TYPE` et `END TYPE`
Initialisation complète des objets dérivés
Définition (déclaration), Instanciation et Initialisation :
Référence (accès) à un champ : `%`
Création d'opérateurs et Surcharge d'opérateurs préexistants
Attribut `PUBLIC` et `PRIVATE` : On rentre dans la POO !

I F2003 et le CONFORT D'ÉCRITURE PROGRAMMATION

Quelques fonctions intrinsèques supplémentaires :

pour savoir ce qui se passe en cours de lecture/écriture :

```
IS_IOSTAT_END
IS_IOSTAT_ERR
```

pour travailler avec la ligne de commande :

```
COMMAND_ARGUMENT_COUNT
GET_COMMAND
GET_COMMAND_ARGUMENT
```

pour communiquer avec son environnement :

```
GET_ENVIRONMENT_VARIABLE
```

pour faire une réalocation plus facilement :

```
MOVE_ALLOC
```

pour ne plus chercher certaines choses évidentes :

```
NEW_LINE
```

pour choisir un jeu de caractères :

```
SELECTED_CHAR_KIND
```

pour permettre une POO bien propre :

```
EXTENDS_TYPE_OF
SAME_TYPE_AS
```

une amélioration dans :

```
NULL
```

L'énumération : (type *enum* du C)

```
ENUM , BIND(C)
    ENUMERATOR :: Blanc , Noir
    ENUMERATOR :: Rouge = 5 , Vert , Bleu
END ENUM
```

Quelques éclaircissements et modifications (*mineures*) :

plus d'interprétation en sortie concernant la colonne 1 !
apparence du zéro dans, par exemple, `Write(*,*) 0.0`
prise en compte du « zéro négatif » comme second opérande dans des fonctions comme `SIGN` , `ATAN2` et `SQRT` , `LOG` lorsque ' Z = X + i Y '
le second argument des fonctions `MOD` et `MODULO` ne peut être nul !

Le module : **`ISO_FORTRAN_ENV`**

Connaître les numéros d'unité logique standards :
`INPUT_UNIT`
`OUTPUT_UNIT`
`ERROR_UNIT`

Connaître les codes d'erreur utilisés lors des E/S :
`IOSTAT_END`
`IOSTAT_EOR`

Connaître la taille en bits des unités de mesure :
`CHARCTER_STORAGE_SIZE`
`NUMERIC_STORAGE_SIZE`
`FILE STORAGE SIZE`

Amélioration des E/S avec des fonctionnalités supplémentaires :

`WAIT`
`FLUSH`
`IOMSG =`
E/S (Sorties) asynchrones

➔Amélioration de quelques formats :
Rappel : ' **:** ' aussi `T` , `TL` , `TR` et `S` , `SS` , `SP` aussi `BN` , `BZ`
Format 'nul' e.g. `I0`

Types dérivés : `DT`

Arrondis : `RU` , `RD` , `RZ` , `RN` , `RC` et `RP`
Séparateur décimal : `DC` ou `DP` (dangereux !)
Rmq : Possibilité de le définir aussi dans `OPEN`

Constructeurs de tableaux :

```
(/ . . . . . /)
 [ . . . . . ]
```

En vrac, quelques autres mots clés (*à mettre, peut-être ailleurs !*) :

```
ASYNCHRONOUS
BIND
PROCTECTED
VALUE
VOLATILE

INTRINSIC
EXTENDS
IMPORT
ASSOCIATE
SELECT TYPE
```

LA GESTION DES EXCEPTIONS IEEE

3 modules intrinsèques : `IEEE_EXCEPTIONS` (« inclus » dans le suivant)
`IEEE_ARITHMETIC`
`IEEE_FEATURES`

Support complet par :

```
USE, INTRINSIC :: IEEE_ARITHMETIC
USE, INTRINSIC :: IEEE_FEATURES
```

En pratique, *je pense* que le premier suffira dans la plupart des cas !

Si l'accès à l'un des (trois) modules du type `IEEE_*` est possible, alors c'est OK pour `IEEE_SUPPORT_FLAG(IEEE_UNDERFLOW)` et `IEEE_SUPPORT_FLAG(IEEE_DIVIDE_BY_ZERO)` et cela ∀ X .

Remarque : Si X est précisé, alors c'est qu'il existe au moins un **X** pour le quel c'est possible.

