

PLAN

- **Sémantique opérationnelle (rappel).**
- Termes et prédicats en Prolog.
- Sémantique déclarative:
 - Logique du premier ordre.
 - Interpretations, Modèles.
- Relations entre les sémantiques.

2ème partie

- Les listes en Prolog.

3

MCours.com

1

PLAN de la 1ère séance

- Les origines du Prolog.
- Deux exemples de programmes.
- Syntaxe
 - Constantes, variables, termes, prédicats.
 - Assertions, Règles, Buts.
- Sémantique.
 - Unification.
 - Arbres de dérivation.
- Les entiers en Prolog.

Sémantique opérationnelle: arbres de dérivation

On considère un programme $P=c_1, \dots, c_k$ et un but $G=g_1, \dots, g_l$.

Soit $c_i = \tau_i :- b_i^1, b_i^2, \dots, b_i^{l_i}$.

(si $l_i = 0$ alors c_i est une assertion, sinon c'est une règle).

On définit noeuds et arcs de l'arbre de dérivation de G pour P par induction (sur la hauteur):

- la racine est G .
- Soit $H=h_1, \dots, h_l$ un noeud de hauteur n , et soit c_s une clause de P dont la tête τ_s s'unifie avec h_1 , avec mgu σ .
Alors on crée le noeud, de hauteur $n + 1$, $H' = b_s^1\sigma, \dots, b_s^{l_s}\sigma, h_2\sigma, \dots, h_l\sigma$ et on étiquette l'arc de H à H' par σ .

Une feuille est soit un but vide (succes), soit un but dont le premier prédicat ne s'unifie avec aucune tête de clause (echec).

4

PLAN

- Sémantique opérationnelle d'un programme logique (rappel).
- Termes et prédicats en Prolog.
- Sémantique déclarative d'un programme logique:
 - Termes et formules de la logique du premier ordre.
 - Interpretations, Modèles.
- Relations entre les sémantiques operationnelle et declarative.

2ème partie

- Les listes en Prolog.

2

PLAN

- Sémantique opérationnelle (rappel).
- Termes et prédicats en Prolog.
- Sémantique déclarative:
 - Logique du premier ordre.
 - Interpretations, Modèles.
- Relations entre les sémantiques.

2ème partie

- Les listes en Prolog.

7

Comment modéliser un problème

Les entités, les objets dont on parle: les termes, construits à l'aide des symboles de fonction.

Les relations entre termes: les formules, construites à l'aide des symboles de prédicats.

exemple du td1 :Adam aime les pommes. Clara aime les carottes. Olivier aime les oranges. Les pommes sont des fruits. Les oranges sont des fruits. Les carottes sont des légumes. Ceux qui aiment les fruits sont en bonne santé.

Les "objets" dont on parle (symboles de fonction):

adam/0, pommes/0, clara/0, carottes/0, olivier/0,
oranges/0

Les relations entre objets (symboles de prédicat):

fruit/1, legume/1, aime/2, bonne_sante/1

8

Sémantique opérationnelle: arbres de dérivation

L'exécution d'un goal G pour un programme P a comme résultat l'ensemble de feuilles succes de l'arbre de dérivation correspondant.

Plus précisément, pour chacune de ces feuilles, le résultat est l'ensemble des instantiations des variables de G qui se trouvent sur le chemin qui mène de la racine à la feuille en question.

exemple:

$p(X, c) : \neg q(X) .$

$p(a, b) .$

$q(X) : \neg t(X) .$

$t(b) .$

$p(a, a) : \neg t(a) .$

$q(c) .$

$G = p(X, Y) .$

5

6

Un choix alternatif

Les objets dont on parle (symboles de fonction):

adam/0, pommes/0, clara/0, carottes/0, olivier/0,
oranges/0, fruit/0, legume/0

Les relations entre objets (symboles de prédicat):

aime/2, bonne_sante/1, est_instance_de/2

Le programme correspondant:

```
aime(adam,pommes).
aime(clara,carottes).
aime(olivier,oranges).
est_instance_de(pommes,fruit).
est_instance_de(oranges,fruit).
est_instance_de(carottes,legumes).
bonne_sante(X):-aime(X,Y),est_instance_de(Y,fruit).
```

Comment modéliser un problème

Le programme correspondant à ce choix de termes et prédicats:

```
aime(adam,pommes).
aime(clara,carottes).
aime(olivier,oranges).
fruit(pommes).
fruit(oranges).
legumes(carottes).
bonne_sante(X):-aime(X,Y),fruit(Y).
```

