

PROLOG

(LISTAS, ARITMÉTICA E TERMOS)

Parcialmente adaptado de
<http://www.learnprolognow.org/>

Aritmética em Prolog (Inteiros e Reais)

- Aritmética

$$2 + 3 = 5$$

$$3 \times 4 = 12$$

$$5 - 3 = 2$$

$$3 - 5 = -2$$

$$4 : 2 = 2$$

1 is the remainder when 7 is
divided by 2

- Prolog

?- 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 de Aritmética

?- 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.
```

```
?- 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

?- 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:
 - 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

?- 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

?-

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.

```
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: acclen/3

- O programa anterior é 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
- 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
- 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

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

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

Acumuladores: acclen/3

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

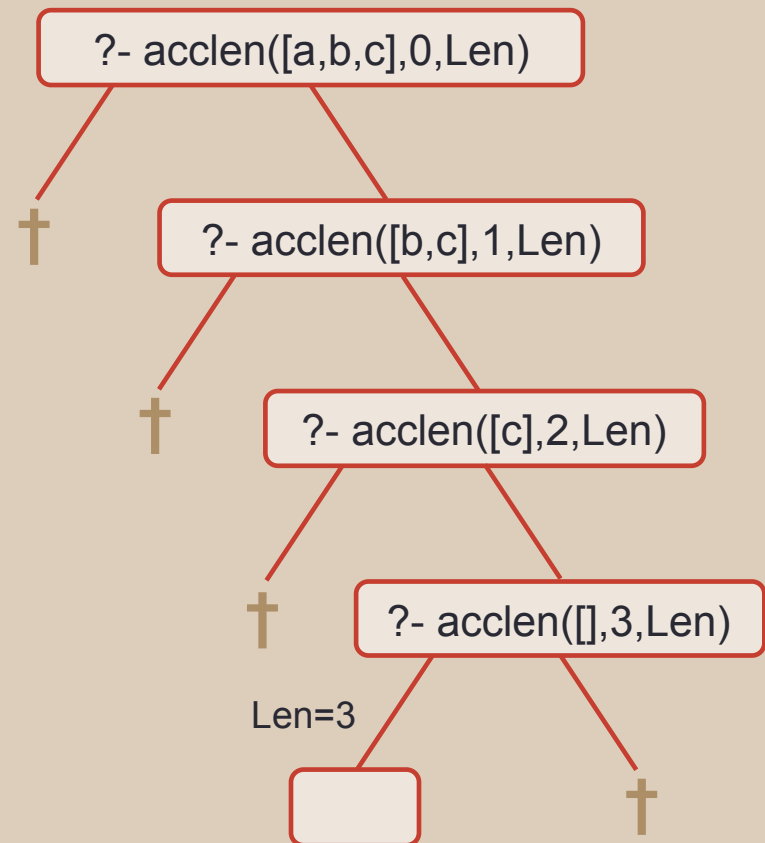
```
acclen([_|L],OldAcc,Length):-  
    NewAcc is OldAcc + 1,  
    acclen(L,NewAcc,Length).
```

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

