

Arithmétique en Prolog

- Prolog fournit un ensemble d'outils pour l'arithmétique, incluant les entiers et les nombres réels.

Arit hmét ique	Prolog
$2 + 3 = 5$?- 5 is 2+3.
$3 \times 4 = 12$?- 12 is 3*4.
$5 - 3 = 2$?- 2 is 5-3.
$3 - 5 = -2$?- -2 is 3-5.
$4 \div 2 = 2$?- 2 is 4/2.
1 est le reste de 7 divisé par 2	?- 1 is mod(7,2).

Exemples d'arithmétique en Prolog

- Quelques exemples d'utilisation de l'arithmétique en Prolog :
 - ?- 10 is 5+5. donne yes
 - ?- 4 is 2+3. donne no
 - ?- X is 3 * 4. unifie X à 12, donnant X=12 et yes
 - ?- R is mod(7,2). unifie R à 1, donnant R=1 et yes

Définir un prédicat avec de l'arithmétique

- On peut définir un prédicat qui effectue des opérations arithmétiques. Par exemple:

```
addThreeAndDouble(X, Y) :-  
    Y is (X+3) * 2.
```

Ici, nous avons le prédicat `addThreeAndDouble/2` qui prend deux arguments, `X` et `Y`, et calcule `Y` comme $(X + 3) \times 2$. Les exemples suivants montrent comment il fonctionne :

- ?- addThreeAndDouble(1,X). ?donne X = 8
- addThreeAndDouble(2,X). donne X = 10

```
addThreeAndDouble(X, Y):-  
    Y is (X+3) * 2.
```

```
?- addThreeAndDouble(1,X).
```

```
X=8
```

```
yes
```

```
?- addThreeAndDouble(2,X).
```

```
X=10
```

```
yes
```

Le prédicat `is/2`

- Les opérateurs `+`, `-`, `/`, et `*` ne font pas d'arithmétique par eux-mêmes. Ils créent des termes complexes.
 - Par exemple, `3+2` est un terme complexe avec le foncteur `+`, une arité de 2, et les entiers 3 et 2 comme arguments.
- Pour forcer Prolog à évaluer des expressions arithmétiques, on utilise le prédicat `is/2`.
`is/2` est une instruction pour que Prolog effectue les calculs. Il y a des restrictions
- sur son utilisation.
 - Il est possible d'utiliser des variables sur la droite du prédicat `is`. Au moment où Prolog réalise l'évaluation, ces variables doivent être instanciées avec un terme Prolog qui n'est pas une variable. Ce terme Prolog doit être une expression arithmétique.
 - `3+2` est en fait `+(3,2)` et ainsi de suite. Les expressions arithmétiques sont des termes complexes.

?- X is $3 + 2$.

- Par exemple :
 - `?- X is 3 + 2. ?- 3 + 2 is X.` unifie `X` à 5, donnant `X = 5` et **yes**.
 - `Result is 2+2+2+2+2.` produit une erreur car `X` n'est pas suffisamment instancié.
 - `Result is 2+2+2+2+2+2.` unifie `Result` à 10, donnant `Result = 10` et **yes**.

?- X is 3 + 2.

X = 5

yes

?- 3 + 2 is X.

?- X is 3 + 2.

X = 5

yes

?- 3 + 2 is X.

ERROR: is/2: Arguments are not sufficiently instantiated

?- Result is 2+2+2+2+2.

Result = 10

yes

?-

Arithmétique et Listes

- Pour calculer la longueur d'une liste :
 - La longueur de la liste vide est zéro.
 - La longueur d'une liste non vide est un plus la longueur de sa queue.

Longueur d'une liste en Prolog

- Voici une implémentation Prolog pour calculer la longueur d'une liste :

```
len([], 0).  
len([_|L], N) :-  
    len(L, X),  
    N is X + 1.
```

- Exemple :
 - `?- len([a,b,c,d,e,[a,x],t], X).` unifie `X` à 7, donnant `x = 7` et `yes`.

```
len([],0).  
len([_|L],N):-  
    len(L,X),  
    N is X + 1.
```

?-

Accumulateurs

- Une autre manière de calculer la longueur d'une liste est d'utiliser des accumulateurs.
 - Les accumulateurs sont des variables qui contiennent des résultats intermédiaires.
- Le prédicat **acclen/3** a trois arguments :
 - La liste dont on veut calculer la longueur.
 - La longueur de la liste, un entier.
 - Un accumulateur, qui conserve la trace des valeurs intermédiaires pour le calcul de la longueur.

```
acclen([],Acc,Length):-  
    Length = Acc.
```

```
acclen([_|L],OldAcc,Length):-  
    NewAcc is OldAcc + 1,  
    acclen(L,NewAcc,Length).
```

**Ajoute 1 à
l'accumulateur chaque
fois que l'on atteint
la tête d'une liste**

?-

Définissons **acclen/3**

- La valeur initiale de l'accumulateur est 0.
- On ajoute 1 à l'accumulateur chaque fois qu'il est possible de traiter la tête d'une liste.
- Quand nous atteignons la liste vide, l'accumulateur contient la longueur de la liste.

```
acclen([_|L], OldAcc, Length) :-  
    NewAcc is OldAcc + 1,  
    acclen(L, NewAcc, Length).
```

```
acclen([], Acc, Length) :-  
    Length = Acc.
```

```
acclen([], Acc, Length) :-  
    Length = Acc.
```

```
acclen([_|L], OldAcc, Length) :-  
    NewAcc is OldAcc + 1,  
    acclen(L, NewAcc, Length).
```

Ajoute 1 à
l'accumulateur chaque
fois que l'on atteint
la tête d'une liste

- Par exemple :

```
?- acclen([a,b,c], 0, Len).  
acclen([], Acc, Acc).
```

donne **Len = 3** et **yes**.

