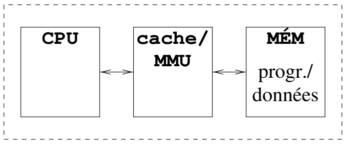
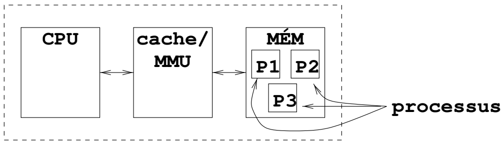
Programmation concurrente

- Dans les programmes concurrents **plusieurs agents collaborent pour effectuer un calcul unique**
- Déf: un **processeur** c'est une unité de traitement (réèle) autonome



- Déf: un **processus** c'est un processeur **virtuel** simulé par un processeur



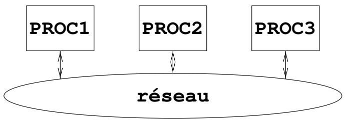
- Un **changement de contexte** ("context-switch") fait passer le contrôle d'un processus au suivant (à intervalles réguliers ou lorsqu'un processus bloquerait)

Processus lourds et légers

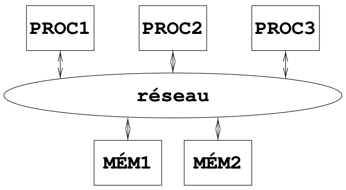
- Déf: un **processus lourd** c'est un processus encapsulé dans son propre espace d'adressage mémoire ("process" de UNIX)
  - Avantage: **sécurité** (impossible de lire/modifier la mémoire d'un autre processus, et si un processus plante, ça ne fait pas planter le processeur)
  - Désavantage: **lent** (temps de création élevé (`fork`) et les tables de pagination doivent être changées à chaque "context-switch")
- Déf: un **processus léger** c'est un processus "virtuel" simulé par un processus lourd ("thread")
  - Avantage: **espace d'adressage commun** à tous les threads (accès facile aux structures de données) et "context-switch" rapide entre threads
  - Désavantage: **aucune sécurité** entre threads et limité à un seul processus lourd

Architectures parallèles (1)

- Déf: un **multi-ordinateur** c'est un système constitué de plusieurs ordinateurs reliés en réseau (réseau JSP)



- Le terme **mémoire distribuée** est aussi utilisé
- Déf: un **multi-processeur** c'est un ordinateur constitué de plusieurs processeurs qui partagent une même mémoire (Dual-Pentium, Sun-E10000, Cray-T3E)

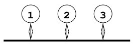


- Le terme **mémoire partagée** est aussi utilisé

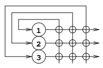
Architectures parallèles (2)

- Plusieurs organisations sont possibles pour les réseaux, chacun a un coût et un débit

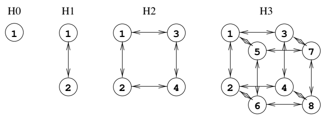
1. Bus: coût=$O(n)$, débit faible (goulot d'étranglement p.e. Ethernet)



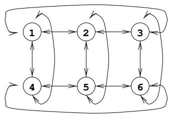
2. Crossbar: coût=$O(n * n)$, débit très élevé (lien entre chaque paire, p.e. SUN-E10000)



3. Hypercube: coût=$O(n * log(n))$, débit moyen (p.e. Connection Machine)



4. Grille: hypercube généralisé à D dimensions, coût=$O(n * D)$, débit moyen (p.e. Cray-T3E)



Objectifs et problèmes

- Objectifs de la programmation concurrente

  1. **Accélérer** l'exécution d'une application par calcul parallèle (p.e. prévision météo)

  2. Implanter une application **physiquement distribuée** (p.e. guichets automatiques, internet (email, web), etc.)

  3. **Meilleure organisation** d'un programme par décomposition en processus (p.e. systèmes réactifs, interfaces usager)

- Problèmes fondamentaux

  1. **Partitionnement** du programme en processus (tâche associée à chaque processus, organisation, nombre fixe/variable)

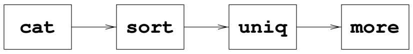
  2. **Communication** entre processus (échange des données à traiter)

  3. **Synchronisation** entre processus (échange des informations de contrôle)

Partitionnement

- Le meilleur partitionnement dépend de l'application, mais il y a deux organisations répandues:

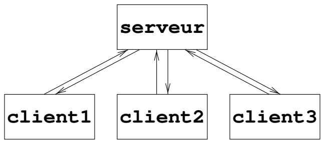
  1. **Pipeline**: les processus se suivent à la queue leu leu et traitent la sortie du processus précédent

     $ cat bottin1 bottin2 | sort | uniq | more

     Autre exemple: compilateur C (`cpp|ccom|as`)

  2. **Client-serveur**: les processus clients envoient des requêtes au processus serveur (un processus central qui offre un certain service, p.e. accès à une BD)

     Le serveur est souvent lui-même un programme concurrent

Création de processus (1)

- Sous UNIX la fonction `fork()` est utilisée pour créer un **clone du processus lourd courant**

- Exemple

```
#include <unistd.h>

void main ()
{ pid_t p1, p2; /* ``Process identifiers'' */
  int sp1, sp2; /* statut des processus */
  int i;

  p1 = fork (); /* retourne 0 dans le clone */
  if (p1 == 0)
    { for (i=0; i<10000000; i++)
        if (i%1000000 == 0) { printf ("1"); fflush (stdout); }
      exit (12); /* terminer le processus p1 */
    }

  p2 = fork (); /* retourne 0 dans le clone */
  if (p2 == 0)
    { for (i=0; i<10000000; i++)
        if (i%1000000 == 0) { printf ("2"); fflush (stdout); }
      exit (34); /* terminer le processus p2 */
    }

  waitpid (p1, &sp1, 0); /* attendre terminaison de p1 */
  waitpid (p2, &sp2, 0); /* attendre terminaison de p2 */

  printf ("\n%d %d\n", WEXITSTATUS(sp1), WEXITSTATUS(sp2));

  /* imprime: 12121212121221121221
              12 34
   */
}
```

- Note: phénomènes de **non-déterminisme** et de "**course**"

Création de processus (2)

- Les processus légers sont moins standards

- La librairie C "POSIX threads" est assez répandue sous UNIX

- Les langages concurrents intègrent des primitives de programmation concurrente à même le langage (p.e. la classe "`Thread`" de Java, le type "`task`" de Ada)

- Certaines implantations de Scheme offrent aussi des primitives de programmation concurrente, par exemple Gambit supporte les primitives suivantes:

  1. (`make-thread` *procédure*) : crée et retourne un "thread" qui appelera la *procédure* sans paramètre

  2. (`thread-start!` *thread*) : démarre l'appel de la *procédure*

  3. (`thread-join!` *thread*) : attends que l'appel de la *procédure* soit complété et retourne le résultat de la *procédure*

Création de processus (3)

- Exemple

```
(define compteur 0)

(define incrementer
  (lambda (n)
    (if (> n 0)
        (begin
          (set! compteur (+ compteur 1))
          (incrementer (- n 1))))))

(define go
  (lambda (nom)
    (begin
      (display "début ")
      (display nom)
      (newline)
      (incrementer 1000000)
      (display "fin ")
      (display nom)
      (newline))))

(define thread1
  (make-thread (lambda () (go "thread1"))))

(define thread2
  (make-thread (lambda () (go "thread2"))))

(thread-start! thread1)
(thread-start! thread2)

(thread-join! thread1)
(thread-join! thread2)

(write compteur)

; sortie:
;
;   début thread1
;   début thread2
;   fin thread1
;   fin thread2
;   1795772       <----- pas égal à 2000000
```


Modèle de concurrence (1)

- Comment expliquer que le résultat n'est pas 2000000?
- Le modèle suivant d'exécution concurrente est utile:
  1. L'exécution de chaque processus est décomposé en **opérations élémentaires** indivisibles
  2. On suppose que deux opérations O_1 et O_2 de deux processus s'exécutent dans l'ordre O_1-O_2 ou O_2-O_1 (**pas en même temps**)
  3. L'exécution globale du programme est une **permutation de toutes les opérations** qui respecte l'ordre séquentiel imposé par chaque processus
- P_1=`AB` et P_2=`xyz`, 10 exécutions possibles:

```
ABxyz  AxByz  AxyBz  AxyzB  xAByz
xAyBz  xAyzB  xyABz  xyAzB  xyzAB
```

