eBook Gratuit

APPRENEZ Prolog Language

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#prolog

Table des matières

À propos 1

Chapitre 1: Premiers pas avec le langage Prolog 2

Remarques 2

Les implémentations 2

Examples 2

Installation ou configuration 2

annexe / 3 3

Contraintes CLP (FD) 3

Programmation de base de données 4

Bonjour le monde 6

Bonjour, World dans l'interpréteur interactif 6

Bonjour, Monde d'un fichier 6

Chapitre 2: Arbres de dérivation 8

Examples 8

Arbre de preuve 8

Chapitre 3: Clauses définies de la clause (DCG) 10

Examples 10

Rien du tout: `// // 0` 10

Analyse avec les DCG 10

Objectifs supplémentaires 10

Arguments supplémentaires 11

Chapitre 4: Directives de codage 12

Examples 12

Appellation 12

Échancrure 12

Ordre des arguments 12

Chapitre 5: Gestion des erreurs et exceptions 14

Examples 14

Erreurs d'instanciation 14

Points généraux sur la gestion des erreurs 14

Nettoyage après exceptions 14
Erreurs de type et de domaine 14
Chapitre 6: Les opérateurs 16
Examples 16
Opérateurs prédéfinis 16
Déclaration de l'opérateur 17
Commande à terme 17
Égalité des termes 18
Chapitre 7: Liste de différences 19
Introduction 19
Examples 19
Utilisation de base 19
Evaluer une expression arithmétique 20
Chapitre 8: Monotonicité 22
Examples 22
Raisonnement sur les prédicats monotones 22
Exemples de prédicats monotones 22
Prédicats non monotones 22
Alternatives monotones pour les constructions non monotones 23
Combiner la monotonie à l'efficacité 23
Chapitre 9: Performance 25
Examples 25
Machine abstraite 25
Indexage 25
Optimisation des appels de queue 25
Chapitre 10: Prédicats Extra-Logiques 26
Examples 26
Prédicats avec effets secondaires 26
Prédicats méta-logiques 26
Prédicats de toutes les solutions 26
! / 0 et les prédicats associés 26
Chapitre 11: Programmation d'ordre supérieur 28

Examples 28
appel / N prédicats 28
mapliste / [2,3] 28
Meta-call 28
foldl / 4 28
Appeler une liste d'objectifs 29
Chapitre 12: Programmation logique par contraintes 30
Examples 30
CLP (FD) 30
CLP (Q) 30
CLP (H) 30
Chapitre 13: Pureté logique 32
Examples 32
dif / 2 32
Contraintes CLP (FD) 32
Unification 32
Chapitre 14: Raisonnement sur les données 33
Remarques 33
Examples 33
Récursivité 33
Accéder aux listes 35
Chapitre 15: Structures de contrôle 37
Examples 37
Disjonction (OU logique), implicite vs explicite 37
Conjonction (ET logique) 37
Couper (supprimer les points de choix) 37
Chapitre 16: Structures de données 39
Examples 39
Des listes 39
Paires 40
Listes d'association 40
termes 40

Termes avec champs nommés utilisant la bibliothèque (enregistrement) 40

Chapitre 17: Utiliser Prolog moderne 42

Examples 42

introduction 42

CLP (FD) pour l'arithmétique des nombres entiers 42

Forall au lieu de boucles pilotées par les pannes 43

Prédicats de fonction 44

Déclarations de flux / mode 45

Crédits 47

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: prolog-language

It is an unofficial and free Prolog Language ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Prolog Language.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Premiers pas avec le langage Prolog

Remarques

Les implémentations

1. SWI-Prolog (gratuit) swi-prolog
    - Implémenté dans c
2. SICStus (commercial) sicstus-prolog
3. YAP (gratuit) yap
4. GNU Prolog (gratuit) gnu-prolog
5. XSB (gratuit) xsb
6. B (commercial) b-prolog
7. SI (commercial)
8. Ciao (libre)
9. Minerva (commercial)
10. ECLiPSe-CLP (gratuit) eclipse-clp
11. Jekejeke Prolog (commercial)
12. Prolog IV
13. Rendement Prolog (gratuit)
    - Implémenté dans c # , javascript et phyton
14. Visual Prolog (commerciale) visuo-Prolog

Examples

Installation ou configuration

SWI-Prolog

Windows et Mac:

- Téléchargez SWI-Prolog sur le site officiel
- Installez simplement en suivant les instructions du programme d'installation.

Linux (PPA):

- Ajoutez le PPA `ppa:swi-prolog/stable` aux sources logicielles de votre système (les développeurs peuvent choisir `ppa:swi-prolog/devel`):
    - Ouvrez un terminal (Ctrl + Alt + T) et tapez: `sudo add-apt-repository ppa:swi-prolog/stable`
    - Ensuite, mettez à jour les informations sur le paquet: `sudo apt-get update`

- Maintenant, installez SWI-Prolog via le gestionnaire de paquets: `sudo apt-get install swi-prolog`
- Vous pouvez maintenant démarrer SWI-Prolog via la ligne de commande avec la commande `swipl`

annexe / 3

```
append([], Bs, Bs).
append([A|As], Bs, [A|Cs]) :-
    append(As, Bs, Cs).
```

`append/3` est l'une des relations Prolog les plus connues. Elle définit une relation entre trois arguments et est vraie *si* le troisième argument est une liste indiquant la concaténation des listes spécifiées dans les premier et deuxième arguments.

Notamment, et comme c'est généralement le cas pour un bon code Prolog, `append/3` peut être utilisé dans *plusieurs directions* : Il peut être utilisé pour:

- *ajouter* deux listes totalement ou partiellement instanciées:

  ```
  ?- A = [1, 2, 3], B=[4, 5, 6], append(A, B, Y)
  Output:
  A = [1, 2, 3],
  B = [4, 5, 6],
  Y = [1, 2, 3, 4, 5, 6].
  ```

- *vérifier* si la relation est vraie pour trois listes complètement instanciées:

  ```
  ?- A = [1, 2, 3], B = [4, 5], C = [1, 2, 3, 4, 5, 6], append(A, B, C)
  Output:
  false
  ```

- *générer* tous les moyens possibles pour ajouter deux listes à une liste donnée:

  ```
  ?- append(A, B, [1, 2, 3, 4]).
  Output:
  A = [],
  B = [1, 2, 3, 4] ;
  A = [1],
  B = [2, 3, 4] ;
  A = [1, 2],
  B = [3, 4] ;
  A = [1, 2, 3],
  B = [4] ;
  A = [1, 2, 3, 4],
  B = [] ;
  false.
  ```

Contraintes CLP (FD)

Les contraintes CLP (FD) sont fournies par toutes les implémentations sérieuses de Prolog. Ils

nous permettent de raisonner sur des **entiers** de manière pure.

```
?- X #= 1 + 2.
X = 3.

?- 5 #= Y + 2.
Y = 3.
```

Programmation de base de données

Prolog classe tout en:

- **Atomes** - Toute séquence de caractères qui ne commence pas par un alphabet majuscule. Par exemple - `a` , `b` , d' `okay`
- **Nombres** - Il n'y a pas de syntaxe spéciale pour les nombres, aucune déclaration n'est requise. Par exemple `1` , `22` , `35.8`
- **Variables** - Chaîne commençant par un caractère majuscule ou un trait de soulignement (_). Par exemple `X` , `Y` , `Abc` , `AA`
- **Termes complexes** - Ils sont fabriqués à partir d'un *foncteur* et d'une séquence d' *arguments* . Le nom d'un terme complexe est toujours un atome, alors que les arguments peuvent être des atomes ou des variables. Par exemple, `father(john,doe)` , `relative(a)` , `mother(X,Y)` .

Une base de données logique contient un ensemble de *faits* et de *règles* .

Un terme complexe avec uniquement des atomes comme arguments est appelé un fait, tandis qu'un terme complexe avec des variables comme arguments est appelé une règle.

Exemple de faits dans Prolog:

```
father_child(fred, susan).
mother_child(hillary, joe).
```

Exemple de règle dans Prolog:

```
child_of(X,Y):-
    father_child(Y,X)
    ;
    mother_child(Y,X).
```

Notez que le `;` ici est comme l'opérateur `or` dans d'autres langues.

Prolog est un langage déclaratif et vous pouvez lire cette base de données comme suit:

> *fred est le père de susan*
>
> *Hillary est la mère de Joe.*
>
> *Pour tout `X` et `Y` , `X` est un enfant de `Y` si `Y` est un père de `X` ou `Y` est une mère de `X`*

En fait, un ensemble fini de faits et / ou de règles constitue un *programme* logique.

L'utilisation d'un tel programme est démontrée par des *requêtes* . Les requêtes vous permettent de récupérer des informations à partir d'un programme logique.

