Prolog
(Listas and Aritmética)

Parcialmente adaptado de

http://www.learnprolognow.org/

Listas

- Uma lista é uma sequência finita de elementos
- Exemplos de listas em Prolog:

[mia, vincent, jules, yolanda]
[mia, robber(honeybunny), X, 2, mia]
[ ]
[mia, [vincent, jules], [butch, friend(butch)]]
[[ ], dead(z), [2, [b,c]], [ ], Z, [2, [b,c]]]

Listas

- Os elementos das listas estão entre parêntesis rectos
- O comprimento de uma lista é o seu número de elementos
- Qualquer termo Prolog pode ser um elemento de uma lista
- Existe uma lista especial:
  a lista vazia [ ]

Cabeça (Head) e Resto (Tail)

- Uma lista não vazia tem duas partes
  - A cabeça
  - O resto
- A cabeça é o primeiro elemento da lista
- O resto é tudo menos o primeiro elemento
  - O resto de uma lista é sempre uma lista

Cabeça e Resto exemplo 1

- [mia, vincent, jules, yolanda]

  Head:
  Tail:

Cabeça e Resto exemplo 1

- [mia, vincent, jules, yolanda]

  Head: mia
  Tail:

Cabeça e Resto exemplo 1

- [mia, vincent, jules, yolanda]

  Head: mia
  Tail: [vincent, jules, yolanda]

Cabeça e Resto exemplo 2

- [[ ], dead(z), [2, [b,c]], [ ], Z, [2, [b,c]]]

  Head:
  Tail:

Cabeça e Resto exemplo 2

- [[ ], dead(z), [2, [b,c]], [ ], Z, [2, [b,c]]]

  Head: [ ]
  Tail:

Cabeça e Resto exemplo 2

- [[ ], dead(z), [2, [b,c]], [ ], Z, [2, [b,c]]]

  Head: [ ]
  Tail: [dead(z), [2, [b,c]], [ ], Z, [2, [b,c]]]

Cabeça e Resto exemplo 3

- [dead(z)]

  Head:
  Tail:

Cabeça e Resto exemplo 3

- [dead(z)]

  Head: dead(z)
  Tail:

Cabeça e Resto exemplo 3

- [dead(z)]

  Head: dead(z)
  Tail: [ ]

Cabeça e resto da lista vazia

- A lista vazia não tem nem cabeça nem resto
- Em Prolog, [ ] é uma lista simples especial sem nenhuma estrutura interna
- A lista vazia tem um papel importante nos predicados recursivos para o processamento de listas em Prolog

O operador |

- O Prolog tem um operador especial | que pode ser usado para decompor a lista em duas partes: cabeça e resto
- O operador | é essencial para escrever predicados de manipulação de listas

O operador |

```
?- [Head|Tail] = [mia, vincent, jules, yolanda].

Head = mia
Tail = [vincent,jules,yolanda]
yes

?-
```

O operador |

```
?- [X|Y] = [mia, vincent, jules, yolanda].

X = mia
Y = [vincent,jules,yolanda]
yes

?-
```

O operador |

```
?- [X|Y] = [ ].

no

?-
```

O operador |

```
?- [X,Y|Tail] = [[ ], dead(z), [2, [b,c]], [], Z, [2, [b,c]]] .

X = [ ]
Y = dead(z)
Z = _4543
Tail = [[2, [b,c]], [ ], Z, [2, [b,c]]]
yes

?-
```

Variáveis Anónimas

- Considere-se que estamos interessados no segundo e quarto elemento de uma lista

```
?- [X1,X2,X3,X4|Tail] = [mia, vincent, marsellus, jody, yolanda].
X1 = mia
X2 = vincent
X3 = marsellus
X4 = jody
Tail = [yolanda]
yes

?-
```

Variáveis Anónimas

- Existe uma forma mais simples de obter apenas a informação requerida:

```
?- [ _,X2, _,X4|_ ] = [mia, vincent, marsellus, jody, yolanda].
X2 = vincent
X4 = jody
yes

?-
```

O underscore é uma variável anónima

Variáveis Anónimas

- É usada quando é necessário usar uma variável mas não nos interessa saber o seu valor
- Cada ocorrência da variável anónima é independente

Member

- Uma das coisas mais básicas que gostaríamos de saber é se algo é um elemento de uma lista ou não
- Vamos escrever um predicado que quando é dado um termo X e uma lista L, indica se X pertence ou não a L
- Este predicado chama-se member/2

member/2

```
member(X,[X|T]).
member(X,[H|T]):- member(X,T).
```

```
?-
```

member/2

```
member(X,[X|T]).
member(X,[H|T]):- member(X,T).
```

```
?- member(yolanda,[yolanda,trudy,vincent,jules]).
```

member/2

```
member(X,[X|T]).
member(X,[H|T]):- member(X,T).
```

```
?- member(yolanda,[yolanda,trudy,vincent,jules]).
yes
?-
```

member/2

```
member(X,[X|T]).
member(X,[H|T]):- member(X,T).
```

```
?- member(vincent,[yolanda,trudy,vincent,jules]).
```

member/2

```
member(X,[X|T]).
member(X,[H|T]):- member(X,T).
```

```
?- member(vincent,[yolanda,trudy,vincent,jules]).
yes
?-
```

member/2

```
member(X,[X|T]).
member(X,[H|T]):- member(X,T).
```

```
?- member(zed,[yolanda,trudy,vincent,jules]).
```

member/2

```
member(X,[X|T]).
member(X,[H|T]):- member(X,T).
```

```
?- member(zed,[yolanda,trudy,vincent,jules]).
no
?-
```

member/2

```
member(X,[X|T]).
member(X,[H|T]):- member(X,T).
```

```
?- member(X,[yolanda,trudy,vincent,jules]).
```

member/2

```
member(X,[X|T]).
member(X,[H|T]):- member(X,T).
```

```
?- member(X,[yolanda,trudy,vincent,jules]).
X = yolanda;
X = trudy;
X = vincent;
X = jules;
no
```

Rescrevendo member/2

```
member(X,[X|_]).
member(X,[_|T]):- member(X,T).
```

Recursividade em listas

- O predicado member/2 acede recursivamente aos elementos da lista
  - faz alguma coisa na cabeça, e em seguida,
  - recursivamente faz a mesma coisa na cauda
- Esta técnica é muito comum em Prolog!

Exemplo: a2b/2

- O predicado a2b/2 recebe 2 listas como argumentos e sucede
  - se o primeiro argumento é uma lista de as, e
  - o segundo argumento é uma lista de bs exactamente do mesmo tamanho

```
?- a2b([a,a,a,a],[b,b,b,b]).
yes
?- a2b([a,a,a,a],[b,b,b]).
no
?- a2b([a,c,a,a],[b,b,b,t]).
no
```

Definir a2b/2: passo 1

```
a2b([],[]).
```

- Frequentemente, o melhor é pensar no caso mais simples
- Neste caso: a lista vazia

Definir a2b/2: passo 2

```
a2b([],[]).
a2b([a|L1],[b|L2]):- a2b(L1,L2).
```

- Agora pensar recursivamente!