- Pour 2 processus avec n et m opérations, il y a un nombre total d'exécutions possibles égal à $(n+m)!/(n! * m!)$

Modèle de concurrence (2)

- L'énoncé "`(set! compteur (+ compteur 1))`" est en fait décomposé en trois opérations élémentaires:

  L) T <- compteur

  A) T <- T+1

  E) compteur <- T

- Si P_1=`LAE` et P_2=`lae` il y a 20 exécutions possibles:

```
LAElae   compteur=2
laeLAE   compteur=2
LAlEae   compteur=1
LAlaeE   compteur=1 (idem pour autres exécutions)
```

- Pour que `compteur` soit mis-à-jour correctement, il faut s'assurer que l'autre processus n'accèdera pas au compteur en même temps (i.e. l'incrémentation est une **section critique**)
- Il faut donc **synchroniser** les 2 processus pour qu'il y ait une **exclusion mutuelle** des processus dans la section critique (au plus 1 processus qui exécute la section critique)

Synchronisation (1)

- Les **sémaphores** permettent d'implanter des sections critiques
- Une sémaphore S c'est une variable entière qui supporte 2 opérations indivisibles
  1. `acquérir(S)`: attendre que S soit positif puis décrémenter S
  2. `céder(S)`: incrémenter S
- On associe une sémaphore à la section critique et on l'initialise au nombre maximal de processus permis dans la section critique (donc normalement S=1, **sémaphore binaire**)

```
  P1                        P2
...                       ...
acquérir(S)               acquérir(S)
compteur <- compteur+1    compteur <- compteur+1
céder(S)                  céder(S)
...                       ...
```

Synchronisation (2)

- Gambit supporte les sémaphores binaires avec les primitives de "mutex" suivantes

  1. `(make-mutex)` : créer une sémaphore binaire libre

  2. `(mutex-lock!` *mutex*`)` : acquérir la sémaphore

  3. `(mutex-unlock!` *mutex*`)` : céder la sémaphore

- Exemple

```
(define s (make-mutex)) ; créer une sémaphore binaire

(define incrementer
  (lambda (n)
    (if (> n 0)
        (begin
          (mutex-lock! s)
          (set! compteur (+ compteur 1))
          (mutex-unlock! s)
          (incrementer (- n 1))))))
```


Synchronisation (3)

- Le programme suivant ne marche pas

```
(define s 1)

(define acquerir
  (lambda ()
    (if (= s 1)
        (set! s 0)
        (acquerir))))

(define ceder
  (lambda ()
    (set! s 1)))

(define incrementer
  (lambda (n)
    (if (> n 0)
        (begin
          (acquerir)
          (set! compteur (+ compteur 1))
          (ceder)
          (incrementer (- n 1))))))
```

- Le problème c'est que le test "`(= s 1)`" et l'affectation "`(set! s 0)`" ne sont pas **atomiques** (il y a un risque qu'entre le test et l'affectation l'autre processus execute son propre test "`(= s 1)`")

- Même si l'acquisition était atomique cette **attente active** risque de gaspiller du temps (mieux vaut suspendre le processus si la section critique est longue ou achalandée)

Synchronisation (4)

- Si mal employées, les opérations de synchronisation peuvent causer un **blocage du programme**

- Les deux cas possibles sont le "**deadlock**" (étreinte fatale) et le "**livelock**"

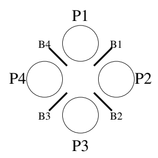
- Exemple de deadlock: accès à 2 ressources (fichiers)

```
   P1                            P2
...                           ...
acquérir(S1)                  acquérir(S2)
acquérir(S2)                  acquérir(S1)
lire "f1" pour créer "f2"     lire "f2" pour créer "f1"
céder(S1)                     céder(S2)
céder(S2)                     céder(S1)
...                           ...
```

- Si les premières acquisitions se font en même temps, ni P1 ni P2 ne pourront faire de progrès

Synchronisation (5)

- Exemple classique: N philosophes chinois qui mangent leur repas en partageant N baguettes



- Pour manger il faut 2 baguettes

- Les philosophes (processus) entrent en compétition pour les baguettes (ressources) avec leurs voisins

```
philosophe Pi:
pour toujours faire
  penser
  acquérir(Bi)
  acquérir(Bi-1)      note: si i=1, i-1 = N
  manger
  céder(Bi-1)
  céder(Bi)
```

- Mène à un deadlock si tous les philosophes commencent au même moment

Synchronisation (6)

- Si les philosophes font plutôt:

```
philosophe Pi:
pour toujours faire
  penser
  encore:
  acquérir(Bi)
  si Bi-1 est prise { céder(Bi), goto encore }
  acquérir(Bi-1) /* atomique avec ‘‘si Bi-1 ...’’ */
  manger
  céder(Bi-1)
  céder(Bi)
```

  il n'y a pas de deadlock, mais un **livelock** est possible:

```
acquérir(B1)
céder(B1)
acquérir(B1)
céder(B1)
...
```

- Dans un livelock un processus ne fait pas de progrès utile bien qu'il continue à exécuter du code

Synchronisation (7)

- Les deadlocks sont évités si tous les processus font l'acquisition des sémaphores **dans le même ordre**

- Exemple: accès à 2 fichiers

```
  P1                            P2
...                           ...
acquérir(S1)                  acquérir(S1)
acquérir(S2)                  acquérir(S2)
lire "f1" pour creer "f2"     lire "f2" pour creer "f1"
céder(S1)                     céder(S1)
céder(S2)                     céder(S2)
...                           ...
```


Synchronisation (8)

- Les "**variables de condition**" permettent des synchronisations plus générales que les sémaphores

- Une variable de condition est un objet qui correspond à un ensemble de processus bloqués en attente qu'une **certaine condition soit vraie de l'état du programme** (existence d'un objet particulier dans une structure de donnée, occurrence d'un évennement particulier, etc)

- Par exemple: supposons une méthode "`retrait(`n`)`" pour un compte banquaire qui doit attendre tant que le solde est plus petit que n, et qui ensuite débite le compte de n (d'autres processus peuvent faire des "`dépot(`n`)`" concurremment):

```
dépot(n):
  solde <- solde + n

retrait(n):
  attendre jusqu'à ce que (solde >= n)
  solde <- solde - n
```

- Tel quel cela ne fonctionne pas car le test de la condition (`solde>=`n) et la modification de `solde` doivent être atomiques

Synchronisation (9)

- Pour obtenir l'exclusion mutuelle on peut utiliser une sémaphore binaire `S`:

```
dépot(n):
  acquérir(S)
  solde <- solde + n
  céder(S)

retrait(n):
  acquérir(S)
  si (solde >= n)
    solde <- solde - n
    céder(S)
  sinon
    céder(S)
    retrait(n)   note: appel terminal = boucle
```

- Malheureusement cette attente active gaspille du temps

Synchronisation (10)

- Il faut utiliser une variable de condition "C" pour bloquer les processus lorsque la condition (solde>=n) est fausse:

```
dépot(n):
  acquérir(S)
  solde <- solde + n
  céder(S)
  réveiller(C)   ***nouveau***

retrait(n):
  acquérir(S)
  si (solde >= n)
    solde <- solde - n
    céder(S)
  sinon
    céder_et_bloquer(S,C)   ***nouveau***
    retrait(n)
```

- L'opération `céder_et_bloquer(S,C)` est **atomique**

- On doit toujours utiliser une variable de condition conjointement avec une sémaphore

Synchronisation (11)

- Gambit supporte les primitives suivantes

1. `(make-condition-variable)` : créer une variable de condition vide
2. `(condition-variable-signal!` *varcond*`)` : débloquer un des processus attachés à *varcond* (le premier qui a bloqué)
3. `(mutex-unlock!` *mutex varcond*`)` : céder le *mutex* et bloquer le processus sur *varcond*

```
(define solde 0)
(define s (make-mutex))
(define vc (make-condition-variable))

(define depot
  (lambda (n)
    (begin
      (mutex-lock! s)
      (set! solde (+ solde n))
      (mutex-unlock! s)
      (condition-variable-signal! vc))))

(define retrait
  (lambda (n)
    (begin
      (mutex-lock! s)
      (if (>= solde n)
        (begin
          (set! solde (- solde n))
          (mutex-unlock! s))
        (begin
          (mutex-unlock! s vc) ; note: 2 paramètres
          (retrait n))))))
```