```
Len=3
```

```
yes
```

```
?-
```



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 sem recursividade terminal, ainda existem goals no stack quando se chega à cláusula base

sem recursividade terminal

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

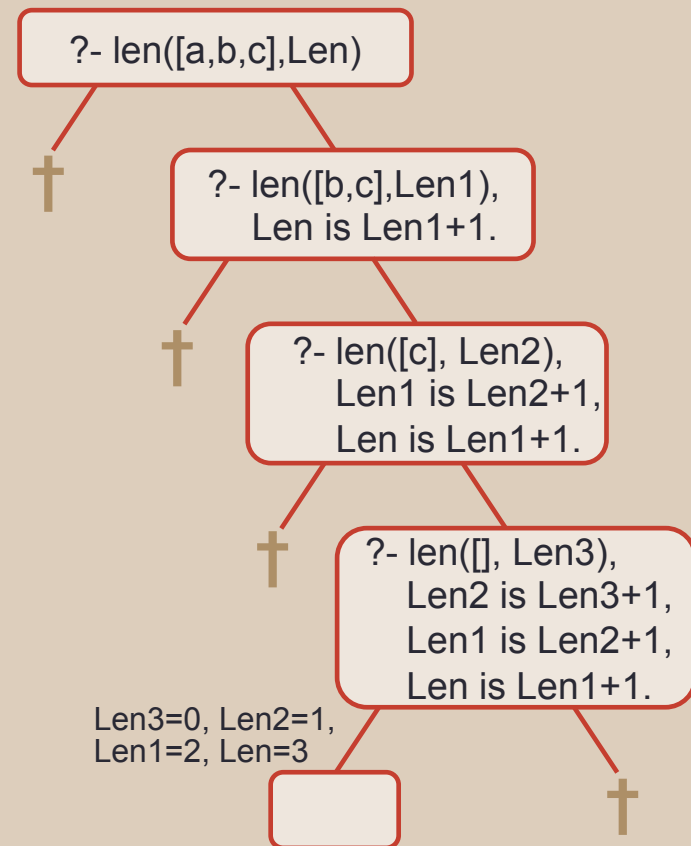
com recursividade terminal

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

len/2

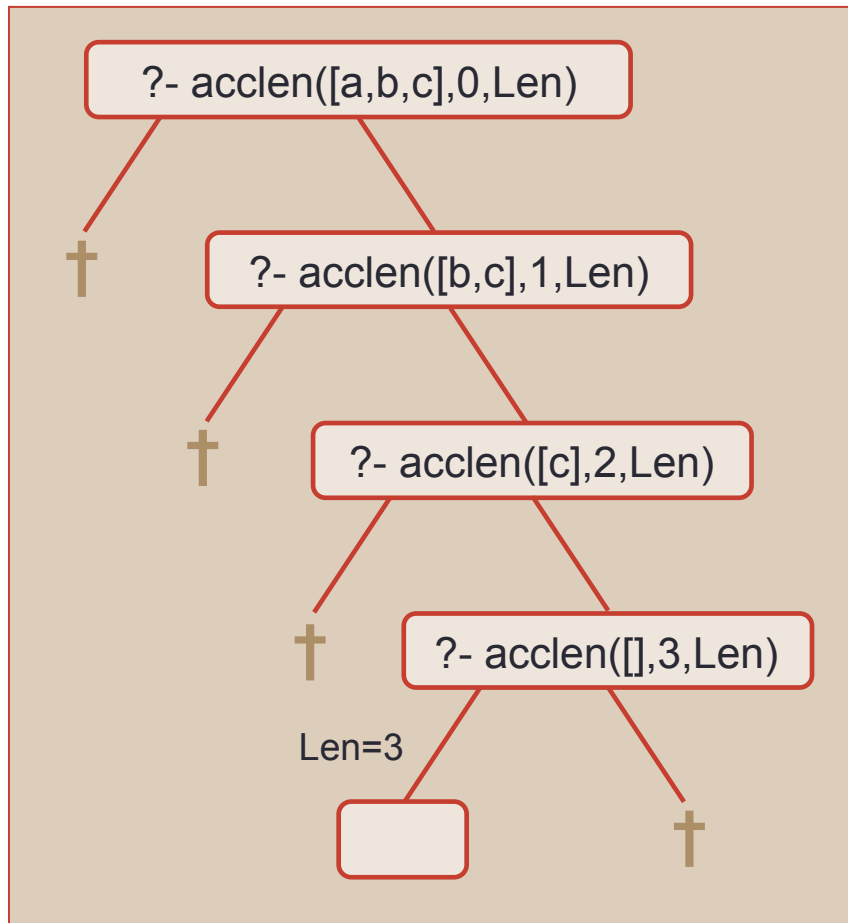
```
len([],0).  
len(_|L,NewLength):-  
    len(L,Length),  
    NewLength is Length + 1.
