Analyse et programmation langage ADA

Séminaire informatique 1ère année

R. Chelouah : rachid.chelouah@eisti.fr

Utilisation en mode diaporama
La page suivante donne le sommaire.
Pour accéder à un chapitre cliquer sur le lien correspondant
De n'importe quel transparent, on revient au sommaire en appuyant sur le logo EISTI

Date : 16/02/2006

Nombres et leurs représentations en mémoire

Nombres entiers

Tout entier positif *n* peut s'écrire sous la forme **:**

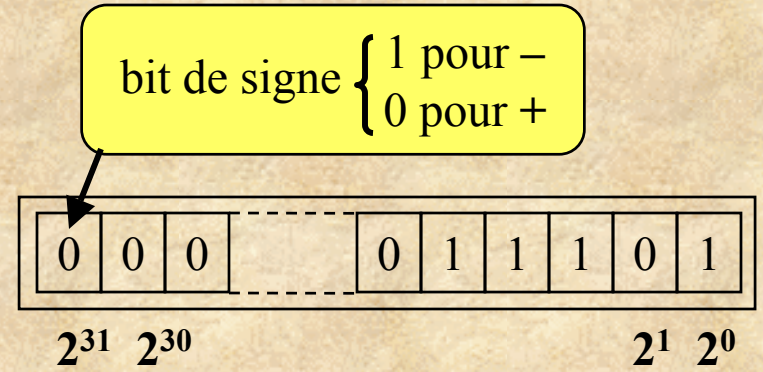
$$n = c_q 2^q + c_{q-1} 2^{q-1} + \ldots + c_1 2^1 + c_0 2^0$$

Si c_q = 1 ou 0 alors cette représentation est une représentation binaire du nombre *n*

Exemple le nombre 29 s'écrit en binaire **11101**

Écriture des entiers en binaire en mémoire

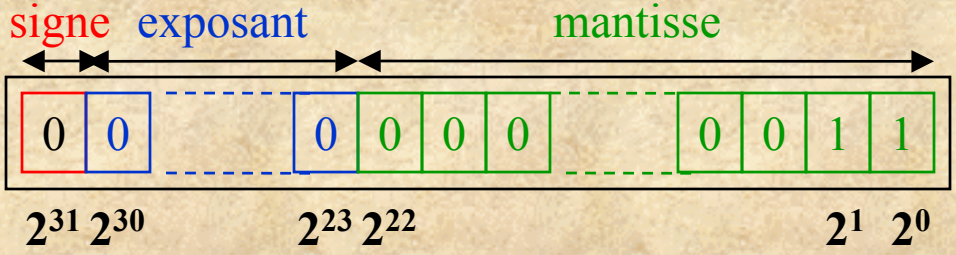
Le nombre 29 en binaire est stocké en mémoire de 32 bits



Nombres réels

Un nombre réel est représenté en machine sous forme : ± m E ±n

- d'une mantisse de 23 bits
- d'un exposant de 8 bits
- d'un bit de signe



Unités de programme

```
with Text_IO ; -- Appel aux bibliothèques

procedure afficher is

                    -- Partie Déclaration

begin --afficher
                    -- Partie Instructions

    Text_IO.Put(" Salut tout le monde !");

end afficher;
```

Comment compiler et lancer un programme écrit en ADA Sous LINUX

- Editer
  - ✓ On ouvre un éditeur de texte vi ou emacs est on saisit le code de la page précédente
  - ✓ On sauvegarde ce fichier sous le nom de afficher.adb
- Compiler Passez dans le répertoire dans lequel vous venez d'enregistrer le fichier
  - ✓ on saisissez la commande suivante **gnatmake afficher.adb**
  - ✓ cette commande compile → édite les liens → construit votre code Ada
- Exécuter

Maintenant, si vous saisissez **./afficher**

vous obtenez le résultat suivant : Salut tout le monde !

Types prédéfinis

- Un **type** définit les valeurs que peut prendre un objet et les opérations qui lui sont applicables. Il existe 5 grandes classes : les types *scalaires* ; les types *composés*; les types *privés*; les types d'*accès*; les types *dérivés*

Les types scalaires, peuvent être soit *discrets* (entier, énumérés), soit *numériques réels*.

Notation algorithmique

```
N : entier { définition }
B : booléen { déf. }
C : caractère { déf. }
R : réel  { définition }

CH : chaîne { déf. }
```

ADA

```
N : Integer;      -- définition
B : Boolean;      -- définition
C : Charater;     -- définition
R : Float;        -- définition

CH : Unbounded_String;    -- définition
```

Remarque: Il existe d'autres types pour les chaînes.

Identificateurs, mots réservés

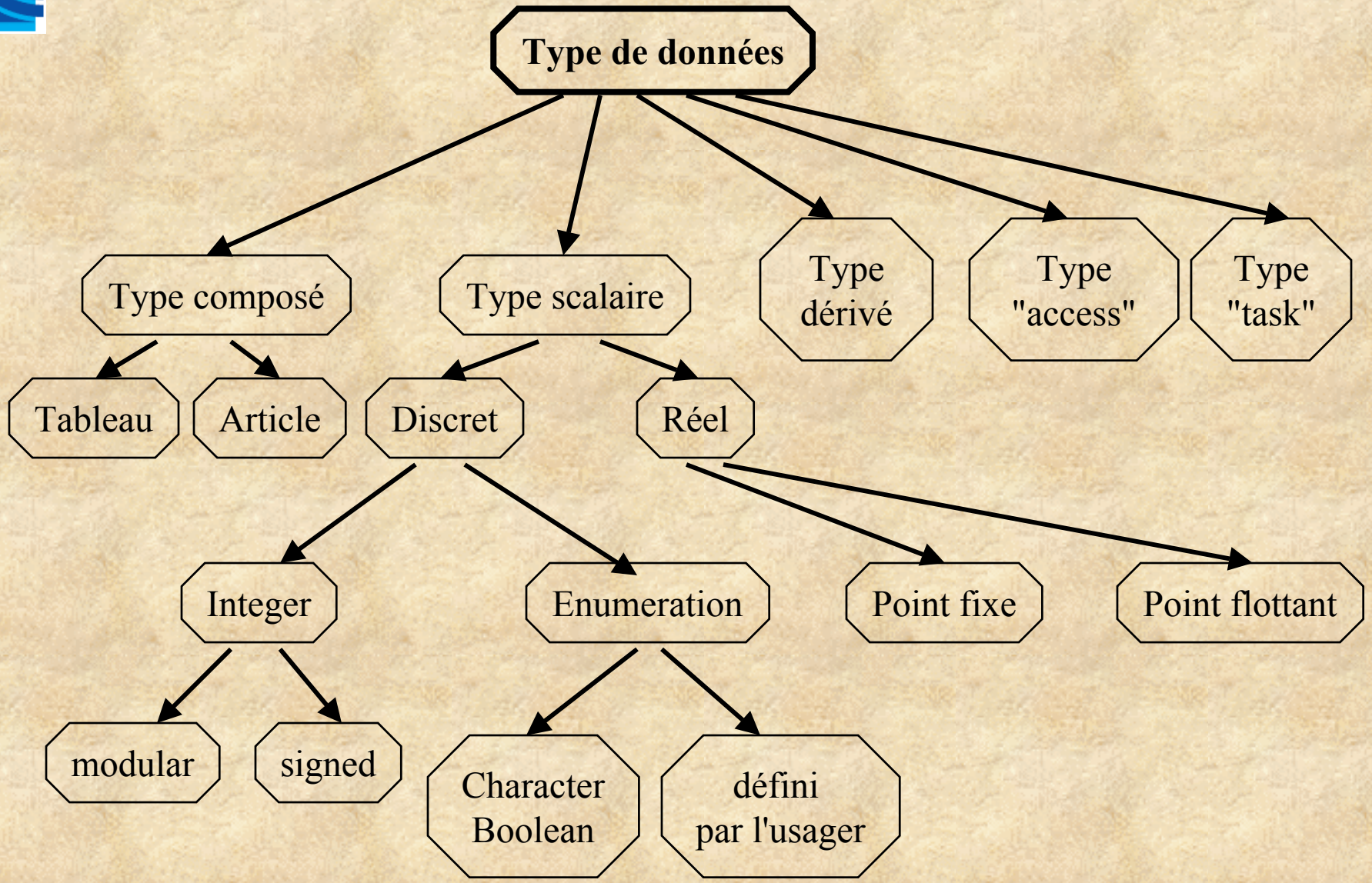
ABORT	ACCEPT	ACCESS	ALL	AND
ARRAY	AT	BEGIN	BODY	CASE
CONSTANT	DECLARE	DELAY	DELTA	DIGITS
DO	ELSE	END	ENTRY	EXCEPTION
EXIT	FOR	FUNCTION	GENERIC	GOTO
IF	IN	IS	LIMITED	LOOP
MOD	NEW	NOT	NULL	OF
OR	OTHERS	OUT	PACKAGE	PRAGMA
PRIVATE	PROCEDURE	RAISE	RANGE	RECORD
REM	RENAMES	RETURN	REVERSE	SELECT
SEPARATE	SUBTYPE	TASK	TERMINATE	THEN
TYPE	USE	WHEN	WHILE	WITH
XOR				

