Le langage Prolog

- Structures de données
- Opérateurs et expressions arithmétiques
- Evaluation des expressions et prédicats de comparaison
- La coupure et la négation
- Les prédicats retardés
- Entrées-sorties

Structures de données

- Déjà vu : objets atomiques (nombres et variables)
- Il existe aussi une structure de données plus complexe appelée **arbre Prolog** ou **structure Prolog**. Prolog connaît aussi les **listes**. Ces structures feront l'objet des prochains cours.

Les opérateurs (1)

- Ils permettent d'utiliser une syntaxe infixée, préfixée ou suffixée pour écrire des prédicats.
- Ce terme n'est pas réservé aux opérateurs arithmétiques (+, - , * et / sont des opérateurs infixés).
- Ils ont des priorités différentes.

Les opérateurs (2)

- Trois types :
  * Opérateur binaire infixé : il figure entre ses 2 arguments et désigne un arbre binaire
    X + Y
  * Opérateur unaire préfixé : il figure avant son argument et désigne un arbre unaire
    -X
  * Opérateur unaire suffixé : il figure après son argument et désigne un arbre unaire (Ex : *X*^2)

Les opérateurs (3)

- Associativité :
  - A gauche yfx:
    - *X op Y op Z* est lu comme *(X op Y) op Z*
  - A droite xfy:
    - *X op Y op Z* est lu comme *X op (Y op Z)*
  - Non associatif xfx: les parenthèses sont obligatoires
    - la syntaxe *X op Y op Z* est interdite

Les opérateurs (4)

- Déclaration des opérateurs :
  - possibilité de modifier la syntaxe Prolog en définissant de nouveaux opérateurs
  - définition : par l'enregistrement de faits de la forme suivante **op(Priorite, Specif, Nom)**
    - Nom : nom de l'opérateur
    - Priorite :compris entre 0 (le plus prioritaire) et 1200
    - Specif :type de l'opérateur (infixé, associatif…)

Les opérateurs (5)

- Déclaration des opérateurs :
  - possibilité de modifier la syntaxe Prolog en définissant de nouveaux opérateurs
  - définition : par l'enregistrement de faits de la forme suivante **op(Priorite, Specif, Nom)**
    - Nom : nom de l' Un opérateur possède une **précédence** (1..1200)
  - par exemple : **+** a une plus forte précédence que */* car

    a+b/c se lit **a+(b/c)**

Les opérateurs (7)

- Exemple de déclaration d'un opérateur :
  - *op(1000, xfx, aime)* définit un opérateur infixé non associatif *aime*
  - Dans ce cas, Prolog traduira une expression du type *X aime Y* en le terme *aime(X, Y)*, et si Prolog doit afficher le terme *aime(X, Y)*, il affichera *X aime Y*.
  - Exercice: définissez les opérateurs *et* (conjonction), *ou* (disjonction), *non* (négation) et => (implication) afin de pouvoir écrire par ex : *(A ou non A)*...

Les expressions arithmétiques (1)

- Prolog connaît les entiers et les nombres flottants.
- Les expressions arithmétiques sont construites à partir de nombres, de variables et d'opérateurs arithmétiques.
- Evaluation : par l'utilisation de l'opérateur **is** par exemple dans *X is 3 - 2 (présuppose un membre droit instancié)*

Les expressions arithmétiques (2)

- Opérations habituelles : addition, soustraction, multiplication, division entière (symbole //), division flottante (symbole /).
- Selon les systèmes Prolog, différentes fonctions mathématiques comme abs(X), ln(X), sqrt(X)
- Représentées par des arbres Prolog

Les expressions arithmétiques (3)

- Expressions et unification : attention à certaines tentatives d 'unification
  - la tentative d'unification entre *3+2* et *5* échouera. En effet, l'expression *3+2* est un arbre alors que 5 est un nombre.
  - L'évaluation des expressions ne fait pas partie de l'algorithme d'unification.

Les prédicats de comparaison

- Comparaison des expressions arithmétiques
  - X =:= Y se traduit par X et Y ont même valeur
  - X =\= Y est la négation du précédent
  - X < Y
  - X =< Y
  - X > Y
  - X >= Y
- Il y a évaluation puis comparaison.