Comment modéliser un problème (4)

Les symboles de prédicat:

hauteur/2, taille/2

Assertions et règles:

```
hauteur(feuille,0).
hauteur(noeud(T1,T2), R):-hauteur(T1,H1),hauteur(T2,H2),
                           max(H1,H2,S), R is S+1.

taille(feuille,0).
taille(noeud(T1,T2), R):-taille(T1,R1),taille(T2,R2),
                           R is R1 + R2 + 1
```

15

Comment modéliser un problème (5)

Prolog n'effectue aucun "contrôle de type"; on peut lancer la requête

```
noeud(taille(X), hauteur(Z,toto), arg)
```

pour le programme précédent, et il n'y aura aucun message d'erreur (juste une réponse négative).

Nous verrons que, dans certains cas, il est utile de mélanger le niveau des termes et celui des prédicats : des prédicats peuvent avoir d'autres prédicats comme argument.

Exemple: `findall(X,bio(X,-,-,henri4,-),L)`.

Pour l'instant, il convient de garder les deux niveaux *strictement* séparés.

16

Comment modéliser un problème (2)

Un deuxième exemple, où les termes ne sont pas forcément des constantes. Voici ceux qu'il faut modéliser:

Un *arbre binaire* est soit une *feuille*, soit un *noeud* dont les deux fils sont des arbres binaires.

La *hauteur* d'une feuille est 0, et la hauteur d'un noeud est le max des hauteurs de ses fils plus 1.

La *taille* d'une feuille est 0, et la taille d'un noeud est la somme des tailles de ses fils plus 1.

13

Comment modéliser un problème (3)

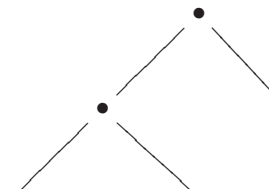
Les symboles de fonction:

feuille/0, noeud/2

Un nombre infini de termes. Par exemple

```
noeud(noeud(feuille,feuille),feuille)
```

qui désigne l'arbre :



14

Logique du 1er ordre: la syntaxe (2)

Définition: l'ensembles de **termes** sur un alphabet (F, P) est défini inductivement par:

- tout élément de V est un terme.
- tout élément de F d'arité 0 (donc toute constante) est un terme.
- Si t_1, \dots, t_n sont des terms et $f/n \in F$, alors $f(t_1, \dots, t_n)$ est un terme.

Voici quelques termes sur l'alphabet F_{int}, P_{int} :

$x, zero, succ(succ(zero)), succ(succ(succ(z)))$

19

Logique du 1er ordre: la syntaxe (3)

Définition: l'ensembles de **formule** sur un alphabet (F, P) est défini inductivement par:

- Si $p/n \in P$ et t_1, \dots, t_n sont des termes, alors $p(t_1, \dots, t_n)$ est une formule (un **atome**).
- Si F et G sont des formules, alors $\neg F, F \vee G, F \wedge G, F \rightarrow G$ sont des formules.
- Si $x \in V$ et F est une formule, alors $\forall x F$ et $\exists x F$ sont des formules.

Voici quelques formules sur l'alphabet F_{int}, P_{int} :

$pair(x), pair(zero) \wedge (\neg pair(succ(zero))),$
 $diviseur(succ(succ(zero)), succ(succ(succ(zero))))),$
 $\forall x(pair(x) \rightarrow diviseur(succ(succ(0)), x)).$

20

PLAN

- Sémantique opérationnelle (rappel).
- Termes et prédicats en Prolog.
- **Sémantique déclarative:**
 - Logique du premier ordre.
 - Interpretations, Modèles.
- Relations entre les sémantiques.

2ème partie

- Les listes en Prolog.

17

Logique du 1er ordre: la syntaxe (1)

Soit V un ensemble dénombrable de symboles de variable (par exemple $V = \{x, y, z, \dots, x_1, \dots\}$)

Définition: un **alphabet** (F, P) consiste de deux ensembles de symboles: fonctions et prédicat. Les éléments de F et P ont chacun une arité prédéfinie. Les éléments de F d'arité 0 sont les constantes (d'individu), les éléments de P d'arité 0 sont les constantes de prédicat.