Testar a2b/2

```
a2b([],[]).
a2b([a|L1],[b|L2]):- a2b(L1,L2).
```

```
?- a2b([a,a,a],[b,b,b]).
yes
?-
```

Testar a2b/2

```
a2b([],[]).
a2b([a|L1],[b|L2]):- a2b(L1,L2).
```

```
?- a2b([a,a,a,a],[b,b,b]).
no
?-
```

Testar a2b/2

```
a2b([],[]).
a2b([a|L1],[b|L2]):- a2b(L1,L2).
```

```
?- a2b([a,t,a,a],[b,b,b,c]).
no
?-
```

Testar a2b/2

```
a2b([],[]).
a2b([a|L1],[b|L2]):- a2b(L1,L2).
```

```
?- a2b([a,a,a,a,a], X).
X = [b,b,b,b,b]
yes
?-
```

Testar a2b/2

```
a2b([],[]).
a2b([a|L1],[b|L2]):- a2b(L1,L2).
```

```
?- a2b(X,[b,b,b,b,b,b,b]).
X = [a,a,a,a,a,a,a]
yes
?-
```

Aritmética em Prolog

- Inteiros e números reais

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

Exemplos

```
?- 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
```

Definir predicados com aritmética

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

Definir predicados com aritmética

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

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

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

Aritmética em Prolog

- É importante saber que apenas +, -, / e * não fazem operações aritméticas
- Expressões 3+2, 4-7, 5/5 são simples termos Prolog
  - Functor: +, -, /, *
  - Aridade: 2
  - Argumentos: inteiros

Aritmética em Prolog

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

Aritmética em Prolog

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

?-
```

Aritmética em Prolog

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

?- 3 + 2 = X.
```

Aritmética em Prolog

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

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

?-
```

O predicado is/2

- Para forçar o Prolog a avaliar expressões aritméticas, é necessário usar

  is

- Como este não é um predicado Prolog como os outros, tem algumas restrições

O predicado is/2

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

O predicado is/2

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

?-
```

O predicado is/2

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

?- 3 + 2 is X.
```

O predicado is/2

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

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

?-
```

O predicado 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.
```

O predicado 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

?-
```

Restrições ao uso do is/2

- Podem-se usar variáveis no lado direito do predicado **is**
- Mas quando o Prolog faz a avaliação, as variáveis têm que estar instanciadas com um termo Prolog sem variáveis
- Este termo Prolog tem que ser uma expressão aritmética

Notação

- Nas expressões aritméticas
  - 3+2, 4/2, 4-5 são simples termos Prolog na notação infixa:
    3+2 é o mesmo que **+(3,2)**
  - O predicado **is** é um predicado Prolog com dois argumentos

Notação

- Nas expressões aritméticas
  - 3+2, 4/2, 4-5 são simples termos Prolog na notação infixa :
    3+2 é o mesmo que **+(3,2)**
  - O predicado **is** é um predicado Prolog com dois argumentos

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

Aritmética e Listas

- Qual o tamanho de uma lista?
  - A lista vazia tem tamanho: zero;
  - A lista não-vazia tem tamanho: um mais o tamanho do seu resto.

Tamanho de uma lista em Prolog

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

```
?-
```

Tamanho de uma lista em Prolog

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

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

Tamanho de uma lista em Prolog

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

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

Acumuladores

- Este é um programa razoável
  - Fácil de entender
  - Relativamente eficiente
- Mas existe outro método para calcular o tamanho de uma lista
  - Introduzir a noção de acumulador
  - Acumuladores são variáveis que guardam resultados intermédios

Definição acclen/3

- O predicado acclen/3 tem três argumentos
  - A lista cujo tamanho queremos calcular
  - O tamanho da lista, um inteiro
  - Um acumulador, que guarda os valores intermédios do tamanho

Definição acclen/3

- O acumulador de acclen/3
  - O valor inicial do acumulador é 0
  - Adicionar 1 ao acumulador cada vez que se retira recursivamente a cabeça da lista
  - Quando se obtém a lista vazia, o acumulador contém o tamanho da lista

Tamanho de uma lista em Prolog

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

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

```
?-
```

Tamanho de uma lista em Prolog

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

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

adicionar 1 ao acumulador de cada vez que se retira a cabeça da lista

```
?-
```