Arbre de recherche pour **acclen/3**

- Pour la requête **?- acclen([a,b,c], 0, Len).** :

```
acclen([], Acc, Acc).  
acclen([_|L], OldAcc, Length) :-  
    NewAcc is OldAcc + 1,  
    acclen(L, NewAcc, Length).
```

On peut représenter l'arbre de recherche comme suit :

```

?- acclen([a,b,c], 0, Len).
/  \
no  ?- acclen([b,c], 1, Len).
      /  \
      no  ?- acclen([c], 2, Len).
            /  \
            no  ?- acclen([], 3, Len).
                  /  \
                  Len=3 no

```

Ajouter un prédicat pour emballer (wrapper)

- On peut ajouter un prédicat `length/2` pour emballer `acclen/3` :

```

length(List, Length) :-
    acclen(List, 0, Length).

```

- Exemple d'utilisation :
 - `?- length([a,b,c], X).` donne `X = 3` et `yes`.

```

acclen([ ],Acc,Acc).

acclen([ _|L],OldAcc,Length):-
    NewAcc is OldAcc + 1,
    acclen(L,NewAcc,Length).

length(List,Length):-
    acclen(List,0,Length).

```

```

?-length([a,b,c], X).
X=3
yes

```


Réversivité terminale

- **Prédicats réversifs terminaux** : Les résultats sont complètement calculés lorsque la fin de la récursion est atteinte.
- **Prédicats réversifs non terminaux** : Il reste des buts à réaliser quand la fin de récursivité est atteinte.
acclen/3 est récursif terminal, mais pas **len/2**.
- Différences:
-

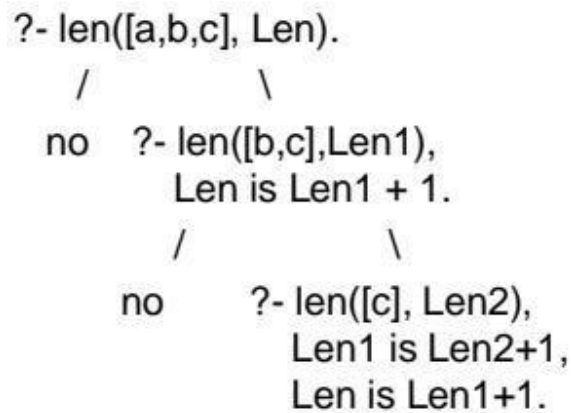
```
acclen([], Acc, Acc).
acclen([_|L], OldAcc, Length) :-
    NewAcc is OldAcc + 1,
    acclen(L, NewAcc, Length).


len([], 0).
len([_|L], NewLength) :-
    len(L, Length),
    NewLength is Length + 1.
```

Comparaison - Récursif terminal Récursif non terminal

- Exemple avec **len/2** :

```
?- len([a,b,c], Len).
```



 Visual representation of a logic or deduction tree related to the calculation of list lengths in a logical or Prolog programming context.

L'arbre de recherche pour **len/2** montre que les calculs sont effectués après l'appel récursif.

- En revanche, l'arbre de recherche pour **acclen/3** montre que les calculs sont effectués pendant l'appel récursif, ce qui en fait une version récursive terminale.

 Tree-like structure illustrating a series of queries and their resolutions in a logical or computational context.

Exercices

- Exercise 5.1
- Exercise 5.2
- Exercise 5.3

Comparaison d'entiers

- Prolog fournit des opérateurs pour comparer des entiers.
- Certains prédicats arithmétiques font des opérations eux-mêmes, sans **is/2**.
- Ces opérateurs forcent l'évaluation des arguments de gauche et de droite.

Arit hmétique	Prolog
$x < y$	$X < Y$
$x \leq y$	$X \leq Y$
$x = y$	$X = Y$
$x \neq y$	$X \neq Y$
$x \geq y$	$X \geq Y$
$x > y$	$X > Y$

- Exemples :
 - $?- 2 < 4+1.$ donne **yes**.
 - $?- 4+3 > 5+5.$ donne **no**.

$?- 4 = 4.$

yes

$?- 2+2 = 4.$

no

$?- 2+2 =:= 4.$

yes

- Autres Exemples :
 - $?- 4 = 4.$ donne **yes**.
 - $?- 2+2 = 4.$ donne **no**.
 - $?- 2+2 =:= 4.$ donne **yes**.

Références :

SWI-Prolog. (s. d.). <https://www.swi-prolog.org/>

Learn ProLog Now ! (s. d.). <https://lpn.swi-prolog.org/lpnpage.php?pageid=top>

PROLOG – Programmation Logique

© Patrick Blackburn, Johan Bos & Kristina Striegnitz

Traduction : Yannick Estève

Cours dispensé à CERI – Avignon Université

[Lien si disponible]