```

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

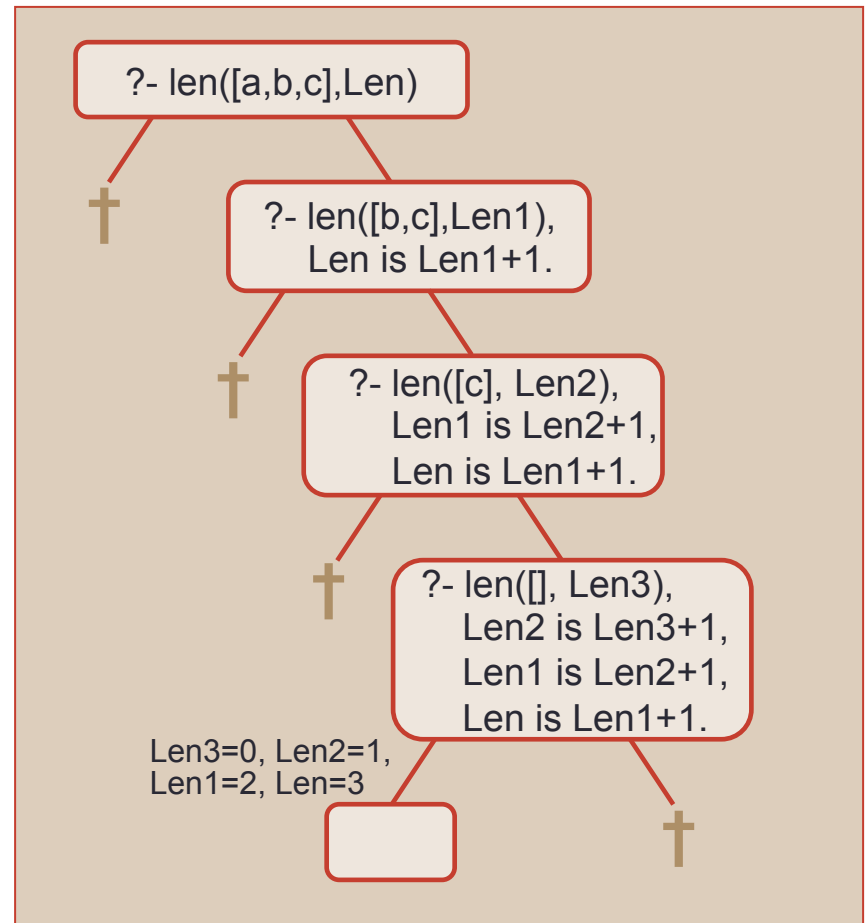


acclen/2 vs. len/2

acclen/2



len/2



Comparação de Inteiros

- Aritmética

$x < y$

$x \leq y$

$x = y$

$x \neq y$

$x \geq y$

$x > y$

- Prolog

?- $X < Y$

?- $X \leq Y$

?- $X =:= Y$

?- $X \backslash= Y$

?- $X \geq Y$

?- $X > Y$

Comparação de Inteiros

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

?- 2 < 4+1.

yes

?- 4+3 > 5+5.

no

?- 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

