Aula 5: Aritmética

- Teoria
  - Introduzir formas de se realizar operações **aritméticas** em Prolog
  - Aplicar estas operações a problemas simples de processamento de listas, usando **acumuladores**
  - Conhecer os predicados com **recursão final** e explicar por que eles são mais eficientes que os predicados que não possuem recursão final.

Aritmética em Prolog

- Prolog oferece uma série de ferramentas básicas de aritmética
- Tanto para números reais quanto para inteiros

Aritmética	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 é o resto da divisão de 7 por 2	?- 1 is mod(7,2).

Exemplos de consultas

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

?- 4 is 2+3.
false

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

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

Definindo predicados com aritmética

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

Definindo predicados com aritmética

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

```
?- somaTresDepoisDuplica(1,X).
X=8
true

?- somaTresDepoisDuplica(2,X).
X=10
true
```

Um olhar mais atento

- É importante saber que +, -, / e ∗, na verdade, não realizam operação aritmética alguma.
- Expressões tais como 3+2, 4-7 e 5/5 são termos comuns do Prolog
  - Funtor: +, -, /, ∗
  - Aridade: 2
  - Argumentos: inteiros

Um olhar mais atento

?- X = 3 + 2.

Um olhar mais atento

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

?-
```

Um olhar mais atento

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

?- 3 + 2 = X.
```

Um olhar mais atento

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

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

?-
```

O predicado is/2

- Para forçar o Prolog a realmente avaliar as expressões aritméticas, temos que usar

  is

  como feito nos exemplos anteriores.
- Isto é uma instrução para o Prolog realizar os cálculos.
- Devido ao fato deste predicado não ser um predicado comum do Prolog, existem algumas restrições em seu uso.

O predicado is/2

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

O predicado is/2

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

?-
```

O predicado is/2

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

?- 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

?- Resultado 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

?- Resultado is 2+2+2+2+2.
Resultado = 10
true

?-
```

Restrições sobre o uso de is/2

- Temos liberdade para usar variáveis no lado direito do predicado **is**.
- Mas, quando o Prolog realmente realizar a avaliação, as variáveis devem estar instanciadas com um termo sem variáveis do Prolog.
- Este termo deve ser uma expressão aritmética.

Notação

- Duas observações finais sobre expressões aritméticas:
  - 3+2, 4/2, 4-5 são apenas termos comuns do Prolog em uma notação mais amigável:
    3+2 é na verdade **+(3,2)** e assim por diante.
  - O predicado **is** é um predicado de dois argumentos em Prolog.

Notação

- Duas observações finais sobre expressões aritméticas:
  - 3+2, 4/2, 4-5 são apenas termos comuns do Prolog em uma notação mais amigável:

    3+2 é na verdade **+(3,2)** e assim por diante.
  - O predicado **is** é um predicado de dois argumentos em Prolog.

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

Aritmética e Listas

- Qual é o comprimento de uma lista?
  - A lista vazia possui comprimento: zero;
  - Uma lista não vazia possui comprimento: um mais o comprimento de sua cauda.

Comprimento de uma lista em Prolog

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

```
?-
```

Comprimento de uma lista em Prolog

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

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

Comprimento de uma lista em Prolog

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

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

Acumuladores

- O programa anterior é bastante bom:
  - Fácil de entender
  - Relativamente eficiente
- Mas existe um outro método de encontrar o comprimento de uma lista:
  - Introduzir a ideia de acumuladores;
  - Acumuladores são variáveis que armazenam resultados intermediários.

Definindo tamAcum/3

- O predicado tamAcum/3 possui três argumentos:
  - A lista cujo comprimento desejamos encontrar;
  - O comprimento da lista: um inteiro
  - Um acumulador, armazenando os valores intermediários para o comprimento.

Definindo tamAcum/3

- O acumulador de tamAcum/3
  - O valor inicial do acumulador é 0;
  - Some 1 ao contador a cada vez que se possa, recursivamente, retirar a cabeça da lista;
  - Quando se alcançar a lista vazia, o acumulador conterá o comprimento da lista.

Comprimento de uma lista em Prolog