Par exemple, un alphabet pour les entiers en notations unaires pourrait être: $F_{int} = \{zero/0, succ/1\}, P_{int} = \{pair/1, diviseur/2\}$

18

Vers une sémantique déclarative:

Le processus itératif qui consiste à unifier un but avec la tête d'une clause, modifier le but et recommencer, jusqu'à l'obtention de la clause vide ou à l'échec (décrit dans le premier cours) s'appelle **résolution**.

La résolution est complète pour les programmes logiques (Robinson, 1965), c.à.d. qu'elle permet de démontrer toutes les conséquences logiques d'un programme.

Mais comment définir exactement la notion de conséquence logique d'un programme, c'est à dire, la **sémantique déclarative** du programme? Il nous faut quelques notions de théories des modèles du 1er ordre.

23

Les clauses (1)

voici les formules autorisées dans un programme PROLOG:

A **clause unitaire** ou **fait** ou **assertion**

$A_1 \wedge \dots \wedge A_k \rightarrow A$ **clause de programme** ou **règle**

($A: -A_1, \dots, A_k$ dans la syntaxe de Prolog)

où A, A_1, \dots, A_k sont des **atomes**, c.à.d. des formules de la forme $p(t_1, \dots, t_n)$, p étant un symbole de prédicat à n places, et t_1, \dots, t_n des termes.

Chaque clause est implicitement préfixée par $\forall x_1 \dots \forall x_m$, où x_1, \dots, x_m sont les variables qui apparaissent dans la clause.

Par exemple $aime(x, y) \wedge fruit(y) \rightarrow bonne_sante(x)$ abrège $\forall x \forall y (aime(x, y) \wedge fruit(y) \rightarrow bonne_sante(x))$

Un programme logique est un ensemble fini de clauses de ces deux types.

21

MCours.com

PLAN

- Sémantique opérationnelle (rappel).
- Termes et prédicats en Prolog.
- **Sémantique déclarative:**
 - Logique du premier ordre.
 - **Interpretations, Modèles.**
- Relations entre les sémantiques.

2ème partie

- Les listes en Prolog.

24

22

Interprétations (2)

Soit \mathcal{I} une interprétation de (F, P) , de domaine D . Un **environnement** est une fonction $\rho : V \rightarrow D$

Définition: interprétation des termes

Soient \mathcal{I} une interprétation et $\rho : V \rightarrow D$ un environnement.

- $\mathcal{I}(x)_\rho = \rho(x)$
- $\mathcal{I}(c)_\rho = \llbracket c \rrbracket$, pour $c/0 \in F$.
- $\mathcal{I}(f(t_1, \dots, t_n))_\rho = \llbracket f \rrbracket(\mathcal{I}(t_1)_\rho, \dots, \mathcal{I}(t_n)_\rho)$, pour $f/n \in F$.

27

Interprétations (3)

Définition: interprétation des formules Soit \mathcal{I} une interprétation de et $\rho : V \rightarrow D$ un environnement.

- $\mathcal{I}(p(t_1, \dots, t_n))_\rho = \begin{cases} \text{vrai} & \text{si } (\mathcal{I}(t_1)_\rho, \dots, \mathcal{I}(t_n)_\rho) \in \llbracket p \rrbracket \\ \text{faux} & \text{sinon} \end{cases}$
- $\mathcal{I}(\neg A)_\rho, \mathcal{I}(A \vee B)_\rho, \mathcal{I}(A \wedge B)_\rho, \mathcal{I}(A \rightarrow B)_\rho$ sont définies en fonction de $\mathcal{I}(A)_\rho$ et $\mathcal{I}(B)_\rho$ avec les tables de vérité usuelles.
- $\mathcal{I}(\forall x A)_\rho = \text{vrai}$ si et seulement si pour tout $d \in D$
 $\mathcal{I}(A)_{\rho[x \leftarrow d]} = \text{vrai}$.
- $\mathcal{I}(\exists x A)_\rho = \text{vrai}$ si et seulement si il existe $d \in D$ tel que
 $\mathcal{I}(A)_{\rho[x \leftarrow d]} = \text{vrai}$.

28

Interprétations (1)

Définition: une interprétation de (F, P) consiste en:

- un ensemble D (le domaine de l'interprétation).
- Pour tout symbole de fonction f/n de F , une fonction $\llbracket f \rrbracket : D^n \rightarrow D$.
- Pour tout symbole de prédicat p/n de P , un ensemble $\llbracket p \rrbracket \subseteq D^n$.