```
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

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
- 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/3

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

```
append([], L, L).
```

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

```
    append(L1, L2, L3).
```

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

```
X=[a,b,c,d,1,2,3,4,5]
```

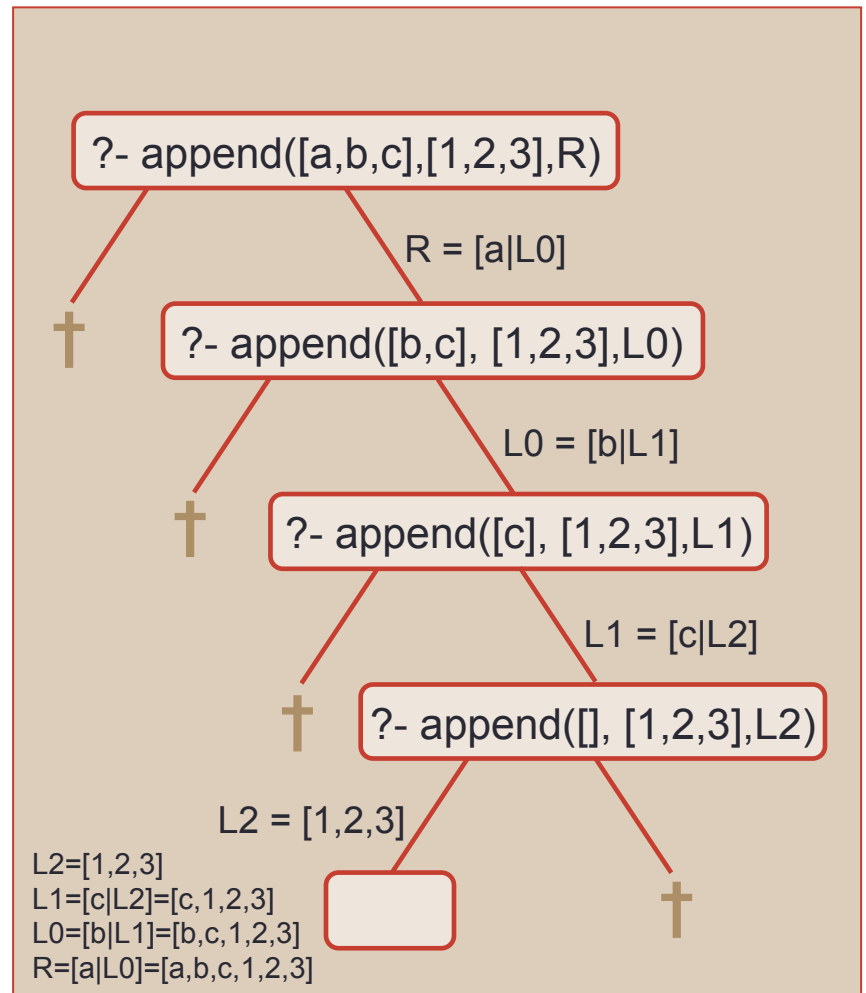
yes

```
?-
```

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,b,c,1,2,3]  
yes
```



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

Usar o append: prefix/2

- 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.

```
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
```

Usar o append: suffix/2

- 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.

```
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
```


Usar o prefix e o suffix: 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).
```

```
?- sublist(X, [a,b,c,d]).
```

```
X = [];
```

```
X = [a];
```

```
X = [a, b];
```

```
X = [a, b, c];
```

```
X = [a, b, c, d];
```

```
X = [];
```

```
X = [b];
```

```
X = [b, c];
```

```
X = [b, c, d];
```

```
X = [];
```

```
X = [c];
```

```
X = [c, d];
```

```
X = [];
```

```
X = [d];
```

```
X = [];
```

```
no
```

```
?-
```

append/3 e eficiência

- O predicado **append/3** é útil, sendo 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 Naïve de listas

- Definição recursiva
 - O inverso da lista vazia, é a lista vazia
 - O inverso da lista $[H|T]$, é a lista obtida por inversão de T e concatenação com $[H]$
- Considere-se a lista $[a,b,c,d]$.
 - Ao inverter o resto obtém-se $[d,c,b]$.
 - Que concatenado com $[a]$ dá $[d,c,b,a]$
- 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.
- A definição está correcta, mas é muito ineficiente
 - Perde imenso tempo a fazer concatenações
- Existe uma alternativa melhor...

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

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.
- Adicionar um predicado para iniciar.

```
accReverse([ ],L,L).  
accReverse([H|T],Acc,Rev):-  
    accReverse(T,[H|Acc],Rev).  
  
reverse(L1,L2):-  
    accReverse(L1,[ ],L2).
```

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

Comparação de termos: ==/2

- O Prolog contém um predicado para comparar termos
- É o predicado da igualdade ==/2
- O predicado da igualdade ==/2 não instancia variáveis, tem um comportamento diferente do =/2

```
?- a==a.
```

```
yes
```

```
?- a==b.
```

```
no
```

```
?- a=='a'.
```

```
yes
```

```
?- a==X.
```

```
X = _443
```

```
no
```

Comparação de variáveis

- Duas variáveis diferentes **não-instanciadas** não são termos iguais
- Variáveis **instanciadas** com um termo T são iguais a T

?- X==X.