Synchronisation structurée (1)

- Les sémaphores et variables de condition offrent des mécanismes de **bas niveau** (non-structurés) pour synchroniser des processus, et il est facile d'obtenir des bugs de logique, par exemple:

```
acquérir(S)
...
acquérir(S) /* on aurait du faire: céder(S) */
            /* on obtient un deadlock */
```

- Pour cette raison, plusieurs langages concurrents possèdent des **énoncés de synchronisation structurés**

- **barrière**: les processus qui arrivent à la barrière attendent que **tous les autres processus** soient arrivés avant de continuer
  - Utile pour séparer les phases d'un calcul parallèle
  - Exemple: jeu de la vie parallèle

```
chaque case de la grille (x,y) est un processus qui fait:
  répéter pour toujours:
     T = f( état, états_voisins )
     barrière()
     état = T
     barrière()
```


Synchronisation structurée (2)

- **moniteur**: un moniteur c'est un objet (au sens OO) avec la contrainte qu'**au maximum un processus à la fois peut exécuter une de ses méthodes** (toutes ses méthodes sont donc mutuellement exclusives)

- Le compilateur implante les moniteurs à l'aide de sémaphores (une sémaphore par objet), mais c'est invisible au programmeur

- Les moniteurs sont utilisés par Java, mais les méthodes mutuellement exclusives sont **indiquées explicitement** (avec le mot clé `synchronized`)

- Exemple: gestion d'un compte banquaire

```
class Compte
 {
  private int solde;
  Compte() { solde = 0; }
  public int lire () { return solde; }
  public synchronized void depot (int n) { solde+=n; }
  public synchronized void retrait (int n) { solde-=n; }
 }
```

Synchronisation structurée (3)

- Exemple (suite)

```
class Proc extends Thread
 {
  private Compte c;
  Proc(Compte compte) { c = compte; }
  public void run()
    { for (int i=0; i<100000; i++) c.depot(1); }
 }

class Banque
 {
  public static void main (String[] args)
     throws InterruptedException
   {
    Compte compte = new Compte();

    Proc p1 = new Proc(compte); // créer processus 1
    Proc p2 = new Proc(compte); // créer processus 2

    p1.start(); // démarrer processus 1
    p2.start(); // démarrer processus 2

    p1.join();  // attendre fin du processus 1
    p2.join();  // attendre fin du processus 2

    System.out.println (compte.lire());
  }
 }
```


Synchronisation structurée (4)

- **rendez-vous**: un rendez-vous c'est une partie de programme ou **2 processus se fusionnent en 1** (temporairement pour cette partie)
- C'est en quelque sorte l'inverse d'une exclusion mutuelle car le rendez-vous est seulement exécuté **si les deux processus sont arrivés au rendez-vous** (le premier arrivé va attendre l'autre)
- En plus de servir de point de synchronisation, le rendez-vous permet d'**échanger des données** entre les 2 processus
- En Ada: le processus P1 déclare qu'il est **prêt à accepter un rendez-vous** et le processus P2 fait **un appel de ce rendez-vous** (possiblement avec des paramètres) pour déclencher le rendez-vous

Synchronisation structurée (5)

- Syntaxe (déclaration d'un rendez-vous):

`accept` ⟨*nom*⟩`(` ⟨*paramètres_formels*⟩ `) do`
  ⟨*énoncés*⟩
`end;`

- Syntaxe (appel d'un rendez-vous):

⟨*processus*⟩`.`⟨*nom*⟩`(` ⟨*paramètres_actuels*⟩ `);`

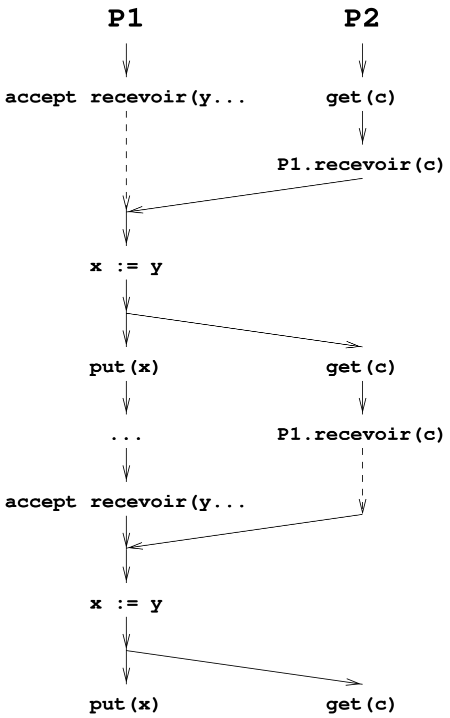
- Exemple: producteur/consommateur

```
task body P1 is  -- le consommateur
  x : character;
begin
  loop
    accept recevoir( y : in character ) do
      x := y;
    end;
    put(x);  -- écrire à l'écran
    ...;     -- autre traitement
  end loop;
end P1;

task body P2 is  -- le producteur
  c : character;
begin
  loop
    get(c);          -- lire du clavier
    P1.recevoir(c);  -- déclencher rendez-vous
  end loop;
end P2;
```


Synchronisation structurée (6)

- Exécution de l'exemple:



Synchronisation structurée (7)

- Le mécanisme de rendez-vous est capable de simuler les sémaphores
- Idée: chaque sémaphore est un processus qui gère l'état de la sémaphore

```
task body S is  -- le processus ‘‘sémaphore’’
begin
  loop
    accept acquerir do
      -- rien
    end;
    accept ceder do
      -- rien
    end;
  end loop;
end S;

procedure incrementer_compteur is
begin
  S.acquerir;
  compteur := compteur+1;
  S.ceder;
end incrementer_compteur;

task body P1 is
begin
  incrementer_compteur;
end P1;

task body P2 is
begin
  incrementer_compteur;
end P2;
```


Programmation logique

- **Prolog** est le langage principal qui offre ce style
- Inventé en 1972 par Alain Colmerauer
- Applications principales:
  - Traitement de la langue naturelle
  - Traitement symbolique et I.A.
- Particularités:
  - Unification et filtrage ("pattern matching")
  - Retour arrière ("backtracking")
  - Interactif
  - Déclaratif (un programme exprime des relations entre objets)

Relations (1)

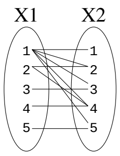
- Déf: une **relation binaire** sur les ensembles X_1 et X_2 est un ensemble de couples de la forme $\langle x_1,x_2 \rangle$, où x_1 est élément de X_1 et x_2 est élément de X_2
- Exemple

soit X1 = {1..5} et X2 = {1..5}

relation R sur X1 et X2: "x1 est un facteur de x2"

R = { <1,1>, <1,2>, <1,3>, <1,4>, <1,5>,
<2,2>, <2,4>, <3,3>, <4,4>, <5,5> }

X1	X2
1	1
1	2
1	3
1	4
1	5
2	2
2	4
3	3
4	4
5	5



- Déf: une **relation *N*-aire** sur les ensembles $X_1 \ldots X_n$ est un ensemble de tuples de la forme $\langle x_1,x_2,\ldots,x_n \rangle$, où x_i est élément de X_i
- Exemple: relation sur les 3 ensembles "Nom", "Code" et "Note"

```
Nom          Code          Note
"Luc Dubuc"  "DUBL123456"  76
"Julie Roy"  "ROYJ123456"  57
"Jean Coutu" "COUJ123456"  89
```


Relations (2)

- Déf: une **fonction** de X_1 (domaine) vers X_2 (l'image) est une relation binaire qui met en relation **au plus un élément** de X_2 avec chaque élément de X_1
- Exemple

```
relation: age de x1 est x2      relation: x1 plus vieux que x2
est une fonction                n'est pas une fonction

X1           X2                 X1           X2
"Luc Dubuc"  45                 "Jean Coutu" "Luc Dubuc"
"Julie Roy"  18                 "Jean Coutu" "Julie Roy"
"Jean Coutu" 60                 "Luc Dubuc"  "Julie Roy"
```