Pour charger la base de données dans l'interpréteur (en supposant que vous avez enregistré la base de données dans le répertoire dans lequel vous exécutez l'interpréteur), il vous suffit d'entrer:

```
?- [nameofdatabase].
```

remplacer le `nameofdatabase` la `nameofdatabase` de `nameofdatabase` par le nom de fichier réel (notez que nous excluons ici l'extension `.pl` du nom de fichier).

Exemple de requêtes dans l'interpréteur pour le programme ci-dessus et les résultats:

```
?- child_of(susan,fred).
true

?- child_of(joe,hillary).
true

?- child_of(fred,susan).
false

?- child_of(susan,hillary).
false

?- child_of(susan,X).
X = fred

?- child_of(X,Y).
X = susan,
Y = fred ;
X = joe,
Y = hillary.
```

Les questions ci-dessus et leurs réponses peuvent être lues comme suit:

> *est susan un enfant de fred? - vrai*
>
> *Joe est-il un enfant de Hillary? - vrai*
>
> *fred est un enfant de susan? - faux*
>
> *est susan un enfant de hillary? - faux*
>
> *qui est un enfant de Susan? - fred*

Voici comment nous programmons la logique dans Prolog. Un programme logique est plus formel: un ensemble d'axiomes ou de règles définissant des relations (aka prédicats) entre objets. Une autre manière d’interpréter la base de données ci-dessus de manière plus formelle est la suivante:

> *La relation* `father_child` *tient entre fred et susan*
>
> *La relation* `mother_child` *tient entre hillary et joe*
>
> *Pour tout* `X` *et* `Y` *la relation* `child_of` *est* `child_of` *entre* `X` *et* `Y` *si la relation* `father_child` *entre* `Y` *et* `X` *, ou la relation* `mother_child` *entre* `Y` *et* `X`

Bonjour le monde

Bonjour, World dans l'interpréteur interactif

Pour imprimer "Bonjour, Monde!" dans l'interpréteur Prolog (nous utilisons ici `swipl` , le shell pour SWI Prolog):

```
$ swipl
<...banner...>
?- write('Hello, World!'), nl.
```

`?-` est l'invite du système: cela indique que le système est prêt pour que l'utilisateur entre une séquence d' *objectifs* (c.-à-d. une *requête*) devant être terminée par un `.` (arrêt complet).

Ici, la requête `write('Hello World!'), nl` a deux objectifs:

- `write('Hello World!')` : `'Hello World!'` doit être affiché **et** (`,`)
- une nouvelle ligne (`nl`) doit suivre.

`write/1` (le `/1` indique que le prédicat prend un argument) et `nl/0` sont *des prédicats intégrés* (la définition est fournie au préalable par le système Prolog). Les prédicats intégrés fournissent des fonctionnalités qui ne peuvent pas être obtenues par une définition de Prolog pure ou pour empêcher le programmeur de les définir.

Le résultat:

> Bonjour le monde!
>
> Oui

se termine par `yes` ce qui signifie que la requête a réussi. Dans certains systèmes, `true` est imprimé au lieu de `yes` .

Bonjour, Monde d'un fichier

Ouvrez un nouveau fichier appelé `hello_world.pl` et insérez le texte suivant:

```
:- initialization hello_world, halt.

hello_world :-
    write('Hello, World!'), nl.
```

La directive d' `initialization` spécifie que l'objectif `hello_world, halt` doit être appelé lorsque le fichier est chargé. `halt` quitte le programme.

Ce fichier peut alors être exécuté par votre exécutable Prolog. Les drapeaux exacts dépendent du système Prolog. Si vous utilisez SWI Prolog:

```
$ swipl -q -l hello_world.pl
```

Cela produira une sortie `Hello, World!` . L' `-q` supprime la bannière qui s'affiche généralement lorsque vous appelez run `swipl` . Le `-l` spécifie un fichier à charger.

Lire Premiers pas avec le langage Prolog en ligne:
https://riptutorial.com/fr/prolog/topic/1038/premiers-pas-avec-le-langage-prolog

Chapitre 2: Arbres de dérivation

Examples

Arbre de preuve

L'arbre de preuve (également arbre de recherche ou arborescence de dérivation) est une arborescence qui montre l'exécution d'un programme Prolog. Cet arbre permet de visualiser le processus de retour en arrière chronologique présent dans Prolog. La racine de l'arbre représente la requête initiale et les branches sont créées lorsque des points de choix apparaissent. Chaque nœud de l'arbre représente donc un objectif. Les branches ne deviennent des leafs que lorsque true / false a été prouvé pour le (les) jeu (s) d'objectifs requis (s) et que la recherche dans Prolog est effectuée de **gauche à droite** .

Prenons l'exemple suivant:

```
% Facts
father_child(paul,chris).        % Paul is the father of Chris and Ellen
father_child(paul,ellen).
mother_child(ellen,angie).       % Ellen is the mother of Angie and Peter
mother_child(ellen,peter).


% Rules
grandfather_grandchild(X,Y) :-
    father_child(X,Z),
    father_child(Z,Y).

grandfather_grandchild(X,Y) :-
    father_child(X,Z),
    mother_child(Z,Y).
```

Lorsque nous interrogeons maintenant:

```
?- grandfather_grandchild(paul,peter).
```

l'arbre de vérification suivant visualise le processus de recherche en profondeur:

```
                                  ?- grandfather_grandchild(paul,peter).
                                      /                              \
                                     /                                \
  ?- father_child(paul,Z1),father_child(Z1,peter).            ?-
father_child(paul,Z2),mother_child(Z2,peter).
             /                    \                                   /
\
      {Z1=chris}              {Z1=ellen}                        {Z2=chris}
{Z2=ellen}
           /                       \                                 /
\
?- father_child(chris,peter).  ?- father_child(ellen,peter).  ?- mother_child(chris,peter). ?-
mother_child(ellen,peter).
```

```
        |                          |                                |
/              \
       fail                       fail                            fail
fail(*)          success
```

(*) échoue pour `mother_child(ellen,angie)` où 'angie' ne correspond pas à 'peter'

Lire Arbres de dérivation en ligne: https://riptutorial.com/fr/prolog/topic/3097/arbres-de-derivation

Chapitre 3: Clauses définies de la clause (DCG)

Examples

Rien du tout: `// // 0`

L'un des non-terminaux DCG les plus élémentaires est `... //0` , qui peut être lu comme "n'importe quoi":

```
... --> [] | [_], ... .
```

Il peut être utilisé pour décrire une liste `Ls` contenant l'élément `E` via:

```
phrase(( ..., [E], ... ), Ls)
```

Analyse avec les DCG

Les DCG peuvent être utilisés pour l'analyse. Mieux encore, *le même* DCG peut souvent être utilisé pour analyser et *générer des* listes décrites. Par exemple:

```
sentence --> article, subject, verb, object.

article --> [the].

subject --> [woman] | [man].

verb --> [likes] | [enjoys].

object --> [apples] | [oranges].
```

Exemple de requêtes:

```
?- phrase(sentence, Ls).
Ls = [the, woman, likes, apples] ;
Ls = [the, woman, likes, oranges] ;
Ls = [the, woman, enjoys, apples] .

?- phrase(sentence, [the,man,likes,apples]).
true .
```

Objectifs supplémentaires

Des objectifs supplémentaires permettent d'ajouter du traitement aux clauses DCG, par exemple, les conditions que les éléments de la liste doivent satisfaire.

Les objectifs supplémentaires sont observés entre accolades à la fin d'une clause DCG.

```
% DCG clause requiring an integer
int --> [X], {integer(X)}.
```

Usage:

```
?- phrase(int, [3]).
true.

?- phrase(int, [a]).
false.
```

Arguments supplémentaires

Les arguments supplémentaires ajoutent des résultats aux prédicats d'une clause DCG, en décorant l'arbre de dérivation. Par exemple, il est possible de créer une grammaire algébrique qui calcule la valeur à la fin.

Étant donné une grammaire qui prend en charge l'ajout d'opération:

```
% Extra arguments are passed between parenthesis after the name of the DCG clauses.
exp(C) --> int(A), [+], exp(B), {plus(A, B, C)}.
exp(X) --> int(X).
int(X) --> [X], {integer(X)}.
```

Le résultat de cette grammaire peut être validé et interrogé:

```
?- phrase(exp(X), [1,+,2,+,3]).
X = 6 ;
```

Lire Clauses définies de la clause (DCG) en ligne:
https://riptutorial.com/fr/prolog/topic/2426/clauses-definies-de-la-clause--dcg-

Chapitre 4: Directives de codage

Examples

Appellation

Lors de la programmation dans Prolog, nous devons choisir deux types de noms:

- noms de **prédicats**
- noms de **variables** .

Un bon nom de *prédicat* indique clairement ce que signifie chaque argument. Par convention, les **traits de soulignement** sont utilisés dans les noms pour séparer la description des différents arguments. Cela est dû au fait que `underscores_keep_even_longer_names_readable` , tandis que `mixingTheCasesDoesNotDoThisToTheSameExtent` .