25

Pour le langage des entiers unaires, une interprétation pourrait être: $D = \text{Nat}$

$$\llbracket \text{zero} \rrbracket = 0$$

$$\llbracket \text{succ} \rrbracket(n) = n + 1$$

$$\llbracket \text{pair} \rrbracket = \{n \mid n \text{ est pair}\}$$

$$\llbracket \text{diviseur} \rrbracket = \{(n, m) \mid n \text{ est un diviseur de } m\}$$

celle-ci est l'interprétation standard, mais celle qui suit est aussi une interprétation:

$$D = \{*\}; \llbracket \text{succ} \rrbracket(*) = *; \llbracket \text{pair} \rrbracket = \llbracket \text{diviseur} \rrbracket = \emptyset$$

26

PLAN

- Sémantique opérationnelle (rappel).
- Termes et prédicats en Prolog.
- Sémantique déclarative:
 - Logique du premier ordre.
 - Interpretations, Modèles.
- **Relations entre les sémantiques.**

2ème partie

- Les listes en Prolog.

31

Sémantique opérationnelle d'un programme logique

Définition: ensemble succes

Soit P un programme; l'ensemble succes de P est l'ensemble des atomes clos A tels que l'arbre de dérivation de racine A possède au moins une branche succes.

Exemple:

- L'ensemble succes du programme:
 $pair(Zero)$
 $pair(succ(succ(X)) \leftarrow pair(X)$
est l'ensemble d'atomes:
 $\{pair(Zero), pair(succ(succ(Zero))), \dots\}$

32

Modèles

Un modèle d'une formule F est une interprétation \mathcal{I} telle que, pour tout ρ , $\mathcal{I}(F)_\rho = vrai$.

Un modèle d'un ensemble de formule est une intepretation qui est un modèle de chaque formule de l'ensemble.

Définition: Un ensemble de formules S est:

satisfaisable s'il a un modèle.

valide si toute interprétation de S est un modèle.

insatisfaisable s'il n'a aucun modèle.

Définition: conséquence logique. Une formule F est conséquence logique d'un ensemble de formule S si tout modèle de S est aussi un modèle de F .

29

Sémantique déclarative d'un programme logique.

Définition:

Soit P un programme. La sémantique déclarative de P est l'ensemble des atomes clos qui sont conséquences logiques de P .

Exemple:

- La sémantique déclarative du programme:

$pair(Zero)$

$pair(succ(succ(X)) \leftarrow pair(X)$

est l'ensemble d'atomes:

$\{pair(Zero), pair(succ(succ(Zero))), pair(succ(succ(succ(succ(Zero))))), \dots\}$

30

PLAN

- Sémantique opérationnelle (rappel).
- Termes et prédicats en Prolog.
- Sémantique déclarative:
 - Logique du premier ordre.
 - Interpretations, Modèles.
- Relations entre les sémantiques.

2ème partie

- Les listes en Prolog.

35

Représentation des listes

- une liste est
 - soit la constante “liste vide”: le terme `[]`
 - soit un terme à deux arguments:
 - le premier élément (*head*) et le reste de la liste (*tail*).
- Le premier élément d’une liste est un terme quelconque, le reste doit être une liste.
- Le symbole fonctionnel `./2` est le constructeur de listes :
`.(Premier, Reste)`
- exemple:
`.(paul, .(pierre, .(marc, .(marie, []))))`
- une autre manière d’écrire la même liste:
`[paul,pierre,marc,marie]`

36

Correction et Complétude

Notations: soit P un programme, on note

$decl(P)$ sa sémantique déclarative.

$op(P)$ son ensemble succes (c.à.d. sa sémantique opérationnelle).

Théorème: correction et complétude de la sémantique opérationnelle vis à vis de la sémantique déclarative:

Pour tout programme logique P , $decl(P) = op(P)$.

33

Problèmes liés à la complétude:

Pour de raisons d’efficacité, l’ensemble des atomes clos sur lesquels l’interpréteur Prolog termine avec succes, que l’on note ici $ef(P)$, est un sous-ensemble strict de $op(P)$.

Exemple: soit P le programme

$p(X) \leftarrow p(X)$

$p(a)$

on a: $decl(P) = op(P) = \{p(a)\}$ mais $ef(P) = \emptyset$

Ceci peut être du à:

- Non-fairness de la règle de sélection des clauses (comme dans l’exemple précédent).
- Prédicats primitifs “extra-logiques” qui “coupent” des parties de l’arbre de dérivation.

