CHAPITRE 3 Langages de plus “haut niveau”: Concepts de base

Langages de plus haut niveau,

Cadre "asynchrone" et "impératif".

Quelque soient les options choisies, au moins trois dimensions fondamentales :

Activités //

Communications

Synchronization

3.1 Dimensions fondamentales

3.1.1 Exécutions parallèles

Parallèle ==>

plusieurs exécutions simultanées

plusieurs ***processus*** dans le cadre "asynchrone" et "impératif"

Au niveau langage:

constructions syntaxiques permettant

de définir, de déclarer des processus

Ada Taks, CSP [ ], objet pour POO//, etc.

Solutions possibles:

Création:

- statique (nb. et type connus à la compilation, e.g CSP)
- dynamique (nb. inconnu, Ada, POO//)

Déclaration:

- à plat
- imbriquée (CSP [[ ]])

Exécution:

- séquentielle

...

- quasi-séquentielle

...

- parallèle

...

3.1.2 Communication

Transfert d'information d'un processus à l'autre

Deux grands types:

- Variables partagées
- Envoi de messages

Envoi de message: différentes sémantiques:

Cas d'une instruction v := p.f (...);

Synchrone:

- Rendez-vous
- RPC (Remote Procedure Call)

Asynchrone

- Uni-directionnelle
- Bi-directionnelle (résultat, futur)

--> •Actors

--> •Objets Actifs

- Sure (préserve l'ordre FIFO)
- Asynchrone avec Rendez-vous

==> Interruption

ou ==> Multithread

3.1.3 Synchronisation

(Synchronisation interne à un processus)

Satisfaction de contraintes relatives à l'enchaînement des actions des différents processus

Mécanismes possibles:

- Attente active

Exemple: while (not cond) do end;

- Sections critiques, Sémaphores, Moniteurs

- Expressions de chemins:

Exemple: {put, get}*

- Rendez-vous ou Com. synchrone

- Services bloquants

Exemple: Accept unlock;

- Remplacement (become)

lg. d'acteurs: changement d'état dans un lg. fonctionnel

Cadre initialement fonctionnel,

Le “become” remplace la notion d’état.

- Verrouillage: lock

read lock: bloque tous les processus qui tentent de modifier la donnée verrouillée

write lock: ... lire ou modifier ...

En fait:

Synchronisation et Communication ne sont pas deux phénomènes indépendants

--> • Communication synchrone est une synchronisation (implicite)

--> • Même une com. asynchrone est une synchronisation:

un processus ne peut pas utiliser un message, une valeur qu'il n'a pas encore reçue

3.2 Autres dimensions

3.2.1 Topologie

Topologie d'une application (dynamique, exécution):

Un graphe orienté:

- Noeud: un processus du système
- Arc: référence d'un processus vers un autre

Une communication potentielle

Choix:

--> • La topologie d'une application est-elle statique ou dynamique?

Statique:

Concurrent Pascal, DP, CSP, OCCAM,

Dynamique:

Ada, Plasma, Act1, ABCL, Orient84/K, ConcurrentSmalltalk, POOL, Hybrid

De plus en plus:

--> • Topologie Dynamique
Création dynamique ==> Topologie Dyn.

Propriétés: Arbre, DAG, cycle

3.2.2 Partage ou non

Dans le cadre d'un langage basé sur la communication:

Est-ce qu'il y a partage d'information?

- variables partagées
- moniteurs (zone ***passive*** partagée)

3.2.3 Passage des paramètres

Lors des communications entre processus:

Inter Process Communication (IPC)

Passage des paramètres:

- Par copie (par valeur)
- Par référence (par nom)

Ou plus sophistiqué:

- Groupé (e.g. Liskov, Lg. Argus)
- Lazy parameter passing (P2P, Eugster)
- Future de première classe

Implication sur le partage:

Par référence généralisée ==> Partage

Par copie ==> n'implique pas le partage

3.2.4 Services et Points d'Entrées

Processus communicants:

- Points d'Entrées:

e.g.: Entry point d'Ada, RPC

Caractéristique importante:

Unification Point d'Entrée et Routine?

a) Modélisation des routines

Possibilité fondamentale:

Passer des routines en paramètre?