Notion de coupure (1)

- Différents noms : **coupure**, **cut**
- Introduit un contrôle du programmeur sur l'exécution de ses programmes
  - * en élaguant les branches de l'arbre de recherche
  - * rend les programmes plus simples et efficaces
- Différentes notations : **!** ou **/** sont les plus courantes

Notion de coupure (2)

- Le coupure permet de signifier à Prolog qu'on ne désire pas conserver les points de choix en attente
- Utile lors de la présence de clauses exclusives
  - * Ex : les 2 règles suivantes
    - * *humain(X) :- homme(X).* s'écrit *humain(X) :- homme(X)!.*
    - * *humain(X) :- femme(X).* s'écrit *humain(X) :- femme(X)!.*

Notion de coupure (3)

- Le coupure permet :
  - * d'éliminer des points de choix
  - * d'éliminer des tests conditionnels que l'on sait inutile
    => plus d'efficacité lors de l'exécution du programme
- Quand Prolog démontre un coupure,
  - * il détruit tous les points de choix créés depuis le début de l'exploitation du paquet des clauses du prédicat où le coupure figure.

Notion de coupure (4)

- Exemples d'utilisation :
  - * Pour prouver que *a* est un élément d'une liste, on peut s'arrêter dès que la première occurrence de *a* a été trouvée.
  - * Pour calculer la racine carrée entière d'un nombre, on utilise un générateur de nombres entiers *entier(k)*. L'utilisation du coupure est alors indispensable après le test K*K>N car la génération des entiers n'a pas de fin.

entier(0).

entier(N) :- entier(N1), N is N1+1.

*racine(N, R) :- entier(K), K*K >N, ! , R is K-1.*

Notion de coupure (5)

- Repeat et fail :
  - * Le prédicat *fail/0* est un prédicat qui n'est jamais démontrable, il provoque donc un échec de la démonstration où il figure.
  - * Le prédicat *repeat/0* est une prédicat prédéfini qui est toujours démontrable mais laisse systématiquement un point de choix derrière lui. Il a une infinité de solutions.
  - * L'utilisation conjointe de *repeat/0*, *fail/0* et du coupure permet de réaliser des boucles.

Notion de coupure (6)

- Dangers de la coupure :
  - – Considérez les programmes suivants :

    enfants(helene, 3). ---> *enfants(helene, 3) :- !.*

    enfants(corinne, 1). ---> *enfants(corinne,1) :- !.*

    enfants(X, 0). ---> *enfants(X, 0).*

    et l'interrogation *enfants(helene, N).* dans les deux cas.

    1) N=3 et N=0

    2) N=3 mais *enfants(helene, 0).* donne aussi *YES*.

    Le programme correct serait :

    enfants(helene,N) :- !, N=3.

Notion de coupure (fin)

- En résumé, les utilités du coupure sont :
  - * éliminer les points de choix menant à des échecs certains
  - * supprimer certains tests d'exclusion mutuelle dans les clauses
  - * permettre de n'obtenir que la première solution de la démonstration
  - * assurer la terminaison de certains programmes
  - * contrôler et diriger la démonstration

La négation (1)

- Problème de la négation et de la différence en Prolog
- Pas de moyen en Prolog de démontrer qu'une formule n'est pas déductible.

La négation (2)

- Négation par l'échec
  * Si *F* est une formule, sa négation est notée *not(F)* ou *not F*. L'opérateur *not* est préfixé associatif.
  * Prolog pratique la **négation par l'échec**, c'est-à-dire que pour démontrer *not(F)* Prolog va tenter de démontrer *F*. Si la démonstration de *F* échoue, Prolog considère que *not(F)* est démontrée.
  * Pour Prolog, *not(F)* signifie que la formule F n'est pas démontrable, et non que c'est une formule fausse. C 'est ce que l'on appelle **l'hypothèse du monde clos**.

La négation (3)

- Elle est utilisée pour vérifier qu'une formule n'est pas vraie.
- La négation par l'échec ne doit être utilisée que sur des prédicats dont les arguments sont déterminés et à des fins de vérification.
  * Son utilisation ne détermine jamais la valeur d'une variable

La négation (4)

- Dangers :
  * Considérez le programme suivant :

    p(a).
  * Et interrogez Prolog avec :

    ?- X = b, not p(X).