X = _443

yes

?- Y==X.

Y = _442

X = _443

no

?- a=U, a==U.

U = a

yes

Comparação de termos: \==/2

- O predicado \==/2 sucede exactamente nos casos em que ==/2 falha
- Ou seja, sucede sempre que dois termos são **diferentes**, e falha caso contrário

?- a \== a.

no

?- a \== b.

yes

?- a \== 'a'.

no

?- a \== X.

X = _443

yes

Termos aritméticos

- $+$, $-$, $<$, $>$, etc são functors e expressões como $2+3$ são termos complexos
- O termo $2+3$ é igual ao termo $+(2,3)$

?- $2+3 == +(2,3)$.

yes

?- $-(2,3) == 2-3$.

yes

?- $(4<2) == <(4,2)$.

yes

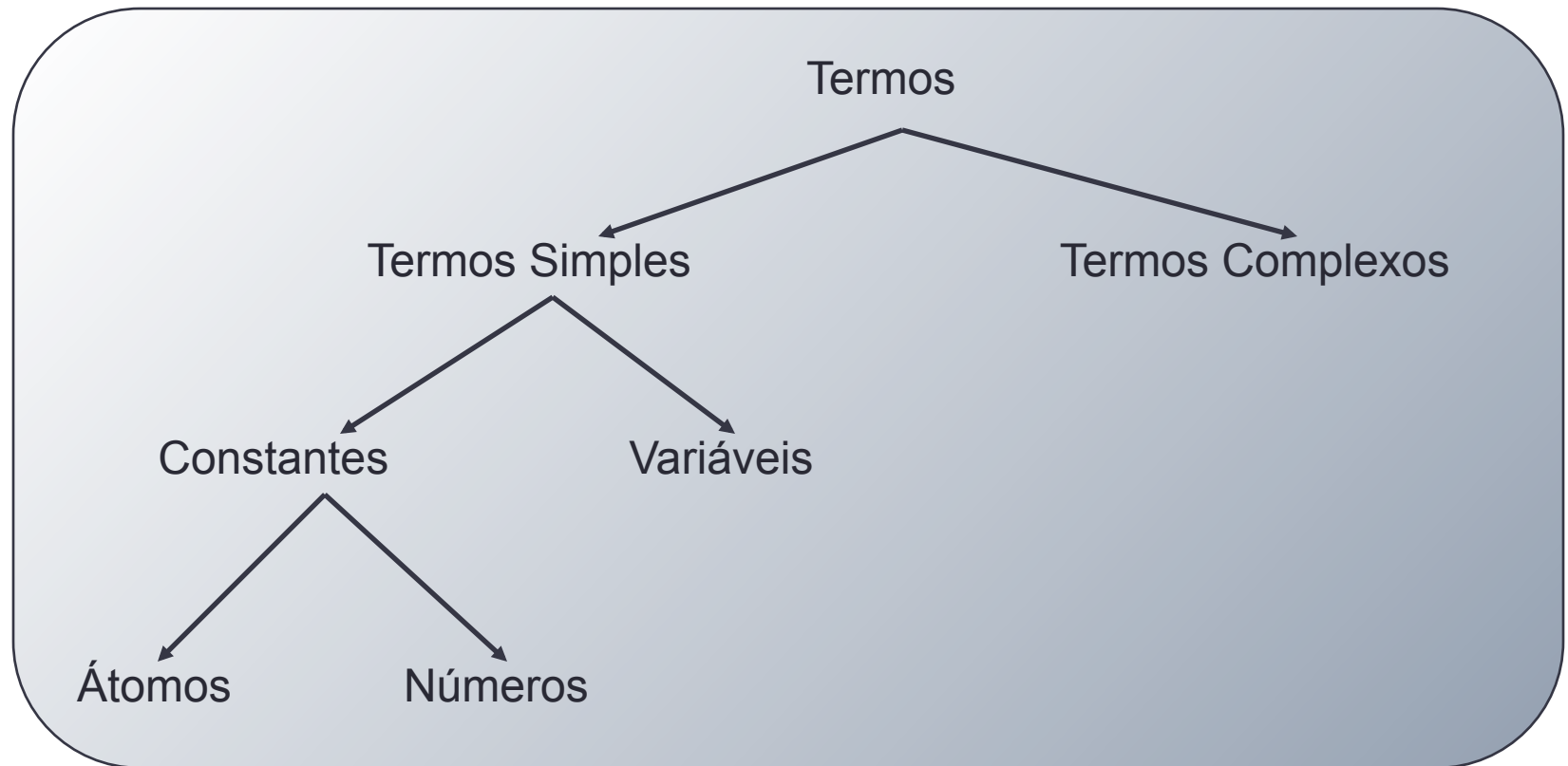
Sumário predicados de comparação

=	Unificação
\=	Negação da unificação
==	Igualdade
\==	Negação da igualdade
:=	Igualdade aritmética
:=\=	Negação da igualdade aritmética

Examinar os termos

- Existem predicados Prolog especializados em examinar termos
 - Predicados que identificam o tipo do termo
 - Predicados fornecem informação sobre a estrutura interna dos termos

Tipos de termos



Verificar o tipo de um termo

atom/1	<i>O argumento é um átomo?</i>
integer/1	<i>... um inteiro?</i>
float/1	<i>... um número real?</i>
number/1	<i>... um inteiro ou número real?</i>
atomic/1	<i>... uma constante?</i>
var/1	<i>... uma variável não-instanciada?</i>
nonvar/1	<i>... uma variável instanciada ou outro termo que não seja uma variável não-instanciada?</i>

Verificar o tipo: atom/1

```
?- atom(a).
```

```
yes
```

```
?- atom(7).
```

```
no
```

```
?- atom(X).
```

```
no
```

Verificar o tipo: atom/1

?- X=a, atom(X).

X = a

yes