Tamanho de uma lista em Prolog

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

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

Quando se obtém a list vazia, o acumulador contém o tamanho da lista

```
?-
```

Tamanho de uma lista em Prolog

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

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

```
?-
```

Tamanho de uma lista em Prolog

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

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

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

?-
```

Árvore de pesquisa do acclen/3

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

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

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

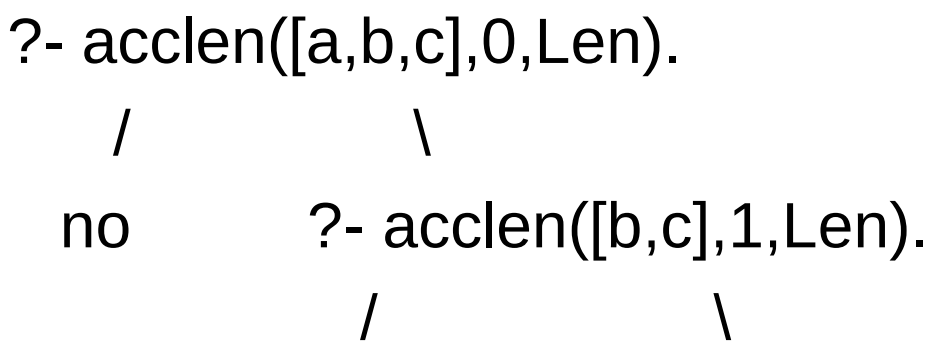
Árvore de pesquisa do acclen/3

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

/ \

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

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

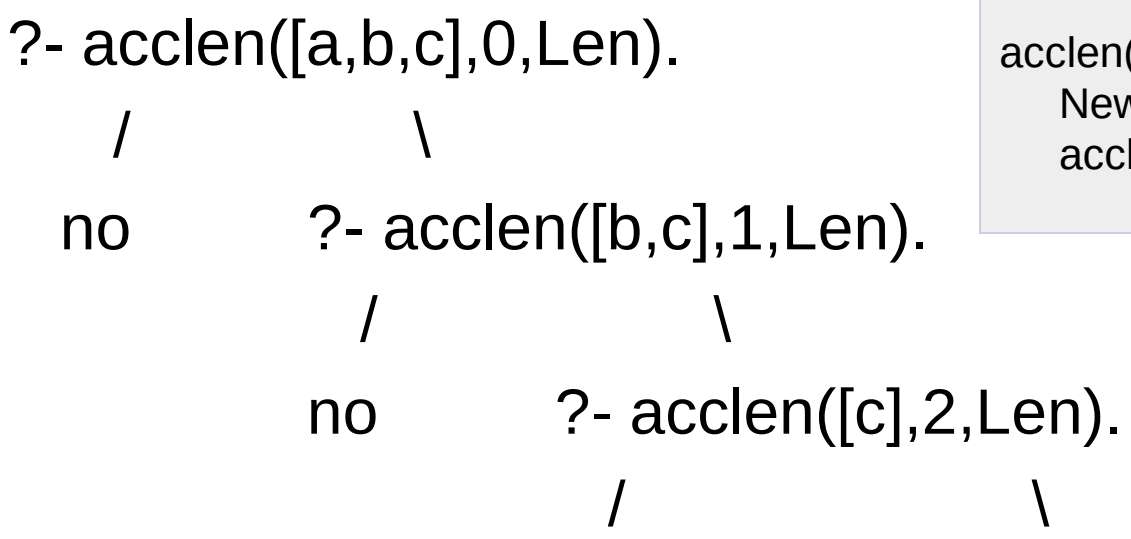
Árvore de pesquisa do acclen/3




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

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

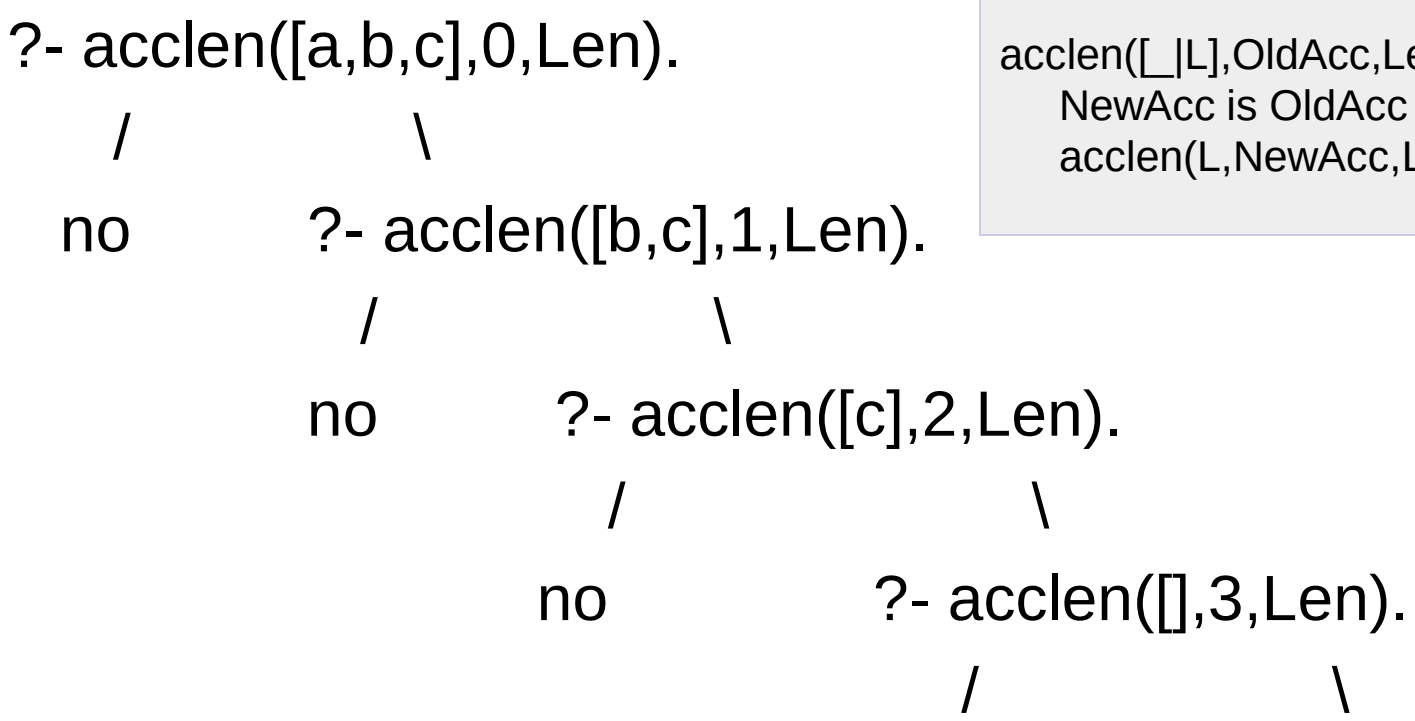
Árvore de pesquisa do acclen/3

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

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



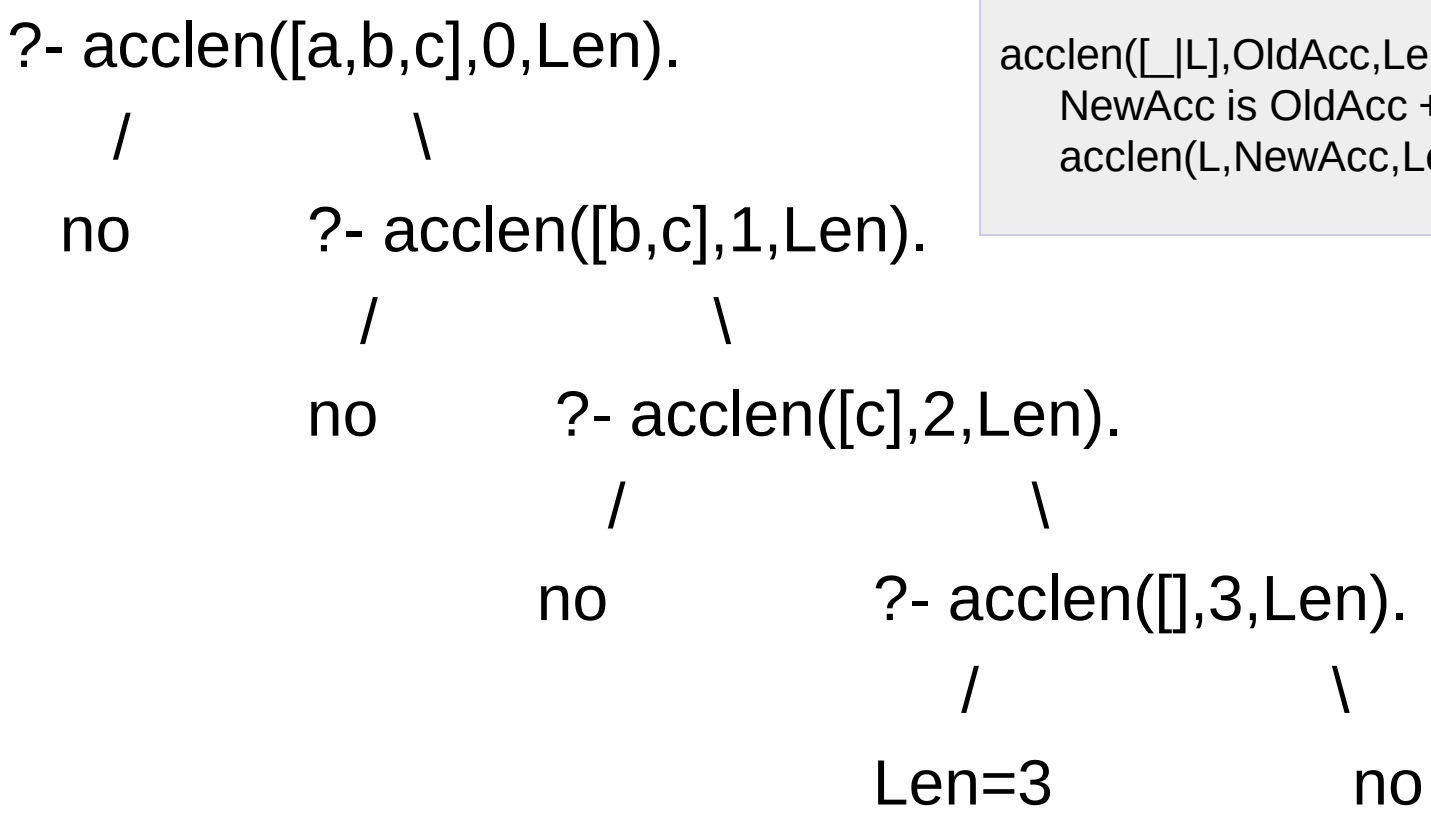
Árvore de pesquisa do acclen/3

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

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



Árvore de pesquisa do acclen/3

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

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



Adicionar um predicado para inicializar

