

PLAN

- Beware of the cut!
- L'algorithme d'unification.
- Spécifier un prédicat: arguments d'entrée et de sortie.
- Algorithmes “générer et tester”:
 - Les n reines.
 - Automates non déterministes.
- Prédicats prédéfinis sur les termes:
 - Prédicats de test.
 - Prédicats de manipulation.
 - Exemples d'utilisation.

PLAN

- Beware of the cut!
- L'algorithme d'unification.
- Spécifier un prédicat: arguments d'entrée et de sortie.
- Algorithmes “générer et tester”:
 - Les n reines.
 - Automates non déterministes.
- Prédicats prédéfinis sur les termes:
 - Prédicats de test.
 - Prédicats de manipulation.
 - Exemples d'utilisation.

Beware of the cut !

Le prédicat `ects_sem(sem,projet,ects)` du TD3:

version sans cut:

```
ects_sem(X, [], 0).
```

```
ects_sem(X, [Y|G], N) :- ue(Y, M, X), ects_sem(X, G, K), N is M+K.
```

```
ects_sem(X, [Y|G], N) :- ue(Y, M, Z), X \= Z, ects_sem(X, G, N).
```

```
?- ects_sem(1, [an1, ia, ig], X).
```

```
X = 9 ? ;
```

```
no
```

```
(rappel: ue(an1,3,1).   ue(ia,6,1).   ue(ig,6,2). )
```

Deuxième version, avec cut:

```
ects_sem(X, [], 0).
```

```
ects_sem(X, [Y|G], N) :- !, ue(Y, M, X), ects_sem(X, G, K), N is M+K.
```

```
ects_sem(X, [Y|G], N) :- ue(Y, M, Z), ects_sem(X, G, N).
```

ou bien

```
ects_sem(X, [], 0).
```

```
ects_sem(X, [Y|G], N) :- ue(Y, M, X), !, ects_sem(X, G, K), N is M+K.
```

```
ects_sem(X, [Y|G], N) :- ue(Y, M, Z), ects_sem(X, G, N).
```

Dans les deux cas, il s'agit d'un “red cut”: si on élimine le cut on obtient

```
ects_sem(X, [], 0).
```

```
ects_sem(X, [Y|G], N) :- ue(Y, M, X), , ects_sem(X, G, K), N is M+K.
```

```
ects_sem(X, [Y|G], N) :- ue(Y, M, Z), ects_sem(X, G, N).
```

```
?- ects_sem(1, [an1, ia, ig], X).
```

```
X = 9 ? ;
```

```
X = 3 ? ;
```

```
X = 6 ? ;
```

```
X = 0 ? ;
```

```
no
```

Version non correcte:

```
ects_sem(X, [], 0).
```

```
ects_sem(X, [Y|G], N) :- !, ue(Y, M, X), ects_sem(X, G, K), N is M+K.
```

```
ects_sem(X, [Y|G], N) :- ue(Y, M, Z), ects_sem(X, G, N).
```

```
?- ects_sem(1, [an1, ia, ig], X).
```

no

Version correcte:

```
ects_sem(X, [], 0).
```

```
ects_sem(X, [Y|G], N) :- ue(Y, M, X), !, ects_sem(X, G, K), N is M+K.
```

```
ects_sem(X, [Y|G], N) :- ue(Y, M, Z), ects_sem(X, G, N).
```

```
?- ects_sem(1, [an1, ia, ig], X).
```

```
X = 9 ? ;
```

no

Le prédicat `inter(liste1,liste2,-resultat)`.