Flexibilité et modularité de l'expression des services

Routine de première classe (1st class citizens, simple si vesion réduite, sinon réflexivité)

Exemple:

opérateur & de C

F (3, v, & g) où g est une fonction

b) Modélisation des requêtes et de leur

service

Possibilité fondamentale:

Une modélisation des requêtes?

Pouvoir les passer en paramètre, les affecter dans une variable, les stocker dans une liste

Réaliser le service comme une opération quelconque

Requêtes de première classe

c) File d'attente des requêtes

Une liste des requêtes en attente accessible par le programmeur?

Enjeux: Flexibilité et modularité

Choisir, sélectionner une requête ==> parcourir la liste

Ada: pas directement accessible

idem majorité des autres lg. (e.g. Java)

3.2.5 Programmation de l'activité des processus

a) Contrôle

Définition de l'activité d'un processus (ou moniteur, Objet Actif, etc.)

- Activité propre
- Service des requêtes

Caractéristiques du contrôle:

Répartition géographique:

Centralisé:

Son expression est regroupée en un point du processus, et (totalement) séparée de celle des routines et des points d'entrées

Exemple: Ada

Décentralisé:

Réparti en plusieurs points du processus, en général mélangé avec le code des routines

Exemple: type moniteur (Lg. Hybrid)

Méthode d'expression:

Contrôle explicite:

Programmer directement l'activité du processus, un fil de contrôle, (select).

Contrôle implicite:

Les autres cas, en général déclaratif

Décentralisé ===> implicite

Centralisé ===> implicite ou explicite

b) Services des requêtes

Exécuter la routine (point d'entrée) sur laquelle porte la requête

Aussi appelé: Acceptation de message

Service explicite:

Une instruction permet de servir explicitement les requêtes reçues.

e.g. En Ada: Accept get;

Service implicite:

Il n'existe pas d'instructions de service, service transparent, en général décidé à partir de propriété (contrôle implicite).

e.g. moniteur: Guide, Hybrid

En général:

Contrôle explicite ===> service explicite

Contrôle implicite ===> service implicite

Sélection sur paramètres:

Si le contrôle d'exécution d'un processus peut être exprimé en fonction de la valeur des paramètres effectifs des requêtes.

e.g. Concurrent C: serve lock (i) such that free (i)

3.2.6 Continuations

Apparaît avec l'appel asynchrone de fonctions

Un processus p appelle une fonction f1 d'un processus p1.

Il y a continuation lorsque le résultat de la fonction f1 est le résultat (sans modification) d'un appel à une fonction f2 d'un autre processus p2

Version dégradée :

cela peut être une partie du résultat

==> Dessin

Continuation explicite

Le programmeur doit spécifier lui-même que le résultat de la fonction f2 doit être retourné à p

Send o1 result o2.foo()
return (result o2.foo())

Continuation implicite

Réalisé sans programmation spécifique.

return (o2.foo())

3.2.7 Terminaison des processus

Terminaison programmée:

Définie par le programmeur: envoi de messages qui indiquent à une tâche qu'elle doit se terminer

Terminaison automatique:

Dépendances statiques:

toutes les tâches mères et les tâches soeurs sont soit terminées, soit en attente (select de Ada)

Impossible dans un modèle où les dépendances sont complètement dynamiques (objets).

Ramasse-miette de processus:

Si un processus n'est plus référencé par un autre processus (actif), il ne recevra plus de requêtes: il peut éventuellement être terminé.

==> Terminaison distribuée

- Pb des cycles.

- Pb. des messages en transit

Perspective: GC of active objets,

leases dealing with cycles

3.3 Notes: Langages d'Actor

An overview of actor languages
Proceedings of the 1986 SIGPLAN workshop on Object-oriented programming >toc
1986 , Yorktown Heights, New York, United States

http://www.math.uwaterloo.ca/~l2nie/actor.pdf

G. Agha, I. Mason, S. Smith, and C. Talcott. A Foundation for Actor Computation. Journal of Functional Programming, Vol. 7, pp.1-72, 1997.