Types en ADA

Un *type* caractérise un ensemble de valeurs et les opérations définies sur cet ensemble.

- Objectifs du typage
  - Fournir une structure et des propriétés aux données
  - Vérifier leur intégrité dans tout le programme
  - Éviter de mélanger accidentellement les données qui ne sont pas comparables
  - Toute violation de type sur les objets est sanctionnée par le compilateur
- Aucun objet ne peut recevoir de valeur ni subir une opération si l'ensemble auquel il appartient n'a pas été préalablement déterminé c'est à dire qu'il faut :
  - présenter un type et le déclarer
  - déclarer que l'objet est de ce type.

Classification des types de données en ADA



Présentation d'un type énumération

- La présentation de type *énumération* est une liste ordonnée de valeurs distinctes représentée par des identificateurs et/ou des caractères littéraux ; cette liste est appelé énumération littérale.

- Exemple

(…, 'A', 'B', ….); type prédéfini caractère
(False, True); type prédéfini booléen
(Rouge, Orange, Vert) type feux défini par l'utilisateur ;
(Lun, Mar, Mer, Jeu, Ven, Sam, Dim); type jours de semaine défini par l'utilisateur

- Syntaxe pour définir un type énumération

type T_Piece **is** (Pile, Face) ; -- *pas de caractere accentue*

Utilisation d'un type énumération

```
procedure Exemple is
   type T_Fruits is (Orange, Banane, Pomme);
   type T_Legumes is (Carotte, Poireau);
   Un_Fruit  : T_Fruits ;
   Un_Legume : T_Legumes ;

    procedure Eplucher (Fruit : T_Fruits) is
   begin -- Eplucher
         null;
   end Eplucher;

begin -- Exemple
   Un_Fruit  := Banane;
   Un_Legume := Banane;              -- erreur !
   Eplucher (Un_Legume);             -- erreur !
end Exemple;
```

Type énumération (La table ASCII)

- Pour Ada 83, la table est sur 7 bits seulement

Déci	Hexa	ASCII	Déci	Hexa	ASCII	Déci	Hexa	ASCII	Déci	Hexa	ASCII
64	40	@	80	50	P	96	60	`	112	70	p
65	41	A	81	51	Q	97	61	a	113	71	q
66	42	B	82	52	R	98	62	b	114	72	r
67	43	C	83	53	S	99	63	c	115	73	s
68	44	D	84	54	T	100	64	d	116	74	t
69	45	E	85	55	U	101	65	e	117	75	u
70	46	F	86	56	V	102	66	f	118	76	v
71	47	G	87	57	W	103	67	g	119	77	w
72	48	H	88	58	X	104	68	h	120	78	x
73	49	I	89	59	Y	105	69	i	121	79	y
74	4A	J	90	5A	Z	106	6A	j	122	7A	z
75	4B	K	91	5B	[	107	6B	k	123	7B	{
76	4C	L	92	5C	\	108	6C	l	124	7C	|
77	4D	M	93	5D	]	109	6D	m	125	7D	}
78	4E	N	94	5E	^	110	6E	n	126	7E	~
79	4F	O	95	5F	_	111	6F	o	127	7F	DEL

Lecture et affichage des types prédéfinis

- Type Integer : --*type entier prédéfini*

  with Ada.Integer_Text_IO**;** **use** Ada.Integer_Text_IO;

- Type Character : --*type caractère prédéfini*

  with Ada.Text_IO; **use** Ada.Text_IO;

- Type Float --*type réelprédéfini*

  with Ada.Float_Text_IO; **use** Ada. Float _Text_IO;

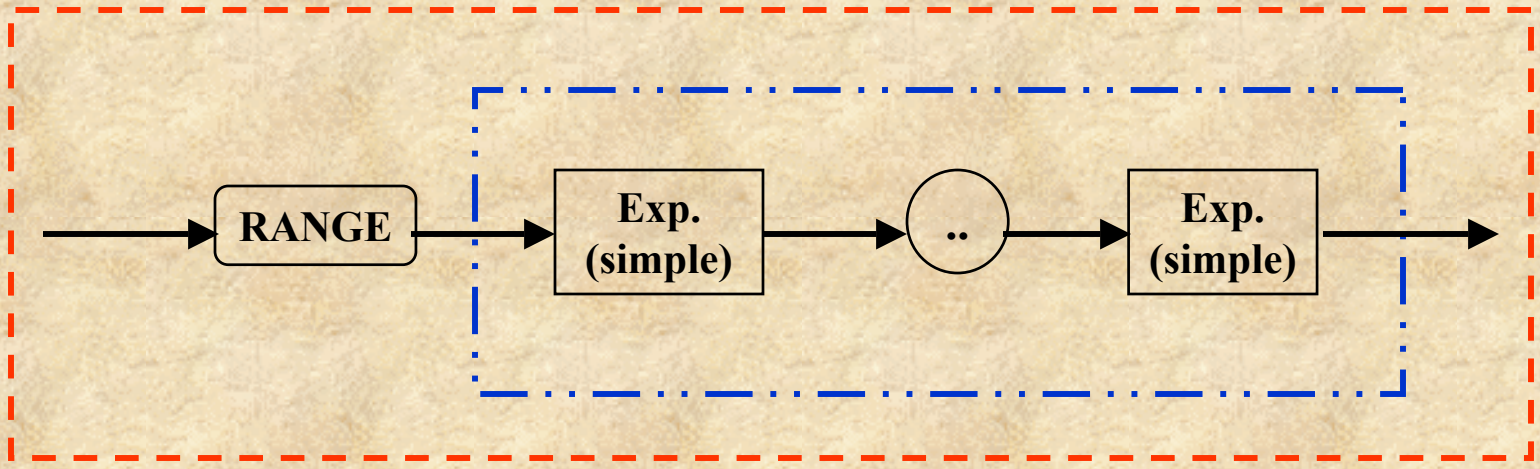
Contraintes

- *Une contrainte* est la restriction des valeurs possibles, définissant un sous ensemble muni des *mêmes opérations* que celles autorisées sur l'ensemble associé au type de base.

