Cours 5: récursivité terminale

- Sujets
  - Réviser les capacités internes à Prolog, lui permettant de faire de l'**arithmétique**
  - Leur appliquer des problèmes de traitement de listes simples, en utilisant des **accumulateurs**
  - Présenter des prédicats **récursif terminaux** et illustrer pourquoi ils sont plus efficaces que des prédicats récursifs simples.
- Ce cours correspond au chapitre 5 ("Arithmétique") du livre LPN, www.learnprolognow.org
- Exercises
  - Exos de LPN: 5.1, 5.2, 5.3

Arithmétique en Prolog

- Prolog fournit quelques outils simples pour l'arithmétique
- Entiers et nombres réels

Arithmétique	Prolog
2 + 3 = 5	?- 5 is 2+3.
3 x 4 = 12	?- 12 is 3*4.
5 – 3 = 2	?- 2 is 5-3.
3 – 5 = -2	?- -2 is 3-5.
4 : 2 = 2	?- 2 is 4/2.
1 is the remainder when 7 is divided by 2	?- 1 is mod(7,2).

Exemples de requêtes

```
?- 10 is 5+5.
yes

?- 4 is 2+3.
no

?- X is 3 * 4.
X=12
yes

?- R is mod(7,2).
R=1
yes
```

Définir des prédicats arithmétiques

```
ajoute_3_et_double(X, Y):-
    Y is (X+3) * 2.
```

Définir des prédicats arithmétiques

```
ajoute_3_et_double(X, Y):-
    Y is (X+3) * 2.
```

```
?- ajoute_3_et_double(1,X).
X=8
yes

?- ajoute_3_et_double(2,X).
X=10
yes
```

Vu de plus près

- Il faut comprendre que +, -, / et * ne font pas d'arithmétique
- Les expressions comme 3+2, 4-7, 5/5 sont des termes simples
  - Foncteur: +, -, /, *
  - Arité: 2
  - Arguments: entiers

Vu de plus près

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

Vu de plus près

```
?- X = 3 + 2.
X = 3+2
yes

?-
```

Vu de plus près

```
?- X = 3 + 2.
X = 3+2
yes

?- 3 + 2 = X.
```

Vu de plus près

```
?- X = 3 + 2.
X = 3+2
yes

?- 3 + 2 = X.
X = 3+2
yes

?-
```

Le prédicat is/2

- Pour forcer Prolog à évaluer les expressions arithmétiques on doit utiliser **is**
  comme nous l'avons fait ici.
- Ceci fait quitter le mode de raisonnement habituel de Prolog
- Donc, pas surprenant qu'il y ait des restrictions à cette capacité d'évaluation.

Le prédicat is/2

```
?- X is 3 + 2.
```

Le prédicat is/2

```
?- X is 3 + 2.
X = 5
yes

?-
```

Le prédicat is/2

```
?- X is 3 + 2.
X = 5
yes

?- 3 + 2 is X.
```

Le prédicat is/2

```
?- X is 3 + 2.
X = 5
yes

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

?-
```

Le prédicat is/2

```
?- 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.
```

Le prédicat is/2

```
?- 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

?-
```

Le prédicat is/2

```
?- 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

?- ajoute_3_et_double(X,12).
```

Le prédicat is/2

```
?- 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

?- ajoute_3_et_double(X,12).
ERROR: is/2: Arguments are not sufficiently instantiated
```

Restrictions à l'utilisation de is/2

- Les expressions à la droite de **is** *peuvent* contenir des variables
- Mais quand Prolog les évalues, ces variables doivent être instanciées par un terme Prolog
  - sans variables
  - représentant une expression arithmétique

Notation

- Deux dernières remarques sur les expressions arithmétiques
  - 3+2, 4/2, 4-5 sont des des termes Prolog simples avec un notation commode: **3+2** est en réalité **+(3,2)** etc.
  - Le prédicat **is** est un prédicat Prolog à deux arguments

is(-Nombre, +Expr)

Notation

- Deux dernières remarques sur les expressions arithmétiques
  - 3+2, 4/2, 4-5 sont des des termes Prolog simples avec un notation commode: **3+2** est en réalité **+(3,2)** etc.
  - Le prédicat **is** est un prédicat Prolog à deux arguments

is(-Nombre, +Expr)

```
?- is(X,+(3,2)).
X = 5
yes
```

Modes de passage d'arguments

de: swiprolog help, notation of predicate descriptions:

\+ Argument must be fully instantiated to a term that satisfies the required argument type. Think of the argument as input.

\- Argument must be unbound. Think of the argument as output.

? Argument must be bound to a partial term of the indicated type. Note that a variable is a partial term for any type. Think of the argument as either input or output or both input and output.

Exo 5.1: Comment Prolog répond-il?

```
1. X = 3*4.

2. X is 3*4.

3. 4 is X.

4. X = Y.

5. 3 is 1+2.

6. 3 is +(1,2).

7. 3 is X+2.

8. X is 1+2.

9. 1+2 is 1+2.
```

Exo 5.1: Comment Prolog répond-il?

```
1. X = 3*4.

2. X is 3*4.

3. 4 is X.

4. X = Y.

5. 3 is 1+2.

6. 3 is +(1,2).

7. 3 is X+2.

8. X is 1+2.

9. 1+2 is 1+2.
```

1. X = 3*4. Variable X est instantiée avec le terme complexe 3*4.

Exo 5.1: Comment Prolog répond-il?

```
1. X = 3*4.

2. X is 3*4.

3. 4 is X.

4. X = Y.

5. 3 is 1+2.

6. 3 is +(1,2).

7. 3 is X+2.

8. X is 1+2.

9. 1+2 is 1+2.
```

1. X = 3*4. Variable X est instantiée avec le terme complexe 3*4.
2. X = 12.

Exo 5.1: Comment Prolog répond-il?

```
1. X = 3*4.

2. X is 3*4.

3. 4 is X.

4. X = Y.

5. 3 is 1+2.

6. 3 is +(1,2).

7. 3 is X+2.

8. X is 1+2.

9. 1+2 is 1+2.
```

1. X = 3*4. Variable X est instantiée avec le terme complexe 3*4.
2. X = 12.
3. ERROR: Arguments are not sufficiently instantiated.