```
tamAcum([], Acum, Tam):-
    Tam = Acum.

tamAcum([_|L], AcumAntigo, Tam):-
    NovoAcum is AcumAntigo + 1,
    tamAcum(L, NovoAcum, Tam).
```

```
?-
```

Comprimento de uma lista em Prolog

```
tamAcum([], Acum, Tam):-
    Tam = Acum.

tamAcum([_|L], AcumAntigo, Tam):-
    NovoAcum is AcumAntigo + 1,
    tamAcum(L, NovoAcum, Tam).
```

some 1 ao contador a cada vez que se possa retirar a cabeça da lista

```
?-
```

Comprimento de uma lista em Prolog

```
tamAcum([], Acum, Tam):-
    Tam = Acum.

tamAcum([_|L], AcumAntigo, Tam):-
    NovoAcum is AcumAntigo + 1,
    tamAcum(L, NovoAcum, Tam).
```

quando se alcançar a lista vazia, o acumulador conterá o comprimento da lista

```
?-
```

Comprimento de uma lista em Prolog

```
tamAcum([], Acum, Acum).

tamAcum([_|L], AcumAntigo, Tam):-
    NovoAcum is AcumAntigo + 1,
    tamAcum(L, NovoAcum, Tam).
```

```
?-
```

Comprimento de uma lista em Prolog

```
tamAcum([], Acum, Acum).

tamAcum([_|L], AcumAntigo, Tam):-
    NovoAcum is AcumAntigo + 1,
    tamAcum(L, NovoAcum, Tam).
```

```
?- tamAcum([a,b,c], 0, Tam).
Tam=3
true

?-
```

Árvore de busca para tamAcum/3

?- tamAcum([a,b,c],0,Tam).

```
tamAcum([ ],Acum,Acum).

tamAcum([_|L],AcumAntigo,Tam):-
    NovoAcum is AcumAntigo + 1,
    tamAcum(L,NovoAcum,Tam).
```

Árvore de busca para tamAcum/3

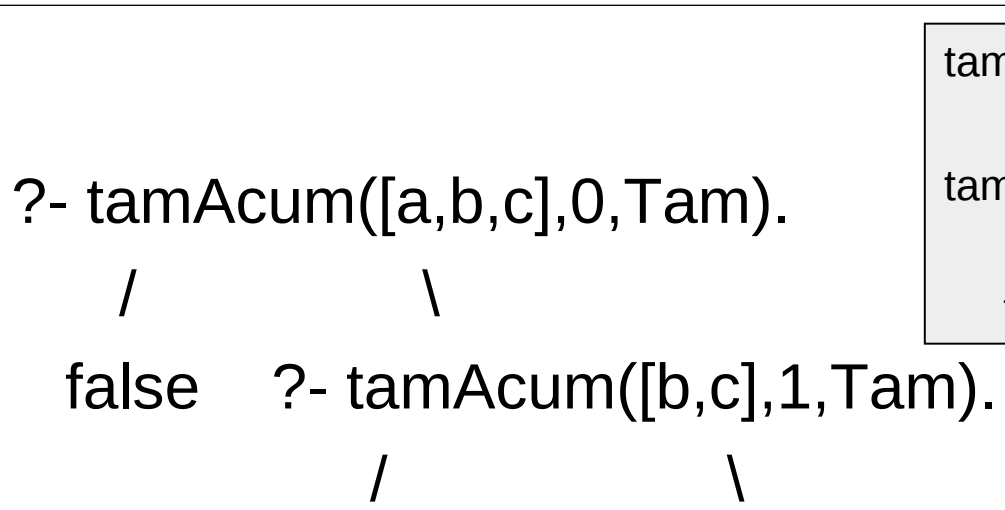
?- tamAcum([a,b,c],0,Tam).

/ \

```
tamAcum([ ],Acum,Acum).