    YES {X = b}
  * Prolog répond positivement car *p(b)* n'est pas démontrable.

La négation (5)

* Par contre, interrogez le avec :

  ?- not p(X), X=b.

  NO
* Prolog répond négativement car *p(X)* étant démontrable avec *{X = a}*, *not p(X)* est considéré comme faux.
* => Incohérence qui peut être corrigée si l'on applique la règle vue précédemment : n'utiliser la négation que sur des prédicats dont les arguments sont déterminés.

La négation (6)

- Le coupure et la négation :

  not P :- call(P), !, fail.

  not P.

  * Le prédicat prédéfini call/1 considère son argument comme un prédicat et en fait la démonstration. Pour démontrer *not P*, Prolog essaie de démontrer *P* à l 'aide de *call(P) (équivaut à P,mais cette écriture augmente la lisibilité du programme)* . S'il réussit, une coupure élimine les points de choix éventuellement créés durant cette démonstration puis échoue. La coupure ayant éliminé la deuxième règle, c'est la démonstration de *not P* qui échoue. Si la démonstration de *call(P)* échoue, Prolog utilise la deuxième règle qui réussit.

La négation (7)

- L'unification :
  * Prédicat binaire infixé : *X = Y*
  * Pour démontrer *X = Y*, Prolog unifie *X* et *Y* ; s'il réussit, la démonstration est réussie, et le calcul continue avec les valeurs des variables déterminées par l'unification.

    ?- X = 2.

    YES

    ?- X = 2, Y = 3, X = Y.

    NO

La négation (fin)

- La **différence** est définie comme le contraire de l'unification
  * Elle est notée : *X \= Y*. Elle signifie que X et Y ne sont pas unifiables, et non qu'ils sont différents.
    * Ex : *Z \= 3.* Sera faux car Z et 3 sont unifiables.
  * Elle pourrait être définie comme:

    X \= Y :- not X = Y.

    ou

    X \= X :- !, fail.

    X \= Y.

Les prédicats retardés

- Prédicats *dif/2* et *freeze/2*
  * Introduits dans le système Prolog II par A. Colmerauer et M. Van Caneghem
  * Apportent une plus grande souplesse dans l'utilisation des prédicats arithmétiques
  * Préservent la modularité des algorithmes classiques de Prolog tout en permettant leurs optimisations

La différence retardée

- Comparaison entre dif(X, Y) et X\=Y
  * Pas de différence si X et Y sont instanciées
  * Si X ou Y n 'est pas instanciée, le prédicat *dif/2* autorise Prolog à retarder la vérification de la différence jusqu'à ce que les deux variables soient instanciées.
  * Permet de poser des contraintes de différence sur les variables. On parle de **contraintes passives** car elles sont simplement mémorisées.

Le gel des prédicats

- Généralisation de la différence retardée à n'importe quel prédicat.
- Syntaxe : *freeze(Variable, Formule)* signifie que cette formule sera démontrée lorsque cette variable sera instanciée.
  * Par exemple :

```
dif(X, Y) :- freeze(X, freeze(Y, X\=Y))
```

Les prédicats retardés (fin)

- Danger :
  * c'est à vous de vous assurer que toutes les formules seront effectivement démontrées. Certains Prolog signalent les prédicats qui n'ont pas été dégelés suite à la non-instanciation d'une variable
- En résumé :
  * disponibles dans tous les systèmes Prolog récents
  * permettent de retarder la démonstration d'une formule
  * utilisation dans la déclaration de contraintes passives
  * prolongement naturel : programmation par contraintes

Les entrées-sorties (1)

- Ecriture sur l'écran ou dans un fichier
- Lecture à partir du clavier ou d'un fichier
- Affichage de termes :
  * *write(1+2)* affiche *1+2*
  * *write(X).* affiche *X* sur l'écran, sous la forme *_245* si c'est le nom interne de la variable.
  * nl/0 permet de passer à la ligne
  * *tab/1* tel que tab(N) affiche N espaces

Les entrées-sorties (2)

- Affichage de termes (suite) :
  - * *display/1* agit comme *write/1* mais en affichant la représentation sous forme d'arbre
    - * Ex :

      write(3+4), nl, display(3+4), nl.