Le module : **IEEE_FEATURES**

Le type dérivé et ses valeurs :

`IEEE_FEATURES_TYPE` peut prendre les valeurs suivantes :

`IEEE_INEXACT_FLAG` ,
Si l'accès y est possible, alors c'est OK pour `IEEE_SUPPORT_FLAG(IEEE_INEXACT,.X )`

`IEEE_INVALID_FLAG` ,
Si l'accès y est possible, alors c'est OK pour `IEEE_SUPPORT_FLAG(IEEE_INVALID,.X.)`

`IEEE_UNDERFLOW_FLAG` ,
Si l'accès y est possible, alors c'est OK pour `IEEE_SUPPORT_FLAG(IEEE_UNDERFLOW,.X.)`

`IEEE_HALTING` ,
Si l'accès y est possible, alors c'est OK pour `IEEE_SUPPORT_HALTING`

`IEEE_DATATYPE` ,
Si l'accès y est possible, alors c'est OK pour `IEEE_SUPPORT_DATATYPE ( X )`

`IEEE_DENORMAL` ,
Si l'accès y est possible, alors c'est OK pour `IEEE_SUPPORT_DENORMAL ( X )`

`IEEE_DIVIDE` ,
Si l'accès y est possible, alors c'est OK pour `IEEE_SUPPORT_DIVIDE ( X )`

`IEEE_INF` ,
Si l'accès y est possible, alors c'est OK pour `IEEE_SUPPORT_INF ( X )`

`IEEE_NAN` ,
Si l'accès y est possible, alors c'est OK pour `IEEE_SUPPORT_NAN ( X )`

`IEEE_SQRT` ,
Si l'accès y est possible, alors c'est OK pour `IEEE_SUPPORT_SQRT ( X )`

`IEEE_ROUNDING` .
Si l'accès y est possible, alors c'est OK pour `IEEE_SUPPORT_ROUNDING(ROUND_VALUE)`

« If the programmer accesses `IEEE_FEATURES`, the processor shall provide support for all of the `IEEE_FEATURES` that are reasonably possible. If the programmer uses an `ONLY` clause on a `USE` statement to access a particular feature name, the processor shall provide support for the corresponding feature, or issue an error message saying the name is not defined in the module. When used this way, the named constants in the `IEEE_FEATURES` are similar to what are frequently called command line switches for the compiler. They can specify compilation options in a reasonably portable manner. »

Le module : **IEEE_EXCEPTIONS**

Les types dérivés et leurs valeurs :

`IEEE_FLAG_TYPE` peut prendre les valeurs suivantes :

`IEEE_INVALID , IEEE_OVERFLOW , IEEE_DIVIDE_BY_ZERO ,`
`IEEE_USUAL ,` (regroupe les **trois** précédents)
`IEEE_UNDERFLOW , IEEE_INEXACT ,`
`IEEE_ALL` (regroupe les **cinq** précédents)

`IEEE_STATUS_TYPE` (*current floating point status*)

Fonctions d'interrogation :

(logical function) `IEEE_SUPPORT_FLAG (FLAG [, X])`
(logical function) `IEEE_SUPPORT_HALTING (FLAG)`

Procédures élémentaires :

`IEEE_GET_FLAG (FLAG,FLAG_VALUE)` (`FLAG_VALUE` est de type logical)
`IEEE_GET_HALTING_MODE (FLAG,HALTING)` (`HALTING` est de type logical)

Procédures **non**-élémentaires :

`IEEE_GET_STATUS (STATUS_VALUE)`
`IEEE_SET_FLAG (FLAG,FLAG_VALUE)` (`FLAG_VALUE` est de type logical)
`IEEE_SET_HALTING_MODE (FLAG,HALTING)` (`HALTING` est de type logical)
`IEEE_SET_STATUS (STATUS_VALUE)`

Exemples :

```
USE, INTRINSIC :: IEEE_ARITHMETIC
LOGICAL HALTING
...
CALL IEEE_GET_HALTING_MODE(IEEE_OVERFLOW,HALTING) ! Store halting mode
CALL IEEE_SET_HALTING_MODE(IEEE_OVERFLOW,.FALSE.) ! No halting
...                                               ! calculation without halting
CALL IEEE_SET_HALTING_MODE(IEEE_OVERFLOW,HALTING) ! Restore halting mode
```

OU

```
USE, INTRINSIC :: IEEE_ARITHMETIC
TYPE(IEEE_STATUS_TYPE) STATUS_VALUE
 ...
CALL IEEE_GET_STATUS(STATUS_VALUE)   ! Get the flags
CALL IEEE_SET_FLAG(IEEE_ALL,.FALSE.) ! Set the flags quiet.
...                                  ! calculation involving exception handling
CALL IEEE_SET_STATUS(STATUS_VALUE)   ! Restore the flags
```

Le module : **IEEE_ARITHMETIC**

Les types dérivés et leurs valeurs :