Cadre initialement fonctionnel,

Le "become" remplace la notion d'état dans le cadre fonctionnel.

D'après:

Peter Wegner: Dimensions of O-O language Design
http://sepc.twi.tudelft.nl/~kees/in4023sheets/out/actors-3.pdf

Principes:

All actors operate in parallel

Message passing is one way, asynchronous communication

one way: how to return a value?

asynchronous: don't wait for an answer; continue processing ==> parallelism

Specify a continuation (where the returned value should go)

The continuation is another actor or the actor itself.


```
(define (Factorial( ))
 (Is-Communication (a eval (with customer =c)
                    (with number =n)) do
  (become Factorial)
   (if (NOT (= n 0))
     (then (send c 1))
     (else (let (x (new FactCust (with customer c)
                          (with number n)))
          (send Factorial (a eval (with customer x)
                    (with number n-1)))))))))

(define (FactCust (with customer =m)
                  (with number =n))
  (Is-Communication (a number k) do
                  (send m n*k)))
```

Autre version:

Avec des constructions appropriées:

"automatic generation of the appropriate customers")

(define (a Factorial (with number = n))

(if (= n 0)

(then 1)

(else (* n (call Factorial (with number n-1))))))

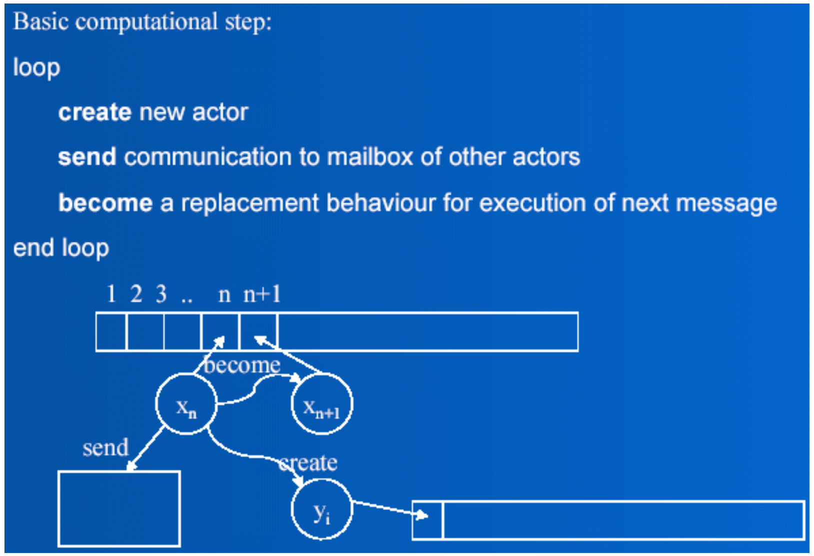

FIGURE 9 Principles of Actors

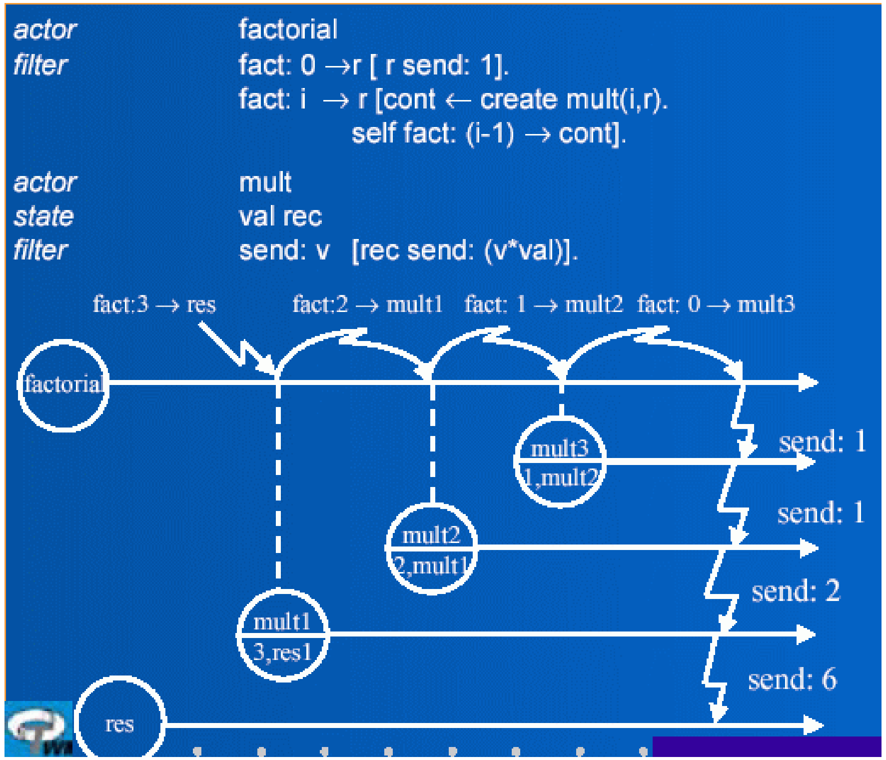

FIGURE 10 Factorial example

CHAPITRE 4 Langages Impératifs

Classiques

Ada

C S P

Occam

4.1 Ada

4.1.1 Notion de tâche

Définition d'un modèle de tâche grâce à une déclaration de type:

- Nom de tâche

- Liste de nom de procédures (points d'entrées)

Définition du corps (implémentation) du modèle:

task body

Création dynamique d'objets de type tâche:

opérateur new

Exemple:

task ressource is

end ressource

type ref_ressource is access ressource ;
ref : ref_ressource;
...
ref := new ressource;

4.1.2 Points d'entrées: procédures

Dans la partie spécification de la tâche, est définie l'en-tête des entrées

Leur corps se trouve dans le corps de la tâche.

Une entrée est pratiquement identique à une procédure:

accept **point_entrée (<P. Form.>)** *do*

<texte du point d'entrée >

end **point_entrée**

Mais *accept* n'est pas une déclaration, en fait c'est une instruction: on peut avoir plusieurs *accept* pour le même point d'entrée.

Exemple:

```
task ressource is
    entry acquerir;
    entry relacher;
end ressource;

task body resource is
begin
    accept acquerir;
    accept relacher;
    accept acquerir
    ...
```


A des instructions *accept* correspondent des instructions de communication utilisant la notation pointée:

```
type ref_ressource is access ressource ;
ref : ref_ressource;
...
ref := new ressource;

ref.acquerir;
ref.relacher;
```

4.1.3 Communications: RDV

La sémantique de l'instruction *accept* est la suivante:

Si une tâche exécute une instruction accept pour un point d'entrée donné:

accept acquerir do ... end;

Si une autre tâche exécute une instruction de communication pour ce même point d'entrée:

ref.acquerir;

alors un Rendez-vous (communication synchrone) se produit entre les deux tâches, pendant lequel :

- les paramètres effectifs sont transmis (s'il y en a)

- les instructions du point d'entrée (s'il y en a) sont exécutées.

Les deux types d'instruction (accept et communication) sont bloquants.

4.1.4 Non-déterminisme: select

On peut exprimer du non-déterminisme ainsi que des gardes et des communications non-bloquantes de la façon suivante:

```
select
    when condition =>
        accept ...
or when condition2 =>
        delay 20.0 ;
        ...
or when condition3
    terminate;
else
    ... instructions ...
end select;
```


L'instruction select peut également être utilisée du côté appelant:

Appel conditionnel d'entrée:

```
select
    t.entree;
else
     ... autres instructions
```

La sémantique étant: si la tâche *ref* est prête à accepter le RDV alors RDV sinon les autres instructions sont exécutées.

Note: sémantique pas très facile à définir (surtout en distribué): que veut dire "prête" ?

ou

```
select
    t.entree;
else
    delay x;
     ... autres instructions
```

Cette seconde possibilité se veut "temps réel".

Exemple d'utilisation:

```
select
    ressource.acquerir;
    put ("Ressource acquise");
else
     put ("Probleme: Ressource non
                         acquise");
```

ou encore

```
select
    ressource.acquerir;
    put ("Ressource acquise");
else
    delay 20.0; -- Secondes
     put ("Probleme: Ressource non
                         acquise");
```

Note la syntaxe est quelque peu bizarre: on entre dans le *else* (on commence *delay*), mais on peut en sortir pour faire la première partie du *select*.

