Chapitre 5

Les langages de la programmation parallèle

Historique

Première génération

1954 à 1958
Première indépendance entre machine et langage
Quelques structures de contrôle
Fortran I et Algol

Deuxième génération

Entre 1959 et 1961
Fortran II : compilation séparée ; Algol : notion de bloc
Manipulation des fichiers (cobol)
Programmation fonctionnelle

Troisième génération

Entre 1962 et 1970
Structures de données complexes
Apparition du langage C dédié à la programmation système
Simula qui annonce l'arrivée des langages de la quatrième génération

Quatrième génération

Apparition de nouveaux paradigmes de programmation
Parallélisme induit par l'évolution technologique
Expression du parallélisme ADA, MODULA 2
Programmes fonctionnels parallèles : CML et Miranda
Programmation logique : Prolog
Programmation par objet : Smaltalk

Modèles de programmation

Programmation séquentielle et parallèle
Ordre total et ordre partiel
Programmation centralisée et distribuée

La programmation parallèle

Plus expressive et plus complexe
Gestion de synchronisation requise de la part du programmeur

La programmation parallèle peut être en deux grands **modèles**

Parallélisme de tâche

Expression de parallélisme fondé sur le calcul
Décomposition du programme en sous ensemble coopérants
Modèle expressif : co-routine, processus, …
Communication, synchronisation, partage de données entre modules
Demande de maîtrise et gestion explicite de synchronisation de la part du programmeur
Construction explicite exprimée par le langage

Parallélisme de données

Expression du parallélisme fondé sur les données
Appliqué sur des architectures parallèles : SIMD, MIMD

Exécution de calcul parallèle sur des données locales
Données parallèles ou agrégats
Application d'opérateur identique sur les éléments
Opération globale
Boucle parallèle
Boucle séquentielle

Les langages de programmation parallèle

Conçu sur les deux modèles
Se base sur l'un des trois axes
Conception de nouveaux langages parallèles
Etendre un langage existant pour supporter le parallélisme
Conception de langage parallèle pour architecture parallèle spécifique

- Le langage Pascal Parallèle
- La communication synchrone : CSP et OCCAM
- Le langage HPF (High Performance Fortran) Portland Group
- La programmation asynchrone : le langage LC3
- Le langage sC++
- Le langage MODULA -2
- Le langage JAVA
- Le langage Concurrent ML
- Le langage ADA

LE LANGAGE ADA

1- Aperçu sur le langage ADA
2- ADA comme langage de programmation parallèle

I. APERCU SUR LE LANGAGE ADA

I.1. JEU DE CARACTERES

Les lettres MAJ de A◊ Z
Les lettres MIN de a ◊ z
Les chiffres de O ◊***9***
Les caractères spéciaux: = # % & ' ( ) * + - / , :: ; < > ! $?
@ [ ] \ ~^ { } ~
Espace ou blanc.

I.2. ELEMENTS LEXICAUX

- Identificateur,
- Les Nombres,
- Les Caracteres Litteraux,
- Chaines de Caracteres,
- Commentaires,
- Pragmas

a) IDENTIFICATEUR

identificateur ◊ nom

<identificateur>: := **{** [blanc-souligné] lettre / chiffre **}**
lettre = lettre MIN / lettre MAJ

Deux identificateurs identiques l'un en MAJ l'autre en MIN sont égaux.

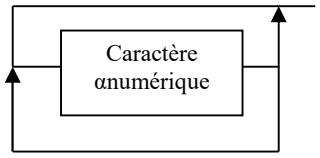
b) LES NOMBRES

2 classes: réels, entiers
Pour les réels, l'exposant est désigné par e/E
En Ada, il existe les nombres basés : outre la base 10, les nombres peuvent s'exprimer dans n'importe quelle base comprise entre 2 et 16.
<nbre basé> ::= base # <entier basé> [, entier basé] # [exposant]
base: := nbre entier

Exemple:
2# 1100 # équivalent en base 2 de 12_{10}
16#C# équivalent en base 16 de 12_{10}
16#F.0# même valeur que 15.0_{10}

c) LES CARACTERES LITTERAUX: un caractère littéral s'écrit en cadrant entre apostrophes l'un des 95 caractères imprimables du code ASCII y compris le blanc exemple : **'B' , '*', ''''**

d) CHAINES DE CARACTERES

<chaîne> : : = « 

»

Exemple: ''A'', ''salut''

e) COMMENTAIRES: il commence par un double trait d'union et se termine à la fin de la ligne.
Les commentaires n'ont aucun effet sur le programme, leur seul but est d'informer le lecteur du programme.

Exemple:
-- partie de calcul
-- les 2 premiers traits marque le début du commentaire

Un commentaire peut être étalé sur plusieurs lignes commençant par **–**

f) PRAGMAS : ont pour rôle de transmettre des informations au compilateur, une pragma débute par le mot réservé PRAGMA suivi de <nom de la pragma> pour la distinguer des autres langages.

Le pragma peut être définie par le langage ou par l'implémentation.

La pragma dont le nom n'est pas reconnu par le compilateur est sans effet.

Exemple:
PRAGMA Optimize (Time)
PRAGMA Inline (Set-Mask)

I.3. DECLARATIONS ET TYPES DE DONNEES

Le langage défini plusieurs formes d'entités nommées, une entité nommée peut être:
Un nombre, un littéral d'énumération, un objet, un discriminant, un article d'enregistrement, un paramètre de boucle, un type ou un sous-type de données, un attribut, un sous-programme, un progiciel, une tâche, un point d'entrée, un bloc,…

Chaque identificateur doit être explicitement déclaré avant d'être utilisé à l'exception des étiquettes, les identificateurs de blocs et les identificateurs de boucle qui sont déclarés implicitement.

Les déclarations sont sous la forme suivante:

<déclaration>::=
<déclaration d'objets> / <déclaration de nombres> /
<déclaration de sous-type> /<déclaration de sous-programme> /
<déclaration de progiciel> / <déclaration de tâche> /
<déclaration d'exception>/ <déclaration de nommage>.

a) déclaration d'objet et de nombres :

Un objet est une entité qui possède une valeur d'un type donné.

déclaration d'objet ::=
<liste d'identificateurs>:[CONSTANT] <indicateur de type>[:=<expression>];
<liste d'identificateurs>:[CONSTANT]<déf de type tableau>[:=<expression>];

déclaration de nombre ::=
<liste d'identificateurs> : CONSTANT [:=<expression>]

<liste d'identificateurs>: :=
identificateur {, identificateur}

Exemple de déclaration de variables:
Suite, Result: **false**;
Nuance : **ARRAY**[1..N] **of** Couleur;

Exemple de déclaration de constantes:
Binf : **CONSTANT** :=1;
Bsup :**CONSTANT** :=Binf/20;

Exemple de déclaration de nombres:
Pi :**CONSTANT** :=3.14l59-26536;
Deux-Pi :**CONSTANT** :=2.O * Pi ;
Depart, debut: **CONSTANT** :=1 ;

b) déclaration de type et de sous-type:

Il existe plusieurs classes de types:
-Type entier
-Type réel
-Type énumération : les valeurs de ce type n'ont pas de valeurs des composants, le type est défini comme étant une énumération.

-Type tableau
-Type enregistrement

} sont composites et leurs valeurs sont formées de plusieurs valeurs de composants

-Type access: les valeurs de ce type donnent accès à d'autres objets.
-Type privé: L'ensemble des valeurs de ce type est bien défini mais n'est pas connu des utilisateurs.

Remarque: les types enregistrements et les types privés peuvent avoir des composants spéciaux appelés discriminants.

L'ensemble de valeurs possibles pour un objet d'un type donné peut être restreint sans modifier l'ensemble des opérations qui s'y appliquent =contrainte.
Une valeur appartient à un sous-type d'un type donné, si elle vérifie cette contrainte

Déclaration de type ::=

Type <identificateur> [partie discriminante] is <déf de type>
/ <déclaration de type incomplète>
<définition de type > ::=
<déf de type énumération>/<déf de type entier>/<déf de type réel>/<déf de type tableau>/<déf de type enregistrement>/<déf de type access>/<déf de type dérivé>/<déf de type privé>

Déclaration de sous-type:

SUBTYPE <identificateur> is <indicateur de type>
<indicateur de type>: := <nom de type>/<nom de sous-type>/[<contrainte>]
<contrainte>::=<contrainte d'intervalle>/<contrainte de précision>/<contrainte d'index>/<contrainte de discriminant>
Remarque: pour tout type ou sous-type T, l'attribut T'BASE est le type de base de T.

Exemple de déclaration de types:

Type deplct **is** (marche, avance, recule, arret) ;
Type num **is** range 1..20;
Type vect 15 **is array** (1. .5) of **integer**;

Exemple de déclaration de sous-types:

Subtype posit **is** deplct **range** marche.. recule ;
Subtype index **is** integer **range** -3.. +3
Subtype feminin **is** personne (sexe => F) ;
Subtype numc **is** num **range** 1..10;

c) définition de type dérivé: Permet de déterminer un type contrainte prenant ses caractéristiques d'un type parent uniquement autorisée dans une déclaration de **Type**.