tamAcum([_|L],AcumAntigo,Tam):-
    NovoAcum is AcumAntigo + 1,
    tamAcum(L,NovoAcum,Tam).
```

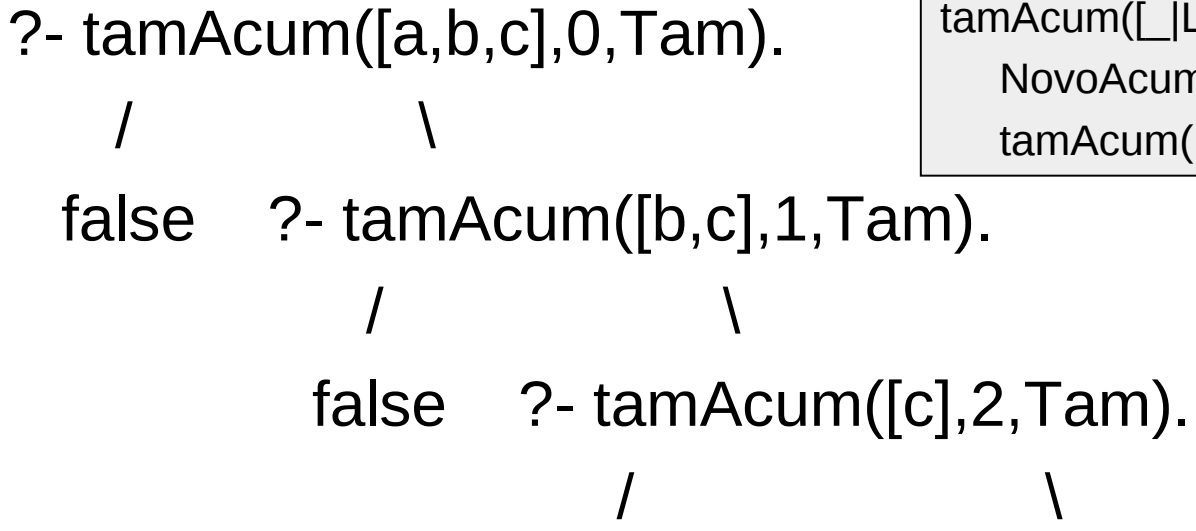
Árvore de busca para tamAcum/3

```
tamAcum([ ],Acum,Acum).

tamAcum([_|L],AcumAntigo,Tam):-
    NovoAcum is AcumAntigo + 1,
    tamAcum(L,NovoAcum,Tam).
```



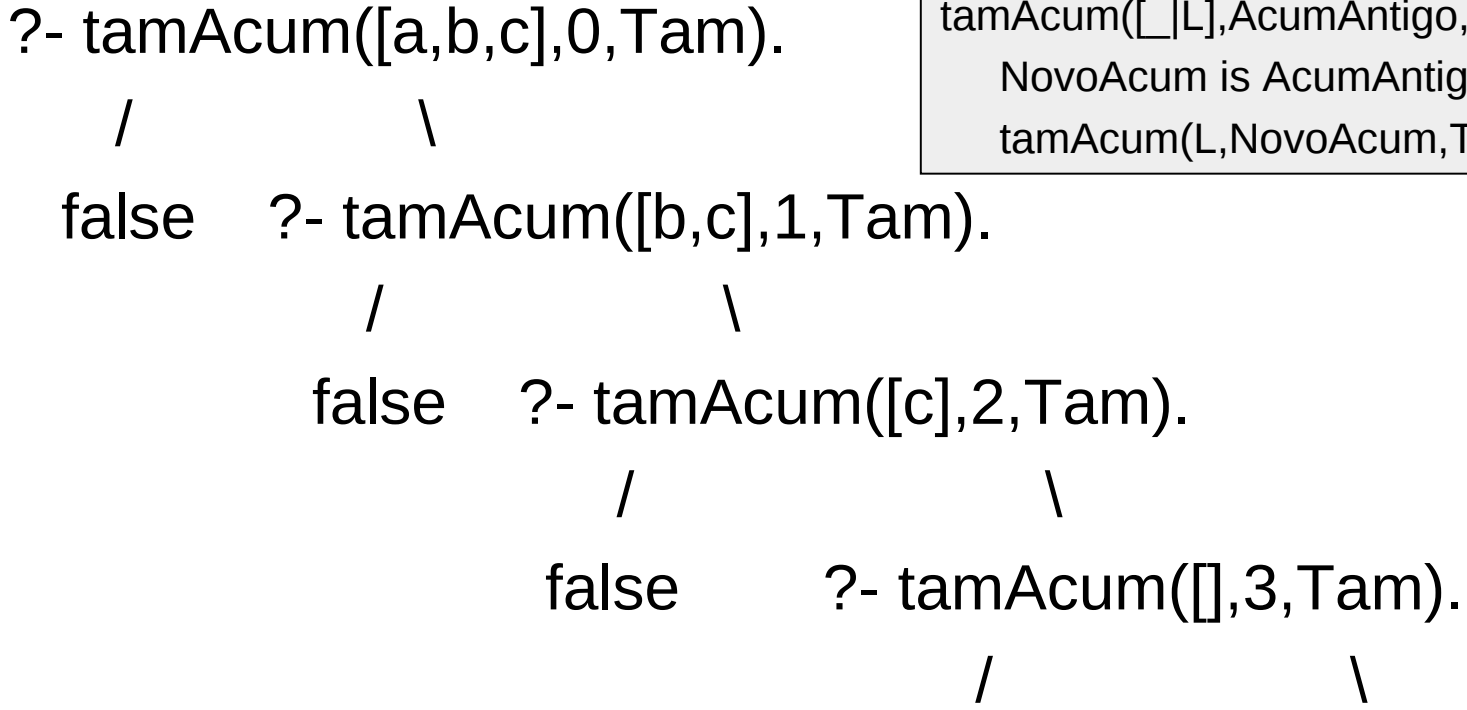
Árvore de busca para tamAcum/3