34

En Prolog:

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

39

Opérations sur les listes (2)

Concaténation:

- On définit la relation: `concat(L1, L2, L3)` telle que `concat(L1, L2, L3)` est vrai si et seulement si la liste L3 est la concatenation de L1 et L2.
- Pour définir `concat(L1, L2, L3)` on peut considérer deux cas, en fonction de la forme du premier argument L1:
 - Si L1 est vide, alors L3 doit être égale à L2 (cas de base).
 - Sinon, L1 est de la forme `[X | L]`. Dans ce cas, le résultat est la liste `[X|L3]` où L3 est la concaténation de L et L2 (cas récursif).
- Ça donne en PROLOG:

```
concat( [], L, L).  
concat( [X | L], L2, [X | L3]) :- concat( L, L2, L3).
```

40

Représentation des listes (2)

- On peut aussi utiliser la notation `[Premier | Reste]` au lieu de `.(Premier, Reste)`
- On peut énumérer plusieurs éléments avant le `|`.
Par exemple:

```
[a,b,c] = [a | [b, c]] =  
[a, b | [c]] = [a, b, c | []]
```

37

Opérations sur les listes

- On définit la relation d'appartenance `membre(X,L)`.
- Le but `membre(X,L)` doit réussir si X est un des éléments de la liste L.
- `membre(b,[a, b, c])` et `membre([b, c],[a, [b, c]])` sont vrais.
`membre(b,[a, [b,c]])` est faux.
- On peut observer que X est un membre de L, si
 - X est le premier élément de L (cas de base), ou
 - X est membre du reste de L (cas récursif)

38

```
?- concat(L1,L2,[3,4,b]).
```

```
L1 = []
L2 = [3, 4, b] ;
```

```
L1 = [3]
L2 = [4, b] ;
```

```
L1 = [3, 4]
L2 = [b] ;
```

```
L1 = [3, 4, b]
L2 = [] ;
```

```
no
```

Exemples d'utilisation de concat:

```
?- concat([a,b,c],[1,2,3],[a,b,c,1,2,3]).
```

```
yes
```

```
?- concat([a,b,c],[1,2,3],L).
```

```
L = [a, b, c, 1, 2, 3]
```

```
?- concat([a,[b,c],[[]]],[a,[],L]).
```

```
L = [a, [b, c], [[]], a, []]
```

- Si X apparaît plusieurs fois dans L , des retours en arrière successifs donnent toutes les possibilités.
- `enlever` échoue si X n'appartient pas à la liste.

47

Exemples d'utilisation de `enlever`:

```
2 ?- enlever( a, [a, b, b, a, b], L).
```

```
L = [b, b, a, b] ;
```

```
L = [a, b, b, b] ;
```

no

```
3 ?- enlever( X, [a, c, c], L).
```

```
X = a
```

```
L = [c, c] ;
```

```
X = c
```

```
L = [a, c] ;
```

48

Appartenance en utilisant `concat`

On peut re-définir la relation `membre` en utilisant `concat`.

```
membre1( X, L) :-
    concat( L1, [X | L2], L).
```

ou

```
membre1( X, L) :-
    concat( _, [X | _], L).
```

45

Enlever un élément:

- Enlever l'élément X d'une liste L peut être réalisé par un prédicat `enlever(X,L,L1)` où $L1$ est la liste L privée de l'élément X
- Il y a deux cas:
 - Si X est le premier de L , alors le résultat est le reste de L (cas de base).
 - Si X est dans le reste de L , alors il est enlevé du reste (cas récursif).
- En Prolog:

```
enlever( X, [X | Q], Q).
```

```
enlever( X, [Y | Q], [Y | Q1]) :- enlever(X, Q, Q1).
```

46

Insérer un élément:

- Avec `enlever`, on peut définir `insérer(X,L,L1)` qui insère l'élément `X` dans la liste `L`, donnant la liste `L1` comme résultat.

```
insérer(X,L,L1) :-  
    enlever(X,L1,L).
```

- On peut aussi définir la relation `membre` en utilisant `enlever`. Un terme `X` est membre d'une liste si on peut l'enlever de cette liste.

```
membre2( X, L ) :-  
    enlever( X, L, _).
```

51

Le module `lists` de YAP

```
use_module(library(lists)).
```

7.5 List Manipulation

The following list manipulation routines are available once included with the `use_module(library(lists))` command.

```
append(?Prefix,?Suffix,?Combined)
```

True when all three arguments are lists, and the members of `Combined` are the members of `Prefix` followed by the members of `Suffix`. It may be used to form `Combined` from a given `Prefix`, `Suffix` or to take a given `Combined` apart.

```
delete(+List, ?Element, ?Residue)
```

True when `List` is a list, in which `Element` may or may not occur, and `Residue` is a copy of `List` with all elements identical to `Element` deleted.