<déf de type dérivé>: := **new** <indicateur de type>
Type <type nouveau> **is new** <type vieux> <contrainte>.

Exemple:
Type jour-ouv **is new range** sam..mer;

d) Définition de types scalaires : types discrets + types réels, tous les types scalaires sont ordonnés.
Types discrets = entiers + énumération

<contrainte d'intervalle>::= **rang** <expression simple>..<expression simple>
T'FIRST: valeur MIN du type T ou valeur inf du type T.
T'LAST: valeur MAX du type T ou valeur sup du type T.

-1- Type énumération:

Exemple:
Type Jour **is** (lundi, mardi, mercredi, jeudi, vendredi, samedi, dimanche) ;
Type Niveau **is** (bas, haut, moyen);
Subtype Jour-sem **is** jour **range** lundi. .vendredi ;

-2- Type caractère : est un type prédéfini CHARACTER, mais si on veut par exemple isoler les majuscules, on peut écrire:

Subtype majuscules **is character range** 'A'.. 'Z';

-3- Type booléen : (FALSE, TRUE) ordonné par la relation FALSE < TRUE.

-4- Type entier: Pour chaque type discret, il existe les fonctions T'POS, T'SUCC, T'PRED, T'VAL.

T'POS(x) : x de type T, le résultat = rang(x) est de valeur entière.
T'SUCC(x) : x de type T, le résultat est la valeur de type T dont le rang est égal à celui de x plus un.
T'PRED(x) : x doit être une valeur de T, le résultat est la valeur de type T dont le rang est égal à celui de x moins un.
T'VAL(N) : N valeur entière, le résultat est la valeur de type T de rang N

Exemple :
Jour'FIRST= lundi
Jour'LAST= dimanche
Niveau'SUCC(haut)= moyen
Niveau'POS(haut)= 1
Niveau'VAL(0)=bas

-5- Type réel:

Fournit des approximations aux nombres réels, avec une borne relative d'erreur pour les réels à point fixe et les réels à point mobile et une borne absolue pour les réels à point fixe.

* **Type réel à point mobile:**
La limite de l'erreur est spécifiée par une précision relative donnant le nombre min de chiffres requis pour la mantisse.

<contrainte pt mobile>: :=digit <expression simple statique>[contrainte d'intervalle]

D: nombre de chiffres décimaux donné par la valeur de <expression statique> après le mot réservé digit, il doit être >0 et entier.
B: nombre minimum de chiffres binaires pour la mantisse binaire.
Les nombres sont de la forme:
Signe * mantisse-binaire * (2.0 ** exposant), le nombre 0 est compris.

Déclaration du type réel à point mobile:

> **Type** <type nouveau> **is digit D** [rang L.. R]
> **Type** <type nouveau> is new <type réel à pt mobile> **digit D** [rang L. .R]
Où L et R expressions statiques de type réel

- Prédéfini- Float - Short-float- Long-float

Prédéfini - Float - Short-float - Long-float

Exemple:

Type coef **is** digits 10 **range** - 1.0 ..I.0; (10 ± 1.0)
Type réel is digits 8;
Type masse **is new** digits 7 **range** 0.0.. 1.0 E10;
Subtype coef-court **is** coef **digits** 5; (5 ± 1.0)

Fonctions sur les réels à point mobile:

F'DIGITS: pour un type F prédéfini, nombre équivalent à la précision des chiffres décimaux (entier universel).
F'MANTISSA: longueur de la mantisse binaire des nombres de F (type entier universel).
F'EMAX: nombre tel que l'intervalle d'exposant binaire des nombres de F est F'EMAX, il est entier.
F'EPSILON: la valeur absolue de la différence entre 1.0 et le premier nombre juste supérieur à 1.0 de type réel.

* **Type réel à point fixe:**
La borne d'erreur pour les réels à point fixe est spécifiée par une valeur absolue appelée le pas du type réel à point fixe.

<contrainte pt fixe>: :=**delta** <expression simple statique>[contrainte d'intervalle]

Le pas est donné par <expression statique> qui suit **delta**, il est positif et de type entier.

Exemple:
Type rpf **is delta** 0.125 rang 0.0 .. 255.0;
Subtype srpf **is** rpf **delta** 0.5; même intervalle que rpf.

Fonction sur les réels à point fixe:

F'DELTA: si F est type ou sous-type, sans contrainte de point fixe, cet attribut est le pas du type de base, i.e c'est le pas spécifié par la contrainte de point fixe, il est du type réel universel.
F'ACTUAL-DELTA: pas effectif de F, de type réel universel.
F'BITS : si les valeurs positives des nombres de F sont exprimées comme K*FACTUAL-DELTA, la fonction F'BITS est le nombre de chiffres binaires utilisés pour représenter l'entier non signé K, il est de type entier universel.
F'LARGE: c'est le plus grand nombre de F.

Remarque: F'LARGE = (2**F'bit – 1) *F'actual-delta.

-6- Type tableau:

Objet composite dont les éléments sont tous d'un même type, un élément du tableau est désigné par un ou plusieurs valeurs d'index appartenant à des types discrets.

<définition de type tableau>::=
array ((<index>{,<index>}) of <ind de type composant)
array <contrainte d'index> of <ind de type composant>
<index>: := marque de type **range** <>
<contrainte d'index>: := (<intervalle discret> {,<intervalle discret>})
<intervalle discret>: := marque de type [contrainte d'intervalle] / intervalle

Remarque : l'ordre des indices est significatif.

- tableaux sans contrainte:
array (index,{index}) **of** <indicateur de sous-type composant>

Exemple:

Type matrice **is array** (**integer range** <>,**integer range** <>) of float;
Type tab **is array** (**integer range** <>) **of character**;

- tableaux avec contraintes:
array <contrainte d'index> **of** <indicateur de sous-type composant>

Exemple:

Type table **is array** (1..20) **of character**;
Type tab **is array** (**integer range** <>) **of character**;

Attributs des tableaux:

T'FIRST: borne inf du premier index.
T'LAST: borne sup du premier index.
T'LENGTH: nombre de valeurs du premier index(0->O),le résultat est entier
T'RANGE: sous-type défini par l'intervalle T'FIRST. .T'LAST.
T'FIRST(N): borne inf du Nième index.
T'LAST(N): borne sup du Nième index.
Idem pour T'LENGTH(N) et T'RANGE(N).

-7- Type chaîne:

Le type prédéfini **STRING** désigne des tableaux de dimension un du type prédéfini **CHARACTER** indexé par des valeurs du sous-type prédéfini **NATURAL**.

Exemple:

Subtype NATURAL is INTEGER range 1 ..INTEGER'LAST.
Type STRING is array (NATURAL range <>) of CHARACTER;

On peut appliquer sur ce type:
- La concaténation représentée par le symbole &.
- Les opérateurs de relation <, >, <= ,>= , /=, >

Exemple:
Phrase: **STRING** (1..120) := (1..120 =>'');
Rep: **CONSTANT STRING** := "dites okey";

-8- Type enregistrement:

Objet composite formé de composants nommés qui sont de différents types.
La valeur d'un objet enregistrement est une valeur composite constituée par les valeurs de ses composants.

<définition de type enregistrement>::=
record
<liste de composants>
end record
<liste de composants>::=
{<déclaration de composant>} [<partie-variante>] / null;
<déclaration de composant>::=

<liste d'identificateurs> : <indicateur de type> [:= <expression>];
/<liste d'identificateurs> : <indicateur de type tableau> [:= < expression>]

Exemple:
Type date **is**
record
j :**integer range** 1..31 ;
m : nom-mois;
an :**integer range** 0. .2004;
end record;
Type complexe **is**
record
re :**float** :=0.0;
im :**float** :=0 .0;
end record

-8-1 Discriminant

Dans un type enregistrement on peut donner une partie discriminante qui constitue les discriminants de ce type.
<partie discriminante> ::=
<liste d'identificateurs>: <indicateur de type> [:=<expression>]

Remarque: tout discriminant doit appartenir à un type discret, les types enregistrements et les types privés sont les seuls types qui peuvent avoir des discriminants.

Exemple :
Type t(long :**integer rang** 0. .max := 200) **is**
record
Pos :**integer rang** 0. .max :=0 ;
Val : **string**(1..long);
end record;

Un discriminant n'est pas nécessairement référencé par un composant d'enregistrement.
Type e(nbre: **NATURAL**) **is**
record
Contenu :integer;
End record

-8-2 Contraintes de discriminant:

Les valeurs de discriminant autorisées pour un objet enregistrement peuvent être fixées par une contrainte de discriminant.