Voici des exemples de bons noms de prédicats:

- `parent_child/2`
- `person_likes/2`
- `route_to/2`

Notez que les noms *descriptifs* sont utilisés. Les impératifs sont évités. L'utilisation de noms descriptifs est recommandée car les prédicats Prolog peuvent généralement être utilisés dans *plusieurs* directions, et le nom doit également s'appliquer à tout ou aucun des arguments n'est instancié.

La capitalisation mixte est plus courante lors de la sélection des noms de *variables* . Par exemple: `BestSolutions` , `MinElement` , `GreatestDivisor` . Une convention courante pour nommer des variables qui indiquent des *états* successifs est d'utiliser `S0` , `S1` , `S2` , ..., `S` , où `S` représente l'état final.

Échancrure

Il y a peu de constructions de langage dans Prolog, et plusieurs façons de les indenter sont communes.

Quel que soit le style choisi, un principe à respecter est de **ne jamais** placer `(;)/2` à la *fin* d'une ligne. En effet `;` et `,` ressemblent beaucoup, et `,` se produit souvent à la fin d'une ligne. Par conséquent, les clauses qui utilisent une disjonction doivent par exemple être écrites comme suit:

```
(  Goal1
;  Goal2
)
```

Ordre des arguments

Idéalement, les prédicats Prolog peuvent être utilisés dans toutes les directions. Pour de

nombreux prédicats purs, cela est également le cas. Cependant, certains prédicats ne fonctionnent que dans des *modes* particuliers, ce qui signifie des schémas d'instanciation de leurs arguments.

Par convention, les arguments les plus courants pour ces prédicats sont:

- **les** arguments d' **entrée** sont placés en premier. Ces arguments doivent être instanciés *avant* l'appel du prédicat.
- *les paires* d'arguments qui appartiennent ensemble sont placées de manière adjacente, telles que `p(..., State0, State, ...)`
- **les** arguments de **sortie** prévus sont placés en dernier. Ces prédicats sont instanciés par le prédicat.

Lire Directives de codage en ligne: https://riptutorial.com/fr/prolog/topic/4612/directives-de-codage

Chapitre 5: Gestion des erreurs et exceptions

Examples

Erreurs d'instanciation

Une erreur d' **instanciation** est lancée si un argument n'est pas suffisamment *instancié* .

De manière critique, une erreur d'instanciation *ne peut pas* être remplacée par *une défaillance silencieuse* : en cas d' **absence de solution** , cela signifie qu'il n'ya **pas de solution** , alors qu'une erreur d'instanciation signifie qu'une *instance* de l'argument peut participer à une solution.

Cela contraste avec, par exemple **, une erreur de domaine** , qui peut être remplacée par une défaillance silencieuse sans modifier le sens déclaratif d'un programme.

Points généraux sur la gestion des erreurs

Prolog propose des **exceptions** , qui font partie de la norme ISO Prolog.

Une exception peut être lancée avec `throw/1` et prise avec `catch/3` .

La norme ISO définit de nombreux cas dans lesquels des erreurs doivent ou peuvent être émises. Les exceptions standardisées sont toutes des `error(E,_)` formulaire `error(E,_)` , où `E` indique l'erreur. Les exemples sont `instantiation_error` , `domain_error` et `type_error` , qui voient.

`setup_call_cleanup/3` est un prédicat important en relation avec les exceptions.

Nettoyage après exceptions

Le prédicat `setup_call_cleanup/3` , dont l'inclusion dans la norme ISO Prolog est en cours et fourni par un nombre croissant d'implémentations, nous permet de nous assurer que les ressources sont correctement libérées après la levée d'une exception.

Une invocation typique est:

```
setup_call_cleanup(open(File, Mode, Stream), process_file(File), close(Stream))
```

Notez qu'une exception ou une interruption peut même se produire immédiatement après l'appel de `open/3` dans ce cas. Pour cette raison, la phase d' `Setup` est effectuée de manière *atomique* . Dans les systèmes Prolog qui ne fournissent que `call_cleanup/2` , cela est beaucoup plus difficile à exprimer.

Erreurs de type et de domaine

Une **erreur de type** se produit si un argument n'est pas du *type* attendu. Des exemples de types sont:

- `integer`
- `atom`
- `list` .

Si le prédicat est du type attendu, mais en dehors du *domaine* attendu, une **erreur de domaine** est générée.

Par exemple, une erreur de domaine est admissible si un nombre entier compris entre 0 et 15 est attendu, mais l'argument est le nombre entier 20.

Déclarativement, une erreur de type ou de domaine équivaut à *une défaillance silencieuse* , car aucune instanciation ne peut aboutir à un prédicat dont l'argument est du type ou du domaine incorrect.

Lire Gestion des erreurs et exceptions en ligne: https://riptutorial.com/fr/prolog/topic/7114/gestion-des-erreurs-et-exceptions

Chapitre 6: Les opérateurs

Examples

Opérateurs prédéfinis

Opérateurs prédéfinis selon ISO / IEC 13211-1 et 13211-2:

Priorité	Type	Les opérateurs)	Utilisation
1200	xfx	`:- -->`	
1200	fx	`:- ?-`	Directive, requête
1100	xfy	`;`	
1050	xfy	`->`	
1000	xfy	`','`	
900	fy	`\+`	
700	xfx	`= \\=`	Unification du terme
700	xfx	`== \\== @< @=< @> @>=`	Comparaison de terme
700	xfx	`=..`	
700	xfx	`is =:= =\= < > =< >=`	Évaluation arithmétique et comparaison
600	xfy	`:`	Qualification de module
500	yfx	`+ - /\ \/`	
400	yfx	`* / div mod // rem << >>`	
200	xfx	`**`	Pouvoir flottant
200	xfy	`^`	Quantification variable, puissance entière
200	fy	`+ - \`	Identité arithmétique, négation; complément binaire

De nombreux systèmes fournissent d'autres opérateurs en tant qu'extension spécifique à la mise en œuvre:

Priorité	Type	Les opérateurs)	Utilisation
1150	fx	`dynamic multifile discontiguous initialization`	Directives standard

Priorité	Type	Les opérateurs)	Utilisation
1150	fx	`mode block volatile public meta_predicate`	
900	fy	`spy nospy`	

Déclaration de l'opérateur

Dans Prolog, les opérateurs personnalisés peuvent être définis avec `op/3` :

```
op(+Precedence, +Type, :Operator)
```

- Déclare l'opérateur comme étant un opérateur d'un type avec une priorité. L'opérateur peut également être une liste de noms, auquel cas tous les éléments de la liste sont déclarés comme étant des opérateurs identiques.
- La priorité est un entier compris entre 0 et 1200, où 0 supprime la déclaration.
- Le type est l'un de: `xf` , `yf` , `xfx` , `xfy` , `yfx` , `fy` ou `fx` où `f` indique la position du foncteur et `x` et `y` indiquent les positions des arguments. `y` désigne un terme avec une priorité inférieure ou égale à la priorité du foncteur, alors que `x` désigne une priorité strictement inférieure.
    - Préfixe: `fx` , `fy`
    - Infix: `xfx` (non associatif), `xfy` (associatif droit), `yfx` (associatif gauche)
    - Postfix: `xf` , `yf`

Exemple d'utilisation:

```
:- op(900, xf, is_true).

X_0 is_true :-
  X_0.
```

Exemple de requête:

```
?- dif(X, a) is_true.
dif(X, a).
```

Commande à terme

Deux termes peuvent être comparés via la commande standard:

variables @ <numéros @ <atomes @ <chaînes @ <structures @ <listes

Remarques:

- Les structures se comparent alphabétiquement par foncteur d'abord, puis par arité et enfin par comparaison de chaque argument.
- Les listes se comparent d'abord par longueur, puis par élément.

Opérateur de commande	Réussit si
X @ <Y	X est inférieur à Y dans l'ordre standard
X @> Y	X est supérieur à Y dans l'ordre standard
X @ = <Y	X est inférieur ou égal à Y dans l'ordre standard
X @> = Y	X est supérieur ou égal à Y dans l'ordre standard

Exemple de requêtes:

```
?- alpha @< beta.
true.

?- alpha(1) @< beta.
false.

?- alpha(X) @< alpha(1).
true.

?- alpha(X) @=< alpha(Y).
true.

?- alpha(X) @> alpha(Y).
false.

?- compound(z) @< compound(inner(a)).
true.
```

Égalité des termes

Opérateur d'égalité	Réussit si
X = Y	X peut être unifié avec Y
X \ = Y	X ne peut pas être unifié avec Y
X == Y	X et Y sont identiques (c'est-à-dire qu'ils s'unifient sans *aucune* liaison de variable)
X \ == Y	X et Y ne sont pas identiques
X =: = Y	X et Y sont arithmétiquement égaux
X = \ = Y	X et Y ne sont pas arithmétiquement égaux

Lire Les opérateurs en ligne: https://riptutorial.com/fr/prolog/topic/2479/les-operateurs

Chapitre 7: Liste de différences

Introduction

Les listes de différences de Prolog désignent le concept pour connaître la structure d'une liste *jusqu'à un point* . Le reste de la liste peut être laissé libre jusqu'à l'évaluation complète d'un prédicat. Une liste dont la fin est inconnue est appelée *liste ouverte* , terminée par un *trou* . Cette technique est particulièrement utile pour valider des syntaxes complexes ou des grammaires.