- Il existe 4 sortes de contraintes :
  - La contrainte de domaine, spécifiant des bornes inf et sup ; elle n'est applicable que à un type autorisant une relation d'ordre, c'est à dire associé :
    - ✓ à un ensemble discret (suite des valeurs ordonnées)
    - ✓ à l'ensemble des réels
  - La contrainte de précision
  - La contrainte d'indice
  - La contrainte de discriminant

Contrainte de domaine

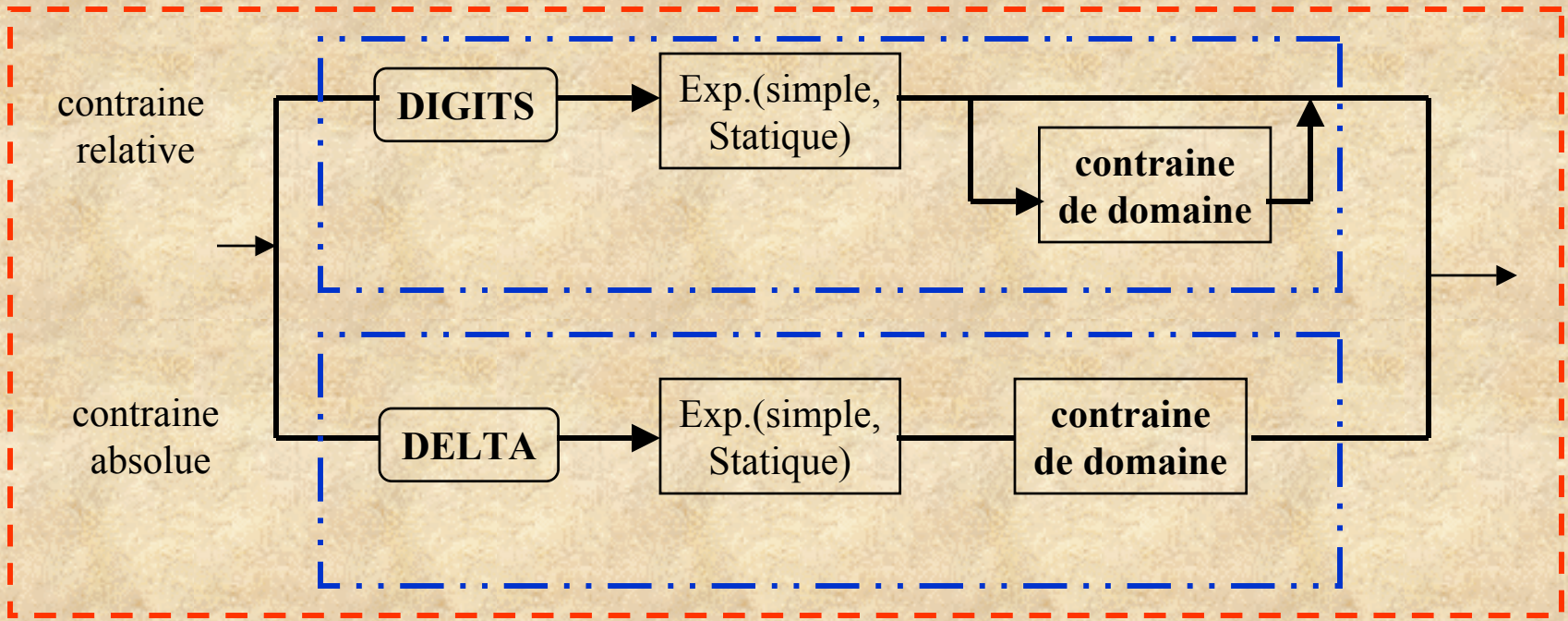
- *Une contrainte de domaine* est définie par le diagramme syntaxique suivant :



Exemple : **range** –10..10 (entier entre –10 à +10 inclus, 0 compris)
range 'A'..'Z' (lettres de l'alphabet)
range –1.0..1.0 (réel entre –1 et +1 inclus)

Contrainte de précision

Une contrainte de précision est définie par le diagramme syntaxique suivant :



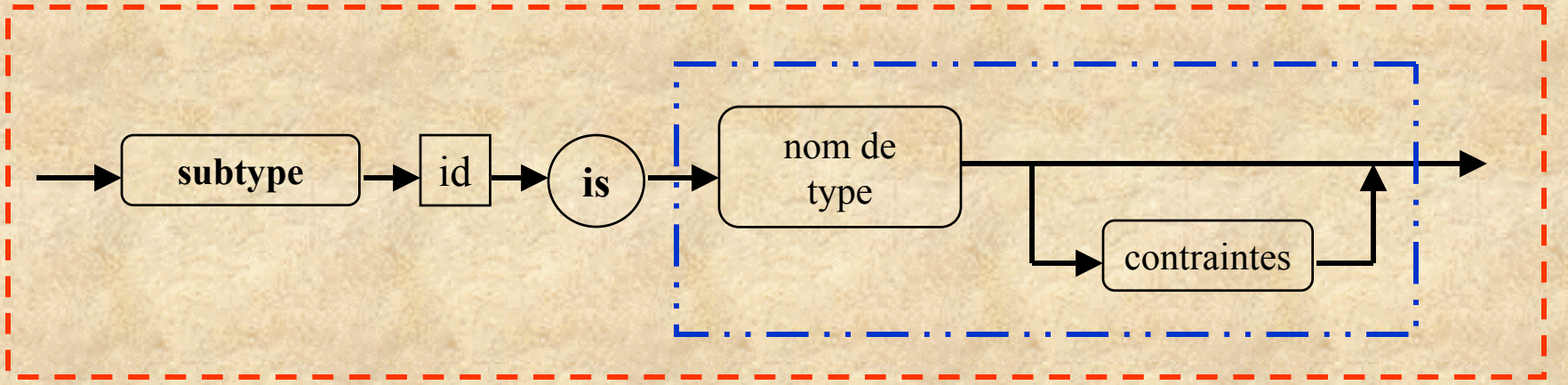
Exemple :

Digits 3 réel au moins à 3 chiffres
digits 3 **range** 20.0..50.0

Delta 0.1 réel avec pas d'un dixième
delta 0.1 **range** 35.0..40.0

Sous type

- Les sous-types : Sous-ensemble des valeurs d'un type

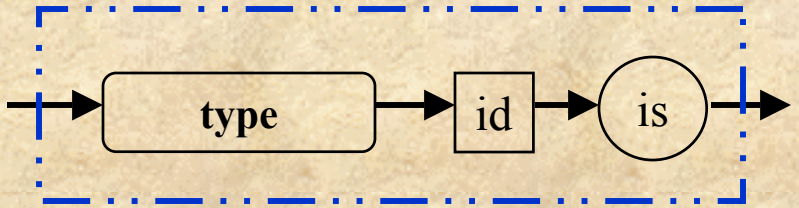


Cette déclaration donne l'identificateur du sous type et sa relation avec le type de **base**.