Exo 5.1: Comment Prolog répond-il?

```
1. X = 3*4.

2. X is 3*4.

3. 4 is X.

4. X = Y.

5. 3 is 1+2.

6. 3 is +(1,2).

7. 3 is X+2.

8. X is 1+2.

9. 1+2 is 1+2.
```

1. X = 3*4. Variable X est instantiée avec le terme complexe 3*4.
2. X = 12.
3. ERROR: Arguments are not sufficiently instantiated.
4. X = Y.

Exo 5.1: Comment Prolog répond-il?

```
1. X = 3*4.

2. X is 3*4.

3. 4 is X.

4. X = Y.

5. 3 is 1+2.

6. 3 is +(1,2).

7. 3 is X+2.

8. X is 1+2.

9. 1+2 is 1+2.
```

1. X = 3*4. Variable X est instantiée avec le terme complexe 3*4.
2. X = 12.
3. ERROR: Arguments are not sufficiently instantiated.
4. X = Y.
5. yes.

Exo 5.1: Comment Prolog répond-il?

```
1. X = 3*4.

2. X is 3*4.

3. 4 is X.

4. X = Y.

5. 3 is 1+2.

6. 3 is +(1,2).

7. 3 is X+2.

8. X is 1+2.

9. 1+2 is 1+2.
```

1. X = 3*4. Variable X est instantiée avec le terme complexe 3*4.
2. X = 12.
3. ERROR: Arguments are not sufficiently instantiated.
4. X = Y.
5. yes.
6. yes.

Exo 5.1: Comment Prolog répond-il?

```
1. X = 3*4.

2. X is 3*4.

3. 4 is X.

4. X = Y.

5. 3 is 1+2.

6. 3 is +(1,2).

7. 3 is X+2.

8. X is 1+2.

9. 1+2 is 1+2.
```

1. X = 3*4. Variable X est instantiée avec le terme complexe 3*4.
2. X = 12.
3. ERROR: Arguments are not sufficiently instantiated.
4. X = Y.
5. yes.
6. yes.
7. ERROR: Arguments are not sufficiently instantiated.

Exo 5.1: Comment Prolog répond-il?

```
1. X = 3*4.

2. X is 3*4.

3. 4 is X.

4. X = Y.

5. 3 is 1+2.

6. 3 is +(1,2).

7. 3 is X+2.

8. X is 1+2.

9. 1+2 is 1+2.
```

1. X = 3*4. Variable X est instantiée avec le terme complexe 3*4.
2. X = 12.
3. ERROR: Arguments are not sufficiently instantiated.
4. X = Y.
5. yes.
6. yes.
7. ERROR: Arguments are not sufficiently instantiated.
8. X = 3.

Exo 5.1: Comment Prolog répond-il?

```
1. X = 3*4.

2. X is 3*4.

3. 4 is X.

4. X = Y.

5. 3 is 1+2.

6. 3 is +(1,2).

7. 3 is X+2.

8. X is 1+2.

9. 1+2 is 1+2.
```

1. X = 3*4. Variable X est instantiée avec le terme complexe 3*4.
2. X = 12.
3. ERROR: Arguments are not sufficiently instantiated.
4. X = Y.
5. yes.
6. yes.
7. ERROR: Arguments are not sufficiently instantiated.
8. X = 3.
9. no. Prolog evalue l'expression arithmétique à droite de is/2. Puis il essaie d'unifier ce terme avec le terme à gauche de is/2. Ceci échoue comme le nombre 3 n'unifie pas avec le terme complexe 1+2.

Exo 5.1: Comment Prolog répond-il?

```
10. is(X,+(1,2)).

11. 3+2 = +(3,2).


12. *(7,5) = 7*5.

13. *(7,+(3,2)) = 7*(3+2).

14. *(7,(3+2)) = 7*(3+2).

15. 7*3+2 = *(7+(3,2))

15. *(7,(3+2)) = 7*(+(3,2)).
```

Exo 5.1: Comment Prolog répond-il?

Réponses:

```
10. is(X,+(1,2)).

11. 3+2 = +(3,2).


12. *(7,5) = 7*5.

13. *(7,+(3,2)) = 7*(3+2).

14. *(7,(3+2)) = 7*(3+2).

15. 7*3+2 = *(7+(3,2))

15. *(7,(3+2)) = 7*(+(3,2)).
```

Exo 5.1: Comment Prolog répond-il?

```
10. is(X,+(1,2)).

11. 3+2 = +(3,2).


12. *(7,5) = 7*5.

13. *(7,+(3,2)) = 7*(3+2).

14. *(7,(3+2)) = 7*(3+2).

15. 7*3+2 = *(7+(3,2))

15. *(7,(3+2)) = 7*(+(3,2)).
```