4.1.5 Exemples:

a) Tampon borné

```
task body tampon is

begin
    loop
        select
            when [non plein] =>
                accept produire (x) do
                    ... mettre dans tampon
                end;
                ... hors RDV
            or
            when [non vide] =>
                accept consommer (x) do
                    ... prendre du tampon
                end;
                ... hors RDV
            or
                terminate;
        end select;
    end loop;
end;
```

Non-déterminisme lorsque le tampon est ni vide ni plein.

b) Exclusion mutuelle

Il suffit de passer par une unique tâche qui gère la structure à accéder en exclusion mutuelle

Le RDV assure l'exclusion mutuelle

```
task structure is
    entry ecrire ( x );
end;

task body structure is
    ... structure de données
begin
    loop
        accept ecrire (x) do
            ... acces a la structure en
            ... exclusion mutuelle
        end ecrire;
    end loop;
end structure;
```


c) Lecteurs-Rédacteurs

```
select
    when STARTWRITE'COUNT= 0 =>
        accept STARTREAD do
            nb_readers := nb_readers +1;
        end;
or
    accept STOPREAD do
            nb_readers := nb_readers - 1;
    end;
or
    when nb_readers = 0 =>
        accept STARTWRITE;
        accept STOPWRITE;
        for i in1..STARTREAD'COUNT loop
            accept STARTREAD do
            nb_readers := nb_readers +1;
        end;
    end loop;
end select;
```

COUNT : Nombre d'appels en attente sur un point d'entrée

Mais pas accès à l'ordre des demandes de RDV!

Priorité: Mixte,

Ni Lecteur, Ni Rédacteur,
Ni équitable

d) Tampon à une place

```
task case is
    entry ecrire ( x: in integer);
    entry lire ( x: out integer );
end;

task body case is
    v: integer;
begin
    loop
        accept ecrire (x) do      -- Bloquant
            v := x;
        end;
        accept lire (x) do        -- Bloquant
            x := v;
        end;
    end loop;
end;
```

Les deux opérations sont bloquantes et forcément successives.

e) Variable partagée

```
task variable_partage is
    entry ecrire ( x: in integer);
    entry lire ( x: out integer );
end;

task body variable_partage is
    v: integer := init value;
begin
    loop
        select
            accept ecrire (x)do--NonBloquant
                v := x;
            end;
        or
            accept lire (x) do --NonBloquant
                x := v;
            end;
        end select;
    end loop;
end;
```

Les opérations deviennent non-bloquantes grâce au select

Le select apporte également l'*attente non-active* si aucun point d'entrée n'est appelé

4.1.6 Bilan sur Ada

Select:

non bloquant

non déterminisme

attente non-active

terminaison

Le langage ne va pas assez loin:

Les points d'entrée sont presque des procédures, mais pas complètement (pb. de réutilisation: un processus n'est pas un ADT) Un peu objet mais un peu court.

Quelques problèmes:

Pb. de mise en oeuvre (Distribué, pooling)

Pb. RDV synchrone uniquement (deadlock)

Pb. d'expressivité, en particulier au niveau de l'expression de la synchronisation (cf. l'exemple des Lecteurs-Rédacteurs)

Pas accès à l'ordre total sur les demandes de RDV!

Symptomatique: extensions adoptées par Ada 95:

protected type

(sorte de région critique automatique)

requeue **entrée;**

requeue **entrée** *with abort*

Avoir accès aux paramètres d'un appel de point d'entrée, puis le remettre en attente

Avec abort ou non du RDV coté appelant

pragma **Prority (nb);**

Statique ou dynamique (pour temps réel)

Nombreux attributs:

T'Callable

Tâche T, vrai si T peut être appelée

T'Identity

Une valeur qui identifie la tâche

E'Caller

Pour une entrée, l'Id de la tâche appelante (dans un *accept*)

T'Terminated

Vrai si la tâche T est terminée

4.2 C S P

Ada inspiré de CSP

Plus ancien, plus formel
(volonté de pouvoir faire des preuves)

4.2.1 Forme générale: Composition //

Un programme CSP est ensemble de processus composé parallèlement:

[nom1 :: <processus1> // ...