<contrainte de discriminant>::=
(<spécification de discriminant> {, <spécification de discriminant>])
<spécification de discriminant> ::=
[<nom de discriminant> {<nom de discriminant>] =>] <expression>

Chaque expression spécifie la valeur d'un discriminant, une contrainte de discriminant doit donner une valeur pour chaque discriminant du type.

Attribut:
Cet attribut est défini pour tout objet A d'un type avec discriminant:
A'CONSTRAINED: est vrai si et seulement si une contrainte de discriminant s'applique à l'objet A; si A est un paramètre formel, la valeur de cet attribut est déterminée en fonction du paramètre effectif, le résultat est booléen.

Exemple:
Grand : t(2000) ◊ 2000 caractères
Grand'**CONSTRAINED** = **TRUE**

Remarques:
> Ces règles garantissent l'existence d'une valeur pour les discriminants.
> Si une déclaration de sous-type contient une contrainte de discriminant tous les objets de ce sous-type sont contraints et leurs discriminants sont initialisés.

-8-3 Partie variante

Un type enregistrement peut être muni de parties variantes.

<partie variante>::=
CASE <nom de discriminant> **is**
{**WHEN** <choix> } / <choix { => <liste de composants> }
END CASE;
<choix>: := <expression simple> / <intervalle discret> > / **OTHERS**

Remarque:
Une variante peut avoir une liste de composants vide qui sera spécifiée par MULL.

-9- Types acces

Les objets déclarés dans un programme sont accessibles par leur nom. Leur durée de vie est la même que celle de la partie déclarative où ils sont locaux, par contre, des objets peuvent également être crées par l'exécution d'allocateurs. Ils n'apparaissent pas dans une déclaration d'objet explicite. Ils ne peuvent pas être désignés par leurs noms. On accède à un tel objet par une valeur d'acces qui est retournée par l'allocateur. On dit que la valeur d'acces désigne l'objet.

<définition de type acces>: := **access** <indicateur de type>

<déclaration de type incomplète>:: = type<identificateur>[<partie discriminante>]

Exemple:
```
Type cellule;  déclaration de type incomplète
Type lien is access cellule;
Type cellule is
record
        Val:    integer;
        Succ: lien;
        Pred:  lien ;
end record

Type individu(sexe :genre) is
record
        Nom : string(1..20);
        Age :integer range 0..100;
       case sexe is
```

```
            when m => épouse :nom-individu(sexe =>f);
            when f => époux: nom-individu(sexe =>m);
          end case ;
end record ;
```

I.4. EXPRESSIONS ET OPERATEURS

Expression = formule de calcul d'une valeur.
Chaque élément a une valeur et un type.
Exemple:
b**2
b**2- 4.0*a*c
(a and b) or c

a) OPERATEURS:

- <opérateur logique>: := and/or/xor
- <opérateur de relation>: := **=** / **/=** / **<** / **<=** / **>** / **>=**
-<opérateur additif>: := + / - / &
-<opérateur monadique>: : + / - / not
-<opérateur multiplicatif: := * / / / mod / rem

reste entier

-<opérateur d'exponentiation>: := **

Remarques:
> Les formes de contrôle **and then** et **or else** ont la même priorité que les opérateurs logiques.
> Les tests d'appartenance **in** et **not in** ont la même priorité que les opérateurs de relation.
> Dans une expression en général les opérateurs de priorité plus élevée sont évalués d'abord.
> Dans le cas d'opérateurs de même priorité l'évaluation a lieu de gauche à droite, on peut utiliser les parenthèses pour imposer un certain ordre d'évaluation.

b) ALLOCATEURS:

L'exécution d'un allocateur crée un objet et donne comme résultat une valeur d'accès qui désigne cet objet.

<allocateur> ::=
new <marque de type> [(<expression>)] / **new** <marque de type> agrégat
/ **new** <marque de type> <contrainte de discriminant> / **new** <marque de type contrainte d'index>.
La marque de type donné dans un allocateur désigne le type de l'objet crée; le type de la valeur d'accès donnée par l'allocateur est défini par le contexte.
Exemple:

```
    New cellule(0, null, null);
    New cellule(value => 0,succ => null,pred => null)
    New matrice(1 ..10,1 . .20)                    -- non initialisé
    New matrice(1..10 =>(1..20 =>0.01)             -- initialisé
```

I.5. INSTRUCTIONS

Instruction = exécution d'actions

<suite d'instructions>: := <instruction>{<instruction>}

I.5.1. Instruction d'affectation:

<instruction d'affectation>: :=<nom de variable> := <expression>

<instruction conditionnelle> ::= **if** <condition>
then <suite d'instructions>
else if <condition>
then <suite d'instructions>}
[**else** <suite d'instructions>]
endif;
<condition>: :=<expression booléenne>

I.5.2. Instruction d'aiguillage:

<instruction d'aiguillage> : := **case** <expression> **is**
{**when** <choix> { /<choix > = <suite d'instructions>}
endcase;

I.5.3. Instruction de boucle:

<instruction de boucle>::=
[identificateur:][<clause d'itération>] <boucle simple>
[identificateur];
<boucle simple>: := **loop** <suite d'instructions> **endloop**

<clause d'itération> ::=
For <identificateur> **in** [**reserve**] <intervalle discret> / **while** <condition>
L'identificateur de boucle doit apparaître 2 fois, une fois au début et une fois à la fin.

I.5.4. Instructions de sortie:

Provoque la sortie d'une boucle englobante, en fonction de la valeur booléenne d'une condition.

<instruction de sortie>: :=**exit** [<nom de boucle>] [**when** <condition>]

• La boucle qui cesse d'être exécutée est la boucle la plus proche à moins que l'instruction de sortie spécifie le nom de la boucle englobante, ainsi c'est la boucle nommée qui cesse d'être exécutée (ainsi que toutes les boucles imbriquées dans la boucle nommée).

• Si l'instruction de sortie contient une boucle, la sortie se fait si la condition a la valeur **true**.

• Une instruction de sortie ne doit figurer qu'à l'intérieur d'une boucle.

I.5.5. Instruction de retour :

Termine l'exécution d'une fonction, d'une procédure ou d'une instruction d'acceptation.

<instruction de retour>: := **return** [<expression>]
Une instruction de retour à l'intérieur d'un corps de procédure ou d'une instruction d'acceptation ne doit pas contenir d'expression

I.5.6. Instruction de branchement:

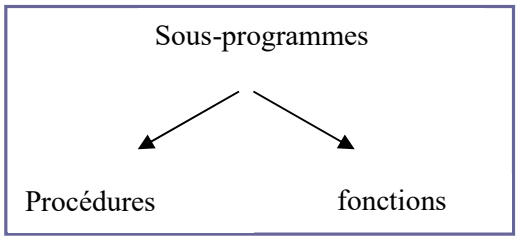
A pour résultat le transfert explicite du contrôle à une autre instruction spécifiée par une étiquette.
<instruction de branchement>: := **goto** <étiquette>;

Exemple:

```
<<test>>
if a(i) < x then
      if b(i) /= 0 then i :=b(i);
           goto test
     endif;
endif;
```

I.6. SOUS-PROGRAMMES

Le sous-programme est une unité de programme dont l'exécution est déclenchée par un appel de ce sous-programme.
Les sous-programmes sont l'une des trois formes de programmes qui permettent de composer un programme, les deux autres formes sont les progiciels et les tâches.



Il y a 3 modes de passage de paramètres:

- • **In** : on n'utilise que la valeur effective du paramètre, le sous-programme ne modifie pas la valeur.
- • **Out**: le sous-programme produit une valeur mais n'utilise pas la valeur du paramètre effectif.
- • **In out**: le sous-programme utilise la valeur du paramètre effectif et peut lui affecter une nouvelle valeur.

SYNTAXE:

<déclaration de sous-programme>: := <spécification de sous-programme> / <déclaration de sous-programme générique>/ <génération de sous-programme>

<spécification de sous-programme>: :=**procédure** <identificateur> [<partie formelle>] / **function** <désignateur> [<partie formelle>] **return** <indicateur de type>

<désignateur> ::= <identificateur>/ <symbole d'opérateur>

<symbole d'opérateur>: := <chaîne de caractères>

<partie formelle>::=(<déclaration de paramètre>{;<déclaration de paramètre>})

<déclaration de paramètre>::=[in]/ [out] / [inout]

Exemple de déclaration de sous-programmes

Procedure incrémenter(x : **inout integer**);

Function trav **return** float in -1.0.. 2.0;

Function premier(m,n :**integer**) **return integer**;

function produit-scalaire(x,y: vecteur) **return float;**

procedure cherche;

Procedure change(x :**out float**);

Remarque : tous les sous-programmes sont réentrant et peuvent être appelés récursivement.

a) Corps de sous-programme : spécifie l'exécution d'un sous-programme.