Version sans cut (explicite):

```
inter([],T,[]).
```

```
inter([C|G],T,[C|P]):- member(C,T),inter(G,T,P).
```

```
inter([C|G],T,R):- not(member(C,T)),inter(G,T,R).
```

```
?- inter([2,1,4],[1,3,4],X).
```

```
X = [1,4] ? ;
```

```
no
```


Version avec cut:

```
inter([],T,[]).  
inter([C|G],T,[C|P]):- !, member(C,T),inter(G,T,P).  
inter([C|G],T,R):- inter(G,T,R).
```

ou

```
inter([],T,[]).  
inter([C|G],T,[C|P]):- member(C,T), ! ,inter(G,T,P).  
inter([C|G],T,R):- inter(G,T,R).
```

Dans les deux cas, il s'agit d'un "red cut": si on élimine le cut on obtient

```
inter([],T,[]).  
inter([C|G],T,[C|P]):- member(C,T),inter(G,T,P).  
inter([C|G],T,R):- inter(G,T,R).
```

```
?- inter([2,1,4],[1,3,4],X).  
X = [1,4] ? ;  
X = [1] ? ;  
X = [4] ? ;  
X = [] ? ;  
no
```

Version non correcte:

```
inter([],T,[]).  
inter([C|G],T,[C|P]):- !, member(C,T),inter(G,T,P).  
inter([C|G],T,R):- inter(G,T,R).
```

```
?- inter([2,1,4],[1,3,4],X).
```

no

Version correcte:

```
inter([],T,[]).  
inter([C|G],T,[C|P]):- member(C,T),!,inter(G,T,P).  
inter([C|G],T,R):- inter(G,T,R).
```

```
?- inter([2,1,4],[1,3,4],X).
```

```
X = [1,4] ? ;
```

no

Le cut peut être placé en première position dans le corps d'une clause. Exemple:

$f(0,1).$

$f(N,R):-N>0, P \text{ is } N-1, f(P,Q), R \text{ is } 2*Q.$

version avec cut:

$f(0,1):-!. .$

$f(N,R):- P \text{ is } N-1, f(P,Q), R \text{ is } 2*Q.$

PLAN

- L'algorithme d'unification.
- Spécifier un prédicat: arguments d'entrée et de sortie.
- Algorithmes “générer et tester”:
 - Les n reines.
 - Automates non déterministes.
- Prédicats prédéfinis sur les termes:
 - Prédicats de test.
 - Prédicats de manipulation.
 - Exemples d'utilisation.

L'algorithme d'unification

Entrée: deux termes t_1 et t_2 .

Sortie: la substitution σ , mgu de t_1 et t_2 , si elle existe, sinon echec.

Empiler $t_1 = t_2$.

Tant que pile non vide

 dépiler $u = v$

 Selon le cas:

L'algorithme d'unification

- u est une variable sans occurrence dans v :
On substitue v à u dans la pile et dans σ
- v est une variable sans occurrence dans u :
On substitue u à v dans la pile et dans σ
- u et v sont des constantes ou variables identiques: On continue.
- $u = f(u_1, \dots, u_n)$ $v = f(v_1, \dots, v_n)$.
On empile $u_i = v_i$ pour i de n à 1 .

Sinon échec.

Exemple

$$f(g(X,A,h(X,b)),Z) = f(g(A,b,Z),Y)$$

On empile les arguments de droite à gauche.

Etat de la pile et substitution calculée, après une étape:

$$\begin{array}{lcl} g(X,A,h(X,b)) & = & g(A,b,Z) \quad \text{Substitution } \{\} \\ Z & = & Y \end{array}$$

Après une deuxième étape:

$$\begin{array}{lcl} X & = & A \quad \text{Substitution } \{\} \\ A & = & b \\ h(X,b) & = & Z \\ Z & = & Y \end{array}$$

Ensuite:

$$A = b \quad \text{Substitution } \{X \leftarrow A\}$$

$$\begin{array}{lcl} h(A,b) & = & Z \\ Z & = & Y \end{array}$$

Ensuite:

$$\begin{array}{lcl} h(b,b) & = & Z \quad \text{Substitution } \{X \leftarrow b, A \leftarrow b\} \\ Z & = & Y \end{array}$$

Ensuite:

$$\begin{array}{lcl} h(b,b) & = & Y \quad \text{Substitution} \\ & & \{X \leftarrow b, A \leftarrow b, Z \leftarrow h(b,b)\} \end{array}$$

Résultat:

$$\{X \leftarrow b, A \leftarrow b, Z \leftarrow h(b,b), Y \leftarrow h(b,b)\}$$

PLAN

- L'algorithme d'unification.
- Spécifier un prédicat: arguments d'entrée et de sortie.
- Algorithmes “générer et tester”:
 - Les n reines.
 - Automates non déterministes.
- Prédicats prédéfinis sur les termes:
 - Prédicats de test.
 - Prédicats de manipulation.
 - Exemples d'utilisation.