- Le sous-type est défini par :
  - une partie de l'ensemble de valeurs du type donné
  - l'ensemble des opérations primitives du type donné
- Le langage le considère comme le même type
- Pour les entiers, on peut citer les sous-types Natural et Positive (prédéfinis)
- Exemple d'un sous-type d'un type prédéfini :

```
subtype T_Heure is Integer range 0 .. 23;
```

Présentation d'un nouveau type



- La présentation d'un nouveau type *entier* est réduit à l'ajout d'une contrainte de domaine, dont les bornes doivent être entières et distinctes (la borne inférieure est strictement plus petite que la borne supérieure).

  range min..max **;** **--** *notion d'ordre sinon type vide*

  Exemple : A entier compris entre 1 et 12 -- *type entier signé*

  Syntaxe :

  type T_Douze **is new** Integer **range** 1..12 **;**

  type T_Douze **is range** 1..12 **;** -- ***new*** *Integer est facultatif pour les entiers*

  A : T_Douze :=1;

  B : Integer ;

  B := A; **- -** *affectation refusée par le compilateur*

package Douze_IO **is new** Ada.Text_IO. Integer_IO(T_Douze);

use Douze_IO**;**

Présentation d'un nouveau type énumération

- **Type** T_Jour **is** (Lun, Mar, Mer, Jeu, Ven, Sam, Dim);
  package Jour_IO **is new** Enumeration_IO(T_Jour); **use** Jour_IO ;

- **Type** T_Eau **is** (Ocean, Mer, Lac);
  package Eau_IO **is new** Enumaration_IO (T_Eau); **use** Eau_IO;

Put(Mer) -- *il y au une ambiguïté pour le compilateur, quel paquetage utilisé?*
-- *pour lever cette ambigüité, utiliser l'expression qualifiée*

Put(T_Jour'(Mer)); -- *utilisation du paquetage Jour_IO*
Put(T_Eau'(Mer)); -- *utilisation du paquetage Eau_IO*

NOTES
Ada.Text_IO.Put('A'); -- *affiche le caractère A*
package Char_IO **is new** Ada.Text_IO.Enumeration_IO(Character);
Char_IO.Put('A'); -- *affiche le caractère 'A', entre avec les apostrophes*

Présentation d'un type nouveau réel contraint

La présentation d'un nouveau type *réel* est réduite à l'ajout d'une contrainte de précision relative ou absolue (donc assortie éventuellement d'une contraine de domaine).

- contrainte de précision relative => Type réel point flottant
- contrainte de précision absolue => Type réel point fixe

Exemple : Digit 6 ; -- *type réel point flottant*
Delta 0.01 Range –1.00..1.00 ; -- *type réel point fixe*

Syntaxe : **type** Identificateur **is digits 3 ;** -- *range –1.00..1.00*
type Identificateur **is delta** 0.01 **range** –1.00..1.00 **;**
type Identificateur **is new** Float **range** –1.00..1.00 **;**

Remarque **:** Dans ce troisième cas, "**is new** Float" est obligatoire

Type dérivé

- Type dérivé d'un type

Reprend les caractéristiques du type dont il dérive ...

- ensemble de valeurs
- ensemble des primitives

... mais considéré comme comme un type différent.

```
type T_poids is new Float;
type T_longueur is new Float;
```

Lecture et affichage des nouveaux types réels

- Type réel point flottant : **type** T_Reel_Flottant **is digits** nbchiffres **;**

Où nbchiffres (statique) représente la précision désirée. On peut rajouter une contrainte de domaine.

  - L'erreur (précision) est relative
  - elle est spécifiée en donnant le nombre minimal de chiffres significatifs désiré.

type T_Reel_Flottant **is digits** 6 **range** 0.0..0.999999; -- *le 0 ne compte pas*
package ES_R_Flottant **is new** Ada.Text_IO.Float_IO (T_Reel_Flottant);
use ES_R_Flottant **;**

- Type réel point fixe : Ce type est utile pour travailler sur des nombres réels consécutifs séparés d'un pas fixe (erreur absolue).
  - L'erreur est absolue
  - elle est spécifiée en donnant l'écart entre deux valeurs immédiatement successives.

type T_Reel_Fixe **is delta** 0.1 **range** -1.0..1.0;
package ES_R_Fixe **is new** Ada.Text_IO.Fixed_IO (T_Reel_Fixe);
use ES_R_Fixe **;**

Affectation

Une affectation qui permet de mettre une valeur donnée ou la valeur calculée d'une expression donnée dans une région de la mémoire centrale accessible par le nom de la variable.

- Une affectation : variable = expression

Notation algorithmique

```
Programme Test
        Variables A, B : Entier
Début
        A ← 5
        B ← 2
        A ← B
        B ← 2A + 1
Fin
```

ADA

```
procedure  Test
    A, B : Integer ;
begin
        A = 5;
        B = 2;
        A = B;
        B = 2*A +1;
end
```

Traduction du conditionnelle simple

Notation algorithmique

```
si  condition
alors actions
fin si

si  Ma_Lettre = 'A'
alors NB_A ← NB_A + 1
fin si
```

ADA

```
if condition then
    actions ;
end if;

if Ma_Lettre = 'A' then
    NB_A := NB_A + 1;
end if;
```

Traduction du conditionnelle avec une alternative

Notation algorithmique

```
si  condition
alors   action1
sinon  action2
fin si

Exemple :
si X < Y
alors  MAX ← Y
sinon MAX ← X
fin si
```

ADA

```
if condition then
     action1;
else
     action2;
end if;
Exemple :
 if X<Y then
       MAX := Y ;
 else
       MAX := X ;
 end if;
```

Traduction du conditionnelle avec plusieurs alternatives ou généralisée

Notation algorithmique

```
selon  conditions
  condition1 : action1
  condition2 : action2
  condition3 : action3
fin selon
```

ADA

```
-- selon  variables
if condition1 then
    action1;
elsif condition 2 then
    action2;
elsif condition 3 then
    action3;
end if;
```