```
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
```

Recursividade terminal

- Porque é que acclen/3 é melhor que len/2 ?
  - acclen/3 tem recursividade terminal, len/2 não
- Diferença:
  - Nos predicados com recursividade terminal, os resultados são completamente calculados quando se chega à cláusula base
  - Nos predicados com recursividade terminal, ainda existem goals no stack quando se chega à cláusula base

Comparação

Not tail-recursive

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

Tail-recursive

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

Árvore de pesquisa do len/2

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

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

Árvore de pesquisa do len/2

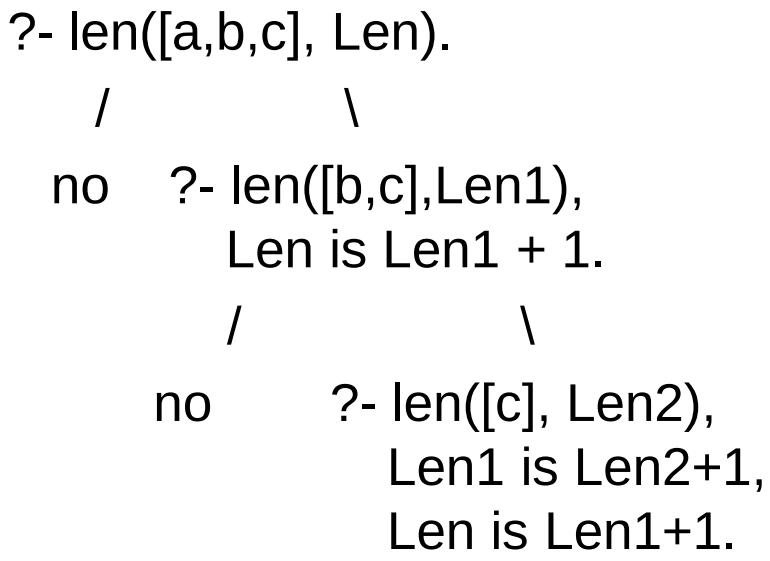
?- len([a,b,c], Len).

/ \

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

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

Árvore de pesquisa do len/2




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

Árvore de pesquisa do len/2

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

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

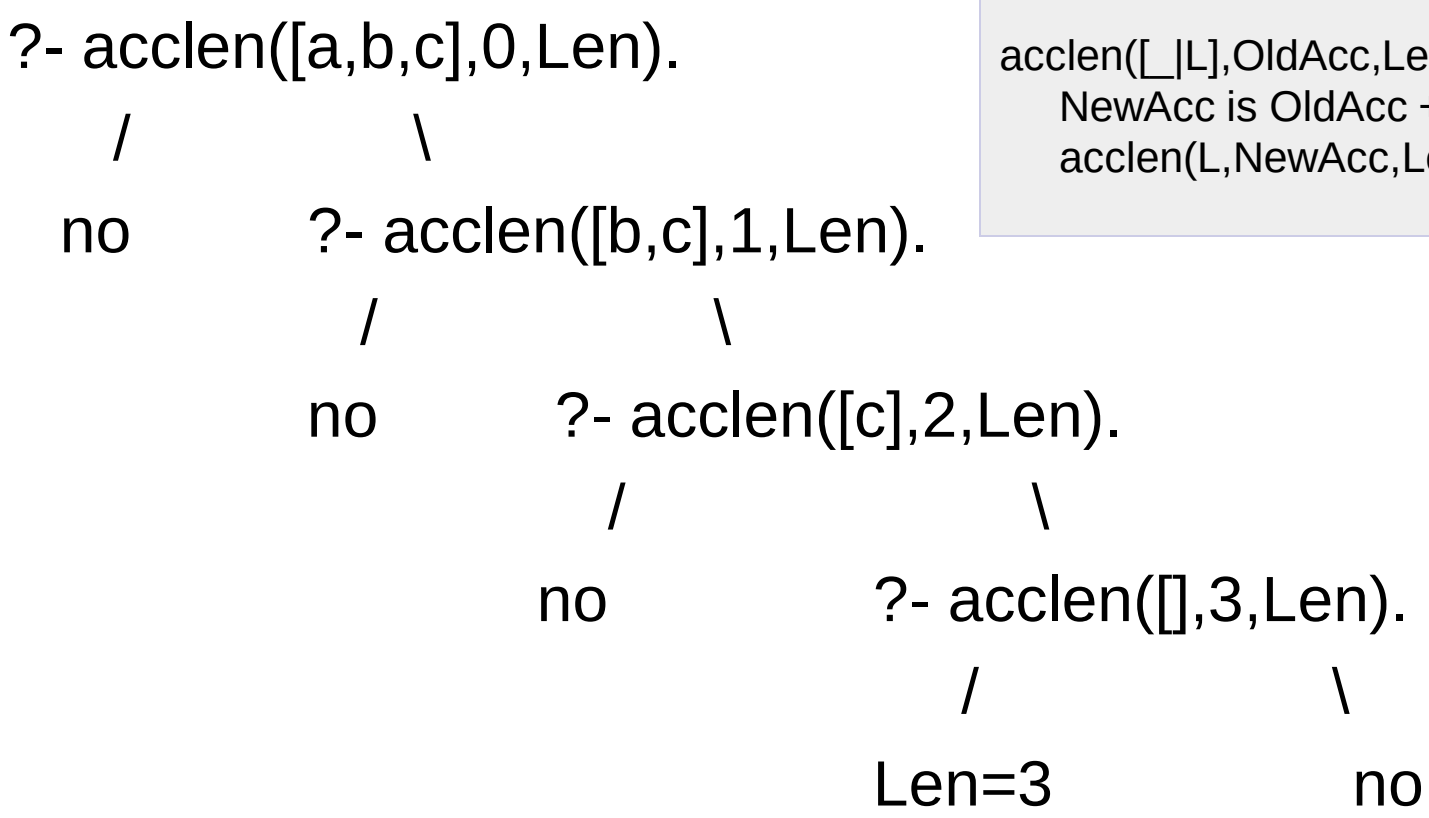
Árvore de pesquisa do len/2