```
tamAcum([ ],Acum,Acum).

tamAcum([_|L],AcumAntigo,Tam):-
    NovoAcum is AcumAntigo + 1,
    tamAcum(L,NovoAcum,Tam).
```



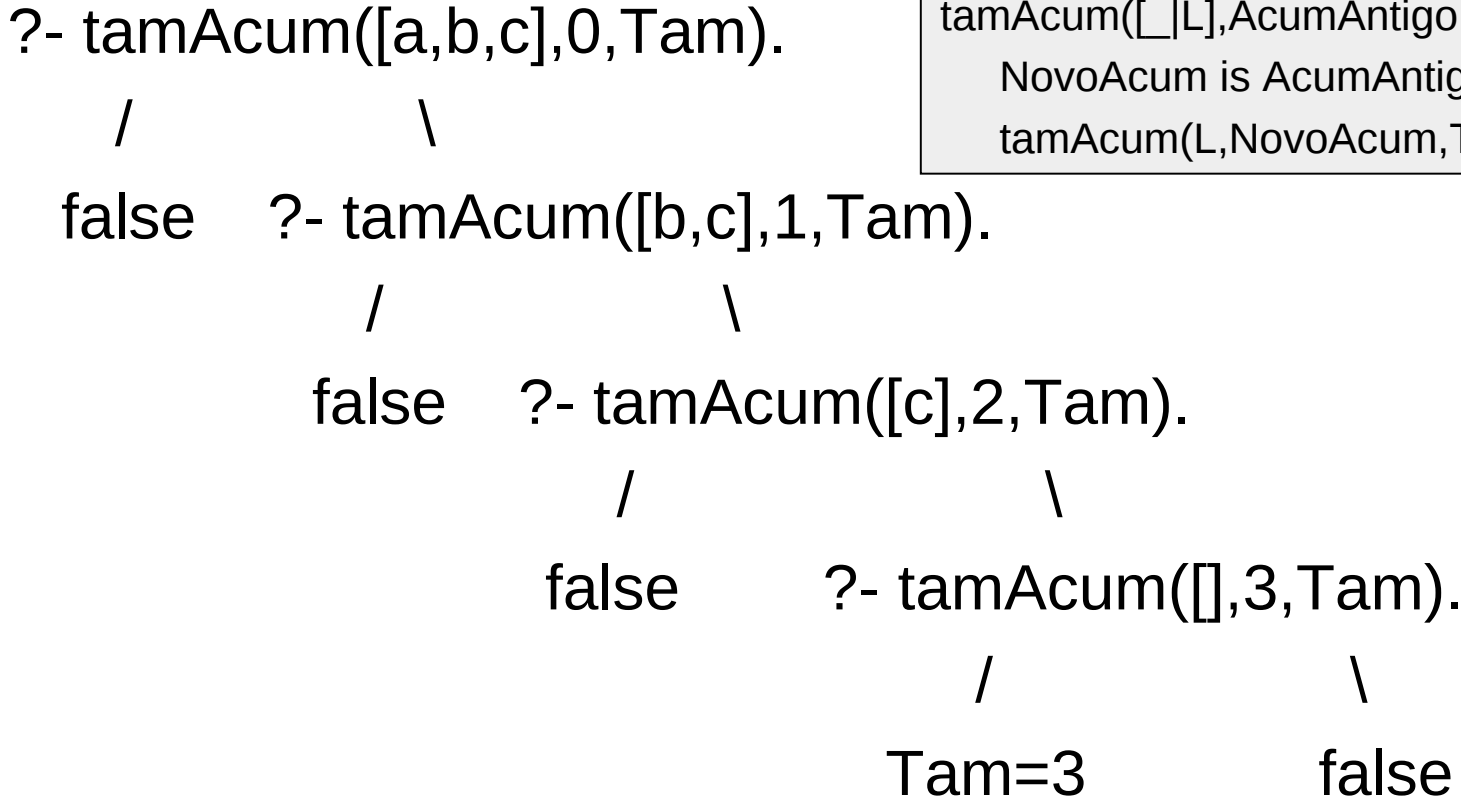
Árvore de busca para tamAcum/3