Réponses:

```
10.  X = 3.
```

Exo 5.1: Comment Prolog répond-il?

```
10. is(X,+(1,2)).

11. 3+2 = +(3,2).

12. *(7,5) = 7*5.

13. *(7,+(3,2)) = 7*(3+2).

14. *(7,(3+2)) = 7*(3+2).

15. 7*3+2 = *(7+(3,2))

15. *(7,(3+2)) = 7*(+(3,2)).
```

Réponses:

```
10.  X = 3.

11.  yes. 3+2 et +(3,2) sont
deux manières d'écrire le
même terme.
```

Exo 5.1: Comment Prolog répond-il?

```
10. is(X,+(1,2)).

11. 3+2 = +(3,2).

12. *(7,5) = 7*5.

13. *(7,+(3,2)) = 7*(3+2).

14. *(7,(3+2)) = 7*(3+2).

15. 7*3+2 = *(7+(3,2))

15. *(7,(3+2)) = 7*(+(3,2)).
```

Réponses:

```
10.  X = 3.

11.  yes. 3+2 et +(3,2) sont
deux manières d'écrire le
même terme.
12.  yes.
```

Exo 5.1: Comment Prolog répond-il?

```
10. is(X,+(1,2)).

11. 3+2 = +(3,2).

12. *(7,5) = 7*5.

13. *(7,+(3,2)) = 7*(3+2).

14. *(7,(3+2)) = 7*(3+2).

15. 7*3+2 = *(7+(3,2))

15. *(7,(3+2)) = 7*(+(3,2)).
```

Réponses:

10. X = 3.

11. yes. 3+2 et +(3,2) sont deux manières d'écrire le même terme.

12. yes.

13. yes.

Exo 5.1: Comment Prolog répond-il?

```
10. is(X,+(1,2)).

11. 3+2 = +(3,2).

12. *(7,5) = 7*5.

13. *(7,+(3,2)) = 7*(3+2).

14. *(7,(3+2)) = 7*(3+2).

15. 7*3+2 = *(7+(3,2))

15. *(7,(3+2)) = 7*(+(3,2)).
```

Réponses:

```
10.  X = 3.

11.  yes. 3+2 et +(3,2) sont
deux manières d'écrire le
même terme.

12.  yes.

13.  yes.

14.  yes.
```

Exo 5.1: Comment Prolog répond-il?

```
10. is(X,+(1,2)).

11. 3+2 = +(3,2).

12. *(7,5) = 7*5.

13. *(7,+(3,2)) = 7*(3+2).

14. *(7,(3+2)) = 7*(3+2).

15. 7*3+2 = *(7+(3,2))

15. *(7,(3+2)) = 7*(+(3,2)).
```

Réponses:

```
10.  X = 3.

11.  yes. 3+2 et +(3,2) sont
deux manières d'écrire le
même terme.

12.  yes.

13.  yes.

14.  yes.

15.  no.
```

Exo 5.1: Comment Prolog répond-il?

```
10. is(X,+(1,2)).

11. 3+2 = +(3,2).

12. *(7,5) = 7*5.

13. *(7,+(3,2)) = 7*(3+2).

14. *(7,(3+2)) = 7*(3+2).

15. 7*3+2 = *(7+(3,2))

15. *(7,(3+2)) = 7*(+(3,2)).
```

Réponses:

```
10.  X = 3.

11.  yes. 3+2 et +(3,2) sont
deux manières d'écrire le
même terme.

12.  yes.

13.  yes.

14.  yes.

15.  no.

16.  yes.
```

Récursivité terminale

- introduite à l'exemple d'un prédicat qui détermine la longueur d'une liste....

Arithmétique et listes

- Quelle est la longueur d'une liste?
  - La longueur de la liste vide est zéro.
  - La longueur d'une liste non vide est n+1, si n est la longueur de sa queue.

Longueur d'une liste en Prolog

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

```
?-
```

Longueur d'une liste en Prolog

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

```
?- long([a,b,c,d,e,[a,x],t],X).
```

Longueur d'une liste en Prolog

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

```
?- long([a,b,c,d,e,[a,x],t],X).
X=7
yes
?-
```

Accumulateurs

- En principe, long/2 est un bon programme
  - compréhensible
  - efficacité raisonnable, il semble... (!!)
- Mais il y a une autre méthode pour déterminer la longueur d'une liste
  - introduit la notion d'accumulateur
  - homologue de variables qui contiennent résultats intermédiaires

Définir accLong/3

- Le prédicat accLong(Liste,Acc,Longueur) avec trois arguments
  1. La liste dont la longueur doit être trouvée
  2. Un accumulateur, pour les valeurs intermédiaires de longueur
  3. La longueur de cette liste: un entier.

Définir accLong/3

- L'accumulateur de accLong/3
  - On lui donne la valeur 0 quand on appelle le prédicat
  - Traverse la liste récursivement. Ajoute 1 à l'accumulateur chaque fois qu'on trouvera un élément tête.
  - Quand on arrive à la liste vide, l'accumulateur contient la longueur de la liste!