      Affiche :

      3+4

      +(3,4)

      YES

Les entrées-sorties (3)

- Affichage de caractères et de chaînes
  - * *put/1* s'utilise en donnant le code ASCII d'un caractère :

    put(97).

    affiche : *a*
- Prolog connaît aussi les chaînes de caractères. Elles sont notées entre " ". Mais pour Prolog la chaîne est une liste de codes ASCII des caractères la composant .

Les entrées-sorties (4)

- Lecture de termes :
  - * *read/1* admet n'importe quel terme en argument.
    - * Il lit un terme au clavier et l'unifie avec son argument. Le terme lu doit être obligatoirement suivi d'un point. Certains systèmes Prolog affichent un signe d'invite lorsque le prédicat read/1 est utilisé.
  - * Exemple :

    ?- read(X).

    : a(1,2).

    YES {X = a(1,2)}

Les entrées-sorties (5)

- Autre exemple :

```
calculateur :- repeat,              (*boucle*)
               read(X),             (*lecture expression*)
               eval(X,Y),           (*évaluation*)
               write(Y), nl,        (*affichage*)
               Y = fin, !.          (*condition d'arrêt*)
eval(fin, fin) :- !.                (*cas particulier*)
eval(X, Y) :- Y is X.               (*calcul d'expressions*)
```

  - * lit des expressions arithmétiques, les évalue et les imprime jusqu'à ce que l'utilisateur rentre « fin » au clavier.

Les entrées-sorties (6)

* Le prédicat *eval/2* traite le cas particulier de l'atome *fin*. Lorsque *fin* est lu, le coupure final est exécuté et met fin à la boucle. Sinon *Y = fin* échoue et le prédicat retourne en arrière jusqu'à *repeat/0*.
* Exemple d'utilisation :

```
?- calculateur.
: 2+3 .  (*noter l'espace entre 3 et . Pour éviter de
   penser qu'il sagit d'un réel*)
5
: fin.
fin
YES
```

Les entrées-sorties (7)

- Lecture de caractères :
  * *get/1* et *get0/1*. Tous les deux prennent en argument un terme unifié avec le code ASCII du caractère lu. Si l'argument est une variable, celle-ci prendra pour valeur le code ASCII du caractère lu. *get/1* ne lit que les caractères de code compris entre 33 et 126.
- Les fichiers :
  * Un fichier peut être ouvert en lecture ou écriture.

Les entrées-sorties (8)

* En écriture :
  * mode *write* : son contenu est effacé avant que Prolog y écrive.
  * mode *append* : Prolog écrira à partir de la fin du fichier.
* Ouverture d'un fichier : prédicat *open/3*
  * argument 1 : nom du fichier
  * argument 2 : mode d'ouverture *write*, *append* ou *read*
  * argument 3 : variable qui va recevoir un identificateur de fichier appelé *flux* ou *stream*. (dépend du système Prolog utilisé).

Les entrées-sorties (9)

- Tous les prédicats *read*, *write* et autres vus auparavant admettent un second argument : le flux identifiant le fichier.
  * Ex : *write(Flux, X).* où Flux est un identificateur de fichier
  * Ex : *read(Flux,X), get(Flux, X), get0(Flux,X)*
  * Fermeture du fichier : prédicat *close/1* qui prend en argument le flux associé au fichier.

Les entrées-sorties (fin)

- Exemple d'utilisation

```
ecrire(T) :-
        open(' a.prl ', append, Flux), (*ouverture*)
        write(Flux, T), nl(Flux),        (*écriture*)
        close(Flux).
  (*fermeture*)
```

Listes et Suites Finies

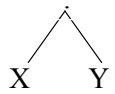
- Représentation
- Propriété
- Accès aux éléments
- Parcours et fonctions récursives,
- Sous-suites et arbres représentatifs

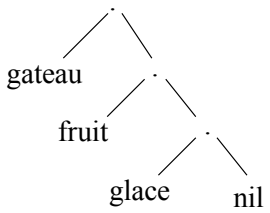
Représentation (1)

– La liste vide est notée []. Elle est la fin de n'importe quelle liste.
  - Il est préférable de noter la liste précédente sous la forme : *[gateau,fruit,glace]*
  - Les listes peuvent alors être représentées sous forme d 'arbres.