```
tamAcum([ ],Acum,Acum).

tamAcum([_|L],AcumAntigo,Tam):-
    NovoAcum is AcumAntigo + 1,
    tamAcum(L,NovoAcum,Tam).
```



Árvore de busca para tamAcum/3

```
tamAcum([ ],Acum,Acum).

tamAcum([_|L],AcumAntigo,Tam):-
    NovoAcum is AcumAntigo + 1,
    tamAcum(L,NovoAcum,Tam).
```



Adicionando um predicado-capa

```
tamAcum([ ],Acum,Acum).

tamAcum([ _|L],AcumAntigo,Tam):-
     NovoAcum is AcumAntigo + 1,
     tamAcum(L,NovoAcum,Tam).

tam(Lista,Tam):-
      tamAcum(Lista,0,Tam).
```

```
?- tam([a,b,c], X).
X=3
true
```

Recursão final

- Por que tamAcum/3 é melhor que tam/2?
  - tamAcum/3 tem recursão final, enquanto tam/2 não.
- Diferença:
  - Em predicados com recursão final, os resultados já estão completamente calculados quando alcançamos a cláusula base.
  - Em predicados recursivos que não possuam recursão final, ainda existirão metas na pilha quando alcançarmos a cláusula base.

Comparação

Sem recursão final

```
tam([],0).
tam([_|L],NovoTam):-
    tam(L,Tam),
    NovoTam is Tam + 1.
```

Com recursão final

```
tamAcum([],Acum,Acum).
tamAcum([_|L],AcumAntigo,Tam):-
    NovoAcum is AcumAntigo + 1,
    tamAcum(L,NovoAcum,Tam).
```

Árvore de busca para tam/2

?- tam([a,b,c], Tam).

```
tam([],0).
tam([_|L],NovoTam):-
    tam(L,Tam),
    NovoTam is Tam + 1.
```

Árvore de busca para tam/2

?- tam([a,b,c], Tam).

/ \

false ?- tam([b,c],Tam1),
Tam is Tam1 + 1.

```
tam([],0).
tam([_|L],NovoTam):-
    tam(L,Tam),
    NovoTam is Tam + 1.
```

Árvore de busca para tam/2

?- tam([a,b,c], Tam).

/ \

false ?- tam([b,c],Tam1),
Tam is Tam1 + 1.

/ \

false ?- tam([c], Tam2),
Tam1 is Tam2+1,
Tam is Tam1+1.