`IEEE_CLASS_TYPE` peut prendre les valeurs suivantes :

```
IEEE_SIGNALING_NAN , IEEE_QUIET_NAN , IEEE_NEGATIVE_INF ,
IEEE_NEGATIVE_NORMAL , IEEE_NEGATIVE_DENORMAL ,
IEEE_NEGATIVE_ZERO , IEEE_POSITIVE_ZERO ,
IEEE_POSITIVE_DENORMAL , IEEE_POSITIVE_NORMAL ,
IEEE_POSITIVE_INF , IEEE_OTHER_VALUE
```

`IEEE_ROUND_TYPE` peut prendre les valeurs suivantes :

```
IEEE_NEAREST , IEEE_TO_ZERO , IEEE_UP , IEEE_DOWN , IEEE_OTHER
```

Fonctions d'interrogation :

```
(logical function) IEEE_SUPPORT_DATATYPE ([X])
(logical function) IEEE_SUPPORT_DENORMAL ([X])
(logical function) IEEE_SUPPORT_INF ([X])
(logical function) IEEE_SUPPORT_NAN ([X])
(logical function) IEEE_SUPPORT_ROUNDING (ROUND_VALUE [, X])

(logical function) IEEE_SUPPORT_DIVIDE ([X])        (« division IEEE »)
(logical function) IEEE_SUPPORT_SQRT ([X])          (« racine IEEE »)

(logical function) IEEE_SUPPORT_STANDARD ([X])
                        Toutes les fonctions IEEE_SUPPORT_* précédentes en même temps !
                        Rmq : .False. si le processeur supporte des réels non IEEE en plus !

(logical function) IEEE_SUPPORT_IO ([X])            (formatage et arrondis)
(logical function) IEEE_SUPPORT_UNDERFLOW_CONTROL ([X])
```

Fonction de transformation :

```
(integer function) IEEE_SELECTED_REAL_KIND ([P, R])
```

Exemple (sur une véritable machine IEEE) :

```
Integer, Parameter ::                              &
    SP = IEEE_SELECTED_REAL_KIND (6,30)    ,   &
    DP = IEEE_SELECTED_REAL_KIND (14,300)
```

Fonctions élémentaires (lorsque les types IEEE correspondants sont supportés) :

```
(logical function) IEEE_IS_FINITE (X)
(logical function) IEEE_IS_NAN (X)
(logical function) IEEE_IS_NEGATIVE (X)
(logical function) IEEE_IS_NORMAL (X)
(logical function) IEEE_UNORDERED (X,Y)
```

`TYPE(IEEE_CLASS_TYPE) Function IEEE_CLASS (X)`
selon la valeur de `X` !

`IEEE_VALUE (X,CLASS)` obtention d'une valeur IEEE donnée

Exemples :

```
Real :: Infini, NaN_doux, NaN_dur, Zero_neg, Zero
....
Infini = IEEE_VALUE(Infini,IEEE_NEGATIVE_INF)
NaN_doux = IEEE_VALUE(NaN_doux,IEEE_QUIET_NAN)
NaN_dur = IEEE_VALUE(NaN_dur,IEEE_SIGNALING_NAN)
Zero_neg = IEEE_VALUE(Zero_neg,IEEE_NEGATIVE_ZERO)
Zero = IEEE_VALUE(Zero,IEEE_POSITIVE_ZERO)
```

`IEEE_COPY_SIGN (X,Y)`	avec un résultat de même type que `X`
`IEEE_LOGB (X)`	avec un résultat de même type que `X` en principe égal à `EXPONENT(X)-1`
`IEEE_NEXT_AFTER (X,Y)`	avec un résultat de même type que `X`
`IEEE_REM (X,Y)`	avec un résultat du meilleur type entre `X` et `Y`
`IEEE_RINT (X)`	avec un résultat de même type que `X`
`IEEE_SCALB (X,I)`	avec un résultat de même type que `X`

Procédures **non**-élémentaires :

```
IEEE_GET_ROUNDING_MODE (ROUND_VALUE)
IEEE_SET_ROUNDING_MODE (ROUND_VALUE)
```

`IEEE_GET_UNDERFLOW_MODE (GRADUAL)`	(`GRADUAL` est de type logical)
`IEEE_SET_UNDERFLOW_MODE (GRADUAL)`	(`GRADUAL` est de type logical)

Exemples :

```
USE, INTRINSIC :: IEEE_ARITHMETIC
TYPE(IEEE_ROUND_TYPE) ROUND_VALUE
...
CALL IEEE_GET_ROUNDING_MODE(ROUND_VALUE)   ! Store the rounding mode
CALL IEEE_SET_ROUNDING_MODE(IEEE_NEAREST)
...                                        ! calculation with round to nearest
CALL IEEE_SET_ROUNDING_MODE(ROUND_VALUE)   ! Restore the rounding mode
```