- Les langages fonctionnels permettent de manipuler aisément les relations binaires qui sont des fonctions
- Les langages logiques permettent de manipuler **n'importe quelles relations**

Relations (3)

- Avantage: permet de faire des calculs "dans les 2 sens" (passer de X_1 à X_2, ou l'inverse)

- Exemple 1

```
relation: le genre de x1 est x2

X1            X2
"Luc Dubuc"   homme
"Julie Roy"   femme
"Jean Coutu"  homme
```

- En programmation fonctionnelle il serait naturel d'implanter une fonction "`genre`" qui retourne le genre étant donné le nom

- Cela permettrait de répondre facilement à la question: **quel est le genre de "Julie Roy"?**

- Cependant cette fonction ne permet pas de répondre à la question: **quels sont tous les hommes?**

- En programmation logique il sera tout aussi facile de spécifier x_2 pour obtenir le ou les x_1 qui sont en relation, que l'inverse

Relations (4)

- Exemple 2: relation ternaire de concaténation de chaînes

```
relation: la concaténation de x1 et x2 donne x3

X1     X2     X3
""     ""     ""
""     "a"    "a"
"a"    ""     "a"
""     "ab"   "ab"
"a"    "b"    "ab"
"ab"   ""     "ab"
""     "abc"  "abc"
"a"    "bc"   "abc"
"ab"   "c"    "abc"
"abc"  ""     "abc"
.
.  et ainsi de suite pour toutes les chaines
.
```

- Quelle est la concaténation de `"a"` et `"b"`?

- Qu'est-ce qui, concaténé à `"a"` donne `"ab"`?

- Quelles sont les 2 chaînes *x* et *y* qui lorsqu'on les concatènes donne `"ab"`?

Prolog (1)

- Programme = ensemble de relations + **requête** qu'une relation existe

- 2 façons de spécifier des relations:

  1. **fait**: indique un tuple particulier dans la relation
  2. **règle**: indique comment déduire qu'un tuple fait partie d'une relation

- Syntaxe de base de Prolog:

  ⟨*variable*⟩ ::= identificateur débutant avec majuscule

  ⟨*atome*⟩ ::= identificateur débutant avec minuscule

  ⟨*terme*⟩ ::= ⟨*nombre*⟩
  ⟨*terme*⟩ ::= ⟨*variable*⟩
  ⟨*terme*⟩ ::= ⟨*atome*⟩
  ⟨*terme*⟩ ::= ⟨*atome*⟩ (⟨*liste_termes*⟩)

  ⟨*liste_termes*⟩ ::= ⟨*terme*⟩ { , ⟨*terme*⟩ }

  ⟨*fait*⟩ ::= ⟨*terme*⟩ .

  ⟨*règle*⟩ ::= ⟨*terme*⟩ :- ⟨*liste_termes*⟩ .

  ⟨*requête*⟩ ::= ⟨*liste_termes*⟩ .

Prolog (2)

- Les ⟨*atome*⟩ sont utilisés comme les symboles en Scheme (donc sont des données) et sont aussi utilisés pour nommer les relations:

  ⟨*atome*⟩ (⟨*liste_termes*⟩)

  on dit que l'atome est ici le **foncteur** du terme et entre parenthèses on a ses **arguments**

- Exemple: `genre(luc,homme)`

- Une relation peut s'exprimer en énumérant les tuples qu'elle contient comme des **faits**

- Exemple: soit X_1 = {`luc`,`julie`,`jean`} et X_2 = {`homme`,`femme`} et la relation binaire "`genre`": le genre de x_1 est x_2

  Cette relation s'exprime avec trois faits:

```
genre(luc,homme).
genre(julie,femme).
genre(jean,homme).
```

Prolog (3)

- Les règles permettent de définir des relations à partir d'autres relations:

  ⟨*terme*⟩ `:-` ⟨*liste_termes*⟩ `.`

  c'est ce qu'on appèle une **clause de Horn** qui est divisée en sa **tête** et ses **conditions**

- Exemple 1:

```
male(X) :- genre(X,homme).
```

  qu'on lis: `male(X)` si `genre(X,homme)`

- Traduction logique: pour tout `X` tel que la relation `genre(X,homme)` existe, la relation `male(X)` existe

- Exemple 2:

```
hermaphrodite(X) :- genre(X,homme), genre(X,femme).
```

  qu'on lis: `hermaphrodite(X)` si `genre(X,homme)` et `genre(X,femme)`

- Traduction logique: pour tout `X` tel que la relation `genre(X,homme)` existe et `genre(X,femme)` existe, la relation `hermaphrodite(X)` existe

Prolog (4)

- Exemple 3: "mariage possible entre *x* et *y*"

  `mp(X,Y)` si `X` et `Y` sont de sexe opposé

```
mp(X,Y) :- genre(X,homme), genre(Y,femme).
mp(X,Y) :- genre(X,femme), genre(Y,homme).
```

- Traduction logique:

  1. pour tout `X` et `Y` tel que la relation `genre(X,homme)` existe et `genre(Y,femme)` existe, la relation `mp(X,Y)` existe
  2. pour tout `X` et `Y` tel que la relation `genre(X,femme)` existe et `genre(X,homme)` existe, la relation `mp(X,Y)` existe

- Relation `mp(X,Y)` sous forme tabulaire:

X	Y
luc	julie
jean	julie
julie	luc
julie	jean

- **conjonction** des termes d'une clause de Horn

- **disjonction** des clauses de Horn séparées

Prolog (5)

- L'interprète Prolog s'utilise interactivement:

```
% prolog
SICStus 2.1 #9: Thu Dec 11 10:46:15 EST 1997
| ?- [prog].              <-- chargement de "prog.pl"
yes
| ?-                      <-- attends une requête
```

- Exécution d'un programme Prolog = dérivation d'une **preuve** par le système que la relation indiquée dans la requête existe

- Si on suppose que `"prog.pl"` contient:

```
genre(luc,homme).
genre(julie,femme).
genre(jean,homme).

male(X) :- genre(X,homme).

hermaphrodite(X) :- genre(X,homme), genre(X,femme).

mp(X,Y) :- genre(X,homme), genre(Y,femme).
mp(X,Y) :- genre(X,femme), genre(Y,homme).
```

  voici ce que donnera certaines requêtes:

```
| ?- genre(luc,homme).
yes
| ?- genre(julie,homme).
no
| ?- genre(martin,homme).
no
```


Prolog (6)

- On peut également spécifier des variables dans des requêtes (**requêtes existentielles**)

- Exemple: `| ?- genre(X,homme).`

  Traduction logique: est-ce qu'il existe un `X` tel que la relation `genre(X,homme)` existe?

- L'interprète Prolog va chercher une solution à cette requête (et toutes les autres solutions si demandé par l'usager en entrant ";")

```
| ?- genre(X,homme).
X = luc ? ;
X = jean ? ;
no
| ?- genre(luc,X).
X = homme ? ;
no
| ?- mp(X,Y).
X = luc, Y = julie ? ;
X = jean, Y = julie ? ;
X = julie, Y = luc ? ;
X = julie, Y = jean ? ;
no
| ?- mp(X,Y), genre(Y,femme).
X = luc, Y = julie ? ;
X = jean, Y = julie ? ;
no
| ?- mp(X,X).
no
```

Modèle d'exécution de Prolog

- Prolog tente de trouver une **preuve** que la requête est une relation qui existe à l'aide de la procédure suivante:

  prouver(*requête* **)**:

  1. Pour chaque **fait** et **tête de règle** qui concorde avec la *requête*:
     - Si un **fait**: on a trouvé une preuve
     - Si une **tête de règle**:
       * Pour chaque *condition* de cette règle: **prouver(** *condition* **)**
       * on a trouvé une preuve
  2. pas trouvé de preuve

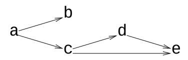
- Les **faits** et **règles** sont examinés dans l'ordre d'apparition dans le programme

- Si une **sous-preuve** échoue, la recherche revient sur ses pas (**retour-arrière**) pour tenter la prochaine alternative de preuve

Arbres de preuve (1)

- À toute preuve complète correspond un **arbre de preuve** qui indique chaque sous-preuve effectuée pour prouver la requête