```
tam([],0).
tam([_|L],NovoTam):-
    tam(L,Tam),
    NovoTam is Tam + 1.
```

Árvore de busca para tam/2

```
?- tam([a,b,c], Tam).
      /              \
  false   ?- tam([b,c],Tam1),
             Tam is Tam1 + 1.
            /                \
        false    ?- tam([c], Tam2),
                    Tam1 is Tam2+1,
                    Tam is Tam1+1.
                   /              \
               false      ?- tam([], Tam3),
                             Tam2 is Tam3+1,
                             Tam1 is Tam2+1,
                             Tam is Tam1 + 1.
```

```
tam([],0).
tam([_|L],NovoTam):-
    tam(L,Tam),
    NovoTam is Tam + 1.
```

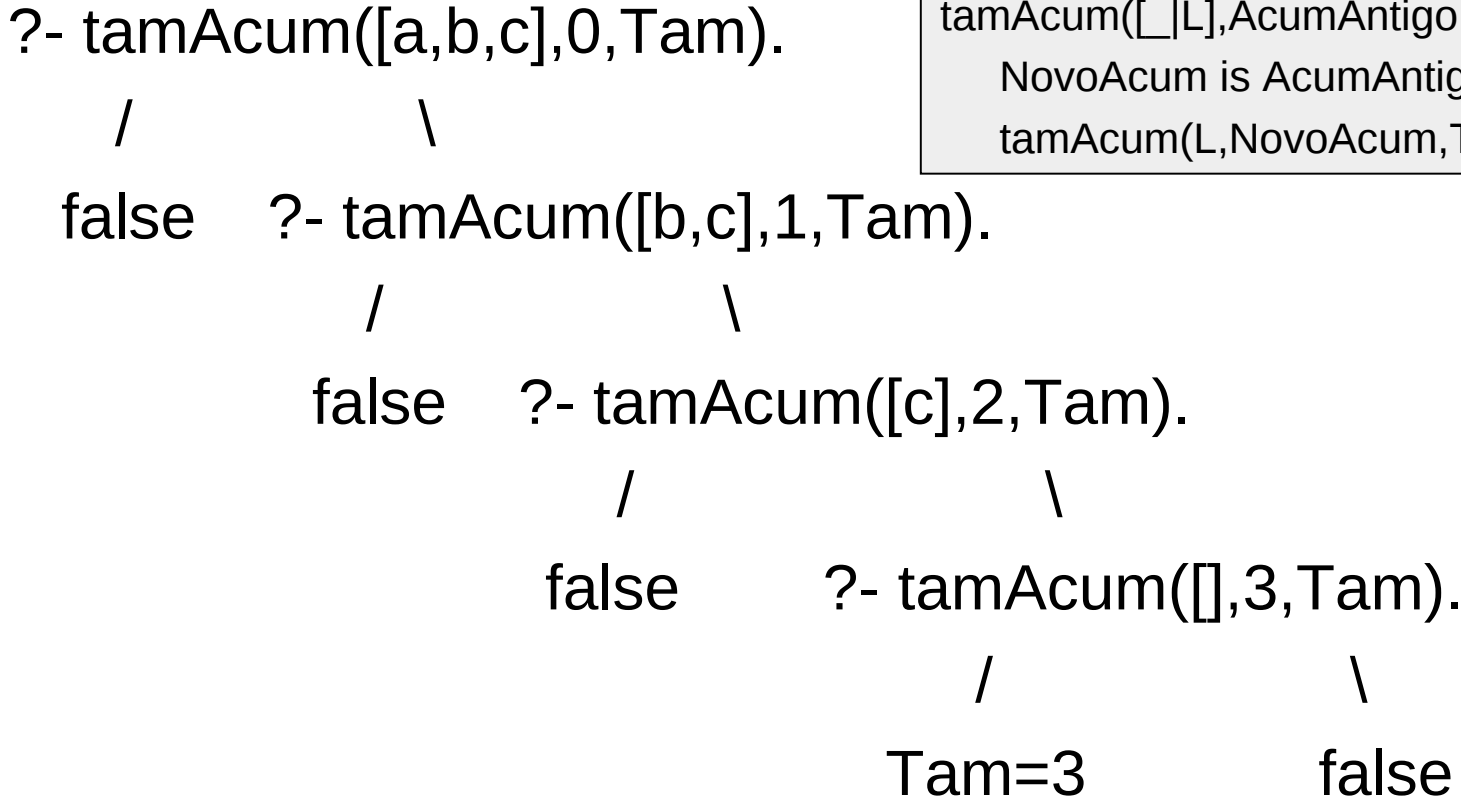
Árvore de busca para tam/2