OU

```
USE,INTRINSIC :: IEEE_ARITHMETIC
LOGICAL SAVE_UNDERFLOW_MODE
...
CALL IEEE_GET_UNDERFLOW_MODE(SAVE_UNDERFLOW_MODE)
CALL IEEE_SET_UNDERFLOW_MODE(GRADUAL=.FALSE.)
...                          ! Perform some calculations with abrupt underflow
CALL IEEE_SET_UNDERFLOW_MODE(SAVE_UNDERFLOW_MODE)
```

Un exemple (un peu plus) complet :

```
 REAL FUNCTION HYPOT(X, Y)
      REAL, INTENT(IN) :: X, Y
      !
      ! In rare circumstances this may lead to the signaling of IEEE_OVERFLOW
      ! The caller needs to ensure that exceptions do not cause halting.

      USE, INTRINSIC :: IEEE_ARITHMETIC
      !
      ! IEEE_OVERFLOW is always available with IEEE_ARITHMETIC
      !
      USE, INTRINSIC :: IEEE_FEATURES, ONLY: IEEE_UNDERFLOW_FLAG     ! ???????
      !
      REAL SCALED_X, SCALED_Y, SCALED_RESULT
      LOGICAL, DIMENSION(2) :: FLAGS
      !
      TYPE(IEEE_FLAG_TYPE), PARAMETER, DIMENSION(2) :: &
      !
      OUT_OF_RANGE = (/ IEEE_OVERFLOW, IEEE_UNDERFLOW /)
      !
      ! The processor clears the flags on entry
      ! Try a fast algorithm first
      !
      HYPOT = SQRT( X**2 + Y**2 )
      !
      CALL IEEE_GET_FLAG(OUT_OF_RANGE,FLAGS)
      !
      IF ( ANY(FLAGS) ) THEN
            CALL IEEE_SET_FLAG(OUT_OF_RANGE,.FALSE.)
            IF ( X==0.0 .OR. Y==0.0 ) THEN
                  HYPOT = ABS(X) + ABS(Y)
            ELSE IF ( 2*ABS(EXPONENT(X)-EXPONENT(Y)) > DIGITS(X)+1 ) THEN
                  HYPOT = MAX( ABS(X), ABS(Y) ) ! one of X and Y can be ignored
            ELSE                                ! scale so that ABS(X) is near 1
                  SCALED_X = SCALE(X, -EXPONENT(X))
                  SCALED_Y = SCALE(Y, -EXPONENT(X))
                  SCALED_RESULT = SQRT( SCALED_X**2 + SCALED_Y**2 )
                  HYPOT = SCALE(SCALED_RESULT,+EXPONENT(X)) ! may cause overflow
            END IF
      END IF
      !
      ! The processor resets any flag that was signaling on entry
      !
      END FUNCTION HYPOT
```

« An attempt is made to evaluate this function directly in the fastest possible way. This will work almost every time, but if an exception occurs during this fast computation, a safe but slower way evaluates the function. This slower evaluation might involve scaling and unscaling, and in (very rare) extreme cases this unscaling can cause overflow (after all, the true result might overflow if `X` and `Y` are both near the overflow limit). If the `IEEE_OVERFLOW` or `IEEE_UNDERFLOW` flag is signaling on entry, it is reset on return by the processor, so that earlier exceptions are not lost. »

L'INTEROPÉRABILITÉ AVEC LE C

II : LA GRANDE NOUVEAUTÉ DE F2003 :

LE LANGAGE ORIENTÉ OBJET

Il faut l'utiliser pour y croire !

Moralité : On peut (doit) oublier le C++ !

Caractéristiques éliminées (définies dans la Norme) :

Détruites déjà en F95 !

1. Boucles DO avec des réels
2. Branchement à une instruction `END IF`
3. Instruction `PAUSE` : à remplacer par `Read(*,*)`
4. Instruction GOTO assignée et Formats assignés
5. Descripteur d'édition `H`

Obsolescence (Définie dans la Norme) :

C'est-à-dire prévu pour être détruit :

1. `IF` arithmétique
2. Fins d'instructions DO multiples et/ou partagées
3. Instruction `RETURN` multiple
4. Instruction `GOTO` calculé
5. Fonction formule *ou* Fonction-Instruction
6. Instruction `DATA` parmi les instructions exécutables
7. Fonction caractère dont le résultat est de longueur `*`
8. Forme fixe du code source (col. 6, 7 et 72)
9. Déclaration des chaînes de caractères par `CHARACTER*n`

Instructions pouvant devenir obsolétes (dépréciées) :

Instructions `SEQUENCE , EQUIVALENCE`
Instructions `COMMON , BLOCK DATA`
Directive `INCLUDE` (??)
Instruction `DO WHILE`
Écriture des `DOUBLE PRECISION`
Noms spécifiques pour les procédures intrinséques