Traduction du conditionnelle avec plusieurs alternatives + un cas par défaut

Notation algorithmique

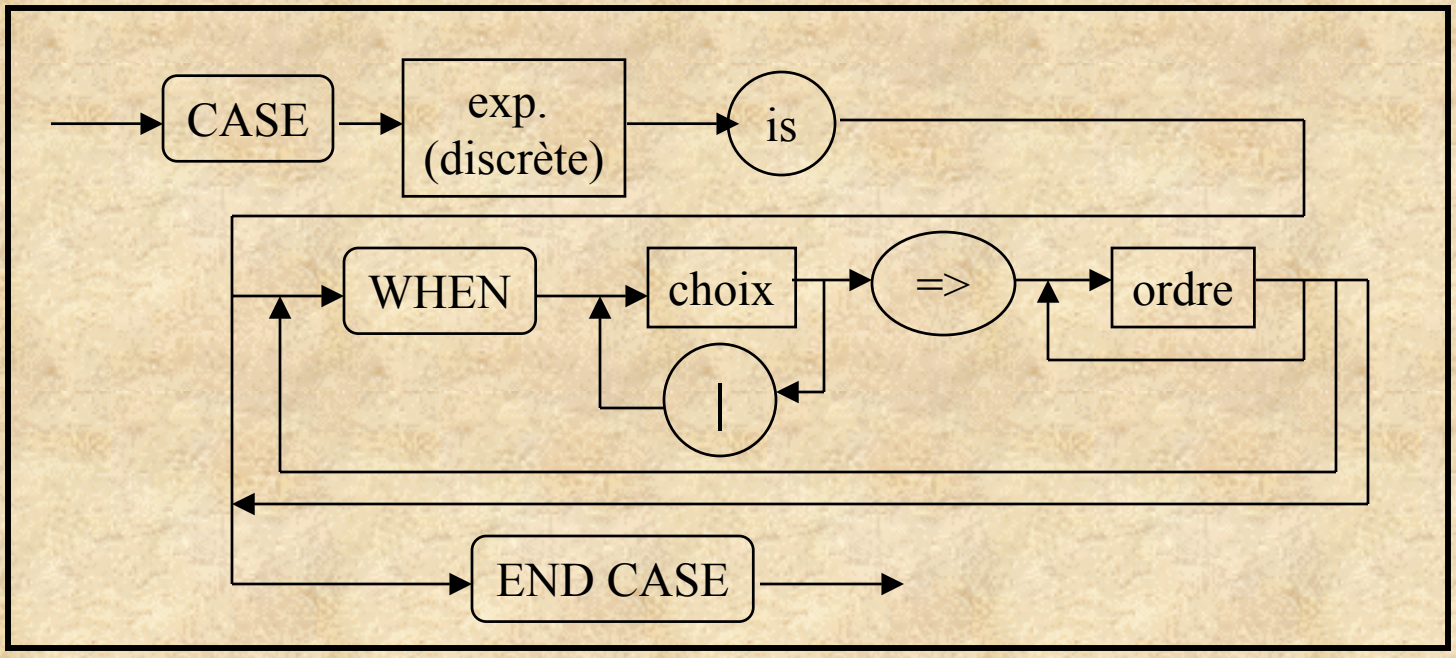
```
selon  conditions
  condition1 : action1
  condition2 : action2
  autrement : action3
fin selon
```

ADA

```
-- selon  variables
if condition1 then
    action1;
else if condition 2 then
    action2;
else
    action3;
end if;
```

Cas

Le cas est un ordre composé défini ainsi :



- Le choix a un rôle analogue à l'alternative, mais c'est l'égalité d'une valeur d'expression de type discret et d'un choix proposé qui détermine la suite des ordres à exécuter.

- Les choix doivent proposer toutes les valeurs possibles une fois et une seules fois, OTHERS servant à proposer toutes les restantes et ne pouvant donc être placer qu'en dernier.

Cas particulier d'analyse par cas

Notation algorithmique

```
selon  V
  V = a : action1
  V = b : action2
  V = c : action3
fin selon
```

ADA

```
case V is
    when a => action1;
    when b => action2;
    when c => action3;
    when others => null;
end case;
```

Espression d'un type discret

Statique, valeurs ou intervalles séparés par des barres verticales

Attention :

Il faut énumérer tous les cas.

Une action peut être l'instruction vide null

Exemple

Notation algorithmique

```
selon  V
  V = 'a' : B ← 1
  V = 'b' : B ← X
  V = 'c' : B ← 3
  V = 'd' : B ← 1
fin selon
```

ADA

```
case V is
    when ‘a’ => B:=1;
    when ‘b’ => B:=X;
    when ‘c’ => B:=3;
    when ‘d’ => B:=1;
    when others => null;
end case;
```

Cas particulier d'analyse par cas

Notation algorithmique

```
selon  V
  V = a :          action1
  V = b :          action2
  V = c :          action3
   ...
   autrement : action-k
fin selon
```

ADA

```
case V is
    when a => action1;
    when b => action2;
    when c => action3;
    …
    when others => action-k;
end case;
```

Exemple

Notation algorithmique

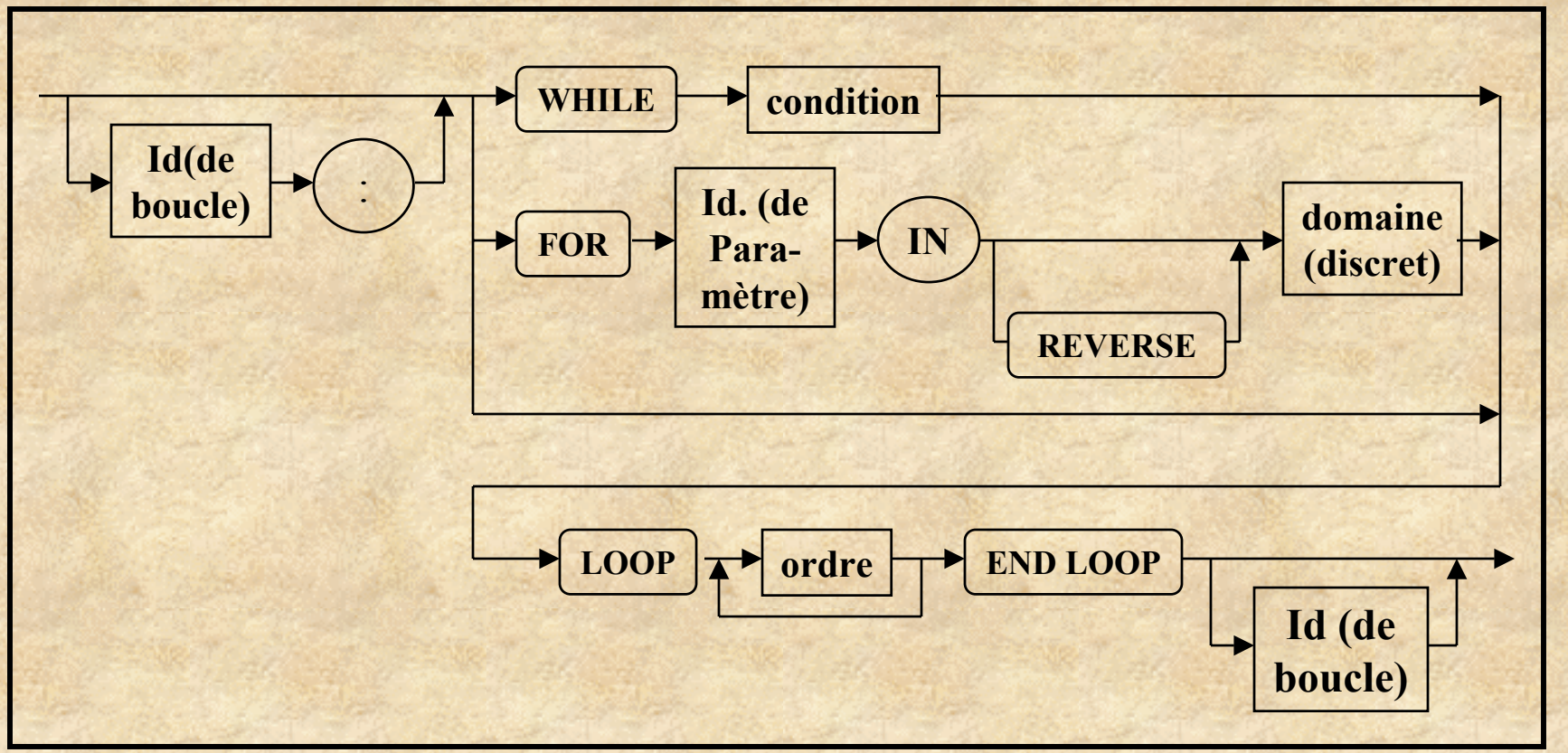
```
selon  V
  V = 'a' : B ← 1
  V = 'b' : B ← X
            C ← V
  V = 'c' : B ← X
  V = 'd' : B ←  X
  autrecas : C ← V
fin selon
```