Longueur d'une liste en Prolog

```
accLong([],Acc,Acc).

accLong([_|L],AccV,Longueur):-
    AccNouv is AccV + 1,
    accLong(L,AccNouv,Longueur).
```

```
?-
```

Longueur d'une liste en Prolog

```
accLong([],Acc,Acc).

accLong([_|L],AccV,Longu
    AccNouv is AccV + 1,
    accLong(L,AccNouv,Longueur).
```

augmenter l'accumulateur chaque fois que nous enlevons la tête de la liste

```
?-
```

Longueur d'une liste en Prolog

```
accLong([],Acc,Acc).

accLong([_|L],AccV,Lor
    AccNouv is AccV + 1,
    accLong(L,AccNouv,Longueur).
```

entrer en récursion avec la longueur mise à jour

```
?-
```

Longueur d'une liste en Prolog

```
accLong([],Acc,Acc).

accLong([_|L],AccV,Longueur) :-
    AccNouv is AccV + 1,
    accLong(L,AccNouv,Longueur).
```

Quand nous arrivons à la liste vide, l'accumulateur contient la longueur de la liste initiale

```
?-
```

Longueur d'une liste en Prolog

```
accLong([],Acc,Acc).

accLong([_|L],AccV,Longueur):-
    AccNouv is AccV + 1,
    accLong(L,AccNouv,Longueur).
```

```
?-
```

Longueur d'une liste en Prolog

```
accLong([],Acc,Acc).

accLong([_|L],AccV,Longueur):-
    AccNouv is AccV + 1,
    accLong(L,AccNouv,Longueur).
```

```
?-accLong([a,b,c],0,Len).
Len=3
yes

?-
```

Arbre de décision pour accLong/3

?- accLong([a,b,c],0,Len).

```
accLong([ ],Acc,Acc).

accLong([_|L],AccV,Longueur):-
    AccNouv is AccV + 1,
    accLong(L,AccNouv,Longueur).
```

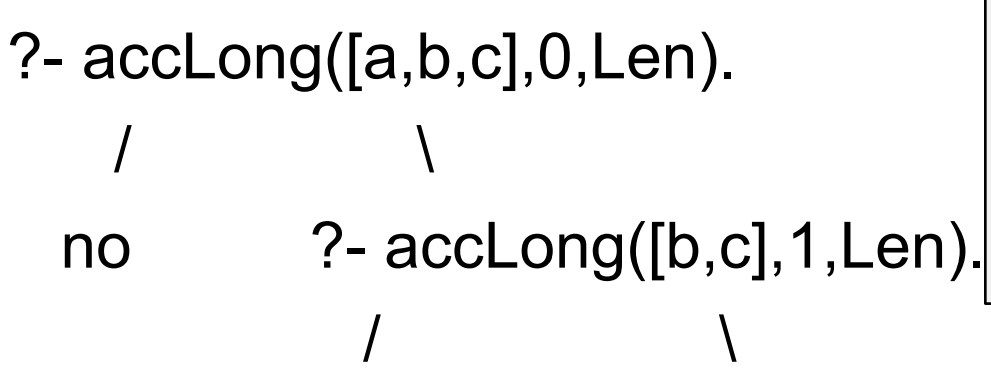
Arbre de décision pour accLong/3

?- accLong([a,b,c],0,Len).

/ \

```
accLong([ ],Acc,Acc).

accLong([_|L],AccV,Longueur):-
    AccNouv is AccV + 1,
    accLong(L,AccNouv,Longueur).
```

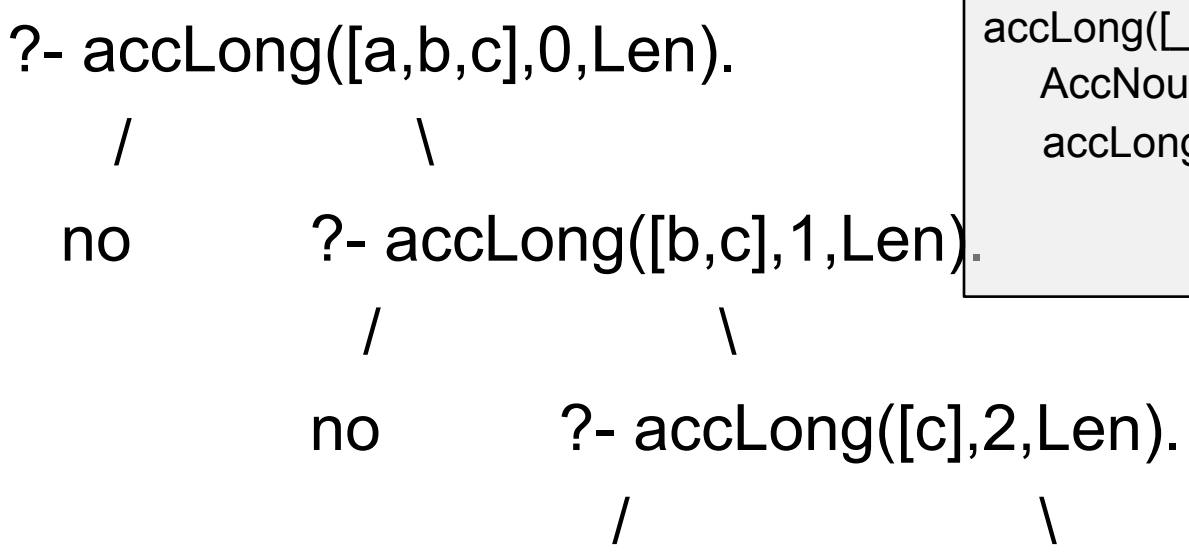
Arbre de décision pour accLong/3




```
accLong([ ],Acc,Acc).

accLong([_|L],AccV,Longueur):-
    AccNouv is AccV + 1,
    accLong(L,AccNouv,Longueur).
```