// nomN :: <processusN>]

ou encore

[nom1 :: <processus1> // ...

// nomK (K=i,j) :: <processusK> // ...]

Dans ce cas, nomI sont des identificateurs de processus; la notation permet de construire un tableau de processus, tous construits sur le même modèle.

La sémantique de la commande // est:

- Tous les processus sont initialisés simultanément
- La commande se termine avec succès seulement quand tous ses composantes se sont terminés avec succès

Note:

La nature et le nombre des processus est défini de façon statique (diff. de Ada par exemple).

Exemple simple:
un traitement de type pipe à trois processus

[entree // traduction // sortie]

Mais attention, les communications utilisent un mécanisme spécifique de communication (pas un simple pipe)

Les processus ne sont pas toujours nommés.

4.2.2 Commande alternative

Une commande alternative a la forme suivante:

```
[ garde1 --> commandes ...
        [] -- Rectangle !!

garde1 ; <com. d'echange> --> com. ...
        []

<com. d'echange> --> com. ...
]
```

Une des gardes évaluée à vrai est sélectionnée -- correspond à un *select* Ada (voir plus loin pour la sémantique).

Le non-déterminisme est possible.

Exemple simple:

calcul du maximum (m) de deux valeurs x et y

```
[ x>= y --> m := x
        []
  x <= y --> m := y
]
```

Dans ce cas, l'instruction revient à un simple test.

4.2.3 Commande répétitive

Une commande répétitive a la forme suivante:

```
* [ garde1 --> commandes ...
        []
garde1 ; <com. d'echange> --> com. ...
        []
<com. d'echange> --> com. ...
]
```

Simplement une commande alternative répétée :

La commande répétitive est exécutée tant qu'une commande peut être sélectionnée, c'est à dire possède une garde qui s'évalue à vrai.

Exemple:

Calcul du maximum d'une fonction discrète f sur un intervalle [0:N]:

```
var int k init 0 ; -- Indice du maximum
var int j  init 1 ; -- Indice parcours [0:N]
*[ j<> N -->
    [ f(k) >= f(j) --> j:= j+1
        []
      f (k) <= f(j) --> k := j; j := j+1
    ]
]--Invariant: k= indice du max. de 0 à j-1
```

Notes: pas de test, pas de boucle

4.2.4 Commande d'échange: RDV

Deux commandes d'échange:

a) Émission

destination ! expression

b) Réception

source ? variable

Sachant que source et destination représentent des noms de processus.

Exemples de commandes d'échanges:

P1, P2, P3 -- Non de Processus
s, y, z -- variables
*P1 ! (x+y)*z -- émission, source*
P2 ? x -- reception, destinataire
P3 ! z

La sémantique d'une commande d'échange est un *RDV* entre les processus *source* (émission, depuis P2 vers P1) et *destination* (réception, depuis P1, reçoit de P2) ssi :

- source et destination concordent

- concordance de type entre la variable de réception et la valeur émise

Dans le cas où l'un ou l'autre n'est pas prêt, il y a attente:

Ces instructions sont bloquantes

4.2.5 Gardes et commandes d'échange

Les gardes pouvant comporter des commandes d'échanges, cela complique considérablement la sémantique -- par rapport à un langage où les gardes ne sont que des valeurs booléennes.

Une garde qui comporte une commande d'échange est passable si une commande correspondante (émission ou réception) est "prête".

Dans le cas mixte: valeur booléenne + Echange:

Passable si les deux sont passables

Dans le cas général, une *commande alternative* s'exécuté comme suit:

- une garde passable est sélectionnée. Si celle-ci comporte une commande d'échange, il se produit (RDV), puis la liste des commandes associées est exécutée.

- si aucune garde n'est passable:
o si com. échange: *attente*
o sinon: *erreur*

Dans le cas général, une *commande répétitive* s'exécuté comme suit:

- tant qu'il existe une garde passable:
en choisir une (non-déterminisme) et l'exécuter

- Lorsque toutes les gardes ne sont pas passables, et ne le seront plus jamais:
la commande se termine normalement

On retrouve un aspect terminaison des processus.