ADA

```
case V is
    when ‘a’ => B:=1;
    when ‘b’ => B:=X;
                C:=V;
    when ‘c’ | ‘d’ => B:=X;
    when ‘e’ .. ‘k’ | ‘s’ => B:=0;
    when others => C:=V;
end case;
```

Boucle

- La *boucle* est un ordre composé défini ainsi :



- Une boucle permet d'exécuter une suite d'ordres un certain nombre de fois.

Boucle while

Notation algorithmique

```
Ident : Tantque condition vérifiée faire
                action
        FinTantque
N← 10
i ← 1
S← 0
Somme : tantque i ≤ N faire
                S ←  S + i
                i ← i + 1
        fin tantque
```

ADA

```
Ident : while Ccont loop
                action;
        end loop;
N := 10;
i := 1;
S := 0;
Somme : while i <= N loop
                S  := S + i ;
                i := i + 1;
        end loop Somme;
```

Remarque :

- Il faut s'assurer que l'expression booléenne devient fausse après un nombre fini d'itérations, sinon le programme exécuterait l'instruction **while** indéfiniment.
- Si l'expression booléenne est initialement fausse, l'instruction **while** n'est pas effectué

Boucle for

Notation algorithmique

Pour Compteur ← Initial **à** Final **Pas** Valeur
    action
FinPour

Pour Compteur I allant de 4 à 1 (4 fois)
    AV(50+I)
    GA(90+2*I)
FinPour

ADA

```
for I in 1..N loop
    action;
end loop;

for I in Reverse 1..4 loop
    AV(50 + I );
    GA(90+  2 * I );
end loop;
```

- Le nombre d'itérations sera égal à l'expression_2 – expression_1 + 1
- La variable de boucle (variable d contrôle) est déclarée implicitement, du type des borne de l'intervalle et n'existe que dans le corps de la boucle **for**
- Il n'est pas possible de changer la valeur de la variable de boucle
- Si l'intervalle est nul, c'est-à-dire que l' expression_1 a une valeur supérieure à expression_2, la boucle n'est pas effectuée
- Si la valeur de expression_1 ou expression_2 est modifiée par une itération, le nombre d'itérations ne change pas!

La boucle générale loop

Notation algorithmique

répéter
    action
Jusquàceque bouléen

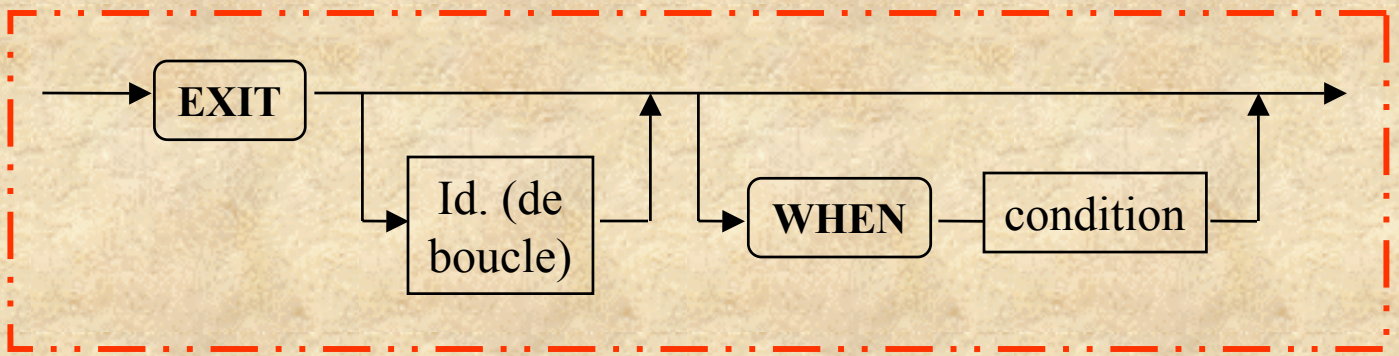
```
N ← 10
i ← 1
S ← 0
répéter
    S ←  S + i
    i ← i + 1
jusquàceque i > N
```

ADA

```
loop
    action;
     if CondArret then exit;
end if;
end loop;
N := 10;
i   := 1;
S :=  0;
loop
    S := S + i ;
     i := i + 1;
     exit when  i > N;
end loop;
```

Sortie de boucle

- L'ordre de *sortie de boucle* est un ordre simple défini ainsi :



- IL cause la sortie d'une *boucle,* c'est à dire l'exécution du premier ordre suivant :

✓ Si aucun identificateur de boucle n'est mentionné, la boucle qui cesse d'être exécuté est celle englobant immédiatement l'ordre de sortie.

✓ Si plusieurs boucles sont imbriquées les unes dans les autres, la mention de l'identificateur permet de sortir simultanément de plusieurs boucles.

Exemple boucles imbriquées

- Il est possible, mais déconseillé, de placer une ou plusieurs instructions **exit** dans une boucle **for** ou **while**.
- La boucle **loop** peut porter une étiquette. Cette étiquette est utile dans des cas tels que deux boucles imbriquées:

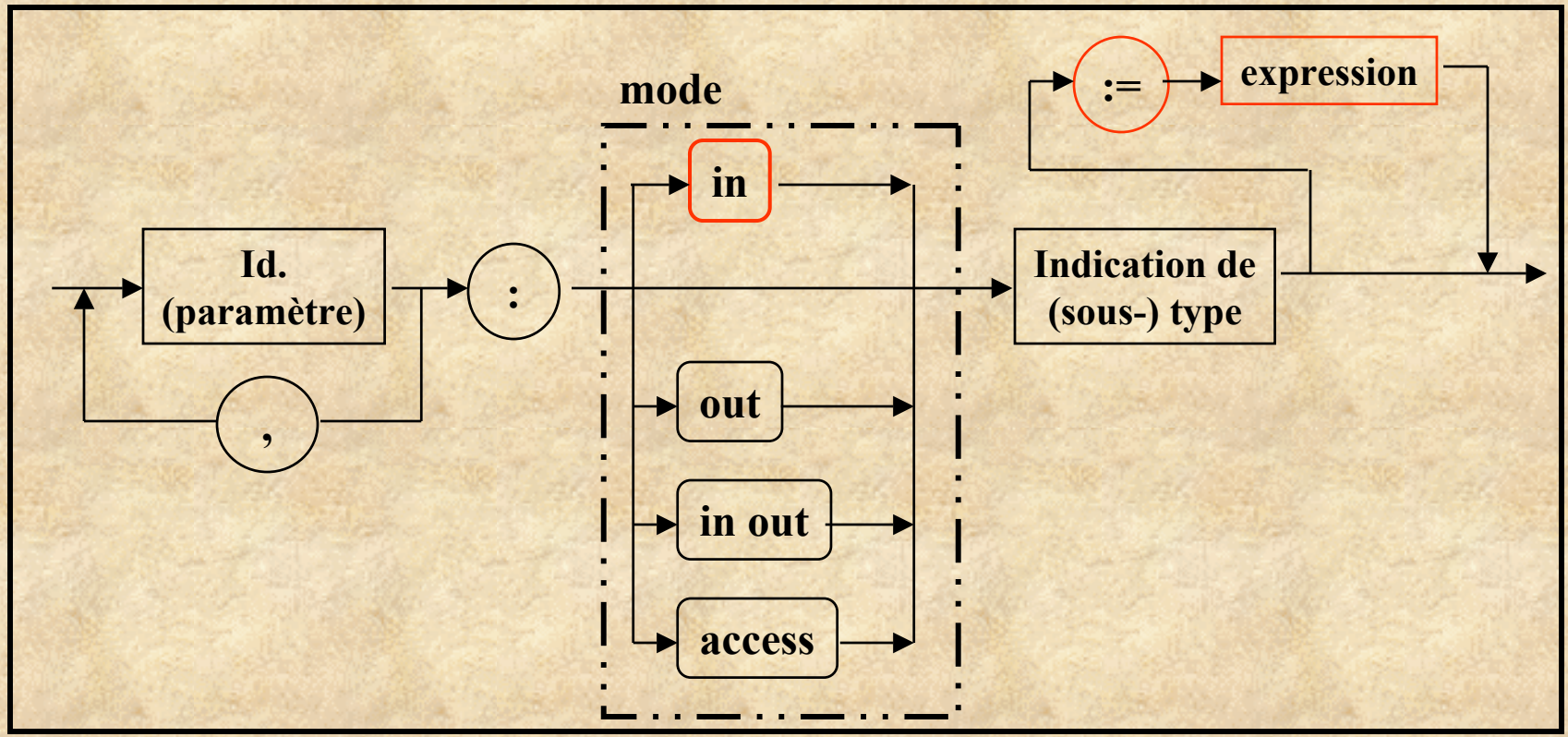
```
Externe: loop
                loop
                      -- Instructions
                       exit when B;                  -- Sortie de la boucle interne si B vraie
                       -- Instructions
                       exit Externe when C;          -- Sortie de la boucle Externe si C vraie
                                    -- Instructions
                 end loop;
                -- Instructions
end loop Externe;
```

- Les boucles **for** et **while** peuvent aussi porter une étiquette.
- L'instruction **exit**; sans condition existe et provoque la sortie inconditionnelle de la boucle.

Déclaration de paramètres dans sous-programme

- Une déclaration de sous-programme définit son nom et la manière dont on l'appelle, c'est à dire dont on fait exécuter les ordres qui sont dans son corps.

- La manière dont on appelle un sous-programme dépend essentiellement de la déclaration de ses paramètres éventuels.



Traduction des paramètres formels

Notation algorithmique	ADA
Consulté X : typeparam	X : **in** typeparam
Elaboré Y : typeparam	Y : **out** typeparam
Modifié Z : typeparam	Z : **in out** typeparam

- Le paramètre est consulté : on met **In** devant son type
- Le paramètre est élaboré : on met **Out** devant son type
- Le paramètre est modifié : on met **In Out** devant son type

Remarque : le langage Ada est très proche de l'algorithmique

Traduction des paramètres formels

- Pour chaque paramètre formel :
  - nom
  - mode
    - **in** : lecture seule
    - **in out** : lecture / mise à jour
    - **out** : écriture ou mise à jour seule
  - type
- Par défaut le mode est **in**
- Un paramètre de mode **in** est considéré comme une constante
- Les fonctions n'autorisent que le mode **in**.

Exemple : on peut déclarer

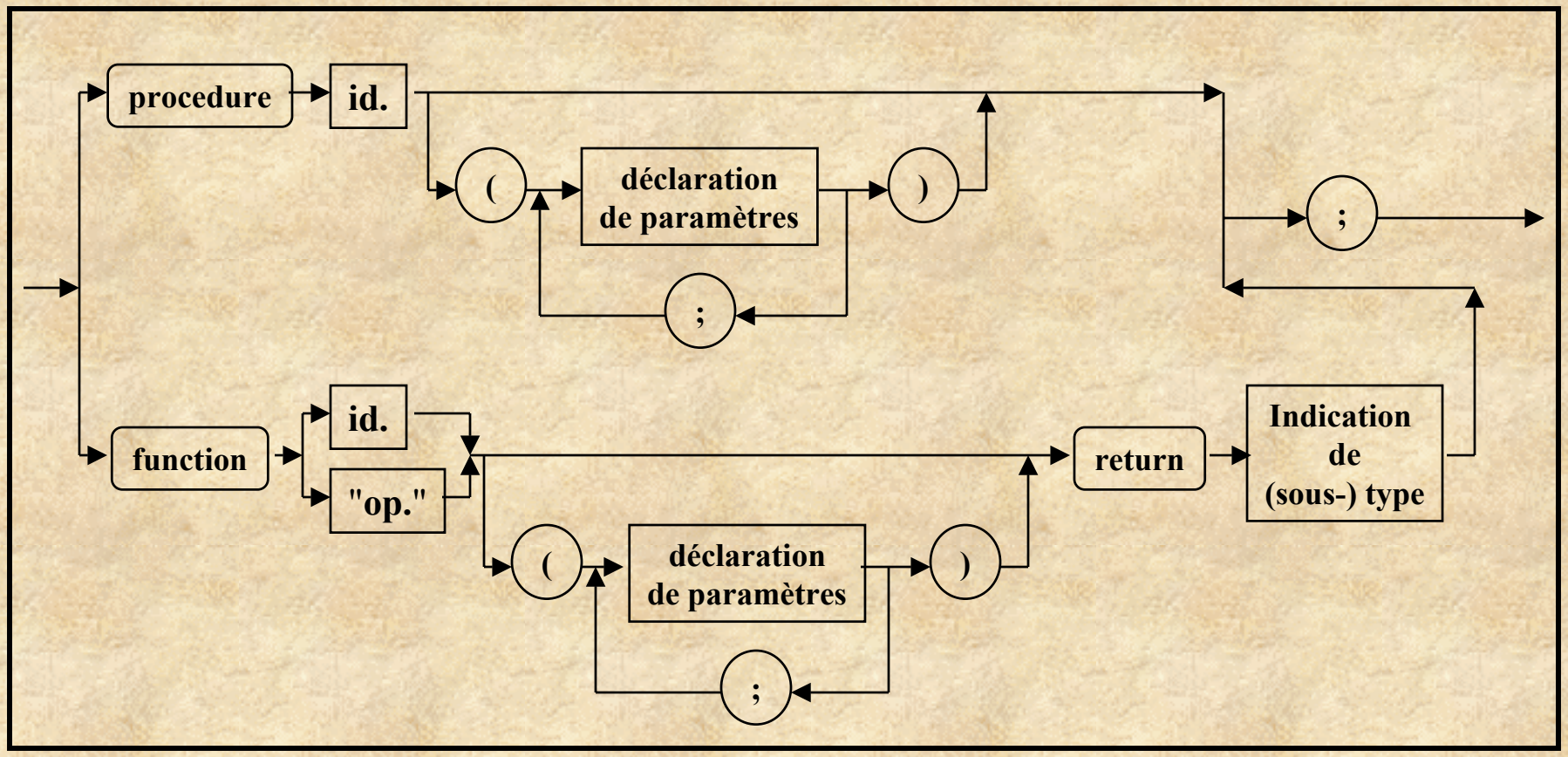
```
X : in Float := 1.2;
Y : in  Float := 1.5;
Z : out  Float   -- ( valeur de retour)
```