Le bien connu Grammars Clause définie (DCG) utilise des listes de différences pour fonctionner sous le capot.

Examples

Utilisation de base

Considérons le prédicat `sumDif/2` , vérifié si la structure d'une liste correspond à plusieurs contraintes. Le premier terme représente la liste à analyser et le second terme une autre liste contenant la partie de la première liste inconnue de nos contraintes.

Pour la démonstration, `sumDif/2` reconnaît une expression arithmétique pour résumer *n* entiers.

```
sumDif([X, +|OpenList], Hole) :-
    integer(X),
    sumDif(OpenList, Hole).
```

Nous savons que le premier élément de la liste à valider est un entier, ici illustré par `X` , suivi du symbole de l'addition (`+`). Le reste de la liste qui doit encore être traité ultérieurement (`OpenList`) est laissé non validé à ce niveau. `Hole` représente la partie de la liste que nous *n'avons pas besoin* de valider.

Donnons une autre définition du prédicat `sumDif/2` pour compléter la validation de l'expression arithmétique:

```
sumDif([X|Hole], Hole) :-
    integer(X).
```

Nous attendons un entier appelé `X` directement au début de la liste ouverte. Il est intéressant de noter que le reste de la liste `Hole` reste inconnu et que les listes de différences servent à l'objectif: la structure de la liste est connue jusqu'à un certain point.

Enfin, la pièce manquante vient quand une liste est évaluée:

```
?- sumDif([1,+,2,+,3], []).
true
```

C'est lorsque le prédicat est utilisé que la fin de la liste est mentionnée, ici `[]` , indique que la liste ne contient pas d'éléments supplémentaires.

Evaluer une expression arithmétique

Définissons une grammaire permettant d'effectuer des additions, des multiplications avec l'utilisation de parenthèses. Pour ajouter plus de valeur à cet exemple, nous allons calculer le résultat de l'expression arithmétique. Résumé de la grammaire:

expression → fois
expression → fois expression '+'
times → element
times → element '*' times
element → "entier"
element → '(' expression ')'

Tous les prédicats ont une arité de 3, car ils doivent ouvrir la liste, le trou et la valeur de l'expression arithmétique.

```
expression(Value, OpenList, FinalHole) :-
    times(Value, OpenList, FinalHole).

expression(SumValue, OpenList, FinalHole) :-
    times(Value1, OpenList, ['+'|Hole1]),
    expression(Value2, Hole1, FinalHole),
    plus(Value1, Value2, SumValue).

times(Value, OpenList, FinalHole) :-
    element(Value, OpenList, FinalHole).

times(TimesValue, OpenList, FinalHole) :-
    element(Value1, OpenList, ['*'|Hole1]),
    times(Value2, Hole1, FinalHole),
    TimesValue is Value1 * Value2.

element(Value, [Value|FinalHole], FinalHole) :-
    integer(Value).

element(Value, ['('|OpenList], FinalHole) :-
    expression(Value, OpenList, [')'|FinalHole]).
```

Pour bien expliquer le principe des *trous* et comment la valeur est calculée, prenons la deuxième `expression` clause:

```
expression(SumValue, OpenList, FinalHole) :-
    times(Value1, OpenList, ['+'|Hole1]),
    expression(Value2, Hole1, FinalHole),
    plus(Value1, Value2, SumValue).
```

La liste ouverte est désignée par le prédicat `OpenList` . Le premier élément à valider est *ce qui précède le symbole d'addition (`+`)* . Lorsque le premier élément est validé, il est directement suivi du symbole d'addition et de la suite de la liste, appelée `Hole1` . Nous savons que `Hole1` est

l'élément suivant à valider et peut être une autre `expression` , donc `Hole1` est alors le terme donné à l' `expression` prédicat.

La valeur est toujours représentée dans le premier terme. Dans cette clause, elle est définie par la somme de `Value1` (tout avant le symbole de l'addition) et `Value2` (tout après le symbole de l'addition).

Enfin, une expression peut être évaluée.

```
?- expression(V, [1,+,3,*,'(',5,+,5,')'], []).
V = 31
```

Lire Liste de différences en ligne: https://riptutorial.com/fr/prolog/topic/9414/liste-de-differences

Chapitre 8: Monotonicité

Examples

Raisonnement sur les prédicats monotones

Les prédicats **monotones** peuvent être débogués en appliquant un raisonnement *déclaratif* .

En pur Prolog, une erreur de programmation peut entraîner l'un ou l'autre des phénomènes suivants:

1. le prédicat *réussit* incorrectement dans un cas où il devrait *échouer*
2. le prédicat *échoue* incorrectement dans un cas où il devrait *réussir*
3. le prédicat *boucle* inopinément là où il ne devrait produire qu'un ensemble fini de réponses.

À titre d'exemple, considérons comment nous pouvons déboguer la casse (2) par un raisonnement déclaratif: nous pouvons systématiquement *supprimer les* objectifs des clauses du prédicat et voir si la requête échoue *toujours* . En code monotone, la suppression des objectifs peut tout au plus rendre le programme résultant *plus général* . Par conséquent, nous pouvons identifier les erreurs en voyant lequel des objectifs conduit à l'échec inattendu.

Exemples de prédicats monotones

Voici des exemples de prédicats **monotones** :

- **unification** avec `(=)/2` ou `unify_with_occurs_check/2`
- `dif/2` , exprimant le déséquilibre des termes
- **Les contraintes CLP (FD)** aiment `(#=)/2` et `(#>)/2` , en utilisant un mode d'exécution monotone.

Les prédicats Prolog qui utilisent uniquement des objectifs monotones sont eux-mêmes monotones.

Les prédicats monotoniques permettent un raisonnement déclaratif:

1. L'ajout d'une contrainte (c.-à-d. Un objectif) à une requête peut tout au plus *réduire* , jamais étendre, l'ensemble des solutions.
2. La suppression d'un objectif de tels prédicats peut au maximum *prolonger* , jamais réduire, l'ensemble des solutions.

Prédicats non monotones

Voici des exemples de prédicats qui **ne** sont **pas** monotones:

- des prédicats méta-logiques tels que `var/1` , `integer/1` etc.
- les prédicats de comparaison de termes comme `(@<)/2` et `(@>=)/2`
- les prédicats qui utilisent `!/0` , `(\+)/1` et d'autres constructions qui rompent la monotonie

- *prédicats tout-solutions* comme `findall/3` et `setof/3` .

Si ces prédicats sont utilisés, l' *ajout d'* objectifs peut conduire à plus de solutions, ce qui va à l'encontre de l'importante propriété déclarative connue de la logique selon laquelle l'ajout de contraintes peut tout au plus *réduire* , jamais étendre, l'ensemble des solutions.

Par conséquent, d'autres propriétés sur lesquelles nous nous appuyons pour le débogage déclaratif et d'autres raisonnements sont également rompues. Par exemple, les prédicats non monotones brisent la notion fondamentale de **commutativité** de la conjonction connue de la logique du premier ordre. L'exemple suivant illustre ceci:

```
?- var(X), X = a.
X = a.

?- X = a, var(X).
false.
```

Toutes les solutions-prédicats comme `findall/3` briserai monotonicité: *Ajout* des clauses peut conduire à *l'échec* des objectifs qui *avaient eu lieu* auparavant. Cela va également à l'encontre de la montonicité telle qu'elle est connue par la logique du premier ordre, où l' *ajout de* faits peut tout au plus *augmenter* , sans jamais *réduire* l'ensemble des conséquences.

Alternatives monotones pour les constructions non monotones

Voici des exemples d'utilisation de prédicats **monotones** *au lieu* de constructions impures et non monotones dans vos programmes:

- `dif/2` est destiné à être utilisé à la *place* de constructions non monotones comme `(\=)/2`
- les **contraintes** arithmétiques (CLP (FD), CLP (Q) et autres) sont destinées à être utilisées à la *place* des prédicats arithmétiques modulés
- `!/0` conduit presque toujours à des programmes non monotones et devrait être complètement **évité** .
- **Les erreurs d'instanciation** peuvent être **commises** dans des situations où vous ne pouvez pas prendre une décision éclairée à ce stade.

Combiner la monotonie à l'efficacité

On prétend parfois que, pour des raisons d'efficacité, nous devons accepter l'utilisation de constructions non monotones dans les programmes Prolog du monde réel.