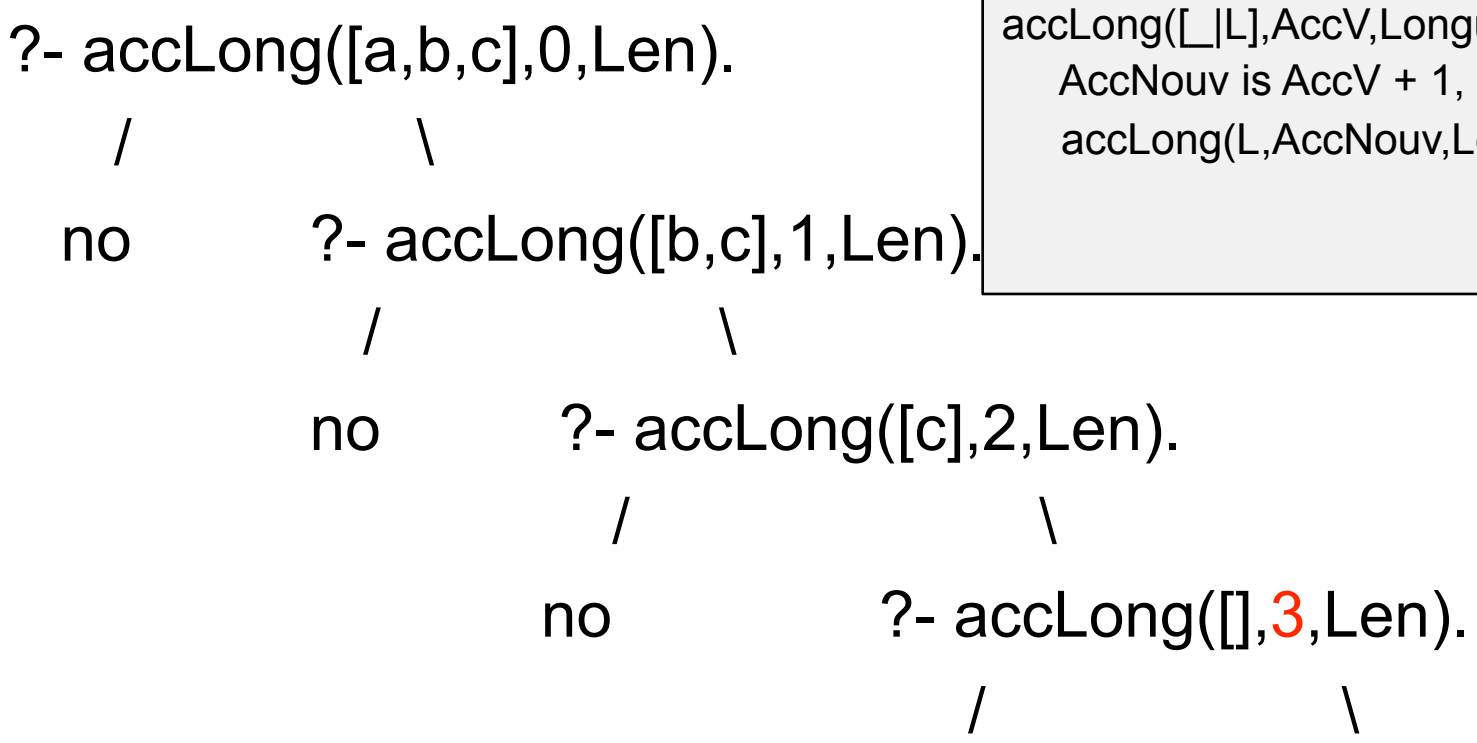
Arbre de décision pour accLong/3

```
accLong([ ],Acc,Acc).

accLong([_|L],AccV,Longueur):-
    AccNouv is AccV + 1,
    accLong(L,AccNouv,Longueur).
```



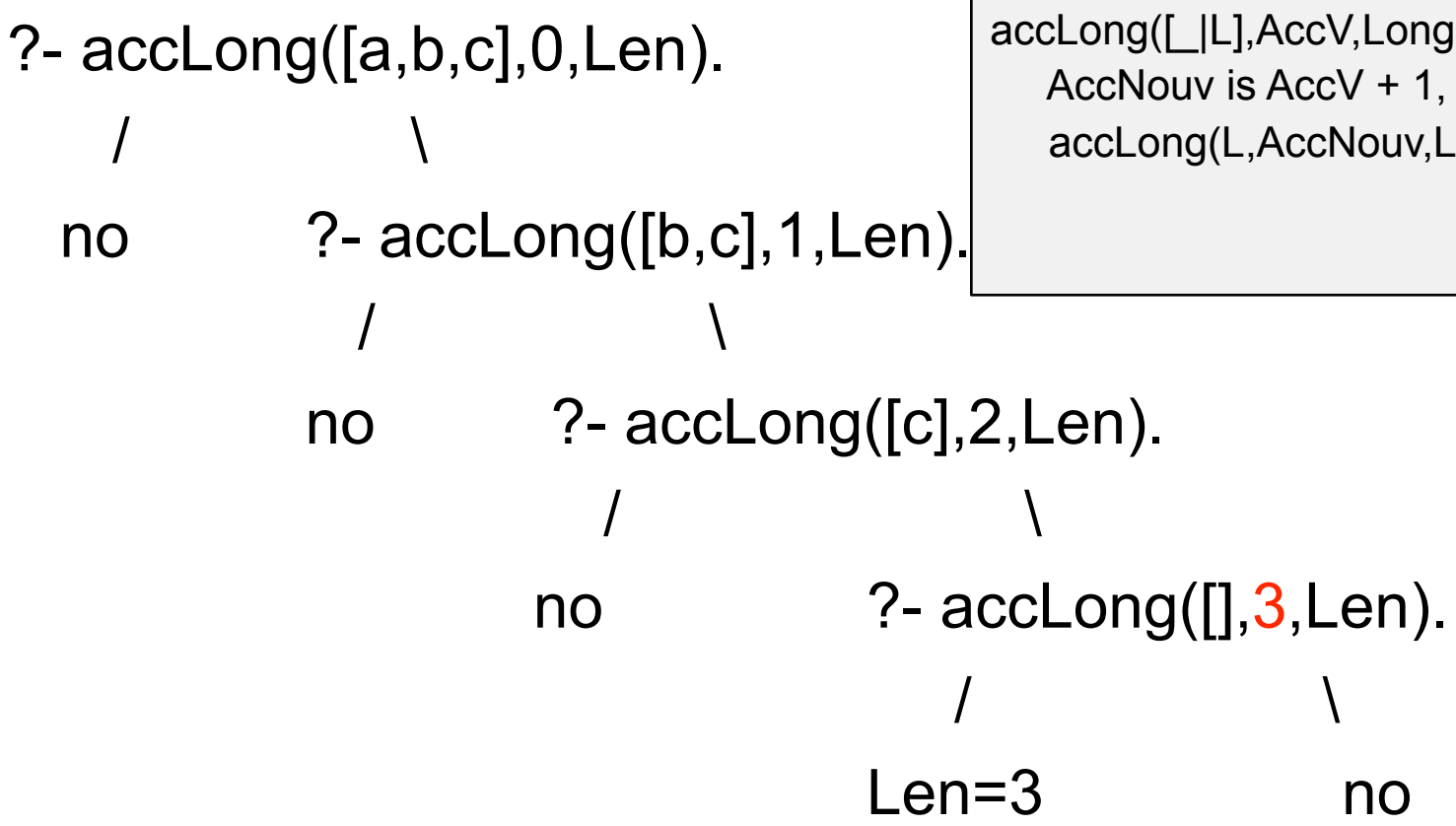
Arbre de décision pour accLong/3

```
accLong([ ],Acc,Acc).

accLong([_|L],AccV,Longueur):-
    AccNouv is AccV + 1,
    accLong(L,AccNouv,Longueur).
```