- Exemple: représentation du graphe




```
lien(a,b).  lien(a,c).  lien(c,d).  lien(c,e).  lien(d,e).

chemin(Z,Z).
chemin(X,Y) :- lien(X,I), chemin(I,Y).
```

- Requête: `chemin(a,b)`

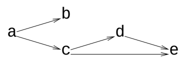
  prouver(`chemin(a,b)` **)**:
  1 essayer `chemin(X,Y) :- lien(X,I), chemin(I,Y).`
  1.1 **prouver(** `lien(X,I)` **)** avec `X=a`:
  1.1.1 essayer `lien(a,b)`, concorde avec `I=b` (**fini 1.1**)
  1.2 **prouver(** `chemin(I,Y)` **)** avec `I=b` et `Y=b`:
  1.2.1 essayer `chemin(Z,Z)`, concorde avec `Z=b` (**fini 1.2**)
  1.3 **fini**

  Arbre de preuve:

```
          chemin(a,b)
            /   \
     lien(a,b)  chemin(b,b)
```


Arbres de preuve (2)

- Requête: `chemin(a,c)`




```
lien(a,b).  lien(a,c).  lien(c,d).  lien(c,e).  lien(d,e).

chemin(Z,Z).
chemin(X,Y) :- lien(X,I), chemin(I,Y).
```

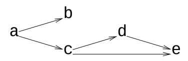
prouver(`chemin(a,c)` **)**:
1 essayer `chemin(X,Y) :- lien(X,I), chemin(I,Y).`
1.1 **prouver(** `lien(X,I)` **)** avec `X=a`:
1.1.1 essayer `lien(a,b)`, concorde avec `I=b` (**fini 1.1**)
1.2 **prouver(** `chemin(I,Y)` **)** avec `I=b` et `Y=c`:
1.2.1 essayer `chemin(X',Y') :- lien(X',I'), chemin(I',Y').`
1.2.1.1 **prouver(** `lien(X',I')` **)** avec `X'=b`:
1.2.1.1.1 **échec de 1.2.1.1** (continuer après 1.2.1)
1.2.2 **échec de 1.2** (continuer après 1.1.1)
1.1.2 essayer `lien(a,c)`, concorde avec `I=c` (**fini 1.1**)
1.2 **prouver(** `chemin(I,Y)` **)** avec `I=c` et `Y=c`:
1.2.1 essayer `chemin(Z,Z)` concorde avec `Z=c` (**fini 1.2**)
1.3 **fini**

Arbre de preuve:

```
          chemin(a,c)
            /   \
     lien(a,c)  chemin(c,c)
```


Arbres de preuve (3)

- Requête: `chemin(a,e)`




```
lien(a,b).  lien(a,c).  lien(c,d).  lien(c,e).  lien(d,e).

chemin(Z,Z).
chemin(X,Y) :- lien(X,I), chemin(I,Y).
```

Construction de l'arbre de preuve:

```
début de la recherche:

               chemin(a,e)
                 /   \
        lien(a,I)     chemin(I,e)
           I=b          /   \
                 lien(I,I') chemin(I',e)
                  *ÉCHEC*

après retour-arrière:

               chemin(a,e)
                 /   \
        lien(a,I)     chemin(I,e)
           I=c          /   \
                 lien(I,I') chemin(I',e)
                   I'=d       /   \
                     lien(I',I'') chemin(I'',e)
                       I''=e        *FINI*
```

Unification (1)

- Prolog utilise l'**unification** pour tester si deux termes sont conformes
- Déf: un **terme clos** c'est un terme qui ne contient pas de variable
- Exemple: `allo` et `f(f(1))` mais pas `X` et `f(1,Y)`
- Un terme non-clos T désigne un ensemble **infini** de termes clos (les **instances de T**) obtenus en substituant les variables dans T par tous les termes clos possible
- Exemple

```
    f(a,X)      f(Y,Y)          f(f(a,Z),f(Z,b))

I   f(a,0)      f(0,0)          f(f(a,0),f(0,b))
N   f(a,1)      f(1,1)          f(f(a,1),f(1,b))
S   f(a,2)      f(2,2)          f(f(a,2),f(2,b))
T    ...         ...             ...
A   f(a,a)      f(a,a)          f(f(a,a),f(a,b))
N   f(a,b)      f(b,b)          f(f(a,b),f(b,b))
C    ...         ...             ...
E   f(a,f(a,a)) f(f(a,a),f(a,a)) f(f(a,f(a,a)),f(f(a,a),b))
S    ...         ...             ...
```


Unification (2)

- Déf: une **substitution** c'est un ensemble de couples variable/terme, p.e. {`A=1,B=f(a,a)`}
- Déf: un **unificateur** de deux termes est une substitution qui rend ces deux termes égaux
- Exemple 1: {`X=a,Y=a`} est un unificateur de `f(X,Y)` et `f(Y,X)`
- Exemple 2: {`X=Y`} est un unificateur de `f(X,Y)` et `f(Y,X)`
- Déf: l'**unificateur le plus général** des termes T1 et T2 c'est l'unificateur qui génère le terme T (clos ou non-clos) qui désigne toutes les instances communes de T1 et T2
- {`X=a,Y=a`} est l'U.P.G. de `f(a,X)` et `f(Y,Y)`
- Pas U.P.G. pour `f(Y,Y)` et `f(f(a,Z),f(Z,b))`
- {`A=B,B=C,C=D`} est l'U.P.G. de `g(A,A,B)` et `g(C,D,D)`

Unification (3)

- Après l'unification de deux termes, l'U.P.G. généré est **conservé pour le reste de l'exécution**
- Un retour arrière peut cependant **défaire** cette unification
- Exemple

```
egal(X,X).

f(X,Y,Z) :- egal(X,Y), egal(Z,1).  % règle 1
f(X,Y,Z) :- egal(Y,Z), egal(Z,2).  % règle 2
```

La requête `f(3,A,A)` va essayer la règle

1. ce qui unifie `A` et `3` (à cause de l'appel à `egal(X,Y)`), puis va essayer d'unifier `A` et `1` (à cause de l'appel à `egal(Z,1)`) causant un **retour arrière** qui défait l'unification de `A` et `3`
2. ce qui unifie `A` et `A` (à cause de l'appel à `egal(Y,Z)`), puis unifie `A` et `2` (à cause de l'appel à `egal(Z,2)`), donnant donc la réponse `A=2`

Unification (4)

- Note 1: avec l'opérateur infixe prédéfini "`=`" (qui unifie ses deux arguments) on aurait pu exprimer le programme de cette façon

```
f(X,Y,Z) :- X=Y, Z=1.
f(X,Y,Z) :- Y=Z, Z=2.
```

- Note 2: mais c'est encore plus simple avec

```
f(X,X,1).
f(_,2,2).
```

- Ici on se sert de la variable spéciale "_" qui indique un terme quelconque qu'on désire ignorer
- Toutes les occurences de "_" sont indépendantes
- Cette définition n'est pas équivalente à `egal(X,X)`.

```
egal(_,_).   % unifie avec egal(0,0), egal(0,1), ...
```

Unification (5)

- L'unification n'a pas un sens de "direction", permettant ainsi de l'utiliser à la fois pour **tester** le type d'un terme, pour **construire** des termes et pour en **extraire** les champs

- Exemple:

```
foo( X, Y, f(X,Y) ).          % programme
bar( f(1,X), g(X) ).

| ?- foo( 1, 2, Q ).          % interaction avec Prolog
Q = f(1,2) ? ;
no
| ?- foo( Q, R, f(1,2) ).
Q = 1, R = 2 ? ;
no
| ?- foo( Q, R, g(1,2) ).
no
| ?- foo( Q, Q, f(1,R) ).
Q = 1, R = 1 ? ;
no
| ?- foo( Q, R, S ), foo( 1, 2, S ).
Q = 1, R = 2, S = f(1,2) ? ;
no
| ?- foo( Q, R, f(S,S) ).
R = Q, S = Q ? ;
no
| ?- bar( Q, R ).
Q = f(1,_A), R = g(_A) ? ;
no
```

Dans le dernier cas on obtient une solution contenant une variable interne

Programmation logique pure (1)

- L'interprétation logique d'un programme et le résultat obtenu en Prolog **ne sont pas toujours les mêmes**

- Il est possible, et relativement facile, d'obtenir des **boucles infinies** lorsqu'on utilise des **règles récursives**

- Ces programmes ne sont pas équivalents:

```
% programme 1

chemin(Z,Z).
chemin(X,Y) :- lien(X,I), chemin(I,Y).

% programme 2

chemin(Z,Z).
chemin(X,Y) :- chemin(I,Y), lien(X,I).
```

car une requête comme `chemin(a,b)` termine dans le premier cas mais pas dans le deuxième

Programmation logique pure (2)

- Trace avec programme 2

prouver(`chemin(a,b)` **)**:
1 essayer `chemin(X,Y) :- chemin(I,Y), lien(X,I).`
  1.1 **prouver(** `chemin(I,Y)` **)** avec `Y`=b:
  1.1.1 essayer `chemin(X',Y') :- chemin(I',Y'), lien(X',I').`
    1.1.1.1 **prouver(** `chemin(I',Y')` **)** avec `Y'`=b:
    1.1.1.1.1 etc

- Prolog tente cette preuve:

```
                 chemin(a,b)
                   /   \
            chemin(I,b)  lien(a,I)
               /   \
      chemin(I',b)  lien(a,I')
          /   \
chemin(I'',b)  lien(a,I'')
   ...
```

- L'ordre des conditions est donc important!