<corps d'un sous-programme>::=
    <spécification d'un sous-programme> is <partie déclarative>
    begin
        <suite d'instructions>
          [exception
              {<traitement d'exception>}]
    end [désignateur]

Règles:

Le désignateur optionnel situé à la fin du corps du sous-programme doit répéter le désignateur de la spécification du sous-programme.
> Un corps de sous-programme peut provoquer un générateur de code à chaque appel, si cela est demandé par la **pragma INLINE**:
Pragma inline (<nom de sous-programme>{,<nom de sous-programme> });
Cette pragma doit être dans la même partie déclarative que les sous- programmes nommés.

b) Appels de sous-proqrammes : c'est soit un appel de procédure, soit un appel de fonction, il provoque l'exécution du corps du sous-programme.
Au moment de l'appel il a association paramètre effectif avec paramètre formel du sous-programme.

<appel de procédure>::=
        <nom de procédure>[< partie-paramètre effectifs>]
<appel de fonction>::=
                        <nom de fonction> <partie-paramètres effectifs> / <nom de fonction()
<partie-paramètres effectifs>: : =
                        (<association-paramètre> {, <association-paramètre>})
<association-paramètre>::=
                        [<paramètre formel =>] <paramètre-effectif>
<paramètre-formel>: : = identificateur
<paramètre-effectif> ::=expression

I.7. PROGICIELS

Les progiciels ou packages permettent de spécifier des groupes d'entités reliées logiquement entre elles, ils peuvent être utilisés pour décrire des groupes d'entités reliées telles que des types, des objets et des sous-programmes dont les fonctions internes sont protégées de leurs usagers.

Un progiciel est généralement composé de deux parties:
- une spécification du logiciel : la première liste d'éléments de déclaration d'une spécification de progiciel est appelée partie visible du logiciel. Les entités déclarées dans la partie visible peuvent être mentionnés dans d'autres unités de programme au moyen de composants sélectionnés et peuvent être rendus visibles à d'autres unités de programmes au moyen de clauses d'utilisation.

la syntaxe de la partie déclaration est la suivante:
[clause **with**]
package <nom-du-progiciel> **is**
[déclaration des types, constantes, variables, exceptions, sous-programmes, sous-progiciels, tâches exportées par le progiciel]
[partie privée]

end <nom-du-progiciel>;
Les crochets délimitent les parties optionnelles.

- un corps du progiciel: les entités déclarées dans le corps du progiciel ne sont pas accessibles à l'extérieur du progiciel.
La syntaxe de la partie corps est la suivante:
[clause **with**]

package body <nom-du-progiciel> **is**
-- réalisation complète des types
-- déclarations locales au corps du progiciel
-- réalisation complète des sous-pgmes et sous-progiciels
[**begin**
-- initialisation du corps
-- partie exception optionnelle]
end <nom-du-progiciel>

Exemple:
Prenons un progiciel ECRAN dans lequel se trouve une procédure capable d'afficher un message de salutation à l'écran, appelée BIENVENUE , que nous utiliserons dans notre programme:

With ECRAN
-- indique au compilateur que nous travaillons avec ce progiciel ECRAN **Procedure** PROGRAMME **is**
-- c'est le nom de notre application
begin
-- le message va s'afficher: ECRAN.BIENVENUE (« SALUT A TOUS »);
……
end PROGRAMME;

I.8. LES ENTREES/SORTIES : PAQUETAGES STANDARDS

Ada définit un certain nombre d'unités standards qui doivent accompagner tout compilateur validé. Parmi celles ci, on trouve les paquetages d'entées/soties tels que par exemple:

TEXT-IO : gère les entrées/sorties textuelles, et garantit leur forme.

SEQUENTIAL_IO et **DIRECT_IO** : gèrent les entrées/sorties non textuelles, et normalisent jusqu'à la gestion des fichiers à accès direct.

CALENDAR: permet de demander l'heure au système.

SYSTEM: interface de langage avec la machine hôte.

I.9. LES EXCEPTIONS

Les exceptions sont utilisées pour gérer les erreurs qui peuvent intervenir lors de l'exécution d'une séquence d'instructions.

Une exception est déclarée et a un nom, soit par exemple:
package truc **is**
EXP_EXCEPTION : **exception**
begin
……
end truc;

ici le nom est truc.EXP_EXCEPTION
Une exception peut être levée par:
Raise EXP_EXCEPTION

Elle peut être récupérée en fin de bloc:
begin
……
exception
when truc.EXP_EXCEPTION => traitement;
end;

II. ADA COMME LANGAGE DE PROGRAMMATION PARALLELE

II.1. TÂCHES ADA

Les tâches sont des entités qui s'exécutent en parallèle.
Elles peuvent être implémentées en pratique

- sur plusieurs ordinateurs,
- sur un multiprocesseur
- ou sur un processeur unique.

II.1.1. Introduction

Les **tâches** constituent un des trois types principaux d'unités de programme en Ada (les deux autres étant les **sous-programmes** et les **paquetages**).
On considère qu'une **unité de programme principal** est implicitement une **tâche**, mais le programmeur peut déclarer d'autres tâches explicitement à l'intérieur d'autres unités.
Le programmeur peut mettre des tâches dans la partie déclarative de n'importe quelle unité: un bloc, un corps de sous-programme, un corps de tâche ou un paquetage de bibliothèque.

II.1.2. Description des tâches Ada

• En Ada, on n'a pas besoin de démarrer explicitement les tâches. Au contraire, les tâches sont activées après l'élaboration de leur corps, à la fin de la partie déclarative du parent. S'il y a plusieurs tâches déclarées dans la même partie déclarative, elles seront toutes activées à la fin de la partie déclarative, bien que l'ordre d'activation ne soit pas déterminé.

• Une tâche en Ada est un bloc classique :
- déclaration
- corps
- et éventuellement traitement d'exceptions

Elle possède de plus une partie «vue externe » dans laquelle sont décrites les caractéristiques des **points d'entrée** et de **synchronisation**.

• Les **points d'entrée**: peuvent être appelés par d'autres tâches,
ces points d'entrée sont le moyen principal de communication **entre** tâches.

• La **synchronisation** se fait au moyen d'un rendez-vous **entre**
- une tâche qui a appelé un point d'entrée
et une tâche qui a accepté cet appel.

II.1.3. Points d'entrée Ada

Une déclaration de point d'entrée est similaire à une déclaration de sous-programme, mais ne peut se trouver que dans une spécification de tâche.
Les actions à exécuter lorsqu'un point d'entrée est appelé sont spécifiées par l'instruction d'acceptation correspondante.
Les appels de points d'entrée et les instructions d'acceptation, sont les moyens élémentaires de communication et de synchronisation entre tâches.

Syntaxe:
Déclaration-de-point-d'entrée::=
entry identificateur [(intervalle discret)] [partie-formelle];
appel-de point-d'entrée::=
nom-de-point-d'entrée [partie-paramètres-effectifs] ;
instruction-d'acceptation ::=
accept nom-de-point-d'entrée [partie-formelle] [**do**
suite d'instructions
end [identificateur]];

> Lors de l'élaboration d'un point d'entrée, l'identificateur de point d'entrée est introduit en premier, un éventuel intervalle discret est ensuite évalué, et enfin la partie formelle si elle existe, est élaborée comme un sous-programme.
> Une déclaration de point d'entrée qui comporte un intervalle discret, déclare une famille de points d'entrée ayant tous la même partie formelle (si elle existe), pour chaque valeur de l'intervalle, il correspond un point d'entrée.

Le moyen normal de communication des valeurs entre tâches est par appel de points d'entrée et instructions **accept**.

Remarque: Un point d'entrée peut être renommé en procédure

II.2. FORME DES TACHES

II.2.1. Spécification de tâche et corps de tâche

Une spécification de tâche qui commence par les mots-clé réservés **task type** définit un type tâche.

Une spécification de tâche précise l'interface entre une tâche de ce type et d'autres tâches de même type ou de types différents.

<déclaration de tâche>: :=<spécification de tâche>
<spécification de tâche>::=
task [type] <identificateur> [is
{ <déclaration de point d'entrée>}
{<spécification de représentation
end [<identificateur>]];

<corps de tâche>: := **task body** <identificateur> **is** [<partie-déclarative>]
begin
<suite-d'instructions>
[**exception**
{<traitement d'exception>}]
end [<identificateur>];

- • Une spécification de tâche sans le mot réservé type définit une tâche unique, elle introduit un nom de tâche.

- • Les déclarations de points d'entrée et les spécifications de représentation, s'il y en a, sont élaborées dans leur ordre d'apparition.

- • Le corps de tâche définit l'action des tâches (partie exécutable de la tâche).
- • Une instruction d'acceptation spécifie les actions à exécuter lors de l'appel d'un point d'entrée, la partie formelle donnée dans l'instruction d'acceptation doit correspondre à celle donnée dans la déclaration correspondante.

- • Une tâche ne peut exécuter d'instruction d'acceptation que pour ses propres points d'entrée.