52

```
X = c  
L = [a, c] ;  
no
```

49

Exemples d'utilisation de `enlever`:

```
4 ?- enlever( a, L, [c, d, e]).
```

```
L = [a, c, d, e] ;
```

```
L = [c, a, d, e] ;
```

```
L = [c, d, a, e] ;
```

```
L = [c, d, e, a] ;
```

```
no
```

50

[H—], H).

`nth0(?N, ?List, ?Elem, ?Rest)`

Unifies Elem with the Nth element of List, counting from 0, and Rest with the other elements. It can be used to select the Nth element of List (yielding Elem and Rest), or to insert Elem before the Nth (counting from 1) element of Rest, when it yields List, e.g. `nth0(2, List, c, [a,b,d,e])` unifies List with [a,b,c,d,e]. `nth/4` is the same except that it counts from 1. `nth0/4` can be used to insert Elem after the Nth element of Rest.

`nth(?N, ?List, ?Elem, ?Rest)`

Unifies Elem with the Nth element of List, counting from 1, and Rest with the other elements. It can be used to select the Nth element of List (yielding Elem and Rest), or to insert Elem before the Nth (counting from 1) element of Rest, when it yields List, e.g. `nth(1, List, c, [a,b,d,e])` unifies List with [a,b,c,d,e]. `nth/4` can be

55

used to insert Elem after the Nth element of Rest.

`permutation(+List, ?Perm)`

True when List and Perm are permutations of each other.

`remove_duplicates(+List, ?Pruned)`

Removes duplicated elements from List. Beware: if the List has non-ground elements, the result may surprise you.

`reverse(+List, ?Reversed)`

True when List and Reversed are lists with the same elements but in opposite orders.

`same_length(?List1, ?List2)`

True when List1 and List2 are both lists and have the same number of elements. No relation between the values of their elements is implied. Modes `same_length(-,+)` and `same_length(+,-)` generate either list given the other; mode `same_length(-,-)` generates two lists

56

`flatten(+List, ?FlattenedList)`

Flatten a list of lists List into a single list FlattenedList.

`?- flatten([[1],[2,3],[4,[5,6],7,8]],L).`

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

`no`

`is_list(+List)`

True when List is a proper list. That is, List is bound to the empty list (nil) or a term with functor '.' and arity 2.

`last(+List, ?Last)`

True when List is a list and Last is identical to its last element.

`list_concat(+Lists, ?List)`

True when Lists is a list of lists and List is the concatenation of Lists.

`member(?Element, ?Set)`

53

True when Set is a list, and Element occurs in it. It may be used to test for an element or to enumerate all the elements by backtracking.

`memberchk(+Element, +Set)`

As `member/2`, but may only be used to test whether a known Element occurs in a known Set. In return for this limited use, it is more efficient when it is applicable.

`nth0(?N, ?List, ?Elem)`

True when Elem is the Nth member of List, counting the first as element 0. (That is, throw away the first N elements and unify Elem with the next.) It can only be used to select a particular element given the list and index. For that task it is more efficient than `member/2`

`nth(?N, ?List, ?Elem)`

The same as `nth0/3`, except that it counts from 1, that is `nth(1,`

54

of the same length, in which case the arguments will be bound to lists of length 0, 1, 2, ...

`select(?Element, ?Set, ?Residue)`

True when Set is a list, Element occurs in Set, and Residue is everything in Set except Element (things stay in the same order).

`sublist(?Sublist, ?List)`

True when both `append(.,Sublist,S)` and `append(S,.,List)` hold.

`suffix(?Suffix, ?List)`

Holds when `append(.,Suffix,List)` holds.

`sum_list(?Numbers, ?Total)`

True when Numbers is a list of numbers, and Total is their sum.

`sumlist(?Numbers, ?Total)`

True when Numbers is a list of integers, and Total is their sum.

The same as `sum_list/2`, please do use `sum_list/2` instead.

57

MCours.com

`max_list(?Numbers, ?Max)`

True when Numbers is a list of numbers, and Max is the maximum.

`min_list(?Numbers, ?Min)`

True when Numbers is a list of numbers, and Min is the minimum

58