```
?- len([a,b,c], Len).
    /              \
  no    ?- len([b,c],Len1),
            Len is Len1 + 1.
            /                \
       no        ?- len([c], Len2),
                    Len1 is Len2+1,
                    Len is Len1+1.
                   /                \
              no          ?- len([], Len3),
                             Len2 is Len3+1,
                             Len1 is Len2+1,
                             Len is Len1 + 1.
                             /                    \
            Len3=0, Len2=1,                      no
             Len1=2, Len=3
```

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

Árvore de pesquisa do acclen/3

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

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



Comparação de Inteiros

Arithmetic	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

Comparação de Inteiros

- Força a avaliação de ambos os argumentos (lado esquerdo e direito)

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

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

Operadores de Comparação

- Força a avaliação de ambos os argumentos (lado esquerdo e direito)

```
?- 4 = 4.
yes

?- 2+2 = 4.
no

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

Comparação de números

- Definir um predicado que recebe dois argumentos, e é verdadeiro quando:
  - O primeiro argumento é uma lista de inteiros
  - O segundo argumento é o maior inteiro da lista
- Ideia Base
  - Usar um acumulador
  - O acumulador guarda o maior valor encontrado até ao momento
  - Quando encontra um valor maior, actualiza o acumulador

Definição de 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
```

Predicado para inicializar max/2

```
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

?-
```

Append

- O **append/3** é um predicado importante cujos argumentos são todos listas
- append(L1,L2,L3) é verdadeiro se a lista L3 é o resultado da concatenação das listas L1 e L2

```
?- append([a,b,c,d],[3,4,5],[a,b,c,d,3,4,5]).
yes

?- append([a,b,c],[3,4,5],[a,b,c,d,3,4,5]).
no
```

Append

- Do ponto de vista procedimental, a utilização obvia do append/3 é para concatenar duas listas
- Pode-se fazer isso simplesmente usando uma variável no terceiro argumento

```
?- append([a,b,c,d],[1,2,3,4,5], X).
X=[a,b,c,d,1,2,3,4,5]
yes

?-
```

Definição do append/3

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

- Definição recursiva
  - Cláusula base: concatenar a lista vazia a qualquer lista resulta nessa mesma lista
  - Passo recursivo: concatenar uma lista não-vazia [H|T] com uma lista L, resulta numa lista com cabeça H e resto igual à concatenação de T e L

Árvore de pesquisa do append/3

?- append([a,b,c],[1,2,3], R).

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

Árvore de pesquisa do append/3

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

?- append([a,b,c],[1,2,3], R).

/ \

Árvore de pesquisa do append/3

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

```
?- append([a,b,c],[1,2,3], R).
   /                        \
†                  R = [a|L0]
                   ?- append([b,c],[1,2,3],L0)
```

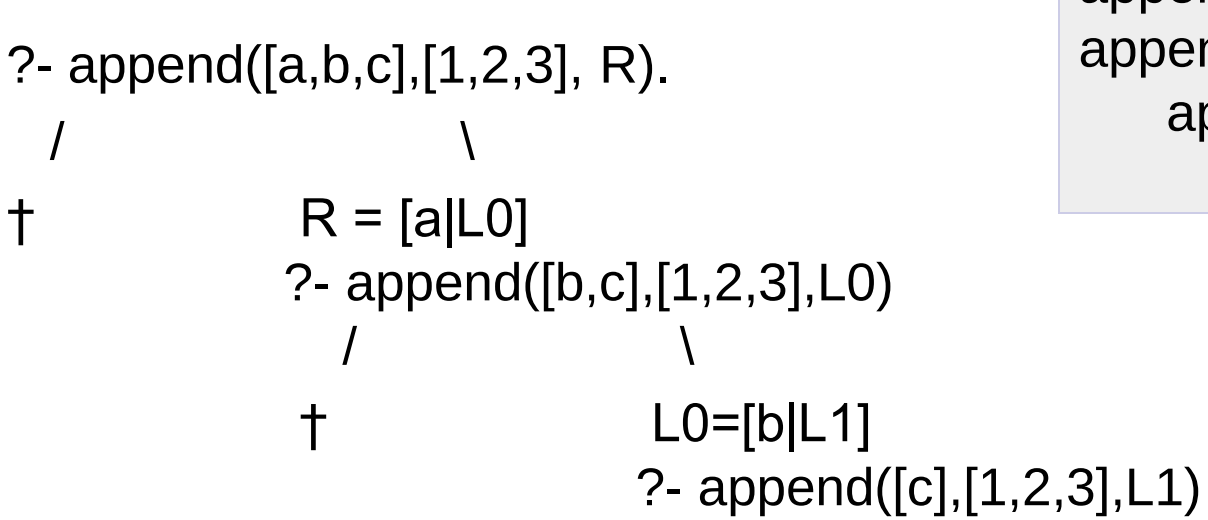
Árvore de pesquisa do append/3

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