Représentation (2)

- Exemples :





Propriété fondamentale (1)

- Jusqu'à présent : arbres particuliers avec des branches gauches sans ramification.
  - On utilise le terme de **peigne** pour les désigner.
- Exercice :
  - résoudre l'équation [*X*|*Y*] = *[gateau,fruit,glace]*
  - par identification on a la solution :
    - {X = gateau; Y = [fruit,glace]}

Propriété fondamentale (2)

La notation [X|L] représente une liste dont la **tête** (le 1er élément) est X et la **queue** (le reste de la liste) est la liste L.

- Cela constitue la base de l'utilisation des listes dans les programmes Prolog.
- On accède aisément à la tête, au reste mais moins facilement au dernier élément ou au 3 élément, à cause de la structure de la liste.

Propriété fondamentale (3)

Exemple du programme *menu* :

entrees([sardine.pate.melon.avocat]).
viandes([poulet,roti,steack]).
poissons([merlan,colin,loup]).
desserts([gateau,fruit,glace]).
S'il n'y a pasd'autres clauses entrees() entrees permet ainsi de stocker une liste.

- Et si on pose la question :
  entrees(E).
- La résolution de ce but donne:
  E=[sardine,pate,melon,avocat]

Propriété fondamentale (4)

– Nous savons donc affecter une liste à une variable.
– Nous avons aussi besoin de pouvoir considérer les éléments d'une liste de façon individuelle, par exemple pour récupérer une entrée particulière.
  - Supposons que l'on dispose du prédicat :
    element-de(X, L) capable d'extraire un objet X d'un peigne L.

Propriété fondamentale (5)

– Nous avions :

menu(X, Y, Z) :- entree(X), plat(Y), dessert(Z).

plat(X) :- viande(X).

plat(X) :- poisson(X).

- Avec la nouvelle déclaration commune des entrées comme liste, ce programme ne convient plus. Pour cela nous allons construire une nouvelle définition du prédicat *entree* (ne pas confondre avec le prédicat *entrees*).

Propriété fondamentale (6)

- On utilise la propriété suivante : si X est une entrée, alors X est un élément quelconque de la liste des entrées. On peut donc écrire :

  entree(X) :- entrees(E), element-de(X, E).

- Et de façon analogue :

  viande(X) :- viandes(V), element-de(X, V).

  poisson(X) :- poissons(P), element-de(X, P).

  dessert(X) :- desserts(D), element-de(X, D).

Accès aux éléments d'une liste(1)

- Il s'agit de rechercher les règles qui vont assurer le fonctionnement du prédicat *element-de(X, L*) qui signifie que X est l'un des éléments de L.
  - Mauvaise méthode inspirée des tableaux: X est élément de L si X est le premier élément, ou le deuxième, … ou le dernier. Cela nécessite la connaissance de la longueur de la liste.

Accès aux éléments d'une liste(2)

- Autre méthode indépendante de la taille de L : on utilise la remarque selon laquelle toute liste peut se décomposer simplement en deux parties, la tête et la queue de liste. Cela conduit à distinguer deux cas :
  - X est élément de L si X est la tête de L.
  - X est élément de L si X est élément de la queue de L.
- Ce qui se traduit directement en Prolog par :
  - R1 : *element-de(X, L) :- est-tete(X, L).*
  - R2 : *element-de(X, L) :- element-de-la-queue(X,L).*
  - R1 et R2 sont là en tant que repères et ne rentrent pas dans les règles.

Accès aux éléments d'une liste(3)

- Mais on peut aller plus loin car L peut toujours être représentée sous la forme [*U|Y*], ce qui nous donne :
  - Si X est en tête de [U|Y] alors X = U.
  - Si X est un élément de la queue de [U|Y] alors X est élément de Y.
- D'où la version finale :
  - R1 : *element-de(X, X.Y).*
  - R2 : *element-de(X, [U|L1]) :- X\=U, element-de(X, L1).*

Accès aux éléments d'une liste(4)

- Commentaires :
  - On interprète ces deux règles de la façon suivante :
    - R1 : X est élément de toute liste qui commence par X.
    - R2 : X est élément de toute liste dont la queue est Y, si X est élément de Y.
  - Le second membre de R2 provoque un nouvel appel à R1 et R2. Il s'agit donc d'un appel récursif. La plupart des problèmes sur les listes ont des solutions mettant en jeu la récursivité.