Exemple1:

```
Task exemplel is
        -- vue externe de la tâche
       entry synchro;
       -- synchro est un point de synchronisation
end exemplel ;

task body exemple1 is
 begin
       -- la tâche démarre et finit par tomber sur le point de synchronisation ;
       accept synchro ;
         TEXT_IO.PUT_LINE  (* synchronisation effectuée *)
                -- quand synchro est appelé, la tâche continue ...
end exemplel;
```

- La différence entre une tâche et une procédure, est qu'on n'appelle pas une tâche, on la déclare: aussitôt déclarée, elle est lancée (au niveau du begin qui suit).
- Ce qu'on peut appeler dans une tâche, ce sont les points d'entrée (ici synchro).

Remarque: exemplel.svnchro

Cette instruction appelle l'entrée synchro de exemplel
Quand synchro est appelé alors:
- Soit ce traitement est déjà fait, et l'appelant peut continuer son cours.
- Soit-il n'est pas terminé (voire pas commencé) et alors, pendant que l'appelant est coince sur l'appel, la même tâche avance jusqu'au point de synchronisation (si le système n'a pas autre chose à faire de plus important par ailleurs), puis l'appelant est libéré.

Exemples de spécification de type tâche:

```
task type ressource is
  entry prendre;
  entry rendre;
end ressource;

task  type  gestion-clavier is
  entry lire(c :out character);
  entry  écrire(c :in character);
end  gestion-clavier;
```

Exemple de spécification de tâche unique:

```
task producteur-consommateur is
        entry lire(v :out élément);
        entry écrire(e :in élément);
end;
```

- Plusieurs tâches appellent la même entrée, une tâche sera autorisée à effectuer le rendez-vous à un moment donné (en commençant par celle qui a appelé l'entrée en premier). Toutes les autres tâches attendront dans la liste d'attente implicite selon leur ordre d'arrivée (FIFO), et non selon leur priorité.

- Dans le cas où deux tâches appellent l'entrée en même temps, la sélection est arbitraire. Une tache peut abandonner la queue implicite, avant d'avoir achevé le rendez-vous. Cela peut se produire lorsqu'un certain temps d'attente maximum a été dépassé (condition de time out).

• Une fois activée, une tâche peut être dans l'un des états suivants:

- **Actif** la tâche possède actuellement un processeur.
- **Prête** la tâche est débloquée et en attente d'exécution.
- **Bloquée** la tâche est en délai ou en attente de rendez-vous.
- **Achevée** la tâche a terminé l'exécution de sa séquence d'instructions.
- **Terminée** la tâche n'a jamais été, ou n'est plus active.

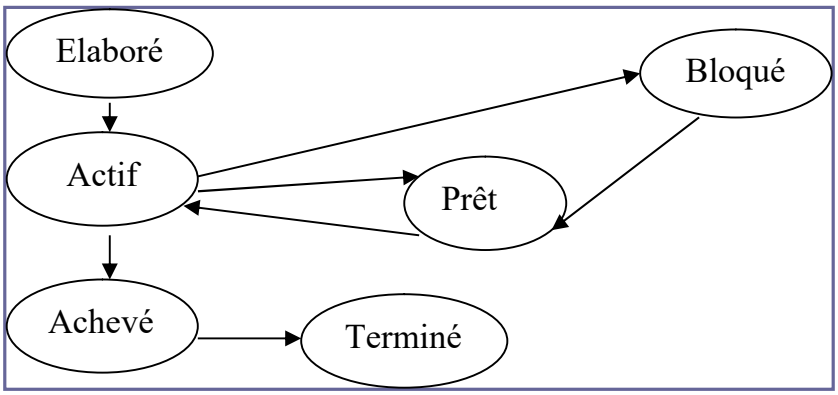


Fig. 4-2-1 : état des tâches.

La figure **Fig. 4-2-1** montre les interactions entre ces états.

La tâche est d'abord élaborée, elle ne sera active qu'à la fin de l'élaboration de région déclarative englobante. Après activation, l'état de la tâche peut être actif, prêt ou bloqué.
Actif: la tâche dispose des ressources de calcul.
Prêt: la tâche attend des ressources de calcul mais est, pour le reste prête à s'exécuter.
Bloqué: la tâche attend un événement, comme un rendez-vous.

• Lorsqu'une tâche est active, elle peut continuer à s'exécuter jusqu'à ce qu'une tâche de plus haute priorité passe à l'état prêt. Lorsqu'une tâche a fini l'exécution de sa suite d'instructions, elle devient **achevée** et **attend** que toutes ses tâches dépendantes se terminent. Une fois achevée, la tâche ne peut plus participer à aucun rendez-vous. Lorsqu'il n'y a aucune tâche « enfant» active, la tâche est terminée.

• A chaque tâche y compris le programme principal, est associée une priorité statique qui indique un degré d'urgence. L'utilisateur peut spécifier cette priorité explicitement par **la pragma PRIORITY** (priorité), si aucune priorité n'est spécifiée pour une tâche donnée, c'est le compilateur qui choisira une valeur.
Les tâches en attente seront servies, puisque les queues d'Ada sont définies comme: « premier entré, premier servi »
But des priorités : aider à allouer les ressources de calcul.

- les processeurs
- l'espace mémoire

II.2.2. Attributs des tâches et des points d'entrée

Pour un objet tâche ou un type tâche T sont définis les attributs suivants:

T'TERMINATED: de type booléen, cet attribut est initialisé à FALSE à la déclaration ou allocation) de la tâche, et devient TRUE quand elle est terminée.

T'PRIORITY : la valeur de cet attribut est la priorité de la tâche T, si elle a une.

T'STORAGE_SIZE: cet attribut de type entier, donne le nombre d'unités de mémoire allouées à la tâche T.

E'COUNT: de type entier, utilisé dans le corps de la tâche T, donne le nombre d'appels de points d'entrée dans la file d'attente associée à E (E: point d'entrée d'une tâche T)

II.2.3. Priorités

Chaque tâche peut avoir éventuellement une priorité qui est une valeur entière du sous-type prédéfini PRIORITY.
Quand une faible valeur est indiquée **=>** un faible degré d'urgence.
L'intervalle des priorités est défini par l'implémentation. La priorité est associée à une tâche au moyen de la pragma:

Pragma PRIORITY expression

On associe une priorité à un programme principal en plaçant cette pragma dans la partie déclarative la plus extérieure. Cette pragma ne peut figurer qu'une seule fois dans une tâche (ou dans le programme principal).

L'effet de priorité de l'ordonnancement est défini par la règle suivante:

Si deux tâches de priorités différentes sont prêtes à être exécutées et peuvent s'exécuter en utilisant les mêmes ressources de traitement, alors la tâche de plus faible priorité ne doit pas s'exécuter si celle de priorité supérieure ne s'exécute pas.

Pour des tâches de même priorité, l'ordre de séquencement n'est pas défini par le langage.
Pour des taches qui n'ont pas de priorité explicite, les règles d'ordonnancement ne sont pas définies, sauf si elles sont engagées dans un rendez-vous.

II.2.4. Avortement d'une tâche

La terminaison anormale d'une tâche ou plusieurs tâches peut être provoquée par une instruction d'avortement.
Syntaxe:

Instruction-d'avortement ::= **abort** nom-de-tâche{,nom-de tâche} ;

- La terminaison anormale d'une tâche provoque celle de toutes les tâches qui en dépendent (directement ou indirectement)

- Si une tâche qui a appelé un point d'entrée est terminée anormalement, elle est retirée de la file d'attente de ce point d'entrée; si le rendez-vous est en train de s'effectuer, la tâche appelante est terminée, mais la tâche appelée peut terminer le rendez-vous normalement.

- Si des tâches sont en attente sur une entrée d'une tâche anormalement terminée, une exception **TASKING_ERROR** est activée dans chacune de ces tâches, à l'endroit de l'appel du point d'entrée, y compris pour une tâche déjà engagée dans un rendez-vous.

II.2.5. Types de tâches

Les tâches peuvent être traitées comme des variables, on peut ainsi les créer dynamiquement. Pour cela, il faut déclarer un type de tâche:

Task type truc i**s**

…….

End truc;

Task body truc is

…….

end truc;

Chaque tâche d'un type de tâche possède les points d'entrées déclarés dans la spécification du type. Au sein du corps d'une tâche chacun de ses points d'entrée (ou famille de points d'entrée) peut être désigné par l'identificateur correspondant.
Le nom d'une des entrées d'une famille prend la forme d'un composant indexé, le nom de la famille étant suivi d'un index entre parenthèses.

II.3. LES RENDEZ-VOUS ADA

Le mécanisme de rendez-vous permet:

>D'échanger des infos avec une tâche: le point d'entrée peut avoir des paramètres, et ainsi recevoir ou transmettre des valeurs.

>Le pouvoir d'attendre plusieurs événements en même temps et traiter le premier qui se présente.

>**Synchronisation**: la tâche appelée doit arriver à une instruction « accept » correspondante.