Il n'y a aucune preuve à ce sujet. Des recherches récentes indiquent que le sous-ensemble purement monotone de Prolog peut non seulement suffire à exprimer la plupart des programmes du monde réel, mais aussi à être efficace dans la pratique. Une construction récemment découverte et encourageant cette vue est `if_/3` : elle combine la monotonie avec une réduction des points de choix. Voir **Indexation dif / 2** .

Par exemple, code du formulaire:

```
pred(L, Ls) :-
```

```
    condition(L),
    then(Ls).
pred(L, Ls) :-
    \+ condition(L),
    else(Ls).
```

Peut être écrit avec `if_/3` comme:

```
pred(L, Ls) :-
    if_(condition(L),
        then(Ls),
        else(Ls)).
```

et *combine la* monotonie avec le déterminisme.

Lire Monotonicité en ligne: https://riptutorial.com/fr/prolog/topic/3989/monotonicite

Chapitre 9: Performance

Examples

Machine abstraite

Pour plus d'efficacité, le code Prolog est généralement compilé en **code machine abstrait** avant son exécution.

De nombreuses architectures et variantes de machines abstraites ont été proposées pour une exécution efficace des programmes Prolog. Ceux-ci inclus:

- **WAM** , la *machine abstraite de Warren*
- **TOAM** , une machine abstraite utilisée dans B-Prolog.
- **ZIP** , utilisé par exemple comme base de la VM de SWI-Prolog
- **VAM** , une architecture de recherche développée à Vienne.

Indexage

Tous les interpréteurs Prolog largement utilisés utilisent l' **indexation des arguments** pour sélectionner efficacement les clauses appropriées.

Les utilisateurs peuvent généralement compter sur au moins un *premier argument d'indexation* , ce qui signifie que les clauses peuvent être efficacement différenciées par le foncteur et l'arité du terme le plus externe du *premier* argument. Dans les appels où cet argument est suffisamment instancié, les clauses d'appariement peuvent essentiellement être sélectionnées en temps *constant* via un hachage sur cet argument.

Plus récemment, l' **indexation JIT** a été implémentée dans plus de systèmes, permettant l'indexation dynamique sur tout argument suffisamment instancié lors de l'appel du prédicat.

Optimisation des appels de queue

Pratiquement tous les systèmes Prolog implémentent l' **optimisation des appels de queue** (TCO). Cela signifie que les appels de prédicats qui sont dans une *position de queue* peuvent être exécutés dans *un* espace de pile *constant* si le prédicat est déterministe.

L'optimisation de la **récursion de la** queue (TRO) est un cas particulier d'optimisation des appels de queue.

Lire Performance en ligne: https://riptutorial.com/fr/prolog/topic/4205/performance

Chapitre 10: Prédicats Extra-Logiques

Examples

Prédicats avec effets secondaires

Les prédicats qui produisent **des effets secondaires** quittent le domaine de la logique pure. Ce sont par exemple:

- `writeq/1`
- `read/1`
- `format/2`

Les effets secondaires sont des phénomènes qui ne peuvent pas être raisonnés dans le programme. Par exemple, suppression d'un fichier ou sortie sur le terminal du système.

Prédicats méta-logiques

Prédit que la raison des *instanciations* est appelée **méta-logique** . Les exemples sont:

- `var/1`
- `ground/1`
- `integer/1`

Ces prédicats sont en dehors du domaine des programmes logiques monotones purs, car ils rompent des propriétés telles que la *commutativité* de la conjonction.

Les autres prédicats qui sont méta-logiques incluent:

- `arg/3`
- `functor/3`
- `(=..)/2`

Ces prédicats pourraient *en principe* être modélisés dans la logique du premier ordre, mais nécessitent un nombre infini de clauses.

Prédicats de toutes les solutions

Les prédicats selon lesquels la raison de *toutes les solutions* est extra-logique. Ce sont par exemple:

- `setof/3`
- `findall/3`
- `bagof/3`

! / 0 et les prédicats associés

Les prédicats qui empêchent ou interdisent une lecture **déclarative** des programmes Prolog sont extra-logiques. Des exemples de tels prédicats sont:

- `!/0`
- `(->)/2` et si-alors-sinon
- `(\+)/1`

Ces prédicats ne peuvent être compris que sur le plan de la procédure, en tenant compte du flux de contrôle réel de l'interprète et, en tant que tels, dépassent le cadre de la logique pure.

Lire Prédicats Extra-Logiques en ligne: https://riptutorial.com/fr/prolog/topic/2282/predicats-extra-logiques

Chapitre 11: Programmation d'ordre supérieur

Examples

appel / N prédicats

La famille de prédicats `call/N` peut appeler des objectifs Prolog arbitraires au moment de l'exécution:

```
?- G=true, call(G).
true.

?- G=(true,false), call(G).
false.
```

mapliste / [2,3]

`maplist/2` et `maplist/3` sont des prédicats d'ordre supérieur, qui permettent de définir un prédicat à partir d'un seul élément pour les *répertorier* . Ces prédicats peuvent être définis à l'aide de `call/2` et `call/3` tant que blocs de construction et livrés avec de nombreux systèmes Prolog.

Par exemple:

```
?- maplist(dif(a), [X,Y,Z]).
dif(X, a),
dif(Y, a),
dif(Z, a).
```

Meta-call

Dans Prolog, le **méta-appel** est une fonctionnalité de langage intégrée. Tout code Prolog est représenté par des *termes* Prolog, ce qui permet de construire des objectifs de manière dynamique et de les utiliser comme d'autres objectifs sans prédicats supplémentaires:

```
?- Goal = dif(X, Y), Goal.
dif(X, Y).
```

En utilisant ce mécanisme, d'autres prédicats d'ordre supérieur peuvent être définis dans Prolog lui-même.

foldl / 4

Un *pli* (de gauche) est une relation d'ordre supérieur entre:

- un prédicat avec 3 arguments

- une liste d'éléments
- un état initial
- un état final, qui résulte de l'application du prédicat à des éléments successifs lors de la traversée d'états intermédiaires.

Par exemple: Utilisez `foldl/4` pour exprimer la *somme* de tous les éléments d'une liste, en utilisant un prédicat comme bloc de construction pour définir la somme de *deux* éléments:

```
?- foldl(plus, [2,3,4], 0, S).
S = 9.
```

Appeler une liste d'objectifs

Pour appeler une liste d'objectifs comme s'il s'agissait d'une conjonction d'objectifs, combinez les prédicats d'ordre supérieur / 1 et maplist / 2:

```
?- Gs = [X = a, Y = b], maplist(call, Gs).
Gs = [a=a, b=b],
X = a,
Y = b.
```

Lire Programmation d'ordre supérieur en ligne:
https://riptutorial.com/fr/prolog/topic/2420/programmation-d-ordre-superieur

Chapitre 12: Programmation logique par contraintes

Examples

CLP (FD)

Les contraintes CLP (FD) (*domaines finis*) implémentent l'arithmétique sur les **entiers** . Ils sont disponibles dans toutes les implémentations sérieuses de Prolog.

Il existe deux cas d'utilisation majeurs de contraintes CLP (FD):

- Arithmétique entière déclarative
- Résoudre des problèmes combinatoires tels que la planification, la planification et les tâches d'allocation.

Exemples:

```
?- X #= 1+2.
X = 3.

?- 3 #= Y+2.
Y = 1.
```

Notez que si `is/2` devait être utilisé dans la deuxième requête, une erreur d'instanciation se produirait:

```
?- 3 is Y+2.
ERROR: is/2: Arguments are not sufficiently instantiated
```

CLP (Q)

CLP (Q) implémente le raisonnement sur *des* nombres *rationnels* .

Exemple:

```
?- { 5/6 = X/2 + 1/3 }.
X = 1.
```

CLP (H)

Prolog lui-même peut être considéré comme **CLP (H)** : programmation par contraintes logiques sur les *termes de Herbrand* . Dans cette perspective, un programme Prolog affiche des contraintes sur les *termes* . Par exemple:

```
?- X = f(Y), Y = a.
```

```
X = f(a),
Y = a.
```

Lire Programmation logique par contraintes en ligne:
https://riptutorial.com/fr/prolog/topic/2057/programmation-logique-par-contraintes

Chapitre 13: Pureté logique

Examples

dif / 2

Le prédicat `dif/2` est un prédicat **pur** : il peut être utilisé dans toutes les directions et avec tous les modèles d'instanciation, ce qui signifie *toujours* que ses deux arguments sont *différents* .

Contraintes CLP (FD)

Les contraintes CLP (FD) sont des relations complètement pures. Ils peuvent être utilisés dans toutes les directions pour l'arithmétique entière déclarative:

```
?- X #= 1+2.
X = 3.

?- 3 #= Y+2.
Y = 1.
```

Unification

L'unification est une relation **pure** . Il ne produit pas d'effets secondaires et peut être utilisé dans toutes les directions, avec l'un des arguments ou les deux complètement ou seulement instanciés.