Récursivité (1)

- On observe deux types de règles et deux types d'arrêt :
  - Une ou plusieurs règles provoquent des appels récursifs, qu'il vaut mieux placer en queue de clause. Dans notre exemple, il s'agit de la règle R2.
  - Une ou plusieurs règles stoppent la boucle. Dans notre exemple, c'est R1 qui s'en charge.
  - Ces règles d'arrêt sont placées en tête.

Récursivité (2)

- Conseils pour l'écriture de programmes récursifs :
  - Chaque règle doit être, sautant que possible, conçue comme une déclaration ou une définition, indépendamment de toute forme d'exécution.
  - L'ordre des règles peut et parfois doit prendre en compte l'ordre d'exécution du programme (et pas seulement pour les parties récursives du programme).

Récursivité (3)

- Il apparaît deux types d'arrêt :
  - Un arrêt explicite. Par exemple, dans R1, l'identité entre l'élément cherché et la tête de la liste fournie, ce qu'exprime la notation *X* et [*X*|*Y*].
  - Un arrêt implicite par exemple lors de la rencontre d'un but impossible à résoudre (on n'explore pas les buts qui suivent).

Construction d'une liste (1)

- Nous avons vu comment parcourir une liste.
- Nous allons voir comment en construire une.
  - Examinons le problème suivant : L une liste contenant un nombre pair d'éléments.
    - Par exemple : L = [0,1,2,3,4,5]
  - Cherchons à construire une nouvelle liste W en prenant les éléments de rang impair dans L.

Construction d'une liste (2)

- Dans notre exemple cela donnerait :
  - W = [0,2,4]
- Nous disposons de deux outils : le découpage d'une liste et la récursivité. Méthode à suivre :
  - Création d'un appel récursif qui parcourt la liste.
  - Modification des règles pour obtenir le résultat désiré.

Construction d'une liste (3)

- D'où le programme :

  R1 : elements-impairs([]).

  R2 : elements-impairs([U,V|L]) :- elements-impairs(L).
- Il reste à modifier le programme de façon à construire la nouvelle liste à partir du parcours de l'ancienne. D'où le programme final :

  R1 : elements-impairs([],[]).

  R2 : elements-impairs([U|M],[U,V|L])

  :- elements-impairs(M, L).

Construction d'une liste (4)

- Interprétation de ces deux règles :
  - R1 : prendre un élément sur deux dans la liste vide donne la liste vide
  - R2 : prendre un élément sur deux dans la liste [U,V|L] donne la liste [U|M] si prendre un élément sur deux dans la liste L donne la liste M.
- Remarque : les éléments dans la liste résultat sont rangés dans le même ordre que dans la liste initiale.

Construction d'une liste (5)

- Concaténation de listes:
  - Conc(Liste1,Liste2,Résultat).
  - Conc([],L,L).
  - Conc([X|L],M,[X|R]):- conc(L,M,R).

Construction d'une liste (6)

- Renverser
  - renverser([],[]).
  - *renverser([X|D],R) :-renverser(D,S), conc(S,[X],R).*

Listes et arbres

- Une liste peut être composées de listes de listes de listes …
- Arbre:
  [a,[b,c,d],[a,b]]

Un classique à la base de Prolog: l'algorithme d'unification

- Un problème P (ensemble d' équations) et une solution courante S
  - si P est vide, succès, on renvoie S ;
  - sinon on pioche une équation, plusieurs cas:
    - décomposition c (s1 , . . . sk) ?= c (t1 , . . . tk) on ajoute les équations si ?= ti `a P
    - conflit c (s1 , . . . , sk) ?= d (u1 , . . . , un) échec
    - trivial t ?= t , on continue avec P
    - élimination de variable X ?= t on continue avec P[X->t] , S[X -> t] si X pas dans Vars (t) [PAS EN PROLOG]
    - cyclicité X ?= t échec si X dans Vars (t) et t ≠ X
    - orientation t ?= X , où t n'est pas une variable on ajoute X ?= t à P
  - pour unifier, on part de P = {t1=t2 }, S = vide_˘