Remarques sur le mécanisme de RDV de CSP:

On ne peut pas déterminer statiquement quel processus va entrer en RDV avec quel autre en tel point du programme. Ceci est déterminé dynamiquement, et de façon non-déterministe.

La commande de RDV est quasi symétrique entre l'émission et la réception: ce n'est pas une qualité:

Le processus receveur doit connaître et mentionner explicitement l'émetteur.

Mauvaise réutilisation et composition des processus (e.g. un serveur).

Doit connaître les noms des clients!!

4.2.6 Exemples

a) Formatage de caractères (prod-cons)

Un processus fournie une suite de caractères. Il faut produire des lignes de 125 caractères pour les envoyer vers une imprimante.

Format:

```
[1:125] var car ligne; -- sortie ligne
var int i init 1 ; -- Indice construction
*[ var car c ;
    Entree ? c --> ligne[i] := c;
                     -- sortie si Entree est finie
                     [ i <= 124 --> i:= i+1
                     []
                       i = 125 -->
                         imprimante!ligne; i:=1
                     ]
];
[ i = 1 --> -- rien à faire, nb. car. x125
  []
  i > 1 --> *[ i <= 125 --> ligne[i] := " " ;
                             i:=i+1
             ];
             imprimante ! ligne ;
]
[ Entree // Format // Imprimante ]
```

b) Crible nombres premiers

On calcule les nombres premiers inférieurs à k

```
[[1:k'] crible ::       -- cas 1 à k'
   -- Texte du processus i
var ent p, mp ;
crible [i-1] ? p ;   -- Bloquant: attente nb. P
imprimante ! p ; -- Sortie du nb. P
   -- mp va contenir successivement tous
   -- les multiples de p
* [ var ent m ;
    crible [i-1] ? m -->
        * [ m> mp --> mp := m+p] ;
        [ m=mp --> -- Rien faire: pas P, mult.
            []
            -- P: on envoie m sur proc i+1
        m<mp --> crible [i+1] ! m
        ]
]
    //
crible [0] ::         -- cas 0
    imprimer ! 2 ;
    -- Elimination des nb. pairs
    *[ n < k --> crible [1] ! n ; n := n + 2;
    ]
    //
crible [k'+1] ::      -- cas k'+1
```



```
        *[ var ent x;
        crible [k'] ? x --> imprimante ! x
        ]
        //
    imprimante ::          -- envoi sur imprimante
    *[  ...
        crible [i] ? n --> print (n)
        ...
    ]
```

Les nombres plus grands que Racine (k) et non éliminés par les cribles inférieurs à Racine (k) sont premiers.

On a Racine (k) cribles

k' égal à la valeur entière de Racine (k)

Question: Nb. P > k' ???????

4.2.7 Conclusions sur CSP

Deux problèmes importants:

- **RDV non statique**
- **RDV symétrique, le receveur devant connaître l'émetteur**

Mais CSP avait essentiellement comme but d'être capable de faire des preuves sur les langages concurrents et parallèles.

Successeur de CSP: Unity

Note:

ADA:
ref. produire --- Pt Entré ---> Accept produire

CSP: P2!v ---- Nom Process ----> P1?r

Pi: c!v ------ Canal --------> c?r

(mobility of name, "labile")

ProActive:
ro.foo (); --- method name ---> serve (foo)

The target is: **an Activity + an Object**

4.3 Occam

1982, "Occam's razor" principe du philosophe minimaliste,

Conception postérieure et inspiré de CSP,

Machine cible et liée au: Transputer

Pas de création dynamique de processus

"OCCAM 2 reference manual", INMOS ltd, Prentice-Hall International, ISBN 0-13-629312-3

4.3.1 Compositions

a) **Séquentielle**

b) **Parallèle**

c) **Conditionnelle**

d) **Itérative**

4.3.2 Communications: canaux

4.3.3 Com. alternative: ALT

4.3.4 Gardes

4.3.5 Définition de tâches

4.3.6 Exemples

a) Tampon borné

b) Clients - Serveurs

c) Calcul de nombres premiers