En ce oui concerne les points d'entrée:
Pour la synchronisation, deux cas de figure sont possibles:

- Si la tâche appelante invoque un point d'entrée avant que l'instruction d'acceptation correspondante n'ait été atteinte par la tâche qui possède ce point d'entrée, alors la tâche appelante est suspendue.

- Si une tâche atteint une instruction d'acceptation avant tout point-d'entrée, l'exécution de la tâche est suspendue.

Si plusieurs tâches appellent le même point d'entrée avant que l'instruction d'acceptation correspondante ne soit atteinte, ces appels sont mis en file d'attente. A chaque point d'entrée est associée une file d'attente, les appels sont traités suivant leur ordre d'arrivée.

En ce qui concerne le partage de variable commune:

- Si deux tâches lisent ou mettent à jour une variable partagée (une variable accessible par les deux), alors aucune d'elles ne peut supposer quoi que ce soit sur l'ordre dans lequel, d'autres effectuent ces opérations, sauf au points où elles se synchronisent. Deux tâches sont synchronisées au début et à la fin de leur rendez-vous. Au début et à la fin de son activation, une tâche est synchronisée avec la tâche qui provoque cette activation.

- Si entre deux points de synchronisation d'une tâche, celle ci lit une variable partagée dont le type est un type scalaire ou accès, alors la variable n'est mise à jour par aucune tâche à aucun moment entre ces deux points.

> **Exclusion mutuelle**: si deux ou plusieurs tâches appellent le même point d'entrée, seul un appel peut être traité à la fois.
Un moyen de passer les paramètres est fournit dans les appels d'entrées.

Exemple2:

task Exemple2 **is**

entry Echange(parameter: **in integer**; résultat :**out character**);

End exemple 2;

```
task body Exemple2 is
   begin
      ……..
      accept Echange(paramètre : in integer; résultat: out character) do
            Traitement(paramètre);
      Résultat:=  … ;
     end Echange;
      ……
  end Exemple2;
```

L'appel aura lieu comme suit:
Exemple2**.** Echange(2,c)

On peut aussi *« renommer »* le point d'entrée en procédure pour en simplifier l'appel:
Procédure Masquée(paramètre :**in integer**; résultat :**out character**) **renames** Exemple2.Echange;

L'appel de Masquée se comportera comme un point de synchronisation mais, en plus les paramètres sont passés et traités.

Tout se passe comme si l'on avait appelé de l'extérieur une procédure
Exemple2**.**Echange dans le contexte de la tâche.

Les variables paramètre et résultat n'ont d'existence dans la tâche qu'entre l'*« accept »* et le *« end Echange »*

Une fois déclarés, les points d'entrée, peuvent être utilisés plusieurs fois dans le corps de la tâche.

Exemple3:

```
Task Exemple3 is
  entry Echange(paramètre :in integer ;résultat :out character);
End Exemple3;
Task body Exemple3 is
   begin
     --  première utilisation
     accept Echange(paramétre:in integer ;résultat :out character) do
           traitement(paramètre);
     résultat :=… ;
     end Echange;
      ……..
    if ... then     -~
              -deuxième utilisation
    accept Echange(paramètre:in integer ;résultat :out character) do
           autre-traitement(paramètre);
    résultat :=…. ;
   end Echange;
   else
        -troisième utilisation
   accept Echange(paramètre:in integer ;résultat :out character) do
           encore-autre-traitement(paramétre);
  résultat :=… ;
  end Echange;
  endif;
  ……..
  end Exemple3;
```

II.4. SIMULATION D'UN SEMAPHORE AVEC UN RENDEZ-VOUS

Soit à simuler un sémaphore binaire avec un rendez-vous, il faut créer une procédure sémaphore pour servir d'intermédiaire entre les processus faisant appel au sémaphore, il vient:

```
Procedure Exc-mut is
Task Semaphore is
        Entry wait;
        Entry signal;
End Semaphore;
Task body Semaphore is
Begin
                Loop
                Accept wait;
                Accept signal;
                End loop;
End Semaphore;

Task P1;
Task body P1 is
Begin
        Loop
                Reml;
                Wait;
                Critl ;
                Signal;
        endloop;
    End P1;

Task P2;
Task body P2 is
Begin
        Loop
                Rem2;
                Wait;
                Crit2;
                Signal;
        End loop;
End P2;
Begin
        Null;
End Exc-mut;
```

Lorsque le processus P1 exécute un appel de wait il attend que le processus Sémaphore exécute le accept correspondant. Dès que le rendez-vous a lieu, P1 entre en section critique, lors du rendez-vous, il n'y a ni paramètre, ni instruction à exécuter. Seul le mécanisme d'attente est utile pour la simulation du sémaphore. Le processus P2 sera bloqué lorsqu'il appela à son tour l'instruction wait puisque le processus Sémaphore attend le rendez-vous signal.

Tant qu'un processus (ici P1) n'aura pas appelé signal, le processus Sémaphore est bloqué et par conséquent P2 aussi. Quand P1 aura fini sa section critique, il effectuera son rendez-vous signal avec le processus sémaphore qui réalise alors une autre boucle et le rendez-vous wait avec P2 se produit.

Remarques :

> Comme les files d'attentes sur les instructions sont FIFO, le sémaphore obtenu sera FIFO: même si P1 rattrape P2, il sera placé après lui dans la file d'attente de l'instruction accept wait.

> Dés l'instant, où un processus entre dans la file d'attente, il est certain d'obtenir son rendez-vous dans un temps fini: le temps que les processus qui sont avant lui réalisent le leur.

> Ceci est un exemple d'implantation d'un seul sémaphore, s'il en faut plusieurs, il faut dupliquer le processus sémaphore et paramétrer les appels pour reconnaître le sémaphore concerné (en Ada, il est possible de déclarer un tableau de tâches sémaphore et les sémaphores seront ainsi adressés par l'indice du tableau).

II.5. ATTENTES MULTIPLES

Attendre simultanément plusieurs événements, et traiter le premier qui se présente.

Exemple4:

```
task Exemple4 is
        entry entrée1;
        entry entrée2 (i :in integer);
        entry entrée3 (c :out character);
        ……
end Exemple4;

task body Exemple4 is
begin
        ……..
        select
                accept entrée1;
        or
                accept entrée2 (i :integer) do
        ……
        or
                accept entrée3 (c :out character) do
        ……..
        end select;
        ……
end Exemple4;
```

• La tâche attendra la première activation de entrée1, entrée2 ou entrée3 indifféremment pour continuer. L'entrée appelée sera « servie » mais le traitement de la tâche continuant, les autres entrées ne le seront pas, dans le cadre de cette instruction **select**.

• Si l'entrée comporte des paramètres, ils seront traités, bien sur comme dans l'Exemple3.

II.6. ATTENTE GARDEE

Il est possible de programmer une attente sélective (avec gardes), en utilisant l'instruction **select**, donnant ainsi la possibilité de réaliser une combinaison d'attentes et de sélection d'une alternative parmi plusieurs. La sélection dépend des conditions associées à chaque alternative.
Une possibilité de sélection est dite ouverte si elle n'est précédée d'aucune clause **when** ou si la condition correspondante est vraie; sinon elle est dite fermée. Donnons la forme générale d'une instruction **select** avec gardes:

```
Select
        When condition1 => accept entry1 do instructions1 end;
                Others instructions;
        Or
        When condition2 => accept entry2 do instructions2 end;
                Others instructions3
                ……..
        else instruction0
end select;
```

Remarque: L'alternative **else** est facultative.

Déroulement de l'instruction select:

1) Evaluer toutes les gardes pour déterminer les alternatives ouvertes.
2) S'il y a des alternatives ouvertes, déterminer qu'elles sont les instructions **accept,** dans les alternatives ouvertes, ayant un rendez-vous en attente.
3) S'il y en a, exécuter une telle alternative. Lorsqu'il y en a plusieurs, le choix de l'alternative à exécuter se fait au hasard.
4) S'il n'y a pas d'alternative ouverte ou s'il y en a mais aucune n'a de processus en attente, exécuter la clause else, si elle est présente.
5) S'il n'y a pas de clause **else** et pas de processus en attente, attendre le premier processus qui se présentera pour un rendez-vous dans une alternative ouverte.
6) S'il n'y a pas d'alternative ouverte et qu'il n'y a pas de clause **else,** c'est qu'il y a erreur.

Exemple :
Prenons l'exemple d'une tâche qui tamponne une entrée, et l'offre à qui la demande: elle doit accepter des caractères, et ceci tant que le tampon n'est pas plein. Quand le tampon est plein, elle doit refuser l'entrée. Par ailleurs, elle doit toujours accepter la lecture du tampon, sauf quand il est vide. Cette tâche est implémentée comme une boucle.