```
?- tam([a,b,c], Tam).
      /              \
  false   ?- tam([b,c],Tam1),
             Tam is Tam1 + 1.
            /                \
        false    ?- tam([c], Tam2),
                    Tam1 is Tam2+1,
                    Tam is Tam1+1.
                  /               \
              false      ?- tam([], Tam3),
                            Tam2 is Tam3+1,
                            Tam1 is Tam2+1,
                            Tam is Tam1 + 1.
                           /                \
                 Tam3=0, Tam2=1,              false
                  Tam1=2, Tam=3
```

```
tam([],0).
tam([_|L],NovoTam):-
    tam(L,Tam),
    NovoTam is Tam + 1.
```

Árvore de busca para tamAcum/3

```
tamAcum([ ],Acum,Acum).

tamAcum([_|L],AcumAntigo,Tam):-
    NovoAcum is AcumAntigo + 1,
    tamAcum(L,NovoAcum,Tam).
```



Exercícios

- Como o Prolog responde às seguintes consultas?

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.
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)).

Exercícios

- Defina um predicado **`incrementa/2`** que é verdadeiro somente quando o seu segundo argumento é um inteiro maior que seu primeiro argumento por uma unidade. Por exemplo,
- ?- incrementa(4,5).
- true

- ?- incrementa(4,6).
- false

Exercícios

- Defina um predicado **soma/3** que é verdadeiro quando o seu terceiro argumento é a soma dos primeiros dois argumentos. Por exemplo,
- ?- soma(4,5,9).
- true

- ?- soma(4,6,12).
- false

Comparando inteiros

- Alguns predicados aritméticos do Prolog realmente realizam aritmética, sem a necessidade de forçar a avaliação.
- Assim são os operadores que comparam inteiros.

Comparando inteiros

Aritmética	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

Operadores de comparação

- Possuem o significado óbvio
- Força os argumentos em ambos os lados a serem avaliados.

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

?- 4+3 > 5+5.
false
```

Operadores de comparação

- Possuem o significado óbvio
- Força os argumentos em ambos os lados a serem avaliados.

```
?- 4 = 4.
true

?- 2+2 = 4.
false

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

Comparando números

- Nós definiremos um predicado que recebe dois argumentos, e é verdadeiro se:
  - O primeiro é uma lista de inteiros; e
  - O segundo argumento é o maior inteiro na lista.
- Ideia básica
  - Usaremos um acumulador;
  - O acumulador armazena o maior valor encontrado até agora;
  - Se encontrarmos um valor maior, o acumulador será atualizado.

Definição de maxAcum/3

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

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

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

```
?- maxAcum([1,0,5,4],0,Max).
Max=5
true
```

Adicionando uma capa: max/2

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

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

maxAcum([],A,A).

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

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

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

?-
```

Resumo desta aula

- Nesta aula foi visto como Prolog realiza operações **aritméticas**.
- Nós demonstramos a diferença entre predicados com **recursão final** e predicados sem recursão final.
- Nós introduzimos uma técnica de programação:
  - a utilização de **acumuladores.**
- Nós também introduzimos a ideia de usar predicados-capa.

Próxima aula

- Sim, mais listas!
  - Definição de concatena/3, um predicado que concatena duas listas
  - Discussão da ideia de inverter uma lista, primeiro, ingenuamente, usando concatena/3 e depois de um modo mais eficiente usando acumuladores.