?- atom(X), X=a.

no

Verificar o tipo: atomic/1

```
?- atomic(mia).
```

```
yes
```

```
?- atomic(5).
```

```
yes
```

```
?- atomic(loves(vincent,mia)).
```

```
no
```

Verificar o tipo: var/1

?- var(mia).

no

?- var(X).

yes

?- X=5, var(X).

no

Verificar o tipo: nonvar/1

```
?- nonvar(X).
```

```
no
```

```
?- nonvar(mia).
```

```
yes
```

```
?- nonvar(23).
```

```
yes
```

A estrutura dos termos

- Dado um termo complexo que informação pode ser extraída?
 - O functor
 - A aridade
 - O argumento
- O Prolog tem um predicado especializado para obter esta informação

O predicado functor/3

- O predicado functor/3 permite obter o functor e a aridade de um termo complexo

```
?- functor(friends(lou,andy),F,A).  
    F = friends  
    A = 2  
yes
```

```
?- functor([lou,andy,vicky],F,A).  
    F = .  
    A = 2  
yes
```

functor/3 e constantes

- O que acontece se usarmos functor/3 com constantes?

```
?- functor(mia,F,A).  
    F = mia  
    A = 0  
yes
```

```
?- functor(14,F,A).  
    F = 14  
    A = 0  
yes
```

functor/3 para construir termos

- O functor/3 pode ser usado para construir termos.

```
?- functor(Term,friends,2).  
   Term = friends(_,_)  
yes
```

Verificação de termos complexos

```
complexTerm(X):-  
    nonvar(X),  
    functor(X,_,A),  
    A > 0.
```

Argumentos: arg/3

- O predicado arg/3 pode ser usado para obter informação sobre os argumentos de um termo complexo
- Tem três argumentos:
 - Um número N
 - Um termo complexo T
 - O N-ésimo argumento de T

?- arg(2,likes(lou,andy),A).

A = andy

yes