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.

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.

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.

Sémantique opérationnelle: arbres de dérivation

On considère un programme $\texttt{P=c}_1\texttt{,...c}_k$ et un but $\texttt{G=g}_1\texttt{,...,g}_l$.

Soit $\texttt{c}_i\texttt{= t}_i\texttt{:- b}_i^1\texttt{,b}_i^2\texttt{,...,b}_i^{l_i}$.

(si $l_i = 0$ alors $\texttt{c}_i$ 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 $\texttt{H=h}_1\texttt{,...h}_l$ un noeud de hauteur n, et soit $\texttt{c}_s$ une clause de P dont la tête $\texttt{t}_s$ s'unifie avec $\texttt{h}_1$, avec mgu σ.

  Alors on crée le noeud, de hauteur $n+1$, $\texttt{H'= b}_s^1\sigma\texttt{,...,b}_s^{l_s}\sigma\texttt{,}$ $\texttt{h}_2\sigma\texttt{,...h}_l\sigma$ 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).

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 précisement, 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):-q(X).
p(a,b).
q(X):-t(X).
t(b).
p(a,a):-t(a).
q(c).


G= p(X,Y).
```

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.

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`

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).
```

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 (2)

Un deuxième exemple, où les termes ne sont pas forcement 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.

Comment modéliser un problème (3)

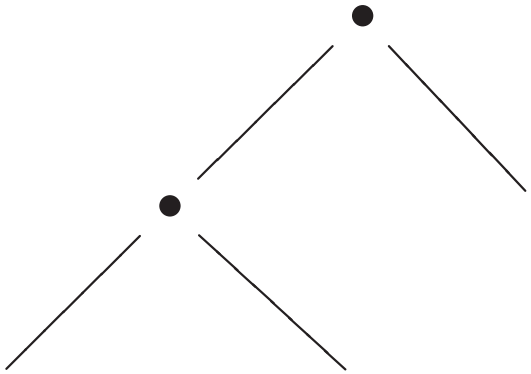
Les symboles de fonction:

`feuille/0, noeud/2`

Un nombre infini de termes. Par exemple

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

qui désigne l'arbre :