- L'ordre des règles a moins d'importance

```
chemin(X,Y) :- lien(X,I), chemin(I,Y).
chemin(Z,Z).
```

requête `chemin(X,b)` donne `X=a` puis `X=b` plutôt que l'inverse (change l'ordre seulement)

Programmation logique pure (3)

- Pour éviter les boucles infinies, il faut s'assurer qu'il y a un certain **progrès** utile dans le calcul à chaque appel récursif

- Un autre problème est la génération de **solutions redondantes**

```
axe(_,0,0).  % axe(X,Y,Z) si point X,Y,Z sur l'axe x ou y
axe(0,_,0).

| ?- axe(0,0,C).
C = 0 ? ;
C = 0 ? ;
no
```

- C'est nuisible à la **performance**, dans certains cas la complexité d'un algorithme peut devenir exponentielle

```
| ?- axe(0,0,A), axe(0,0,B), axe(0,0,C). % 8 fois A=B=C=0
```

- C'est aussi nuisible à la **convivialité** du programme pour l'usager

Arithmétique en programmation logique pure (1)

- Les opérateurs arithmétiques peuvent être exprimés "purement" sous la forme de relations ternaires `op(X,Y,Z)`
- Exemple: la relation `add(X,Y,Z)` existe si `X` et `Y` sont des entiers non-négatifs et `Z` est la somme de `X` et `Y`
- Cette relation a de multiples usages:
  1. Soustraire deux entiers:
     ```
     sub(X,Y,Z) :- add(Y,Z,X).
     ```
  2. Vérifier que l'entier `X` est inférieur ou égal à un autre entier `Y`:
     ```
     lesseq(X,Y) :- add(X,_,Y).
     ```
  3. Vérifier que l'entier `X` est pair:
     ```
     even(X) :- add(Y,Y,X).
     ```


Arithmétique en programmation logique pure (2)

- Ces relations permettent également de générer des entiers:

```
| ?- lesseq(X,3).
X = 0 ? ;
X = 1 ? ;
X = 2 ? ;
X = 3 ? ;
no
| ?- add(X,Y,3).
X = 0, Y = 3 ? ;
X = 1, Y = 2 ? ;
X = 2, Y = 1 ? ;
X = 3, Y = 0 ? ;
no
| ?- even(X).
X = 0 ? ;
X = 2 ? ;
X = 4 ? ;
X = 6 ?
yes
```

- Cela est pratique pour implanter l'approche **générer et tester** pour trouver une solution

```
| ?- lesseq(X,10), lesseq(Y,10), tester(X,Y).
...
```


Arithmétique en programmation logique pure (3)

- Avec ces relations arithmétiques on peut implanter des fonctions numériques "réversibles"

```
fact(0,1).
fact(N,FN) :- sub(N,1,X), fact(X,FX), mult(N,FX,FN).

| ?- fact(5,X).
X = 120 ? ;
no
| ?- fact(X,Y).
X = 0, Y = 1 ? ;
X = 1, Y = 1 ? ;
X = 2, Y = 2 ? ;
X = 3, Y = 6 ? ;
X = 4, Y = 24 ? ;
X = 5, Y = 120 ?
yes
| ?- fact(X,X).
X = 1 ? ;
X = 2 ? ;
... boucle infinie
| ?- fact(X,120).
X = 5 ? ;
... boucle infinie
```

- Dans le premier cas, puisque `N` est liée, `sub` va simplement unifier `X` à `N-1`
- Dans les 3 derniers cas, `N` n'est pas liée dans le premier appel à `sub`, donc `sub` agit comme générateur, unifiant `N` à 1 puis à 2, etc

Arithmétique impure

- Prolog possède des opérateurs arithmétiques prédéfinis mais ils sont **impurs**
- Syntaxe: ⟨*terme1*⟩ `is` ⟨*terme2*⟩

  où ⟨*terme2*⟩ est une expression numérique en notation infixe (les variables contenues dans ⟨*terme2*⟩ **doivent être liées**)
- Prolog calcule la valeur de ⟨*terme2*⟩ et unifie le résultat avec ⟨*terme1*⟩
- Ce ne sont pas des relations réversibles...

```
fact(0,1).
fact(N,FN) :- X is N-1, fact(X,FX), FN is N*FX.

| ?- fact(5,X).
X = 120 ? ;
... boucle infinie
| ?- fact(X,Y).
X = 0, Y = 1 ? ;
{INSTANTIATION ERROR: _25 is _22-1 - arg 2}

fact(0,1).
fact(N,FN) :- N>0, X is N-1, fact(X,FX), FN is N*FX.

| ?- fact(5,X).
X = 120 ? ;
no
```

Structures de données (1)

- Les termes composés sont essentiellement des **enregistrements**
- Le type de l'enregistrement c'est le foncteur et les arguments sont ses champs
- Les champs n'ont pas de nom, c'est leur position qui les identifient
- Exemple: point 3D

  `point(3,0,5)` -> coord x=3, y=0, z=5

- L'unification permet d'extraire les champs et de construire des enregistrements

```
origine(point(0,0,0)).  % point à l'origine?

addpt(point(X1,Y1,Z1),point(X2,Y2,Z2),point(X3,Y3,Z3)) :-
    add(X1,X2,X3),
    add(Y1,Y2,Y3),
    add(Z1,Z2,Z3).

x(point(X,_,_),X).   % extraction de champ
y(point(_,Y,_),Y).
z(point(_,_,Z),Z).
```

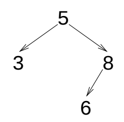

Structures de données (2)

- Les termes peuvent également servir à représenter des **types récursifs** (tels les listes, arbres, graphes)
- Par exemple, pour représenter des **arbres de recherche binaire** on utilise les termes:

  `noeud(Cle,Gauche,Droite)` représente un noeud interne

  `vide` représente un arbre vide

- Donc l'arbre



  est représenté par

```
noeud(5,noeud(3,vide,vide),noeud(8,noeud(6,vide,vide),vide))
```


Structures de données (3)

- Extraction de la plus petite clé

```
ppetit(noeud(X,vide,_),X).
ppetit(noeud(_,G,_),X) :- ppetit(G,X).
```

- Recherche d'une clé dans un a.r.b.

```
element(X,noeud(X,_,_)).
element(X,noeud(C,G,_)) :- X<C, element(X,G).
element(X,noeud(C,_,D)) :- X>C, element(X,D).
```

- Note 1: cela ne permet que de **tester** si une clé existe dans l'arbre car les deux arguments doivent être des termes clos
- Note 2: la clé peut être un terme quelconque (entier, atome, liste, etc) car la relation `X<Y` est définie sur tous les types
- Insérer une clé dans un a.r.b.

```
ins(X,vide,noeud(X,vide,vide)).
ins(X,noeud(C,G,D),noeud(C,G2,D)) :- X<C, ins(X,G,G2).
ins(X,noeud(C,G,D),noeud(C,G,D2)) :- X>C, ins(X,D,D2).
```

- `ins` peut servir à retirer une clé d'un a.r.b. mais seulement si une feuille

Traitement de liste (1)