Spécifier un prédicat: arguments d'entrée et de sortie

Un argument est *d'entrée* pour un prédicat si, au moment du `call` du prédicat, l'argument en question doit être clos, sans variables.

Un argument est *de sortie* pour un prédicat si, au moment du `exit` du prédicat, l'argument en question doit contenir le résultat de l'appel.

Par exemple dans la définition:

```
fact(0,1).
```

```
fact(N,R):- N>0, M is N-1, fact(M,P), R is P*N.
```

le premier argument est d'entrée et le deuxième de sortie.

Une spécification correcte de `fact` est donc `fact(+Arg, -Res)`

Spécifier un prédicat: arguments d'entrée et de sortie (2)

Les erreurs comme

```
[INSTANTIATION ERROR- in arithmetic:  expected bound  
value ]
```

viennent de l'utilisation de termes ouverts pour des argument d'entrée.

Par exemple:

```
? - X is Y + 4
```

ou

```
? - Z < 7
```

Spécifier un prédicat: arguments d'entrée et de sortie (3)

Il est parfois possible d'utiliser un prédicat pour calculer à la fois une relation R et sa relation inverse R^{-1} . Dans ce cas il n'y a pas d'arguments d'entrée et de sortie.

Un exemple vu en TD:

```
transform(0,0).
```

```
transform(s(Y),N):- transform(Y,M), N is M+1.
```

Deux possibilités d'utilisation de `transform`:

```
transform(s(s(s(0))),N).
```

```
transform(X,3).
```

Une spécification correcte dans ce cas serait

```
transform(Terme,Entier), ou transform(?Terme,?Entier).
```

Spécifier un prédicat: arguments d'entrée et de sortie (4)

Comme exemple d'utilisation à *l'invers* de `transform`, voyons l'arbre du but:

`transform(X,3)`

dont voici la trace:

```
?- transform(X,3).  
  (1) call:transform(_172,3) ?  
  (2) call:transform(_247,_250) ?  
  (2) exit:transform(0,0) ?  
  (2) redo:transform(0,0) ?  
  (3) call:transform(_310,_313) ?  
  (3) exit:transform(0,0) ?  
  (2) exit:transform(s(0),1) ?  
  (2) redo:transform(s(0),1) ?  
  (3) redo:transform(0,0) ?
```

(4) call:transform(_373,_376) ?

(4) exit:transform(0,0) ?

(3) exit:transform(s(0),1) ?

(2) exit:transform(s(s(0)),2) ?

(1) exit:transform(s(s(s(0))),3) ?

X = s(s(s(0))) ?

yes

Spécifier un prédicat: arguments d'entrée et de sortie (5)

La symétrie de `transform` n'est pourtant pas complète:

```
?- trans(s(s(s(0))),X).
```

```
X = 3 ? ;
```

```
no
```

```
?- trans(X,3).
```

```
X = s(s(s(0))) ? ;
```

boucle

PLAN

- L'algorithme d'unification.
- Spécifier un prédicat: arguments d'entrée et de sortie.
- Algorithmes “générer et tester”:
 - Les n reines.
 - Automates non déterministes.
- Prédicats prédéfinis sur les termes:
 - Prédicats de test.
 - Prédicats de manipulation.
 - Exemples d'utilisation.

Algorithmes “générer et tester”

Le “backtracking” (retour en arrière) de Prolog est très bien adapté aux problèmes qui nécessitent l’exploration de plusieurs configurations possibles à la recherche d’une (ou de plusieurs) solutions (coloriage d’un plan, n reines, carrés magiques,...)

Deux exemples:

Le problème des n reines.

La reconnaissance d’un mot par un automate fini non déterministe.

PLAN

- L'algorithme d'unification.
- Spécifier un prédicat: arguments d'entrée et de sortie.
- Algorithmes “générer et tester”:
 - Les n reines.
 - Automates non déterministes.
- Prédicats prédéfinis sur les termes:
 - Prédicats de test.
 - Prédicats de manipulation.
 - Exemples d'utilisation.