Arbre de décision pour accLong/3

```
accLong([ ],Acc,Acc).

accLong([_|L],AccV,Longueur):-
    AccNouv is AccV + 1,
    accLong(L,AccNouv,Longueur).
```



Ajouter un emballeur

```
accLong([ ],Acc,Acc).

accLong([ _|L],AccV,Longueur):-
    AccNouv is AccV + 1,
    accLong(L,AccNouv,Longueur).

longueur(Liste,Longueur):-
     accLong(Liste,0,Longueur).
```

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

Récursivité terminale

- Pourquoi préférer accLong/3 à len/2 ?
  - accLong/3 est récursif terminal, len/2 pas
- Différence:
  - Pour les prédicats récursifs terminaux le résultat est complètement calculé une fois qu'on atteint la clause de base.
  - Pour les prédicats récursifs qui ne sont pas récursifs terminaux, il reste des buts à résoudre quand on atteint la fin de la récurrence.

Comparaison: syntaxe

Pas récursif terminal

```
long([],0).

long([_|L],NouvLongueur):-
    long(L,Longueur),
    NouvLongueur is Longueur + 1.
```

Récursif terminal

```
accLong([],Acc,Acc).

accLong([_|L],AccV,Longueur):-
     AccNouv is AccV + 1,
     accLong(L,AccNouv,Longueur).
```

Arbre de décision pour len/2

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

```
long([],0).
long([_|L],NouvLongueur):-
    long(L,Longueur),
    NouvLongueur is
     Longueur + 1.
```

Arbre de décision pour len/2

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

/ \

no ?- long([b,c],Len1),
Len is Len1 + 1.

```
long([],0).
long([_|L],NouvLongueur):-
    long(L,Longueur),
    NouvLongueur is
     Longueur + 1.
```

Arbre de décision pour len/2

```
?- long([a,b,c], Len).
      /                \
   no    ?- long([b,c],Len1),
              Len is Len1 + 1.
              /                \
          no       ?- long([c], Len2),
                      Len1 is Len2+1,
                      Len is Len1+1.
```