- Prolog et Lisp représentent les listes de la même façon (chaîne de paires avec champs contenant n'importe quel type)
- Notation de base
  - `[]` = la liste vide
  - `.(X,Y)` = une paire contenant `X` dans son champ `car` et `Y` dans son champ `cdr`
- Notation usuelle plus élégante
  - `[X,Y,Z]` = `.(X,.(Y,.(Z,[])))`
  - `[X|Y]` = `.(X,Y)`
  - `[X,Y|Z]` = `.(X,.(Y,Z))`

Traitement de liste (2)

- Exemple: relation de construction sur les listes

```
cons(A,B,[A|B]).                  % programme

| ?- cons(1,2,X).                 % interaction avec Prolog
X = [1|2] ? ;
no
| ?- cons(1,[2,3],X).
X = [1,2,3] ? ;
no
| ?- cons(X,Y,[4,5,6]).
X = 4, Y = [5,6] ? ;
no
```

- La relation `member(X,L)` est vraie si `X` est un élément de la liste `L`

```
member(X,[X|_]).                  % programme
member(X,[_|R]) :- member(X,R).

| ?- member(2,[1,2,3]).           % interaction avec Prolog
yes
| ?- member(X,[1,2]).
X = 1 ? ;
X = 2 ? ;
no
| ?- member(5,[1,X,3]).
X = 5 ? ;
no
| ?- member(5,X).                 % énumere toutes les
X = [5|_A] ? ;                    % listes contenant 5
X = [_A,5|_B] ? ;
X = [_A,_B,5|_C] ? ;
...
```


Traitement de liste (3)

- La relation `intersection(L1,L2)` est vraie si `L1` et `L2` ont au moins un élément en commun (c'est-à-dire s'il existe un `X` élément de `L1` qui est aussi élément de `L2`)

- Approche "**générer et tester**":

```
intersection(L1,L2) :- member(X,L1), member(X,L2).

| ?- intersection([1,2,3],[0,2,9]).
yes
```

Cette requête va énumérer les éléments de `[1,2,3]` et tester l'appartenance à `[0,2,9]` (succès à `X=2`)

- Cette implantation ne donne pas une relation commutative si `L1` ou `L2` ne sont pas clos

```
| ?- intersection(L,[1,2]).
L = [1|_A] ? ;
L = [2|_A] ? ;
L = [_A,1|_B] ? ;
L = [_A,2|_B] ? ;
...
| ?- intersection([1,2],L).
L = [1|_A] ? ;                    % on ne passe jamais à 2
L = [_A,1|_B] ? ;
L = [_A,_B,1|_C] ? ;
...
```


Traitement de liste (4)

- La relation `append(L1,L2,L3)` est vraie si la liste `L3` est la concaténation des listes `L1` et `L2`

```
append( [], L, L ).
append( [X|R1], L, [X|R2] ) :- append( R1, L, R2 ).

| ?- append( [1,2], [3,4], Y ).
Y = [1,2,3,4] ? ;
no
| ?- append( [1,2], Y, [1,2,3,4] ).
Y = [3,4] ? ;
no
| ?- append( Y, [3,4], [1,2,3,4] ).
Y = [1,2] ? ;
no
| ?- append( Y, Z, [1,2,3,4] ).
Y = [], Z = [1,2,3,4] ? ;
Y = [1], Z = [2,3,4] ? ;
Y = [1,2], Z = [3,4] ? ;
Y = [1,2,3], Z = [4] ? ;
Y = [1,2,3,4], Z = [] ? ;
no
```

- La relation `append` est puissante: plusieurs relations peuvent s'exprimer uniquement avec elle

Traitement de liste (5)

- La relation `dernier(X,L)` est vraie si `X` est le dernier élément de la liste `L`

```
dernier(X,L) :- append(_,[X],L).

| ?- dernier(Y,[1,2,3]).
Y = 3 ? ;
no
| ?- dernier(5,Y).
Y = [5] ? ;
Y = [_A,5] ? ;
...
```

- La relation `member`: `member(X,L) :- append(_,[X|_],L).`

- La relation `insere(X,L1,L2)` est vraie si `L2` est la liste `L1` dans laquelle `X` a été inséré

```
insere(X,L1,L2) :- append(A,B,L1), append(A,[X|B],L2).

| ?- insere(5,[1,2],Y).
Y = [5,1,2] ? ;
Y = [1,5,2] ? ;
Y = [1,2,5] ? ;
no
| ?- insere(5,Y,[5,1,5]).
Y = [1,5] ? ;
Y = [5,1] ? ;
... boucle infinie
```

Traitement de liste (6)

- La relation `perm(L1,L2)` est vraie si la liste `L1` est une permutation de la liste `L2`

```
perm( [], [] ).
perm( [X|R], L ) :- perm(R,RP), insere(X,RP,L).

| ?- perm([1,2,3],Y).
Y = [1,2,3] ? ;
Y = [2,1,3] ? ;
Y = [2,3,1] ? ;
Y = [1,3,2] ? ;
Y = [3,1,2] ? ;
Y = [3,2,1] ? ;
no
| ?- perm(Y,[1,2,3]).
Y = [1,2,3] ? ;
Y = [2,1,3] ? ;
Y = [3,1,2] ? ;
Y = [1,3,2] ? ;
Y = [2,3,1] ? ;
Y = [3,2,1] ? ;
... boucle infinie
```


Traitement de liste (7)

- Tri "**générer et tester**"

- La relation `tri(L1,L2)` est vraie si la liste `L2` est la liste `L1` triée

```
tri(L1,L2) :- perm(L1,L2), ordonnee(L2).

ordonnee([]).
ordonnee([_]).
ordonnee([X,Y|R]) :- Y>=X, ordonnee([Y|R]).

| ?- tri( [4,1,3,2], L ).
L = [1,2,3,4] ? ;
no
```

- "Quicksort" est plus rapide

```
qsort([],[]).
qsort([X|R],L) :-
  part(X,R,PP,PG),
  qsort(PP,PPT),
  qsort(PG,PGT),
  append(PPT,[X|PGT],L).

part(_,[],[],[]).
part(X,[Y|R],[Y|PP],PG) :- Y<X, part(X,R,PP,PG).
part(X,[Y|R],PP,[Y|PG]) :- Y>=X, part(X,R,PP,PG).

| ?- qsort( [4,1,3,2], L ).
L = [1,2,3,4] ? ;
no
```


Applications de Prolog (1)

- Calcul de la dérivée **symbolique** d'une expression (par exemple: dérivée de `x*x` est `2*x`)

- On se sert du fait qu'en Prolog `X+Y` est une syntaxe conviviale pour le terme `+(X,Y)`

```
deriv( A, 0 ) :- number(A).
deriv( x, 1 ).
deriv( A+B, DA+DB ) :- deriv(A,DA), deriv(B,DB).
deriv( A-B, DA-DB ) :- deriv(A,DA), deriv(B,DB).
deriv( A*B, A*DB+DA*B ) :- deriv(A,DA), deriv(B,DB).
deriv( A/B, (B*DA-DB*A)/(B*B) ) :- deriv(A,DA), deriv(B,DB).
deriv( sin(A), cos(A)*DA ) :- deriv(A,DA).
deriv( cos(A), -sin(A)*DA ) :- deriv(A,DA).

| ?- deriv( x, D ).
D = 1 ? ;
no
| ?- deriv( 2*x, D ).
D = 2*1+0*x ? ;
no
| ?- deriv( cos(x*x), D ).
D = -(sin(x*x)*(x*1+1*x)) ? ;
no
| ?- deriv( F, 1 ).
F = x ? ;
no
| ?- deriv( F, 2 ).
no
| ?- deriv( F, 2*1+0*x ).
F = 2*x ? ;
no
```


Applications de Prolog (2)

- Analyse syntaxique de langue naturelle (pour tester qu'une phrase est valide)

- Une phrase sera une liste de symboles

  `[le,chat,est,sur,la,chaise]`

- La syntaxe est spécifiée par une grammaire

```
<phrase> ::= <chose> <verbe> <relation> <chose>
<chose> ::= le chat | la chaise | la table
<verbe> ::= est
<relation> ::= sur | sous
```