Dans Prolog, l'unification peut se produire

- **explicitement** , en utilisant des prédicats intégrés comme `(=)/2` ou `unify_with_occurs_check/2`
- **implicitement** , lorsque l'unification est utilisée pour sélectionner une clause appropriée.

Lire Pureté logique en ligne: https://riptutorial.com/fr/prolog/topic/2058/purete-logique

Chapitre 14: Raisonnement sur les données

Remarques

Une nouvelle section intitulée **Data Structures a** été créée pour fournir des explications sur certaines structures et des exemples simples de création. Pour que son contenu reste concis et concis, il ne doit contenir aucune documentation sur la manipulation des données.

Par conséquent, cette section a été renommée "Reasoning about data" avec pour but de généraliser le raisonnement sur les données dans Prolog. Cela pourrait inclure des sujets allant de «l'inférence descendante» à la «traversée de listes», ainsi que de nombreux autres. En raison de sa large généralisation, des sous-sections claires devraient être établies!

Examples

Récursivité

Prolog n'a pas d'itération, mais toutes les itérations peuvent être réécrites en utilisant la récursivité. La récursivité apparaît lorsqu'un prédicat contient un objectif qui se réfère à lui-même. Lors de l'écriture de tels prédicats dans Prolog, un motif récursif standard comporte toujours au moins deux parties:

- **Clause de base (non récursive)** : En règle générale, la ou les règles de base représentent le ou les plus petits exemples possibles du problème que vous essayez de résoudre - une liste sans membres, ou un seul membre, ou si vous Travailler avec une structure arborescente, cela peut concerner un arbre vide, ou un arbre avec un seul nœud, etc. Il décrit de manière non récursive la base du processus récursif.
- **Clause récursive (continue)** : Contient toute logique requise, y compris un appel à elle-même, poursuite de la récursivité.

A titre d'exemple, nous définirons l' `append/3` prédicat bien connu `append/3` . Vu de manière déclarative, `append(L1,L2,L3)` est conservé lorsque la liste `L3` est le résultat des listes `L1` et `L2` . Lorsque nous essayons de comprendre la signification déclarative d'un prédicat, nous essayons de décrire les solutions pour lesquelles le prédicat est présent. La difficulté réside ici dans le fait d'essayer d'éviter les détails récurrents étape par étape tout en gardant à l'esprit le comportement procédural du prédicat.

```
% Base case
append([],L,L).

% Recursive clause
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

Le cas de base déclare de manière déclarative que "tout L ajouté à la liste vide est L", notez que ceci ne dit rien sur le fait que L soit vide - ou même une liste (rappelez-vous que dans Prolog tout

se résume en termes):

```
?- append(X,some_term(a,b),Z).
X = [],
Z = some_term(a, b).
```

Pour décrire la règle récursive, bien que Prolog exécute les règles de gauche à droite, nous omettons la tête pendant une seconde et regardons d'abord le corps - en lisant la règle de droite à gauche:

```
    append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

Maintenant, nous disons que si le corps contient: «en supposant que `append(L1,L2,L3)` est valide»

```
    append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

Alors, la tête aussi: “alors alors `append([X|L1],L2,[X|L3])` ”

En anglais, cela se traduit simplement par:

> En supposant que L3 est la concaténation de L1 et L2, alors [X suivi de L3] est également la concaténation de [X suivi de L1] et L2.

Dans un exemple pratique:

> «En supposant que [1,2,3] est la concaténation de [1] et [2,3], alors [a, 1,2,3] est aussi la concaténation de [a, 1] et [2,3]. ”

Maintenant, regardons quelques requêtes:

C'est toujours une bonne idée de tester initialement votre prédicat avec la **requête** la **plus générale** plutôt que de lui fournir un scénario de test spécifique. Pensez-y: en raison de l'unification de Prolog, nous n'avons pas besoin de fournir de données de test, nous leur transmettons simplement des variables gratuites!

```
?- append(L1,L2,L3).
L1 = [],
L2 = L3 ;                                   % Answer #1
L1 = [_G1162],
L3 = [_G1162|L2] ;                          % Answer #2
L1 = [_G1162, _G1168],
L3 = [_G1162, _G1168|L2] ;                  % Answer #3
L1 = [_G1162, _G1168, _G1174],
L3 = [_G1162, _G1168, _G1174|L2] ;          % Answer #4
...
```

`_G1162` variable libre `_G1162` avec des lettres alphabétiques pour obtenir un meilleur aperçu:

```
?- append(L1,L2,L3).
L1 = [],
L2 = L3 ;                                   % Answer #1
```

```
L1 = [_A],
L3 = [_A|L2] ;                              % Answer #2
L1 = [_A, _B],
L3 = [_A, _B|L2] ;                          % Answer #3
L1 = [_A, _B, _C],
L3 = [_A, _B, _C|L2] ;                      % Answer #4
...
```

Dans la première réponse, le cas de base correspond à un motif et Prolog instancie `L1` à la liste vide et `L2` et `L3` unifiés prouvant que `L3` est la concaténation de la liste vide et L2.

A la réponse n ° 2, en revenant chronologiquement, la clause récursive entre en jeu et Prolog tente de prouver que certains éléments dans la tête de `L1` concaténés avec `L2` sont `L3` avec ce même élément dans son en-tête de liste. Pour ce faire, une nouvelle variable libre `_A` est unifiée avec la tête de L1 et il est prouvé que L3 est maintenant `[_A|L2]` .

Un nouvel appel récursif est effectué, maintenant avec `L1 = [_A]` . Encore une fois, Prolog tente de prouver que certains éléments placés dans la tête de `L1` , concaténés avec `L2` sont `L3` avec ce même élément dans sa tête. Notez que `_A` est déjà la tête de `L1` , ce qui correspond parfaitement à la règle, alors maintenant, par récursivité, Prolog place `_A` devant une nouvelle variable libre et nous obtenons `L1 = [_A,_B]` et `L3 = [_A,_B|L2]`

Nous voyons clairement que le motif récursif se répète et peut facilement voir que, par exemple, le résultat de la 100ème étape de la récursivité ressemblerait à ceci:

```
L1 = [X1,X2,..,X99],
L3 = [X1,X2,..,X99|L2]
```

Note: comme c'est le cas pour un bon code Prolog, la définition récursive de `append/3` nous donne non seulement la possibilité de *vérifier* si une liste est la concaténation de deux autres listes, mais *génère* toutes les réponses satisfaisant les relations ou des listes partiellement instanciées.

Accéder aux listes

Membre

`member/2` a un `member(?Elem, ?List)` signature `member(?Elem, ?List)` et indique `true` si `Elem` est membre de `List` . Ce prédicat peut être utilisé pour accéder à des variables dans une liste, où différentes solutions sont récupérées par retour en arrière.

Exemple de requêtes:

```
?- member(X, [1,2,3]).
X = 1 ;
X = 2 ;
X = 3.

?- member(X,[Y]).
X = Y.

?- member(X,Y).
```

```
Y = [X|_G969] ;
Y = [_G968, X|_G972] ;
Y = [_G968, _G971, X|_G975] ;
Y = [_G968, _G971, _G974, X|_G978]
...
```

Correspondance de motif

Lorsque les index dont vous avez besoin sont petits, la correspondance de modèle peut être une bonne solution, par exemple:

```
third([_,_,X|_], X).
fourth([_,_,_,X|_], X).
```

Lire Raisonnement sur les données en ligne:
https://riptutorial.com/fr/prolog/topic/2005/raisonnement-sur-les-donnees

Chapitre 15: Structures de contrôle

Examples

Disjonction (OU logique), implicite vs explicite

Prolog essaie des clauses alternatives pour un prédicat dans l'ordre d'apparition:

```
likes(alice, music).
likes(bob, hiking).

// Either alice likes music, or bob likes hiking will succeed.
```

L'opérateur de disjonction (OR) `;` peut être utilisé pour exprimer ceci dans une règle:

```
likes(P,Q) :-
    ( P = alice , Q = music ) ; ( P = bob , Q = hiking ).
```

Les parenthèses sont importantes ici pour plus de clarté. Voir cette question sur la `,` `;` préséance relative pour la conjonction `,` et disjonction `;` .

Conjonction (ET logique)

La conjonction (ET logique) est représentée par la virgule `,` opérateur (entre autres rôles).

La conjonction entre les clauses peut apparaître dans une requête:

```
?- X = 1, Y = 2.
```

La conjonction peut également apparaître entre les clauses sous-jacent du corps d'une règle:

```
triangleSides(X,Y,Z) :-
    X + Y > Z, X + Z > Y, Y + Z > X.
```

Couper (supprimer les points de choix)

Parfois, il est souhaitable d'empêcher Prolog de revenir en arrière dans des solutions alternatives. L'outil de base à la disposition du programmeur pour empêcher Prolog de continuer à se poursuivre dans son backtrack est l'opérateur cut. considérer ce qui suit.