Les n reines

Problème: placer n reines sans prises sur un échiquier $n \times n$.

Réprésentation des configurations: une liste de n entiers, inclus entre 1 et n .

Le i -ème entier donne la position de la reine sur la i -ème ligne.

Par exemple, pour $n = 4$, la configuration:

		R	
			R
R			
	R		

est représentée par

$[3, 4, 1, 2]$

Les n reines

Il y a autant de configurations que de permutations des entiers de 1 à n .

Remarque: si une configuration est représentée par le choix de n case de l'échiquier, on a $C_{n^2}^n$ configuration au lieu de $n!$.

Pour $n = 8$ cela donne 4426165368 configurations au lieu de 40320.

Une représentation bien choisie est essentielle pour éviter un trop grand nombre de configurations.

Les n reines

Les solutions sont des configuration telles que chaque diagonale et chaque anti-diagonale contient au plus une reine. Par exemple:

	R		
			R
R			
		R	

représentée par $[2, 4, 1, 3]$

Les n reines

Deux reines en positions (i, j) et (i', j') se trouvent sur la même diagonale ssi $i - j = i' - j'$.

0	-1	-2	-3
1	0	-1	-2
2	1	0	-1
3	2	1	0

Les n reines

Deux reines en positions (i, j) et (i', j') se trouvent sur la même anti-diagonale ssi $i + j = i' + j'$.

2	3	4	5
3	4	5	6
4	5	6	7
5	6	7	8

Les n reines

Pour résoudre n reines on peut donc:

- générer une permutation P de $1, \dots, n$.
- Calculer les listes D de différences et la liste S de sommes associés à P . Le i -ème élément de D est $i - P(i)$, le i -ème élément de S est $i + P(i)$.
- vérifier que tous les éléments de D sont différents entre eux. Même chose pour S .
- passer à la permutation suivante et recommencer.

Exemple: on applique l'algorithme du transparent précédent à la permutation $p = [2, 4, 3, 1]$, qui représente la configuration (qui n'est pas une solution):

	R		
			R
		R	
R			

La liste de différences de p est $[1-2, 2-4, 3-3, 4-1] = [-1, -2, 0, 3]$

Donc le test sur les diagonales passe (les reines se trouvent sur des diagonales toutes différentes les unes des autres).

La liste des sommes de p est $[1+2, 2+4, 3+3, 4+1] = [3, 6, 6, 5]$. La deuxième et troisième reine se trouvent sur la même anti-diagonale.

On élimine cette configuration et on passe à la suivante.

Les n reines

Voici un prédicat pour générer les permutations, défini par récurrence sur n :

```
perm( [], [] ) .
```

```
perm( [X|L] , Z ) :- perm(L, W) , insertion(X, W, Z) .
```

```
insertion(X, L, [X|L]) .
```

```
insertion(X, [Y|L] , [Y|G]) :- insertion(X, L, G) .
```

Exemple d'utilisation de `perm` avec $n = 3$:

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

Les n reines

Voici un prédicat pour générer les listes des sommes et des différences: (le 1er argument contient les indices de ligne, le 2ème ceux de colonne, le 3ème les sommes et le 4ème les differences).

```
combiner([], [], [], []).
```

```
combiner([X1|X], [Y1|Y], [S1|S], [D1|D]) :-
```

```
    S1 is X1 +Y1,
```

```
    D1 is X1 - Y1,
```

```
    combiner(X,Y,S,D).
```


Exemple d'utilisation de `combiner` pour l'exemple précédent:

```
?- combiner([1,2,3,4],[2,4,3,1],S,D).
```

```
D = [-1,-2,0,3],
```

```
S = [3,6,6,5] ? ;
```

```
no
```

Les n reines

Voici un prédicat pour vérifier que tous les éléments d'une liste sont différents:

```
tous_diff([X]).
```

```
tous_diff([X|Y]) :- not(member(X,Y)), tous_diff(Y).
```

Les n reines

Voici un prédicat qui resout n reines, dans le cas $n = 4$:

`resout(P) :-`

`perm([1,2,3,4],P),`

`combiner([1,2,3,4],P,S,D),`

`tous_diff(S),`

`tous_diff(D).`

Les n reines

On calcule les solutions:

```
?- resout(X) .
```

```
X = [3,1,4,2] ? ;
```

```
X = [2,4,1,3] ? ;
```

```
no
```

Les n reines

Les deux solutions pour $n = 4$ sont: $[3, 1, 4, 2]$

		R	
R			
			R
	R		

et $[2, 4, 1, 3]$

	R		
			R
R			
		R	

Les n reines

Pour $n = 8$, (après modification de `resout`):

```
?- resout(X).
```

```
X = [5,2,6,1,7,4,8,3] ? ;
```

```
X = [6,3,5,7,1,4,2,8] ? ;
```

```
X = [6,4,7,1,3,5,2,8] ? ;
```

```
X = [3,6,2,7,5,1,8,4] ?
```

```
...
```

```
?- setof(X,resout(X),L),length(L,N).
```

```
L = [[1,5,8,6,3,7,2,4],[1,6,8,3,7,4,2,5],  
[1,7,4,6,8,2,5,3],....]
```

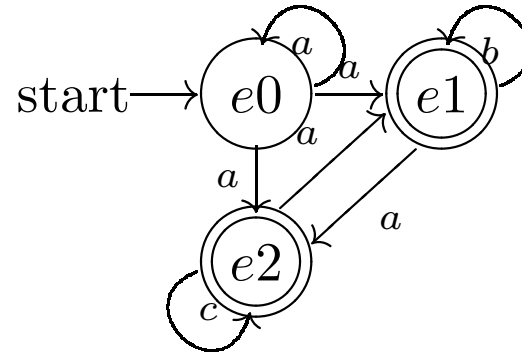
```
N = 92 ?;
```

```
no
```


PLAN

- L'algorithme d'unification.
- Spécifier un prédicat: arguments d'entrée et de sortie.
- Algorithmes “générer et tester”:
 - Les n reines.
 - Automates non déterministes.
- Prédicats prédéfinis sur les termes:
 - Prédicats de test.
 - Prédicats de manipulation.
 - Exemples d'utilisation.

AFND



est représenté par:

`init(e0).`

`final(e1). final(e2).`

`delta(e0,a,e0). delta(e0,a,e1). delta(e0,a,e2).`

`delta(e1,b,e1). delta(e1,a,e2).`

`delta(e2,a,e1). delta(e2,c,e2).`

AFND

Pour vérifier si un mot est accepté:

- générer un chemin de lecture du mot à partir de l'état initial.
- vérifier que le chemin termine sur un état final.
- Sinon, recommencer.

AFND

```
parse(L) :- init(S),  
            trans(S,L).
```

```
trans(X, [A|B]) :-  
    delta(X,A,Y),  
    trans(Y,B).
```

```
trans(X, []) :-  
    final(X).
```

Dans l'exemple, la lecture de [a] termine avec succes à la 2ème tentative:

```
?- parse([a]).
```

```
(1) call:parse([a]) ?
```

```
(2) call:init(_91) ?
```

```
(2) exit:init(e0) ?
```

```
(3) call:trans(e0,[a]) ?
```

```
(4) call:delta(e0,a,_253) ?
```

```
(4) exit:delta(e0,a,e0) ?
```

```
(5) call:trans(e0,[]) ?
```

```
(6) call:final(e0) ?
```

```
(6) fail:final(e0) ?
```

```
(5) redo:trans(e0,[]) ?
```

(5) fail:trans(e0,[]) ?
(4) redo:delta(e0,a,e0) ?
(4) redo:delta(e0,a,_253) ?
(4) exit:delta(e0,a,e1) ?
(7) call:trans(e1,[]) ?
(8) call:final(e1) ?
(8) exit:final(e1) ?
(7) exit:trans(e1,[]) ?
(3) exit:trans(e0,[a]) ?
(1) exit:parse([a]) ?

yes

PLAN

- L'algorithme d'unification.
- Spécifier un prédicat: arguments d'entrée et de sortie.
- Algorithmes “générer et tester”:
 - Les n reines.
 - Automates non déterministes.
- **Prédicats prédéfinis sur les termes:**
 - **Prédicats de test.**
 - Prédicats de manipulation.
 - Exemples d'utilisation.