- Chaque catégorie syntaxique se traduit par une relation:

```
chose([le,chat|L],L).
chose([la,chaise|L],L).
chose([la,table|L],L).

verbe([est|L],L).

relation([sur|L],L).
relation([sous|L],L).

phrase(P) :- chose(P,A), verbe(A,B), relation(B,C), chose(C,[])
```

Applications de Prolog (3)

```
| ?- phrase([le,chat,est,sous,la,table]).
yes
| ?- phrase(P).
P = [le,chat,est,sur,le,chat] ? ;
P = [le,chat,est,sur,la,chaise] ? ;
P = [le,chat,est,sur,la,table] ? ;
P = [le,chat,est,sous,le,chat] ? ;
P = [le,chat,est,sous,la,chaise] ? ;
P = [le,chat,est,sous,la,table] ? ;
P = [la,chaise,est,sur,le,chat] ? ;
P = [la,chaise,est,sur,la,chaise] ? ;
P = [la,chaise,est,sur,la,table] ? ;
P = [la,chaise,est,sous,le,chat] ? ;
P = [la,chaise,est,sous,la,chaise] ? ;
P = [la,chaise,est,sous,la,table] ? ;
P = [la,table,est,sur,le,chat] ? ;
P = [la,table,est,sur,la,chaise] ? ;
P = [la,table,est,sur,la,table] ? ;
P = [la,table,est,sous,le,chat] ? ;
P = [la,table,est,sous,la,chaise] ? ;
P = [la,table,est,sous,la,table] ? ;
no
```


L'opérateur de coupure (1)

- Par défaut Prolog tente de prouver la requête de toutes les façons possibles (suivant le programme) jusqu'à ce qu'une preuve soit trouvée
- Si une preuve échoue, Prolog fait un **retour-arrière** pour tenter une autre ligne de preuve
- Dans certains cas ce retour-arrière peut être nuisible:
  1. si le retour-arrière va **répéter une solution**
  2. si le retour-arrière **n'a aucune chance de trouver une solution**
- L'opérateur de **coupure** ("**cut**") permet de signaler qu'un retour-arrière n'est pas désiré
- Syntaxe:

  `rel(...) :-` $\langle G \rangle$, `!`, $\langle D \rangle$.
- Si le système trouve une preuve pour $\langle G \rangle$ alors un échec de prouver $\langle D \rangle$ sera un échec pour cette règle et **aucune autre règle pour** `rel(...)` **ne sera essayée**

L'opérateur de coupure (2)

- Exemple

```
couleur(bleu).
couleur(orange).
couleur(chocolat).

aliment(pomme).
aliment(orange).
aliment(chocolat).

deuxsens1(X) :- couleur(X), aliment(X).

deuxsens2(X) :- couleur(X), !, aliment(X).

deuxsens3(X) :- couleur(X), aliment(X), !.

| ?- deuxsens1(orange).
yes

| ?- deuxsens2(orange).
yes

| ?- deuxsens3(orange).
yes

| ?- deuxsens1(X).
X = orange ? ;
X = chocolat ? ;
no

| ?- deuxsens2(X).
no

| ?- deuxsens3(X).
X = orange ? ;
no
```


L'opérateur de coupure (3)

- L'opérateur de coupure est **impur** car une relation qui l'utilise peut ne plus avoir la même interprétation logique
- Déf: Une utilisation de "!" est mauvaise ("**red cut**") si la relation ne donne plus les mêmes résultats (exemple précédent)
- Déf: Une utilisation de "!" est bonne ("**green cut**") si la relation donne les mêmes résultats (mais possiblement plus rapidement ou sans solution redondante)
- Exemple

```
% sans opérateur de coupure

element(X,noeud(X,_,_)).
element(X,noeud(C,G,_)) :- X<C, element(X,G).
element(X,noeud(C,_,D)) :- X>C, element(X,D).

% avec opérateur de coupure

element(X,noeud(X,_,_)) :- !.
element(X,noeud(C,G,_)) :- X<C, !, element(X,G).
element(X,noeud(C,_,D)) :- element(X,D).
```

L'opérateur de coupure (4)

```
max1(X,Y,Y) :- X=<Y.
max1(X,Y,X) :- X>=Y.

| ?- max1(1,2,X).
X = 2 ? ;
no
| ?- max1(1,1,X).
X = 1 ? ;
X = 1 ? ;
no

max2(X,Y,Y) :- X<Y.
max2(X,Y,X) :- X>=Y.

| ?- max2(1,2,X).
X = 2 ? ;
no
| ?- max2(1,1,X).
X = 1 ? ;
no

max3(X,Y,Y) :- X<Y, !.    % green cut
max3(X,Y,X) :- X>=Y.

| ?- max3(1,2,X).
X = 2 ? ;
no
| ?- max3(1,1,X).
X = 1 ? ;
no
| ?- max3(1,2,1).
no

max4(X,Y,Y) :- X<Y, !.    % red cut
max4(X,Y,X).

| ?- max4(1,2,X).
X = 2 ? ;
no
| ?- max4(1,1,X).
X = 1 ? ;
no
| ?- max4(1,2,1).
yes
```


La négation (1)

- La négation logique ne s'exprime pas complètement en Prolog
- Lorsque Prolog répond "**no**" à une requête c'est qu'il n'a **pas trouvé de preuve que la requête est vraie**
- Ce n'est pas la même chose que de **trouver une preuve que la requête est fausse**
- C'est ce qu'on appèle "**négation par l'échec**"
- La négation (partielle) d'une relation est possible avec l'opérateur de coupure:

```
pasrel(X) :- rel(X), !, fail.
pasrel(_).
```

- Si `rel(X)` est vraie, alors `pasrel(X)` aboutira à un échec (car la relation `fail` n'existe pas), sinon `pasrel(X)` sera vraie

La négation (2)

- Exemple montrant que cela ne correspond pas exactement à la négation logique:

```
petit(0).
petit(1).
petit(2).

paszero(0) :- !, fail.
paszero(_).


| ?- petit(X), paszero(X).
X = 1 ? ;
X = 2 ? ;
no

| ?- paszero(X), petit(X).
no
```

- Ici la relation `paszero(X)` fonctionne bien à condition que `X` soit un terme clos
- Lorsque `X` n'est pas liée, il y aura unification de `X` avec `0` et donc un échec après le "!"

Implantation du retour-arrière (1)

- Basé sur les **continuations**
- Une requête c'est une série de **buts** à prouver (chaque but est un appel d'une relation)
- Deux "continuations" possibles pour un but B
  1. **succès** (une preuve a été trouvée pour B)
  2. **échec** (une preuve n'a pas été trouvée)
- Un "succès" fait avancer le calcul à la preuve du **prochain but**
- Un "échec" fait un retour-arrière au **but précédent** pour trouver une autre preuve
- La continuation de succès d'un but B1, si elle échoue la preuve du prochain but B2, doit pouvoir retourner à B1 pour trouver une autre preuve
- Donc la continuation de succès a elle même une **continuation d'échec en paramètre**

Implantation du retour-arrière (2)

- Exemple: traduction d'une requête Prolog en Scheme

```
; | ?- member(X,[1,2,3]).
; X = 1 ? ;
; X = 2 ? ;
; X = 3 ? ;
; no

(define generer
  (lambda (lst echec succes)
    (if (null? lst)
      (echec)
      (succes (car lst)
              (lambda ()
                (generer (cdr lst)
                         echec
                         succes))))))

(generer '(1 2 3)
         (lambda () (display "no"))
         (lambda (x retour)
           (write x)
           (read)
           (retour)))
```


Implantation du retour-arrière (3)

```
; somme(L,N,X,Y) :- member(X,L),
;                   member(Y,L),
;                   N is X+Y.
;
; | ?- somme([2,5,8],10,X,Y).
; X = 2, Y = 8 ? ;
; X = 5, Y = 5 ? ;
; X = 8, Y = 2 ? ;
; no

(define somme
  (lambda (lst n echec succes)
    (generer lst
             echec
             (lambda (x retour1)
               (generer lst
                        retour1
                        (lambda (y retour2)
                          (if (= (+ x y) n)
                            (succes x y retour2)
                            (retour2)))))))))

(somme '(2 5 8)
       10
       (lambda () (display "no"))
       (lambda (x y retour)
         (write (list x y))
         (read)
         (retour)))
```