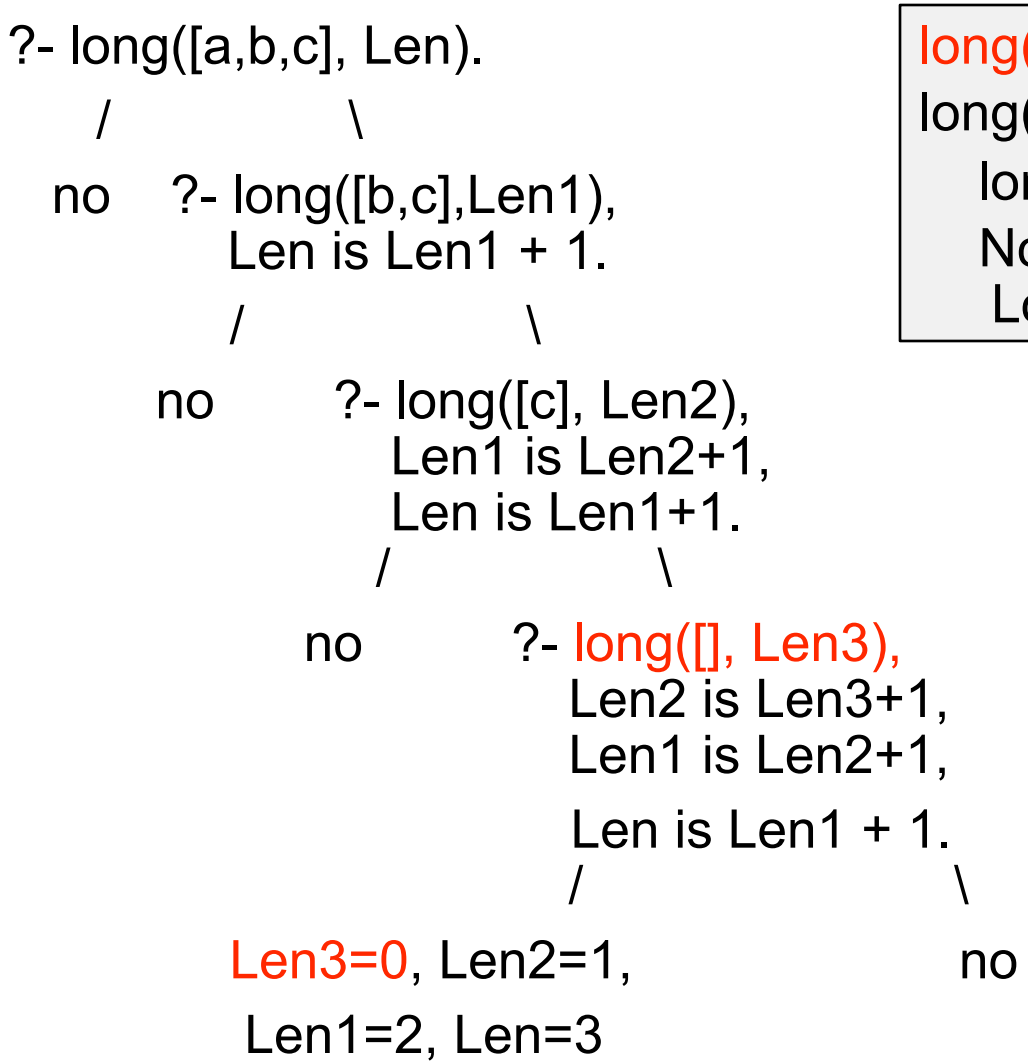
```
long([],0).
long([_|L],NouvLongueur):-
    long(L,Longueur),
    NouvLongueur is
     Longueur + 1.
```

Arbre de décision pour len/2

```
?- long([a,b,c], Len).
      /              \
   no    ?- long([b,c],Len1),
              Len is Len1 + 1.
              /              \
         no       ?- long([c], Len2),
                     Len1 is Len2+1,
                     Len is Len1+1.
                    /              \
                no       ?- long([], Len3),
                            Len2 is Len3+1,
                            Len1 is Len2+1,
                            Len is Len1 + 1.
```

```
long([],0).
long([_|L],NouvLongueur):-
    long(L,Longueur),
    NouvLongueur is
     Longueur + 1.
```

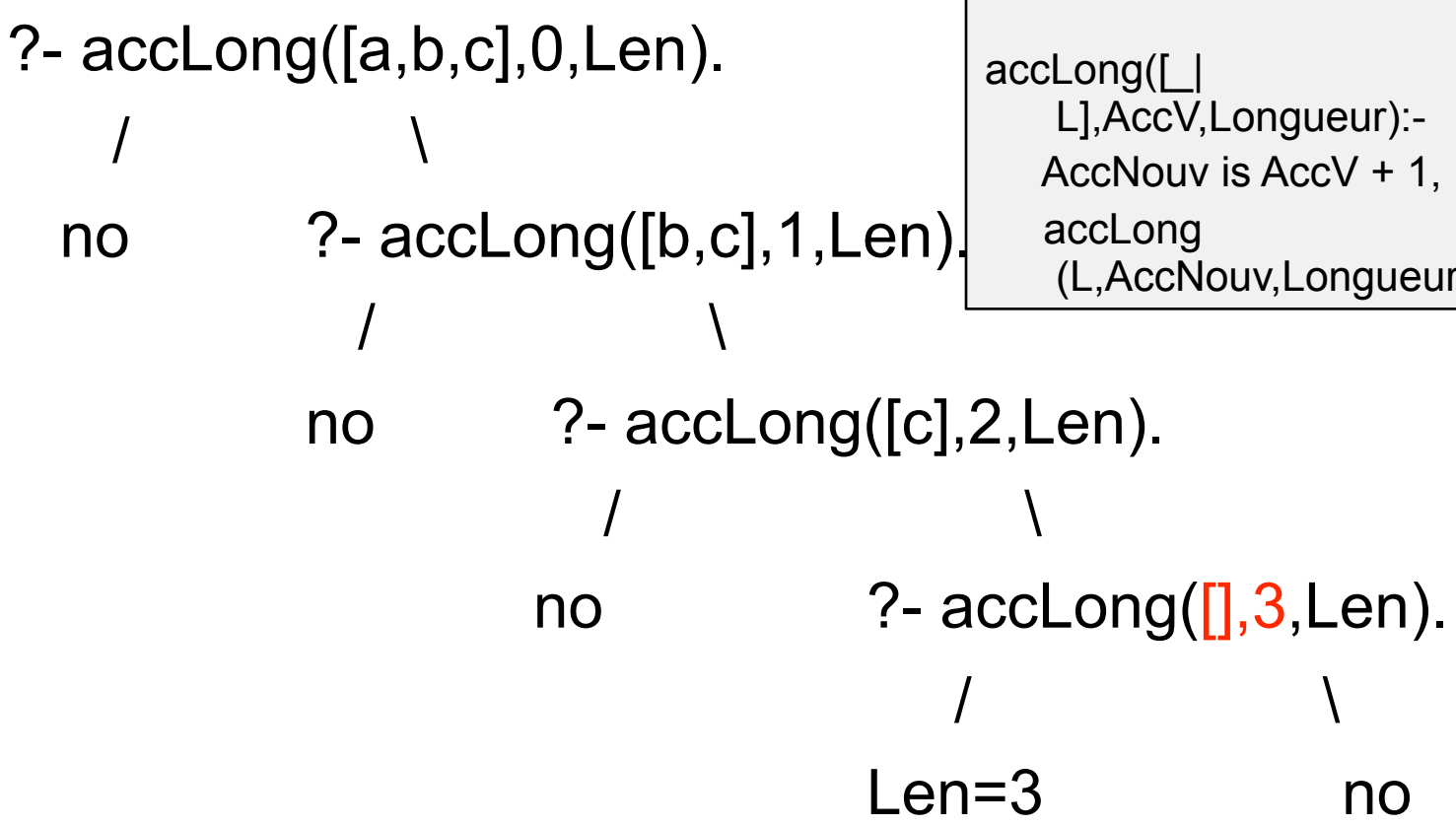
Arbre de décision pour len/2




```
long([],0).
long([_|L],NouvLongueur):-
    long(L,Longueur),
    NouvLongueur is
     Longueur + 1.
```