```
% (percent signs mean comments)
% a is the parent of b, c, and d.
parent(a,b).
parent(a,c).
parent(a,d).
```

Ici, le prédicat `parent/2` réussit plus d'une fois lorsque

```
?- parent(a,X).
```

est appelé. Pour empêcher Prolog de rechercher d'autres solutions après la découverte du premier, vous devez utiliser l'opérateur cut, comme ceci.

```
?- parent(a,X), !.
```

Cela aura X égal à b (car c'est la première solution possible) et ne cherche plus de solutions.

Lire Structures de contrôle en ligne: https://riptutorial.com/fr/prolog/topic/4479/structures-de-controle

Chapitre 16: Structures de données

Examples

Des listes

Les listes sont un type particulier de *terme composé* . Les listes sont définies par induction:

- l'atome `[]` est une liste indiquant la *liste vide* .
- *si* `Ls` est une liste, alors le terme `'.'(L, Ls)` est *également* une liste.

Il existe une syntaxe spéciale pour dénoter les listes de manière pratique dans Prolog:

1. La liste `'.'(a, '.'(b, '.'(c, [])))` peut aussi être écrite comme `[a,b,c]` .
2. Le terme `'.'(L, Ls)` peut aussi être écrit comme `[L|Ls]` .

Ces notations peuvent être combinées de n'importe quelle manière. Par exemple, le terme `[a,b|Ls]` est une liste *si* `Ls` est une liste.

Créer des listes

Une liste de littéraux unifiés avec la variable List:

```
?- List = [1,2,3,4].
List = [1, 2, 3, 4].
```

Construire une liste en consommant:

```
?- Tail = [2, 3, 4], List = [1|Tail].
Tail = [2, 3, 4],
List = [1, 2, 3, 4].
```

Construire une liste de valeurs inconnues à l'aide de la `length/2` intégrée `length/2` :

```
?- length(List,5).
List = [_G496, _G499, _G502, _G505, _G508].
```

Puisque dans Prolog tout est essentiellement un terme, les listes se comportent de manière hétérogène:

```
?- List = [1, 2>1, this, term(X), 7.3, a-A].
List = [1, 2>1, this, term(X), 7.3, a-A].
```

Cela signifie qu'une liste peut également contenir d'autres listes, également appelées listes internes:

```
List = [[1,2],[3,[4]]].
```

Paires

Par convention, le foncteur `(-)/2` est souvent utilisé pour désigner des **paires** d'éléments dans Prolog. Par exemple, le terme `-(A, B)` désigne la paire d'éléments `A` et `B` Dans Prolog, `(-)/2` est défini comme un *opérateur infixe* . Par conséquent, le terme peut être écrit de manière équivalente à `AB` .

De nombreux prédicats couramment utilisés utilisent également cette syntaxe pour désigner des paires. Des exemples de ceci sont `keysort/2` et `pairs_keys_values/3` .

Listes d'association

Dans tous les systèmes Prolog sérieux, **les listes d'associations** sont disponibles pour permettre un accès plus rapide que linéaire à une collection d'éléments. Ces listes d'association sont généralement basées sur *des arbres équilibrés* tels que des **arbres AVL** . Une bibliothèque de domaine public appelée `library(assoc)` avec de nombreux systèmes Prolog et fournit des opérations *O (log (N))* pour insérer, récupérer et modifier des éléments dans une collection.

termes

À un très haut niveau, Prolog ne possède qu'un seul type de données, appelé **terme** . Dans Prolog, toutes les données sont représentées par des termes Prolog. Les termes sont définis par induction:

- un **atome** est un terme. Des exemples d'atomes sont: `x` , `test` et `'quotes and space'` .
- une **variable** est un terme. Les variables commencent par une lettre majuscule ou un trait de soulignement `_` .
- les entiers et les nombres à virgule flottante sont des termes. Exemples: `42` et `42.42` .
- un **terme composé** est un terme défini inductivement comme suit: *Si* `T1` , `T2` , ..., `T_n` sont des termes, *alors F* (`T1` , `T2` , ..., `T_n`) est aussi un terme, où *F* est appelé le **foncteur** de le terme composé.

Termes avec champs nommés utilisant la bibliothèque (enregistrement)

La bibliothèque `[record][1]` permet de créer des termes composés avec des champs nommés. La directive `:- record/1 <spec>` compile en une collection de prédicats qui initialisent, définissent et obtiennent des champs dans le terme défini par `<spec>` .

Par exemple, nous pouvons définir une structure de données `point` avec les champs nommés `x` et `y` :

```
:- use_module(library(record)).

:- record point(x:integer=0,
                y:integer=0).

/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

?- default_point(Point), point_x(Point, X), set_x_of_point(10, Point, Point1).
```

```
Point = point(0, 0),
X = 0,
Point1 = point(10, 0).

?- make_point([y(20)], Point).
Point = point(0, 20).

?-  is_point(X).
false.

?- is_point(point(_, _)).
false.

?- is_point(point(1, a)).
false.

?- is_point(point(1, 1)).
true.

- - - -
```

Chapitre 17: Utiliser Prolog moderne

Examples

introduction

De nombreux systèmes Prolog modernes sont en développement continu et ont ajouté de nouvelles fonctionnalités pour résoudre les problèmes classiques du langage. Malheureusement, de nombreux manuels Prolog et même des cours d'enseignement ne présentent toujours que le prologue obsolète. Ce sujet est destiné à illustrer comment Prolog moderne a surmonté certains des problèmes et syntaxe plutôt inutile qui apparaît dans les anciens Prolog et peut encore être introduite.

CLP (FD) pour l'arithmétique des nombres entiers

Traditionnellement, Prolog exécute l'arithmétique à l'aide des opérateurs `is` et `=:=` . Cependant, plusieurs Prologs actuels offrent CLP (FD) (Programmation par Logique de Contrainte sur Domaines Finis) comme alternative plus propre à l'arithmétique entière. CLP (FD) est basé sur le stockage des contraintes qui s'appliquent à une valeur entière et à la combinaison de celles-ci en mémoire.

CLP (FD) est une extension dans la plupart des Prolog qui la supporte, elle doit donc être chargée explicitement. Une fois qu'il est chargé, le `#=` syntaxe peut prendre la place des deux `is` et `=:=` . Par exemple, dans SWI-Prolog:

```
?- X is 2+2.
X = 4.

?- use_module(library(clpfd)).
?- X #= 2+2.
X = 4.
```

A la différence `is` , `#=` est en mesure de résoudre des équations simples et unifier dans les deux sens:

```
?- 4 is 2+X.
ERROR: is/2: Arguments are not sufficiently instantiated

?- 4 #= 2+X.
X = 2.
```

CLP (FD) fournit sa propre syntaxe de générateur.

```
?- between(1,100,X).
X = 1;
X = 2;
X = 3...
```

```
?- X in 1..100.
X in 1..100.
```

Notez que le générateur ne s'exécute pas réellement: seule la contrainte d'intervalle est stockée, prête à être combinée avec des contraintes ultérieures. Le générateur peut être forcé à exécuter (et à forcer des contraintes) en utilisant le prédicat d' `label` :

```
?- X in 1..100, label([X]).
X = 1;
X = 2;
X = 3..
```

L'utilisation de CLP peut permettre une réduction intelligente des cas de force brute. Par exemple, en utilisant l'arithmétique de type entier ancien:

```
?- trace.
?- between(1,10,X), Y is X+5, Y>10.
...
Exit: (8) 6 is 1+5 ? creep
Call: (8) 6 > 10 ? creep
...
X = 6, Y = 11; ...
```

Prolog continue à parcourir les valeurs 1-5 même s'il est mathématiquement prouvable que les valeurs données ne permettent pas de les utiliser. En utilisant CLP (FD):

```
?- X in 1..10, Y #= X+5, Y #> 10.
X is 6..10,
X+5 #= Y,
Y is 11..15.
```

CLP (FD) effectue immédiatement les calculs et calcule les plages disponibles. L'ajout de `label([Y])` entraînera une boucle de X uniquement à travers les valeurs utiles 6..10. Dans cet exemple de jouet, cela n'augmente pas les performances car, avec une plage aussi petite que 1-10, le traitement d'algèbre prend autant de temps que la boucle l'aurait fait; mais lorsque des nombres plus importants sont traités, cela peut réduire le temps de calcul.

La prise en charge de CLP (FD) est variable entre Prologs. Le meilleur développement reconnu de CLP (FD) se trouve dans SICStus Prolog, qui est commercial et coûteux. SWI-Prolog et d'autres prologs ouverts ont souvent une implémentation. Visual Prolog n'inclut pas CLP (FD) dans sa bibliothèque standard, bien que des bibliothèques d'extension soient disponibles.

Forall au lieu de boucles pilotées par les pannes

Certains manuels "classiques" de Prolog utilisent toujours la syntaxe de boucle, déroutante et sujette à erreur, où une construction par `fail` est utilisée pour forcer le retour en arrière d'appliquer un objectif à chaque valeur d'un générateur. Par exemple, pour imprimer tous les nombres jusqu'à une limite donnée:

```
fdl(X) :- between(1,X,Y), print(Y), fail.
```

```
fdl(_).
```

La grande majorité des prologues modernes n'ont plus besoin de cette syntaxe, fournissant plutôt un prédicat d'ordre supérieur pour y remédier.