```
?- append([a,b,c],[1,2,3], R).
   /                        \
†                  R = [a|L0]
                  ?- append([b,c],[1,2,3],L0)
                     /                    \
```

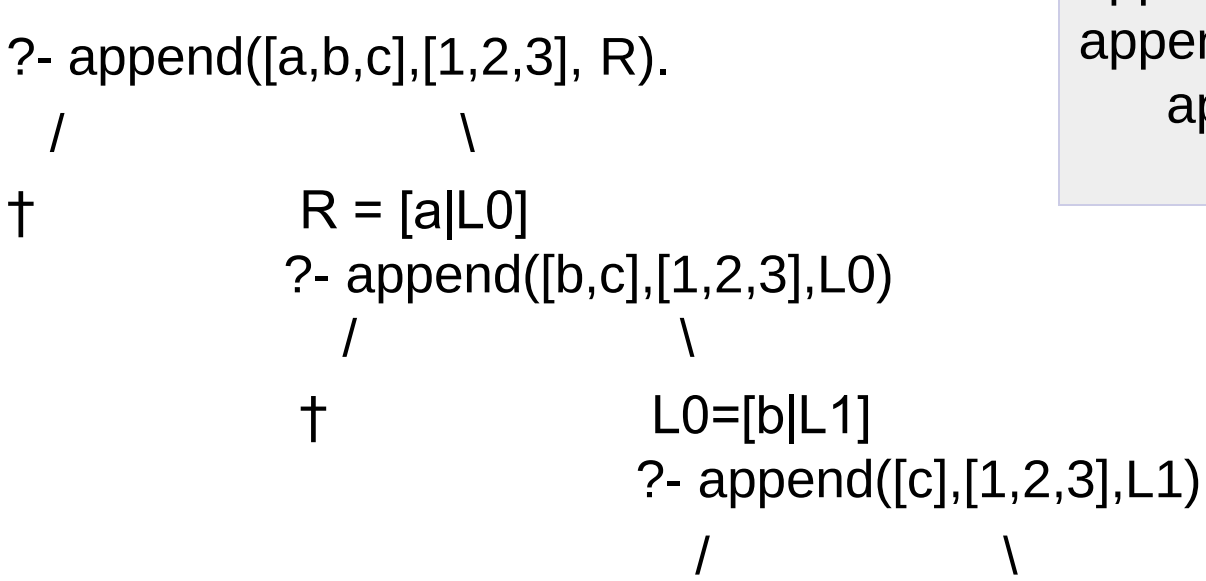
Árvore de pesquisa do append/3

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



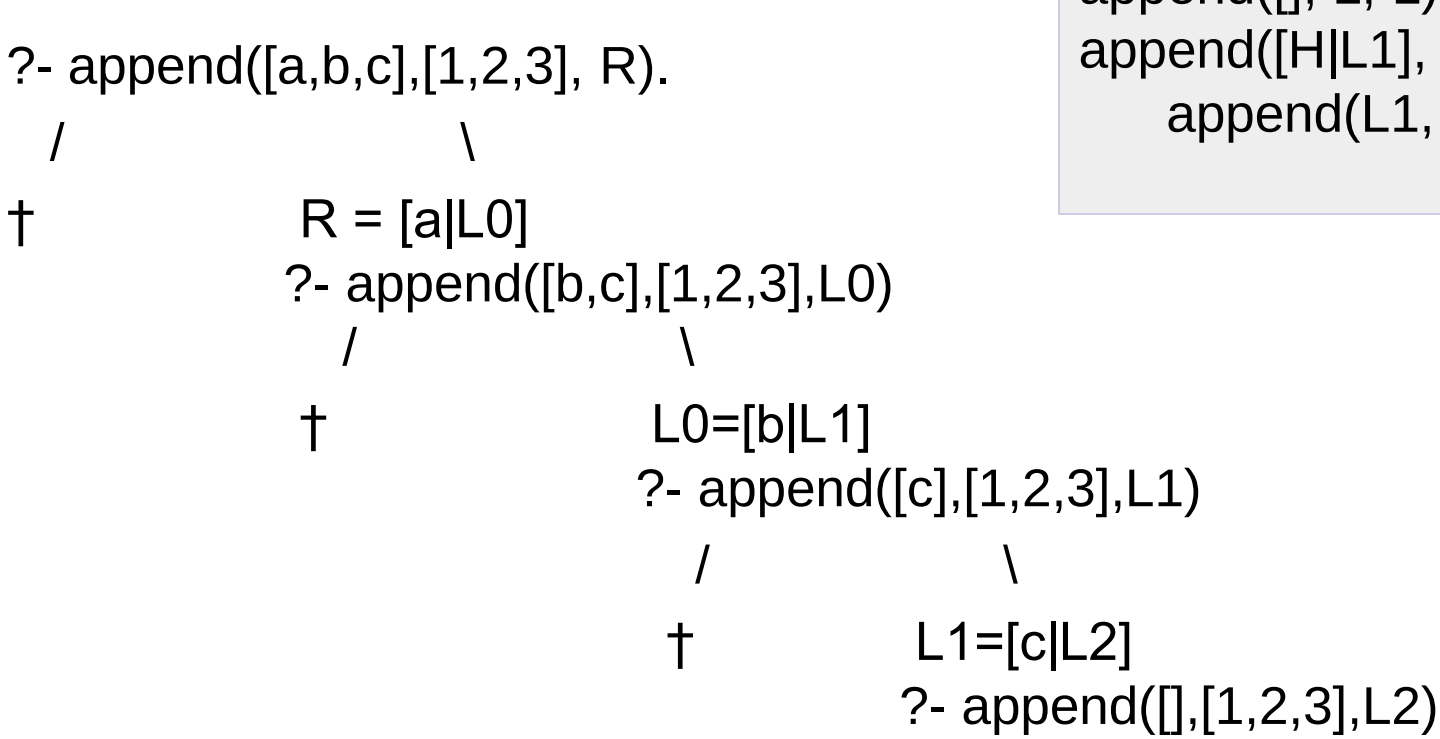
Árvore de pesquisa do append/3

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



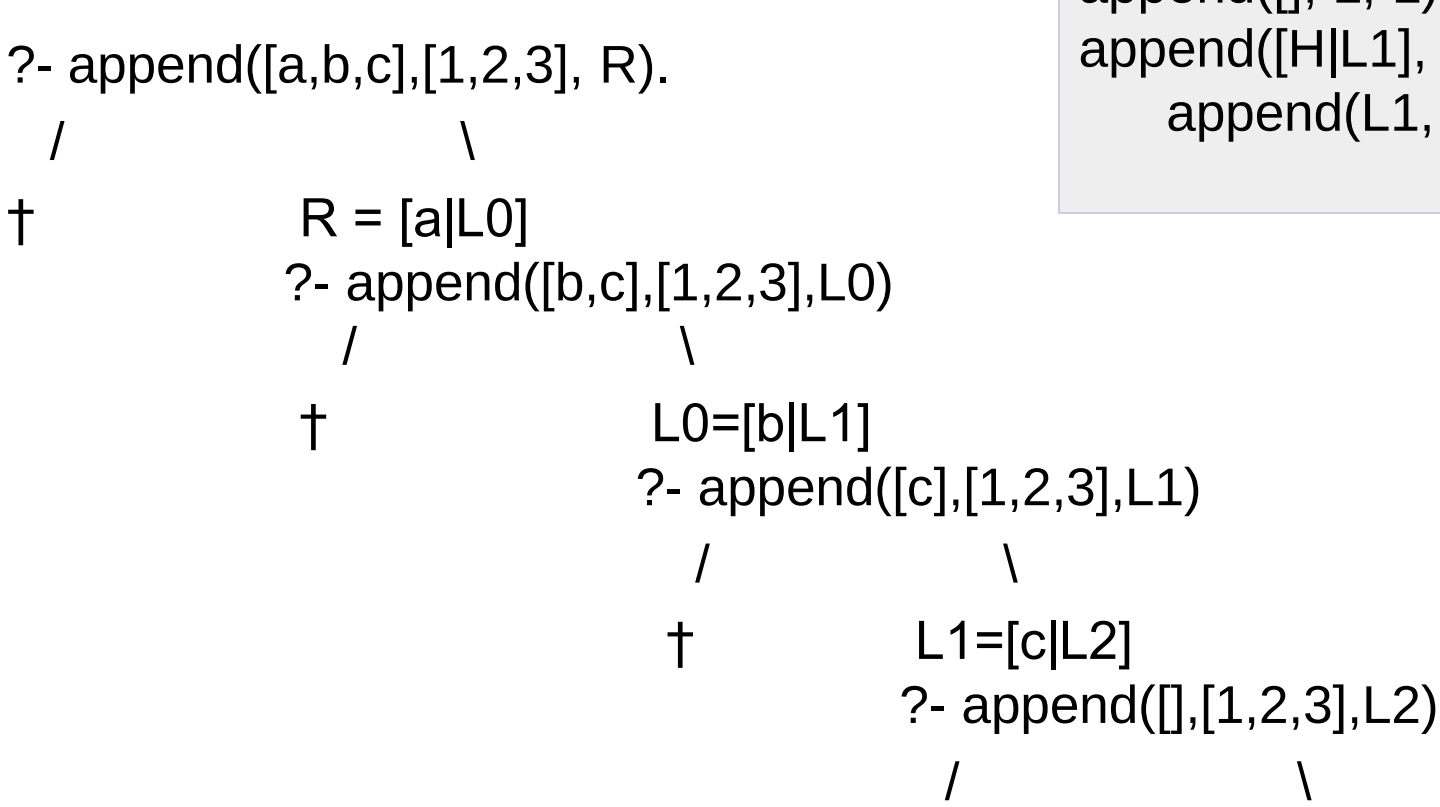