Exemple 5:

```
Task Exemple5 is
    entry input-character(c :in character);
    entry output-string(s :out ;last :out integer);
end Exemple5;
task body Exemple5 is
     max-string :constant integer :=80;
     buffer :string(1 . . max_string);
    index :integer :=O;
begin
        Loop
                Select
                        When index <max-string =>
                        -- on accepte un caractère si le tampon n'est pas plein
                        accept input_character(c :in character) do
                                        index := index+1;
                                        buffer(index) :=c;
                        end input_character;
                        or
                        When index > 0 =>
                            -- on accepte la lecture du tampon s'il n'est pas vide
                        accept output_string(s :out string ;last :out integer) do
```

```
                        S(S'FIRST..S'FIRST+index-l):=buffer(1..index);
                            last :=S'FIRST+index-l;
                            index =0;
                    end output_string;
            end select;
        end loop;
end Exemple5;
```

L'attente gardée peut se terminer par un de ces trois choix:

Choix: else

Si aucun rendez-vous n'est possible (condition du when fausse ou simplement personne n'a appelé l'entrée), alors les instructions suivant le else sont exécutées sans attendre et le traitement se poursuit après le **end select**

Exemple:

```
select
accept es-tu-la;
else TEXT_IO.PUT_LINE ("il n'est pas la");
end select;
```

Choix :delay

Si aucun rendez-vous n'est possible (même raisons) alors:
- on attend le temps spécifié (en secondes), puis si rien ne se passe:
- on exécute les instructions qui suivent le delay.
- On passe en séquence après le end select.

Si pendant le temps d'attente, un rendez-vous venait à être possible:
- la branche correspondante est activée.
- On passe en séquence après le end select.

Exemple:

```
    Select
        When ... =>accept ES TU_LA_AHMED;
                -- accept sans paramètres gardée par un when
        or
        accept ES_TU_LA....ALI(...) do;
                --accept avec paramètre non gardé
        …….
        end ES_TU_LA_ALI;
        or
        delay 0.05;
        -- clause delay
        PUT_LINE (« personne n'est venue, j'ai attendu pourtant");
    End select;
```

Il est possible de mettre plusieurs choix delay dans une instruction select le choix ayant le délai le plus court sera bien sur sélectionné: cette possibilité n'a donc d'intérêt que si les différents délais sont calculés dynamiquement.

Choix: terminate

Il sert à gérer la terminaison propre des tâches dans un système multitâches, il devient rapidement très difficile de savoir si le traitement est terminé.

Exemple:
Select
    Accept ES_TU_LA ;
or terminate;
end select;

-- la tâche accepte d'être tuée dans cet état,
-- Si ES_TU_LA n'est pas demandé, et si toutes les tâches
-- dépendantes sont dans la même situation.
Si dans une instruction select gardée, toutes les conditions when venaient à être fermées, alors l'exception **TASKING-ERROR** serait provoquée. Il vaut mieux toujours prévoir une branche d'échappement (**else, delay** ou **terminate).**

II.7. CONCLUSION

Le concept de rendez-vous tel qu'il vient d'être décrit est souvent trop restrictif, il apparaît souvent nécessaire de permettre à un processus d'accepter un appel parmi plusieurs possibles suivant une condition. Au cas où plusieurs appels sont possibles (remplissent la condition), un choix non déterministe est effectué. Nous avons montré qu'Ada permet de simuler des sémaphores ou des événements et de masquer complètement, à un certain niveau, le mécanisme de rendez-vous.

III. PROCESSUS ET COMMUNICATIONS EN ADA

Dans cette partie, nous présentons successivement :

- la notion de **tâche** permettant de décrire des processus,
- les **outils** permettant d'exprimer l'appel de procédure à distance,
- les **mécanismes** offerts pour la programmation non déterministe
- et enfin quelques **exemples** de programmes.

III.1. LA NOTION DE TACHE

Une **tâche** est une unité de programmation séquentielle exécutable en parallèle avec d'autres tâches et avec l'unité de programme principal.

La déclaration d'une tâche s'effectue en **trois** étapes:

- définition d'un **modèle** de tâche grâce à une déclaration de type,
- définition du **corps** du modèle, c'est-à-dire de la partie algorithmique de ce modèle (texte de la tâche, texte des procédures...),
- **création** effective d'objets de type tâche.

La définition d'un **modèle de tâche** introduit :
- un nom de tâche
- une liste de noms de procédures (entrées) qui pourront être utilisées par des tâches externes. C'est en quelque sorte l'interface de la tâche.

La **création d'objets** de type tâche s'effectue par application de l'opérateur ***new*** à l'identificateur d'un modèle de tâche.

Voici un **exemple** classique **d'utilisation des tâches**:

Declare ***#partie*** *déclaration* **#**
(α1) ***task*** *ressource* ***is*** *# modèle de ressource* **#**

```
            entry acquerir ;
            entry relacher;
        end ressource;
        …
        type ref_ressource is access ressource;
        ref : ref_ressource ; # ref est un objet susceptible de repérer un objet
        de type ressource#
        …

(α2)    task body ressource is
            # déclarations #
        begin
            # partie algorithmique, donc entre autres choses, définition
              concrète de acquerir et relacher #
        end ressource;
        ….

        begin # programme principal #

(α3)     ref := new ressource;
        ….
        End
```

En (α1) on trouve la définition d'un modèle de tâche dont le corps est précisé en (α2).
Un exemplaire de tâche est créé en (α3).
On peut ensuite citer cet exemplaire grâce au nom *ref.*

Dans le cas où il n'existe qu'un objet d'un type tâche donné, il est possible d'utiliser une notation simplifiée : la définition du modèle et la déclaration de l'objet sont simultanées. Il est alors impossible de faire référence au type tâche correspondant qui de ce fait est anonyme.

III.2. DEFINITION ET APPEL DES ENTREES

L'en-tête des entrées se trouve dans la partie spécification des tâches (ou modèles de tâches) et leur texte se trouve dans le corps de tâche.
La définition du corps d'une entrée est très similaire à celle d'une procédure. Elle prend la forme suivante:

```
accept nom_entree (<paramètres formels>) do
        <texte d'entrée>
end nom_entree;
```

Cependant ***accept*** est plus qu'une déclaration, c'est aussi une instruction qui ne s'exécute que si une autre tâche accomplit un appel de cette entrée, c'est-à-dire une instruction telle que

```
        nom.entree (<paramètres effectifs>)
```

Ainsi, si lors d'un appel la tâche appelée est en attente sur l'entrée,
alors le "rendez-vous" a lieu immédiatement,
sinon l'appelant est mis en attente et est donc interrompu.

Lorsqu'une tâche atteint une instruction ***accept p,***
s'il y a des demandes d'exécution de p en attente**,**
alors la "plus ancienne" est prise en compte pour un rendez-vous immédiat,

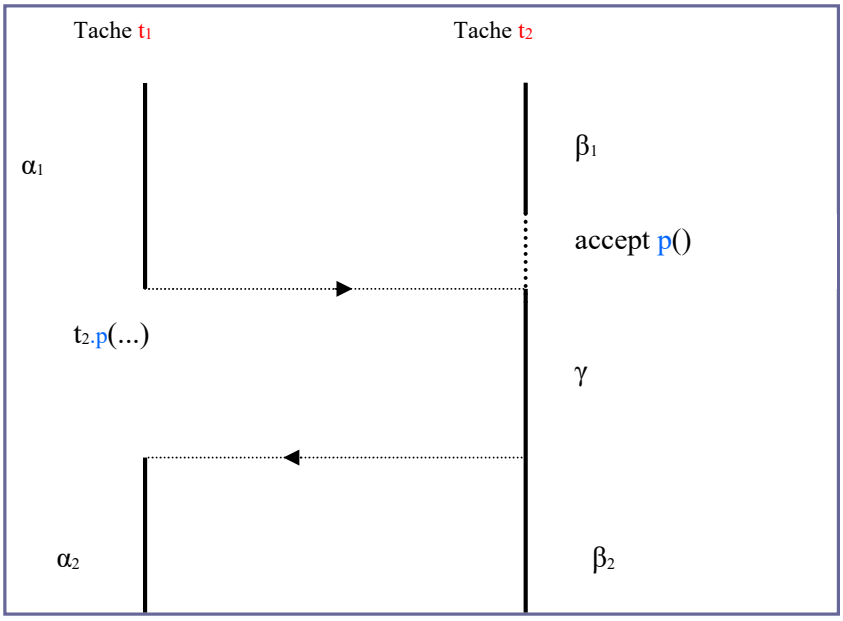
sinon la tâche se met en attente d'appel.

L'instruction d'appel est terminée lorsque l'exécution du corps de l'entrée est terminée, c'est-à-dire son instruction **end.**
Notons qu'il peut y avoir, dans le corps d'une même tâche, plusieurs instructions ***accept*** désignant la même entrée. Ainsi, en fonction de l'état de la tâche, des traitements distincts peuvent être effectués.