```
nicer(X) :- forall(between(1,X,Y), print(Y)).
```

Non seulement cela est beaucoup plus facile à lire, mais si un objectif susceptible d'échouer était utilisé à la place de *print* , son échec serait correctement détecté et transmis - alors que les échecs des objectifs dans une boucle pilotée par une défaillance sont confondus avec l'échec forcé. qui dirige la boucle.

Visual Prolog a un sucre syntaxique personnalisé pour ces boucles, associé à des prédicats de fonction (voir ci-dessous):

```
vploop(X) :- foreach Y = std::fromTo(1,X) do
                console::write(X)
            end foreach.
```

Bien que cela ressemble à un impératif *pour la* boucle, il suit toujours les règles Prolog: en particulier, chaque itération de *foreach a* sa propre portée.

Prédicats de fonction

Traditionnellement, dans Prolog, les "fonctions" (avec une sortie et des entrées liées) étaient écrites en tant que prédicats réguliers:

```
mangle(X,Y) :- Y is (X*5)+2.
```

Cela peut créer la difficulté que si un prédicat de style fonction est appelé plusieurs fois, il est nécessaire de "chaîner" les variables temporaires.

```
multimangle(X,Y) :- mangle(X,A), mangle(A,B), mangle(B,Y).
```

Dans la plupart des Prologs, il est possible d'éviter cela en écrivant un autre opérateur d'infixe à utiliser à la place de `is` qui développe des expressions, y compris la fonction alternative.

```
% Define the new infix operator
:- op(900, xfy, <-).

% Define our function in terms of the infix operator - note the cut to avoid
% the choice falling through
R <- mangle(X) :- R is (X*5)+2, !.

% To make the new operator compatible with is..
R <- X :-
    compound(X),            % If the input is a compound/function
    X =.. [OP, X2, X3],     % Deconstruct it
    R2 <- X2,               % Recurse to evaluate the arguments
    R3 <- X3,
```

```
    Expr =.. [OP, R2, R3],  % Rebuild a compound with the evaluated arguments
    R is Expr,              % And send it to is
    !.
R <- X :- R is X, !.        % If it's not a compound, just use is directly
```

Nous pouvons maintenant écrire:

```
multimangle(X,Y) :- X <- mangle(mangle(mangle(Y))).
```

Cependant, certains Prolog modernes vont plus loin et proposent une syntaxe personnalisée pour ce type de prédicat. Par exemple, dans Visual Prolog:

```
mangle(X) = Y :- Y = ((X*5)+2).
multimangle(X,Y) :- Y = mangle(mangle(mangle(X))).
```

Notez que l'opérateur `<-` et le prédicat de style fonctionnel ci-dessus se comportent toujours comme des *relations* - il est légal d'avoir des points de choix et d'effectuer une unification multiple. Dans le premier exemple, nous évitons cela en utilisant des coupes. Dans Visual Prolog, il est normal d'utiliser la syntaxe fonctionnelle pour les relations et les points de choix sont créés de manière normale - par exemple, l'objectif `X = (std::fromTo(1,10))*10` réussit avec les liaisons X = 10 , X = 20, X = 30, X = 40, etc.

Déclarations de flux / mode

Lors de la programmation dans Prolog, il n'est pas toujours possible, ou souhaitable, de créer des prédicats unifiant pour chaque combinaison possible de paramètres. Par exemple, le prédicat `between(X,Y,Z)` qui exprime que Z est numériquement compris entre X et Y. Il est facilement implémenté dans les cas où X, Y et Z sont tous liés (soit Z est compris entre X et Y, soit ce n'est pas) ou où X et Y sont liés et Z est libre (Z unifie avec tous les nombres entre X et Y ou le prédicat échoue si Y <X); mais dans d'autres cas, tels que X et Z sont liés et Y est libre, il existe potentiellement un nombre infini d'unifications. Bien que cela puisse être mis en œuvre, ce ne serait généralement pas le cas.

Les déclarations de mode ou les *déclarations de mode* permettent une description explicite du comportement des prédicats lorsqu'ils sont appelés avec différentes combinaisons de paramètres liés. Entre les `between` , la déclaration serait:

```
%! between(+X,+Y,+Z) is semidet.
%! between(+X,+Y,-Z) is nondet.
```

Chaque ligne spécifie un motif d'appel potentiel pour le prédicat. Chaque argument est décoré avec `+` pour indiquer les cas où il est lié, ou `-` pour indiquer les cas où il ne l'est pas (il existe également d'autres décorations pour les types plus complexes tels que les tuples ou les listes pouvant être partiellement liées). Le mot clé after *is* indique le comportement du prédicat dans ce cas et peut être l'un de ceux-ci:

- `det` si le prédicat réussit toujours sans point de choix. Par exemple, `add(+X,+Y,-Z)` est `det` car l'ajout de deux nombres donnés X et Y aura toujours exactement une réponse.

- `semidet` si le prédicat réussit ou échoue, sans point de choix. Comme ci-dessus, `between(+X,+Y,+Z)` est `semidet` parce que Z est soit entre X et Y ou ce n'est pas le cas.
- `multi` si le prédicat réussit toujours, mais peut avoir des points de choix (mais aussi peut-être pas). Par exemple, `factor(+X,-Y)` serait `multi` car un nombre a toujours au moins un facteur - lui-même - mais peut avoir plus.
- `nondet` si le prédicat peut réussir avec des points de choix ou échouer. Par exemple, `between(+X,+Y,-Z)` est `nondet` car il peut y avoir plusieurs unifications possibles de Z à des nombres entre X et Y, ou si Y <X, il n'y a pas de nombres entre eux et le prédicat échoue.

Les déclarations de flux / mode peuvent également être combinées avec l'étiquetage des arguments pour clarifier quels termes signifient ou en tapant. Par exemple, `between(+From:Int, +To:Int, +Mid:Int) is semidet .`

Dans les Prologs purs, les déclarations de flux et de mode sont facultatives et ne sont utilisées que pour la génération de documentation, mais elles peuvent être extrêmement utiles pour aider les programmeurs à identifier la cause des erreurs d'instanciation.

Dans Mercury, les déclarations de flux et de mode (et les types) sont obligatoires et validées par le compilateur. La syntaxe utilisée est comme ci-dessus.

Dans Visual Prolog, les déclarations et les types de flux et de mode sont également obligatoires et la syntaxe est différente. La déclaration ci-dessus serait écrite comme suit:

```
between : (int From, int To, int Mid) determ (i,i,i) nondeterm (i,i,o).
```

La signification est la même que ci-dessus, mais avec les différences qui:

- Les déclarations de flux / mode sont séparées des déclarations de type (car on suppose que le flux / mode pour un seul prédicat ne varie pas avec la surcharge de type);
- `i` et `o` sont utilisés pour `+` et `-` et sont associés aux paramètres basés sur la commande;
- Les termes utilisés sont différents. `det` devient `procedure` , `semidet` devient `determ` , et `nondet` devient `nondeterm` (`multi` est toujours `multi`).

Lire Utiliser Prolog moderne en ligne: https://riptutorial.com/fr/prolog/topic/5499/utiliser-prolog-moderne

Crédits

S. No	Chapitres	Contributeurs
1	Premiers pas avec le langage Prolog	coder, Community, Eyal, Limmen, manlio, mat, Nick the coder, Norman Creaney, Rajat Jain, Ruslan López Carro, SeekAndDestroy, Willem Van Onsem, Xevaquor
2	Arbres de dérivation	SeekAndDestroy
3	Clauses définies de la clause (DCG)	false, mat, Will Ness, ZenLulz
4	Directives de codage	mat
5	Gestion des erreurs et exceptions	mat
6	Les opérateurs	false, mat, SeekAndDestroy
7	Liste de différences	ZenLulz
8	Monotonicité	manlio, mat
9	Performance	mat
10	Prédicats Extra-Logiques	false, mat
11	Programmation d'ordre supérieur	4444, Eyal, mat
12	Programmation logique par contraintes	mat, SeekAndDestroy
13	Pureté logique	mat
14	Raisonnement sur les données	Daniel Lyons, manlio, mat, SeekAndDestroy
15	Structures de contrôle	hardmath, Nick the coder
16	Structures de données	Eyal, mat, SeekAndDestroy

17	Utiliser Prolog moderne	Mark Green