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
```

Comment modéliser un problème (5)

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

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

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

Dans certain cas, il est utile de melanger le niveau des termes et celui des prédicats : des prédicats peuvent avoir d'autres prédicats comme argument.

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

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.

Logique du 1er ordre: la syntaxe (1)

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

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: $F_{int} = \{zero/0, succ/1\}$, $P_{int} = \{pair/1, diviseur/2\}$

Logique du 1er ordre: la syntaxe (2)

Définition: l'ensembles de termes sur un alphabet F, P donné 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 $t_1, \ldots, t_n$ sont des terms et $f/n \in F$, alors $f(t_1, \ldots, t_n)$ est un terme.

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

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

Logique du 1er ordre: la syntaxe (3)

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

- Si $p/n \in P$ et $t_1, \ldots, t_n$ sont des termes, alors $p(t_1, ..., t_n)$ 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}:

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

Les clauses (1)

On introduit la classe de formule qui correspond aux clauses des programmes PROLOG:

Définition: un littéral est un atome ou la négation d'un atome. Une clause est une formule de la forme:
$\forall x_1 \ldots \forall x_n (L_1 \vee \ldots \vee L_k)$
où les L_i sont des littéraux et les x_j sont toutes et seules les variables de la formule.

notation: la clause
$\forall x_1 \ldots \forall x_n (A_1 \vee \ldots \vee A_j \vee \neg B_1 \ldots \vee \neg B_l)$
où $A_1, ..., A_j, B_1, ..., B_l$ sont des atomes, est notée
$A_1, \ldots, A_j \leftarrow B_1, ..., B_l$.

La clause ci-dessus se lit: si B_1 et ... et B_l alors A_1 ou ... ou A_j.

Les clauses (2)

Les programmes sont composé de deux types de clauses:

$A \leftarrow B_1, \ldots, B_k$ clause de programme ou règle

$A \leftarrow$ clause unitaire ou fait ou assertion

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

Vers une sémantique declarative:

Le processus itératif qui consiste à unifier le but avec la tête d'une clause, modifier le but et recommencer, jusqu'à l'obtention de la clause vide ou à l'echec (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 dir, la sémantique declarative du programme? Il nous faut quelques notions de théories des modèles du 1er ordre.

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.

Interprétations (1)

Définition: une intérpretation d'un langage du premier ordre $L = F, P$ consiste en:

- un ensemble D (le domaine de l'interprétation).
- Pour tout symbole de fonction f/n de L, une fonction $[\![f]\!] : D^n \to D$.
- Pour tout symbole de prédicat p/n de L, un ensemble $[\![p]\!] \subseteq D^n$.

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

$[\![zero]\!] = 0$

$[\![succ]\!](n) = n + 1$

$[\![pair]\!] = \{n \mid n \text{ est pair}\}$

$[\![diviseur]\!] = \{(n, m) \mid n \text{ est un diviseur de } m\}$

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

$D = \{*\}$; $[\![succ]\!](*) = *$; $[\![pair]\!] = [\![diviseur]\!] = \emptyset$

Interprétations (2)

Soit $\mathcal{I}$ une interprétation de $L = F, P$, de domaine D. Un environnement est une fonction $\rho : V \to D$

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

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

Interprétations (3)

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

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

Modèles

Un modèle d'une formule F est une interprétation $\mathcal{I}$ telle que, pour tout ρ, $\mathcal{I}(F)_\rho = vrai$.
Un modéle d'un ensemble de formule est une inteprétation qui est un modéle de chaque formule de l'ensemble.

Définition: Un ensemble de formules S est:
satisfaisable s'il a un modéle.
valide si toute interprétation de S est un modèle.
insatisfaisable s'il n'a aucun modéle.

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.

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.

Example:

- La sémantique déclarative du programme:

  $pair(zero)$

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

  est l'ensemble d'atomes:

  $\{pair(zero), pair(succ(succ(zero))), pair(succ(succ(succ(succ(zero))))), ...\}$

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.

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.

Example:

- L'ensemble succes du programme:

  $pair(zero)$

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

  est l'ensemble d'atomes:

  $\{pair(zero), pair(succ(succ(zero))), ...\}$

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 semantique opérationnelle vis à vis de la sémantique déclarative:
Pour tout programme logique P, $decl(P) = op(P)$.

Problèmes liés à la complétude:

Pour de raisons d'efficacité, l'ensemble des atomes clos sur lesquels l'intrepréteur 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 précedent).
- Prédicats primitifs "extra-logiques" qui "coupent" des parties de l'arbre de dérivation.

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.

Représentation des listes

- une liste est
  - soit la constante “liste vide”: le terme `[]`
  - soit un terme à deux arguments:
    le premier élément (la tête) et le reste de la liste (la queue).
- La tête peut être tout terme Prolog, la queue doit être une liste.
- La tête et la queue sont combinées par le symbole fonctionnel `./2`: `.( Tete, Queue)`
- exemple:
  `.( paul, .( pierre, .( marc, .(marie, [])))))`
- une autre manière d’écrire la même liste:
  `[paul,pierre,marc,marie]`

Représentation des listes (2)

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

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

Opérations sur les listes

- On définit la relation d'appartenance comme `membre(X,L)` où X est un objet et L une liste.
- Le but `membre(X,L)` est vrai si X est membre de 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 la tête de L, ou
  - X est membre de la queue de L
- En Prolog:

```
membre( X, [X | Queue]).
membre( X, [Tete | Queue]) :-
     membre( X, Queue).
```

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 L2 et L3 doivent être les mêmes.
  - Si L1 est non vide, alors L1 est de la forme `[X | L]`. Dans ce cas concaténer L1 avec L2 donne une liste `[X|L3]` où L3 est la concaténation de L avec L2.
- Ça donne en PROLOG:

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

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]

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


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

Yes
```

```
?- 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
```

Appartenance en utilisant concat

On peut définir la relation `membre` aussi avec `concat`.

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

ou

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

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 sans l'élément X
- Il y a deux cas:
  - Si X est la tête de la liste, alors le résultat est la queue de la liste.
  - Si X est dans la queue, alors il est enlevé de la queue.
- En Prolog:

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

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

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

Exemples d'utilisation pour 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] ;
```

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

Exemples d'utilisation pour 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
```

Insérer un élément:

- Avec enlever on peut définir `inserer(X,L,L1)` qui insère l'élément X quelque part dans la liste L en donnant L1.

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

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

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

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.

```
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)
```

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,

[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

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

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.

`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