Illustrons ce mécanisme d'appel d'entrée par un **exemple**.
Considérons l'exécution de deux tâches t1 et t2 dont les corps sont respectivement:

```
task body t1 is
   begin
        α1;
        t2.p,
        α2
    end;

task body t2 is
   begin
        β1;
        accept p(...) do
          γ
        end;
        β2
    end
```



Les suites d'instructions **α1** et **β1** appartenant respectivement aux tâches ***t1*** et ***t2***, sont d'abord exécutées.
Après exécution de **β1**, la tâche t***2*** se met en attente de l'exécution d'un appel de l'entrée ***p*** par ***t1.***
Le rendez-vous n'a lieu que lorsque la tâche ***t1*** exécute l'appel **t2.p**(...).
Ensuite l'exécution du corps **γ** de **p** a lieu puis les suites d'instructions **α2** et **β2** sont exécutées en parallèle.

On remarque que le schéma d'appel d'entrée correspond tout à fait à un appel de procédure à distance suivant le modèle non hiérarchique.

Voici un **exemple** de programme ADA permettant le partage d'une ressource.

```
task ressource is
        entry acquerir;
        entry relacher;
end ressource;
task body ressource is
begin
        loop
                accept acquerir;
                accept relacher;
        end loop
end ressource;
```

acquerir et ***relacher*** sont les entrées d'une tâche ressource dont le rôle est essentiellement de sérialiser les appels à ***acquerir*** et ***relacher.***
Remarquons que les deux entrées ne possèdent ni paramètres, ni corps explicites, elles permettent de mettre en oeuvre une synchronisation pure.

III.3. EXPRESSION DU NON-DETERMINISME

Le concept de rendez-vous tel qu'il vient d'être décrit est souvent trop restrictif. En effet, il apparaît très souvent nécessaire de permettre à un processus ***d'accepter*** un appel parmi plusieurs possibles suivant qu'une condition est réalisée ou non.
Au cas où plusieurs appels sont possibles (c'est-à-dire remplissent la condition) un choix non-déterministe est effectué.
En vue de modéliser ce comportement, on utilise l'instruction ***select*** dont la forme générale (côté appelé) peut être décrite ainsi:

<instruction_select> ***::=***
select
{when *<condition>* ***=>} o /1*** *<alternative>*
{or {when *<condition>**} o/1*** *=> <alternative>**} ****
{else *<suite d'instructions>**}***
end select;

<*alternative****> ::= <****instruction_accept****> {****suite d'instructions****} /***
<instruction delay> ***{****suite d'instructions****}/***
terminate

<instruction_accept> ::= ***accept*** *id (<parametre>) do.. .end;*

<instruction delay> ::= ***delay*** *time;*

Ainsi une instruction ***select*** est une instruction structurée dont chacune des composantes est constituée d'une garde éventuellement vide (partie ***when****)* suivie d'une branche.
Une branche est dite ouverte si la garde associée est ***vraie*** *ou si* cette garde est vide.
Une branche est franchissable lorsqu'elle est ouverte avec au moins un appel en attente sur l'entrée associée. Le choix entre plusieurs branches franchissables est non-déterministe.

En plus de l'instruction ***accept,*** les branches peuvent être constituées d'une instruction ***delay*** permettant de définir un délai maximum d'attente d'appel ou d'une instruction ***terminate*** dont l'exécution implique la terminaison de la tâche correspondante.

Enfin, signalons qu'une instruction ***select*** peut également comporter une partie ***else.*** Dans ce cas, toutes les autres branches doivent être des branches de rendez-vous et si aucun rendez-vous n'est franchissable immédiatement, alors la partie else est exécutée et l'instruction ***select*** est terminée.

Exemple du tampon borné
Le tampon borné peut être décrit en ADA à l'aide d'une instruction ***select*** qui permet de réaliser un choix non-déterministe entre tâche productrice et tâche consommatrice.

Le programme ADA qui suit traite d'un tampon de dix éléments.
Une variable *i* indique le premier emplacement disponible
et une variable j indique le prochain élément à consommer.
- La condition *i = j + 10* correspond à un tampon plein,
- la condition *i* = j à un tampon vide
- et la condition j < *i <j + 10* correspond à un tampon dans lequel il est possible de produire et à partir duquel il est possible de consommer.

```
task body t is
        tampon : array (0.. .9) of character;
        i, j integer:= 0;
begin
        loop
                select
                        when i <j + 10 =>
                                accept produire (e: in  character) do
                                        tampon (i mod 10) :=  e;
                                end;
                                i:=i+1
                or

               when j < i =>
                        accept consommer (e: out character) do
                                e:= tampon (j mod 10)
                        end;
                        j:=j+1
                or
                        terminate
                end select;
        end loop;
end t;
```

Côté appelant, l'instruction ***select*** peut être utilisée dans deux situations:

- d'une part la demande de rendez-vous immédiat,
- d'autre part l'appel limité dans le temps.

- Dans le premier cas, si un rendez-vous avec la tâche appelée est impossible immédiatement, la tâche appelante exécute une autre suite d'instructions (partie *else),*
- dans le second cas, si une demande de rendez-vous ne peut être satisfaite dans un délai donné, une suite d'instructions appropriées est exécutée.

La syntaxe de **select** (côté appelant) peut se résumer ainsi:

<appel conditionnel> ::=

```
        select
                <appel d'entrée>
        else
                <suite d'instructions>
        end   /
        select
                <appel d'entrée>
        or
                <instruction delay>;
                <suite d'instructions>
        end
```

Ainsi, on pourra écrire des instructions *select* comme:

```
        select
                ressource.acquerir;
                put ("ressource acquise");
        else
                put ("ressource non acquise");
        end select;
```

ou bien

```
select
        ressource.acquerir;
        put ("ressource acquise");
or
        delay 20.0;
        put ("ressource non acquise")
end select;
```

III.4. QUELQUES EXEMPLES DE PROGRAMMES ADA

Dans cette partie, nous présentons quelques exemples classiques de programmes ADA.

Exemple 1 : Exclusion mutuelle

Considérons le problème de l'exclusion mutuelle à une structure de données.
Une solution possible est la suivante:

```
task structure is
        entry ecrire (elt : in out type_elt)
end;
…….
Task body structure is
        …
        table:… # définition de la structure de données #
        …
        begin
                loop
                        accept( ecrire (elt : in out type_ elt) do
                                …..
                                # accès à la structure de données#
                                ……
                        end ecrire;
                end loop;
        end structure;
```

L'accès à *structure* se fait en exclusion mutuelle: seule une entrée *écrire* peut être traitée à un instant donné.

Exemple 2 : Tampon à un emplacement

On veut construire un programme permettant la gestion d'un tampon à une place.
Ce tampon est accessible via deux procédures *lire* et *écrire* et les synchronisations introduites doivent assurer l'exécution d'une suite de couples *(écrire* ; *lire),* en aucun cas une information ne peut être lue avant d'avoir été écrite.

```
 task boite is
                entry ecrire (x : in integer);
                entry lire (x : out integer)
end boite;
task body boite is
        v : integer;
        begin
                loop
                        accept  ecrire (x : in integer) do
                                v :=x
                          end;
```

```
                    accept  lire (x : out integer) do
                            x :=v
                        end
                end loop;
        end boite;
```

Exemple 3 : Variable partagée

Le problème consiste à construire un programme permettant de gérer une variable partagée. Cette variable peut être écrite une fois puis lue un nombre indéterminé de fois avant d'être réécrite, la première écriture étant effectuée lors de l'initialisation. Voici un programme solution :

```
task variable_partagee is
                entry lire (x : out elt);
                entry ecrire (x : in elt)
end;
…..

task body variable_partagee is
                v : elt := init;
        begin
            loop
                    select
                        accept  lire (x : out elt) do
                                x :=v;
                            end;
                    or
                        accept  ecrire  (x : in elt) do
                                v :=x;
                            end;
                    end select;
                end loop;
        end variable_partagee;
```

Au moment de l'exécution de l'instruction ***select***, plusieurs situations sont envisageables :

(i) ni *lire*, ni *écrire* n'ont été appelées, la tâche *variable_partagee* est alors mise en attente ;

(ii) l'une des opérations *lire* ou *écrire* a été appelée, elle est immédiatement traitée ;

(iii) les deux opérations ont été appelées : un choix non-déterministe est opéré.

Dans tous les cas, les synchronisations désirées sont bien mises en œuvre.

III.5. UN POINT DE VUE SUR LE PARALLELISME EN ADA

Il est intéressant de remarquer que l'instruction ***accept*** possède quasiment la même syntaxe qu'une procédure traditionnelle.
De même, un appel à une instruction ***accept*** est semblable à un appel de procédure.
Le schéma de contrôle suivi après acceptation d'un appel est également le schéma procédural.
En fait, seul le fait qu'un appel à une instruction ***accept*** puisse être retardé par le serveur constitue une différence importante.

Le choix effectué par ADA quant à l'outil d'expression du parallélisme est bon.