Convention :

On commence par les parmètres de mode **in**, puis **in out** et enfin **out**

Déclaration de sous programme

- La déclaration de ***sous programme*** est définie par le diagramme suivant :



Exemple de specifications

- Soit le type : **type** TDecimal **is range** 1 .. 10 ;

- Une *fonction* est un sous programme qui, lorqu'il est appelé, exécute ses ordres et retourne une valeur se substituant à l'appel. Le (sous-) type du *résultat* est donc indiqué après le mot **RETURN**.

function Somme(X, Y : **in** TDecimal) **return** TDecimal ;

- Une *procedure* est un sous programme qui, lorqu'il est appelé, exécute ses ordres en communiquant des valeurs par ses paramètres. Elle est donc déclarée par les seules indications de son nom et de ses paramètres.

procedure Somme(X, Y : **in** TDecimal ; Z : **out** TDecimal);

Fonctions

- Déclaration

  function Somme(X, Y : **in** TDecimal) **return** TDecimal ;

- Corps

```
function Somme(X, Y : in TDecimal) return TDecimal is
        Z : TDecimal ; -- Partie déclarative
begin
        Z:= X+Y ;   --  Partie instructions
        return Z ;   -- On peut utiliser return X+Y
end Somme;
```

Une fonction s'achève avec une instruction **return**.

Une fonction peut contenir plusieurs instructions **return**.

Procédures

- Déclaration

```
procedure Somme(X, Y : in TDecimal ; Z : out TDecimal);
```

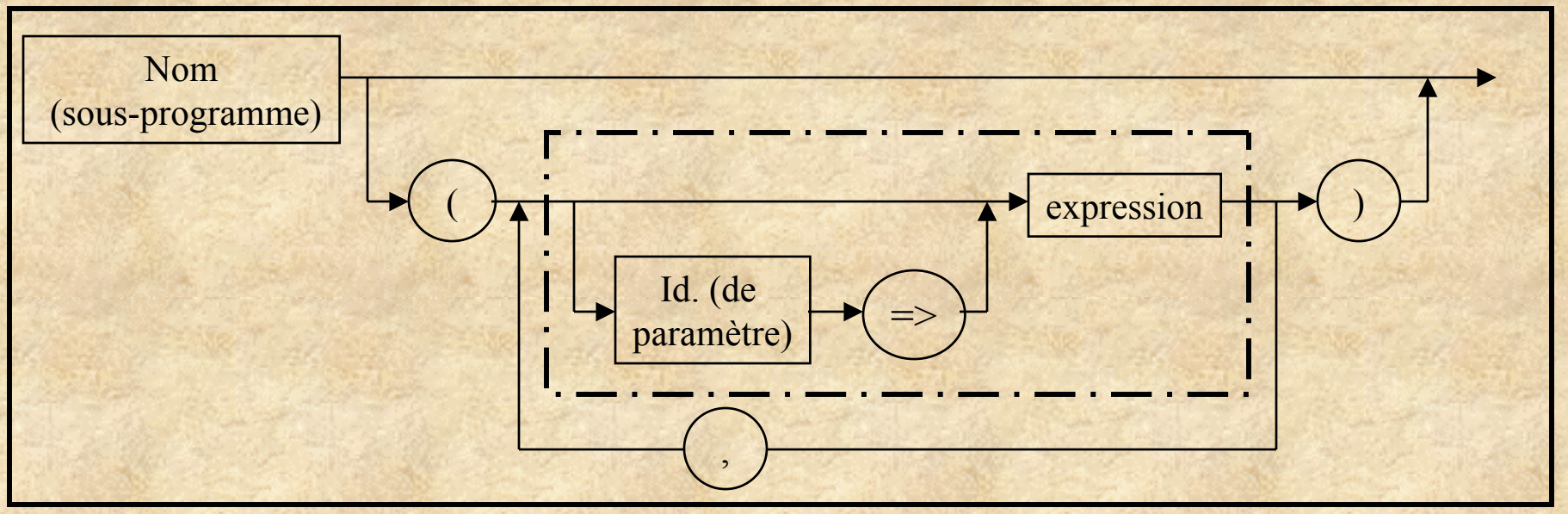
- Corps

```
procedure Somme(X, Y : in TDecimal ; Z : out TDecimal) is
        -- Partie déclarative ici aucun besoin
begin
        Z := X+Y; -- Partie instructions
end Somme;
```

Une procédure s'achève avec le **end** final.

Appel de sous programme

- L'appel de *procédure ou fonction*, est un *ordre* simple défini par le diagramme suivant



- Cet ordre entraîne l'exécution de la procédure dont le nom est donné, après avoir donner à chaque paramètre IN ou IN OUT une valeur, dites argument d'*appel*, sauf éventuellement si le paramètre correspondant à une valeur par défaut.

```
Get(A);                    New_Line(Spacing =>3);          Skip_Line;
Z := Sin(X+Y);             X := Random(Generator =>Generateur);
```

Arguments d'appel

- La liste des arguments d'appel peut être :

-*positionnelle*, c'est à dire que les valeurs des arguments sont données respectivement et dans l'ordre aux paramètres (tel qu'ils sont déclarés dans la spécification de procédure) ;

-*nommée*, les valeurs étant données aux paramètres dont le nom les précède, séparé par " =>" L'ordre est alors quelconque ;

- *mixte*, la partie positionnelle précédant la partie nommée.

Exemple : la procédure "Agenda" imprime le calendrier d'un mois, de jour à jour, à partir d'une date donnée est déclarée ainsi :

procedure Agenda (An, Mois : **in** natural; Jour : **in** natural := 1) ;

```
Agenda(1999, 3, 8) ;                                  -- positionnelle
Agenda(Jour => 8, Mois => 3, An => 1999) ;            -- nommée
Agenda(1999, Jour => 8, Mois => 3) ;                  -- mixte
Agenda(Mois => 3, An => 1999) ;  ou Agenda(1999, 3) ; -- la valeur du Jour = 1
```