Prédicats de test

- `integer/1` teste si l'argument est un entier.
 `?- integer(3).`
 Yes.
 `?- integer(X).`
 No.
- `number/1` teste si l'argument est un nombre.
- `float/1` teste si l'argument est un flottant.
- `atomic/1` teste si l'argument est une constante.
- `var/1` teste si l'argument est une variable non instanciée.
- `nonvar/1` teste si l'argument n'est pas une variable non instanciée.
- `compound/1` teste si l'argument est un terme composé.
- `ground/1` teste si l'argument est un terme sans variables.

PLAN

- L'algorithme d'unification.
- Spécifier un prédicat: arguments d'entrée et de sortie.
- Algorithmes “générer et tester”:
 - Les n reines.
 - Automates non déterministes.
- **Prédicats prédéfinis sur les termes:**
 - Prédicats de test.
 - **Prédicats de manipulation.**
 - Exemples d'utilisation.

Prédicats de manipulation de termes

- `functor(Terme, Foncteur, Arite).`

Mode d'utilisation `(+, -, -)` ou `(-, +, +)` seulement.

`?- functor(pere(jean, isa), F, A).`

`F = pere, A = 2.`

`?- functor(T, pere, 2).`

`T = pere(X, Y).`

- `arg(N, Terme, X)` unifie `X` et le `N`-ième argument de `Terme`.

`?- arg(1, pere(jean, isa), X).`

`X = jean.`

Prédicats de manipulation de termes

Terme =.. Liste transforme un terme en une liste.

?- pere(jean, isa) =.. L.

L = [pere, jean, isa].

?- T =.. [a, b, c].

T = a(b, c).

PLAN

- L'algorithme d'unification.
- Spécifier un prédicat: arguments d'entrée et de sortie.
- Algorithmes “générer et tester”:
 - Les n reines.
 - Automates non déterministes.
- **Prédicats prédéfinis sur les termes:**
 - Prédicats de test.
 - Prédicats de manipulation.
 - **Exemples d'utilisation.**

Exemple d'utilisation

```
/*Addition*/
```

```
plus(X, Y, Z) :-  
    nonvar(X),  
    nonvar(Y),  
    Z is X + Y.
```

```
plus(X, Y, Z) :-  
    nonvar(Y),  
    nonvar(Z),  
    X is Z - Y.
```

```
plus(X, Y, Z) :-  
    nonvar(X),  
    nonvar(Z),  
    Y is Z - X.
```


Exemple d'utilisation

```
/* Unification*/
```

```
/* Unifier deux Variables*/
```

```
unify(X, Y) :-  
    var(X),var(Y),X = Y.
```

```
/* Premier argument variable*/
```

```
unify(X, Y) :-  
    var(X),nonvar(Y),X = Y.
```

```
/* Deuxieme argument variable*/
```

```
unify(X, Y) :-  
    nonvar(X),var(Y),Y = X.
```

```
/* Arguments sont constantes*/  
unify(X, Y) :-  
    nonvar(X), nonvar(Y),  
    atomic(X), atomic(Y),  
    X = Y.
```

```
/* Arguments sont composes.*/  
unify(X, Y) :-  
    nonvar(X), nonvar(Y),  
    compound(X), compound(Y),  
    termUnify(X, Y).
```



```

/* Unifier deux termes composes*/
termUnify(X, Y) :-
    functor(X, F, N),
    functor(Y, F, N), /* meme symb. fonctionnelle*/
    argUnify(N, X, Y).

/* Unifier les arguments */
argUnify(N, X, Y) :-
    N>0,
    argUnify1(N, X, Y),
    Ns is N - 1,
    argUnify(Ns, X, Y).

argUnify(0, X, Y).

```



```
/* Unifier les arguments N */  
argUnify1(N, X, Y) :-  
    arg(N, X, ArgX),  
    arg(N, Y, ArgY),  
    unify(ArgX, ArgY).
```