Árvore de pesquisa do append/3

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



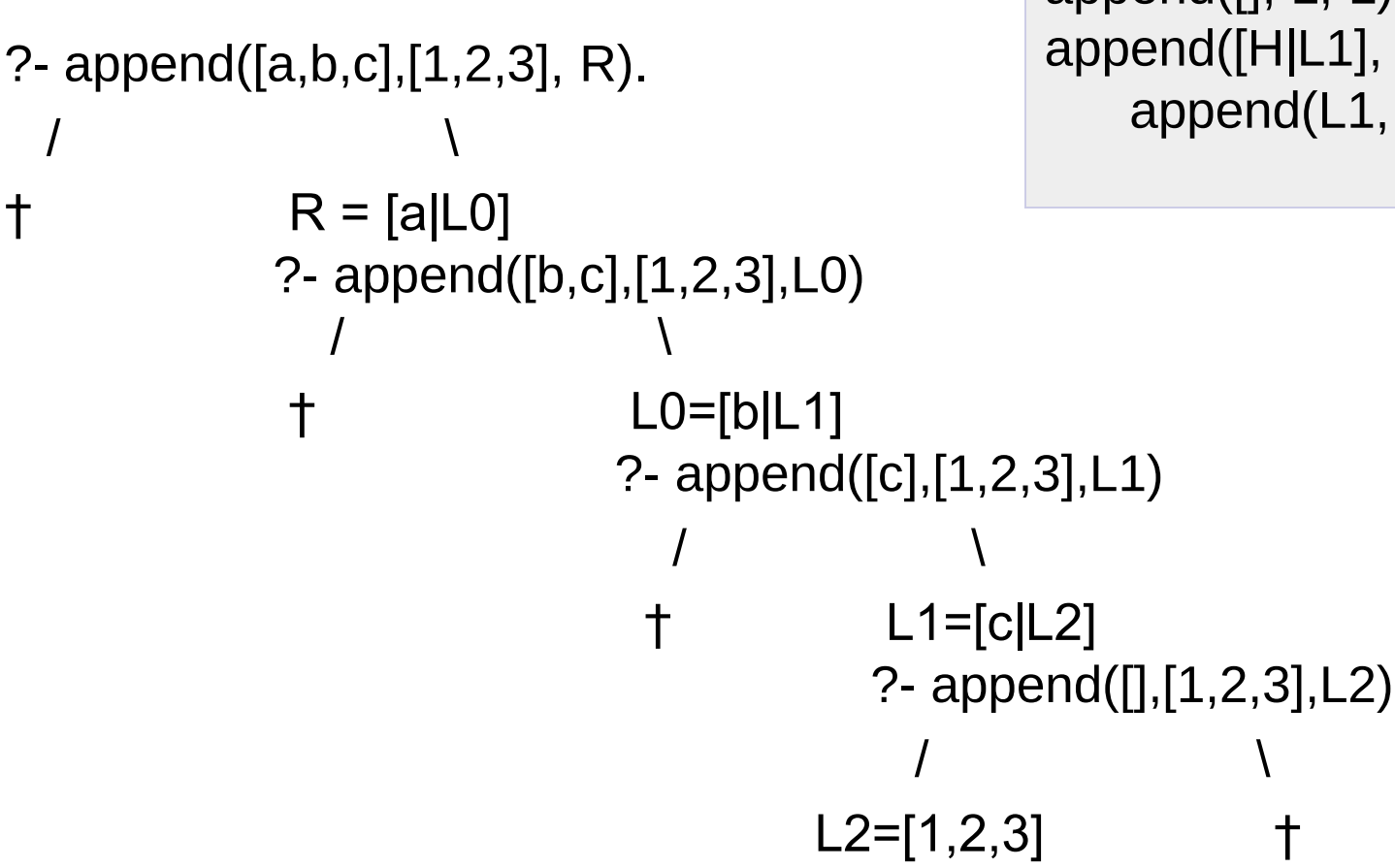
Árvore de pesquisa do append/3

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



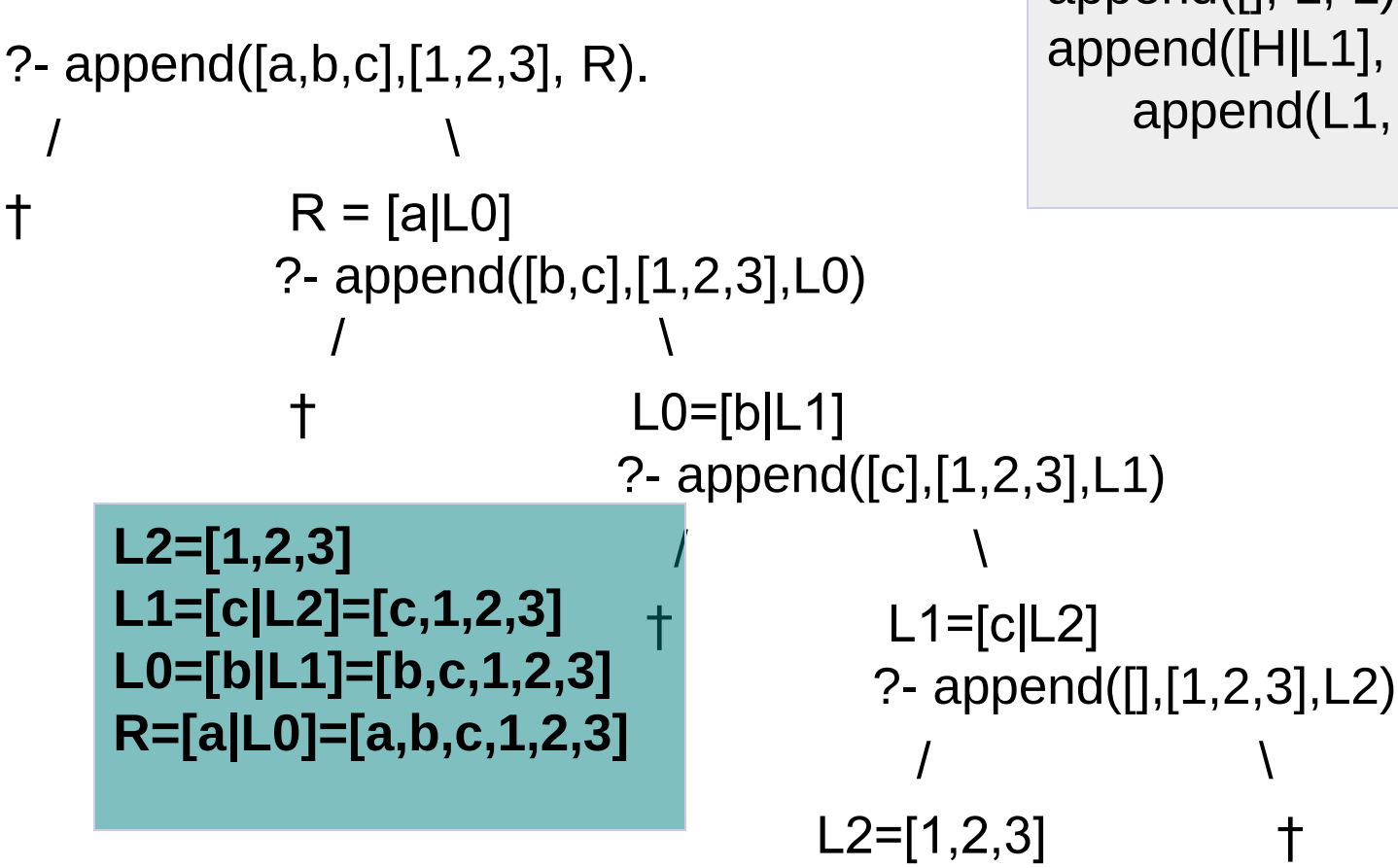
Árvore de pesquisa do append/3

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



Árvore de pesquisa do append/3

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



Usar o append/3

- Dividir uma lista:

```
?- append(X,Y, [a,b,c,d]).
X=[ ]          Y=[a,b,c,d];
X=[a]          Y=[b,c,d];
X=[a,b]        Y=[c,d];
X=[a,b,c]      Y=[d];
X=[a,b,c,d]    Y=[ ];
no
```

Prefixo e sufixo

- Pode-se usar o append/3 para definir outros predicados úteis
- Um exemplo é obter prefixos e sufixos de uma lista

Definição do prefix/2

```
prefix(P,L):-
    append(P,_,L).
```

- Uma lista P é prefixo de outra lista L quando existe uma lista tal que L é o resultado da concatenação de P com essa lista.
- Usa-se a variável anónima porque não interessa o conteúdo dessa lista.

Uso do prefix/2

```
prefix(P,L):-
    append(P,_,L).
```

```
?- prefix(X, [a,b,c,d]).
X=[ ];
X=[a];
X=[a,b];
X=[a,b,c];
X=[a,b,c,d];
no
```

Definição do suffix/2

```
suffix(S,L):-
    append(_,S,L).
```

- Uma lista S é sufixo de outra lista L quando existe uma lista tal que L é o resultado da concatenação dessa lista com S.
- Usa-se a variável anónima porque não interessa o conteúdo dessa lista.

Uso do suffix/2

```
suffix(S,L):-
    append(_,S,L).
```

```
?- suffix(X, [a,b,c,d]).
X=[a,b,c,d];
X=[b,c,d];
X=[c,d];
X=[d];
X=[];
no
```

Definição de sublist/2

- É agora simples escrever um predicado que obtém sub-listas de listas
- As sub-listas de uma lista L são apenas os prefixos dos sufixos de L

```
sublist(Sub,List):-
    suffix(Suffix,List),
    prefix(Sub,Suffix).
```

append/3 e eficiência

- O predicado **append/3** é útil e é importante saber como o usar
- É também importante saber que o **append/3** pode ser fonte de ineficiência
- Porquê?
  - Concatenar uma lista não é feito com uma só acção
  - É necessário percorrer uma das listas

Inversão de uma Lista

- Vamos ilustrar o problema do append/3 usando-o para inverter os elementos de uma lista
- Definir um predicado que transforme a lista [a,b,c,d,e] na lista [e,d,c,b,a]
- É útil uma vez que o Prolog apenas permite o acesso directo à cabeça da lista

Inversão Naïve

- Definição recursiva
  1. O inverso da lista vazia, é a lista vazia
  2. O inverso da lista [H|T], é a lista obtida por inversão de T e concatenação com [H]
- Exemplo, considere-se a lista [a,b,c,d].
  - Ao inverter o resto obtém-se [d,c,b].
  - Que concatenado com [a] dá [d,c,b,a]

Inversão Naïve em Prolog

```
naiveReverse([],[]).
naiveReverse([H|T],R):-
    naiveReverse(T,RT),
    append(RT,[H],R).
```

- A definição está correcta, mas é muito ineficiente
- Perde imenso tempo a fazer concatenações
- Mas existe uma alternativa melhor…

Inversão com um acumulador

- A alternativa é usar um acumulador
- O acumulador é uma lista, inicialmente vazia
- Retira-se a cabeça da lista a inverter e adiciona-se como a cabeça do acumulador
- Repete-se até obter a lista vazia
- Nesta altura o acumulador contém a lista invertida!

Inversão com um acumulador

```
accReverse([ ],L,L).
accReverse([H|T],Acc,Rev):-
    accReverse(T,[H|Acc],Rev).
```

Adicionar um predicado para iniciar

```
accReverse([ ],L,L).
accReverse([H|T],Acc,Rev):-
    accReverse(T,[H|Acc],Rev).
```

```
reverse(L1,L2):-
    accReverse(L1,[ ],L2).
```

Exemplo do acumulador

- List: [a,b,c,d] Accumulator: []

Exemplo do acumulador

- List: [a,b,c,d] Accumulator: []
- List: [b,c,d] Accumulator: [a]

Exemplo do acumulador

- List: [a,b,c,d]
- List: [b,c,d]
- List: [c,d]

Accumulator: []
Accumulator: [a]
Accumulator: [b,a]

Exemplo do acumulador

- List: [a,b,c,d]
- List: [b,c,d]
- List: [c,d]
- List: [d]

Accumulator: []
Accumulator: [a]
Accumulator: [b,a]
Accumulator: [c,b,a]

Exemplo do acumulador

- List: [a,b,c,d]
- List: [b,c,d]
- List: [c,d]
- List: [d]
- List: []

Accumulator: []
Accumulator: [a]
Accumulator: [b,a]
Accumulator: [c,b,a]
Accumulator: [d,c,b,a]