Arbre de décision pour accLong/3

```
accLong([ ],Acc,Acc).

accLong([_|
    L],AccV,Longueur):-
    AccNouv is AccV + 1,
    accLong
     (L,AccNouv,Longueur).
```



Exo 5.2a

- Définissez un prédicat à deux places **incremente/2** qui est vrai seulement quand son deuxième argument est un entier supérieur de un à son premier argument.
- Par exemple, **incremente(4,5)** est vrai, alors que **incremente(4,6)** est faux.

Solution

```
incremente(X,Y):- Y is X + 1.
```

cette solution permet des requêtes
- pour tester
- pour générer en position du deuxième argument

Exo 5.2b

- Définissez un prédicat à trois places **somme/3** qui est vrai seulement quand son troisième argument est la somme des deux premiers.
- Par exemple **somme(4,5,9)** est vrai, alors que **somme(4,6,12)** est faux.

Solution

```
somme(X,Y,Z):-  Z is X + Y.
```

Exo 5.3

- Ecrivez le prédicat **ajouteUn/2** dont le premier argument est une liste d'entiers, et dont le deuxième argument est une liste d'entiers obtenue en ajoutant 1 à chaque entier de la première liste. Par exemple, la requête

  ajouteUn([1,2,7,2],X)

  doit donner

  X=[2,3,8,3].

Solution

```
ajouteUn([],[]).

ajouteUn([H|T],[H1|T1]):-
        H1 is H + 1,
        ajouteUn(T,T1).
```

Comparer les entiers

- Certains prédicats arithmétiques de Prolog font vraiment des opérations arithmétique par eux-mêmes.
- Ce sont les opérateurs qui comparent les entiers.

Comparer les entiers

Arithmétique	Prolog
$x < y$	X < Y
$x \leq y$	X =< Y
$x = y$	X =:= Y
$x \neq y$	X =\= Y
$x \geq y$	X >= Y
$x > y$	X > Y

Opérateurs de comparaison

- Ont les significations auxquelles on s'attend
- Forcent arguments gauches et droits à être évalués

```
?- 2 < 4+1.
yes

?- 4+3 > 5+5.
no
```

Opérateurs de comparaison

- Ont les significations auxquelles on s'attend
- Forcent arguments gauches et droits à être évalués

```
?- 4 = 4.
yes

?- 2+2 = 4.
no

?- 2+2 =:= 4.
yes
```

Comparer des entiers

- Nous allons définir un prédicat max/2 qui prend deux arguments, et est vrai si:
  - Le premier argument est une liste d'entiers
  - Le deuxième argument l'entier le plus grand de la liste
- Idée: passage par accMax/3
  - On utilise un accumulateur
  - L'accumulateur conserve l'entier le plus grand trouvé jusque là en parcourant la liste
  - Si on trouve une valeur supérieure, on met l'accumulateur à jour

Définition accMax/3

```
accMax([H|T],A,Max):-
    H > A,
    accMax(T,H,Max).

accMax([H|T],A,Max):-
    H =< A,
    accMax(T,A,Max).

accMax([],A,A).
```

```
?- accMax([1,0,5,4],0,Max).
Max=5
yes
```

Un emballeur max/2 pour accMmax/3

```
accMax([H|T],A,Max):-
    H > A,
    accMax(T,H,Max).

accMax([H|T],A,Max):-
    H =< A,
    accMax(T,A,Max).

accMax([],A,A).

max([H|T],Max):-
    accMax(T,H,Max).
```

```
?- max([1,0,5,4], Max).
Max=5
yes

?- max([-3, -1, -5, -4], Max).
Max= -1
yes

?-
```

Résumé

- Nous avons résumé l'arithmétique en Prolog
- Nous avons mis en évidence la différence entre les prédicats **récursif terminaux** et ce qui ne le sont pas
- Nous avons introduit une technique de programmation: les **accumulateurs**
- Nous avons montré l'idée de prédicats **emballeurs.**

TP demain, et chez vous SVP!

- Oui, encore plus de listes!
  - Définir le prédicat append/3, qui enchaîne deux listes
  - invertir une liste, d'abord naïvement en utilisant append/3, puis plus efficacement avec des accumulateurs.
  - et d'autres... avec récursivité terminale
  - Clé: l'accumulateur est une liste.
    - FIFO: first in first out (queue d'attente)
    - LIFO: last in first out (stack, pile)
    - ... (autres structures de données possibles!)

La semaine prochaine

- La coupure
- La négation par échec